І. М. Дудзяний

ПРОГРАМУВАННЯ МОВОЮ С++

Частина 1. Парадигма процедурного програмування

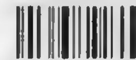
Міністерство освіти і науки України
Львівський національний університет імені Івана Франка

І. М. Дудзяний

Програмування мовою С++

Частина 1. Парадигма процедурного програмування

Навчальний посібник

НБ ПНУС

788344

Рекомендовано
Міністерством освіти і науки України

Львів 2013

32.973я73

УДК 004.438 С++ (075.8)
ББК 3973.2–018я73 С++
Д 81

Рецензенти:

д-р фіз.-мат. наук, проф. *В. А. Кривень*
(Тернопільський державний технічний університет імені Івана Пулюя);
д-р фіз.-мат. наук, проф. *І. В. Огірко*
(Українська академія друкарства, м. Львів);
канд. фіз.-мат. наук, доц. *І. І. Лазурчак*
(Дрогобицький державний педагогічний університет)

Рекомендовано Міністерством освіти і науки України
як навчальний посібник для студентів вищих навчальних закладів
(лист № 14/18–Г–1484 від 23.06.08 р.)

Дудзяний І.М.

Д 81 Програмування мовою С++. Частина 1 : Парадигма процедурного програмування : навчальний посібник / І. М. Дудзяний. – Львів : ЛНУ імені Івана Франка, 2013. – 468 с.

ISBN 978-617-10-0016-2
ISBN 978-617-10-0017-9 Ч. 1

У посібнику систематизовано базові засоби процедурного програмування мовою С++ в інтегрованому середовищі розробки Microsoft Visual Studio 2010: елементарні типи даних, визначення змінних і констант, конструкції галужень і циклів, масиви, вказівники, рядки та функції. Головну увагу звернуто на реалізацію типових алгоритмів опрацювання скалярних величин, скінченних послідовностей, рядів, цілих чисел, масивів, рядків тощо. Розглянуто реалізацію класичних алгоритмів розв'язування задач комбінаторної оптимізації на базі жадібного методу та методу динамічного програмування.

Для студентів вищих навчальних закладів, які вивчають сучасні інформаційні технології.

Прикарпатський національний університет імені Василя Стефаника код 02125266 НАУКОВА БІБЛІОТЕКА

УДК 004.438 С++ (075.8)
ББК 3973.2–018я73 С++

ISBN 978-617-10-0016-2
ISBN 978-617-10-0017-9 Ч. 1

© Дудзяний І. М., 2013
© Львівський національний університет імені Івана Франка, 2013

Зміст

ПЕРЕДМОВА 7
1. **ШВИДКИЙ СТАРТ** 9
1.1. Перші програми на С++ 9
1.2. Підготовка програми С++ до виконання 21
1.3. Директиви передпроцесора 25
1.4. Введення / виведення даних 31
1.5. Консольні застосування Microsoft Visual C++ 36
1.5.1. Загальні положення 36
1.5.2. Створення консольних програм ISO/ANSI C++ 38
1.5.3. Особливості виконання програм у CLR 43
1.5.4. Бібліотека класів платформи .NET 46
1.5.5. Створення керованих консольних програм 49
Запитання для самоперевірки..... 50
Завдання для програмування..... 50
2. **РОБОТА З ДАНИМИ** 51
2.1. Символи та лексеми 51
2.2. Вбудовані типи даних..... 53
2.3. Визначення змінних 60
2.4. Конструювання виразів 67
2.5. Перетворення типів даних 72
2.6. Приклади розробки програм 75
2.7. Початкове знайомство з функціями 84
2.8. Робота з даними у С++/CLI 102
2.8.1. Спільна система типів платформи .NET 102
2.8.2. Форматування під час виведення результатів 105
2.8.3. Приклади розробки керованих консольних програм 108
Запитання для самоперевірки..... 112
Завдання для програмування..... 113

3. **ГАЛУЖЕННЯ І ЦИКЛИ** 121
3.1. Опис конструкцій та інструкцій C++ 121
3.2. Конструкції галуження та вибору 125
3.3. Конструкції циклу 133
3.4. Програми з простим повторенням 138
3.4.1. Цикли з покроковим уведенням даних 138
3.4.2. Тестування програми за допомогою випадкових чисел 141
3.4.3. Цикли з покроковим виведенням даних 144
3.4.4. Обчислення скінченних сум і добутків 149
3.4.5. Найпростіші рекурентні співвідношення 153
3.5. Обчислення із заданою точністю 160
3.5.1. Обчислення числових рядів 160
3.5.2. Обчислення степеневих рядів 166
3.5.3. Обчислення визначених інтегралів 171
3.5.4. Розв'язування рівнянь з однією змінною 180
3.6. Рекурсивні та ітераційні процеси 183
3.6.1. Загальні методи конструювання алгоритмів .. 183
3.6.2. Реалізація найпростіших рекурентних співвідношень 187
3.6.3. Зменшення розміру задачі на постійний множник 198
3.6.4. Рекурсивна функція та стек 207
3.7. Програмування задач цілочислової арифметики 217
3.7.1. Теоретичні положення 217
3.7.2. Визначення найбільшого спільного дільника 222
3.7.3. Прості числа 226
3.7.4. Виокремлення та опрацювання цифр числа 233
Запитання для самоперевірки 237
Завдання для програмування 239

4. **МАСИВИ ТА ВКАЗІВНИКИ** 251
4.1. Масиви 252
4.1.1. Одновимірні масиви 252
4.1.2. Багатовимірні масиви 260
4.2. Вказівники 262
4.2.1. Загальні положення 262
4.2.2. Зв'язок масивів і вказівників 264
4.2.3. Динамічний розподіл пам'яті 265
4.3. Взаємодія функцій та вказівників 271
4.3.1. Передача функції аргументів-масивів 271
4.3.2. Вказівник на функції 280
4.4. Багатомодульні проекти 285
4.4.1. Відокремлене компілювання модулів 285
4.4.2. Шаблон багатомодульного консольного проекту 291
4.4.3. Узагальнений алгоритм обчислення визначених інтегралів 296
4.4.4. Сервісний модуль обробки масивів 299
4.5. Стандартні задачі обробки одновимірних масивів 303
4.5.1. Загальні положення 303
4.5.2. Схема Горнера обчислення многочлена 305
4.5.3. Прості методи сортування масиву 308
4.5.4. Визначення сусіднього розміщення елементів масивів 313
4.5.5. Переміщення елементів масиву 316
4.5.6. Злиття двох упорядкованих масивів 320
4.5.7. Алгоритми пошуку 321
4.6. Рядки символів у стилі С 325
4.6.1. Загальні положення 325
4.6.2. Стандартні функції обробки рядків 330
4.6.3. Кодування символів 335

4.6.4. *Функції* Windows API *відображення літер кирилиці* ... 339
4.6.5. *Засоби локалізації у стилі* C ... 344
4.6.6. *Стандартні функції класифікації символів* ... 351
4.6.7. *Сканування формату рядка* ... 355
Запитання для самоперевірки ... 361
Завдання для програмування ... 362
5. ЗАДАЧІ КОМБІНАТОРНОЇ ОПТИМІЗАЦІЇ ... 367
5.1. Вступ у теорію оптимізації ... 368
5.1.1. *Постановка задачі оптимізації* ... 368
5.1.2. *Задачі безумовної оптимізації* ... 370
5.1.3. *Задачі умовної оптимізації* ... 371
5.1.4. *Задачі математичного програмування* ... 373
5.1.5. *Методи оптимізації* ... 375
5.2. Метод динамічного програмування ... 378
5.2.1. *Загальні положення* ... 378
5.2.2. *Реалізація однопараметричного методу* ДП ... 380
5.2.3. *Реалізація двопараметричного методу* ДП ... 394
5.3. Метод динамічного програмування розв'язування задач про наплічник ... 407
5.4. Метод динамічного програмування на графах ... 414
5.4.1. *Базові поняття теорії графів* ... 414
5.4.2. *Постановка задачі про найкоротші шляхи* ... 420
5.4.3. *Алгоритм Флойда-Уоршолла* ... 424
5.5. Застосування жадібного методу ... 433
5.5.1. *Загальні положення* ... 433
5.5.2. *Виокремлення мінімального каркаса в простих графах* ... 435
5.5.3. *Алгоритм Пріма* ... 437
5.5.4. *Алгоритм Краскала* ... 441
5.5.5. *Алгоритм Дейкстри* ... 446
5.5.6. *Алгоритм Беллмана-Форда* ... 451
Запитання для самоперевірки ... 457
Завдання для програмування ... 457
БІБЛІОГРАФІЧНИЙ СПИСОК ... 459
ПРЕДМЕТНИЙ ПОКАЖЧИК ... 463

ПЕРЕДМОВА

Мову C++ спроектовано і розроблено у фірмі Bell Laboratories (США) упродовж 1979 – 1983 рр. як розширення (надмножина) мови C. Розробник мови C++ Б. Страуструп, окрім мови C, називає ще одне джерело C++ – мову Simula67, з якої запозичено поняття *класу*.

Класи реалізують парадигму *об'єктно-орієнтованого програмування* (ООП), за якою програми проектують у вигляді множини взаємодіючих об'єктів. Структуру і поведінку цих об'єктів описують ієрархічними класами – абстрактними типами даних з відкритим інтерфейсом і прихованою внутрішньою реалізацією. Класи уможливлюють також ініціалізацію даних, неявне перетворення типів, механізми перевантаження операцій тощо.

Парадигма програмування – це система ідей та понять, які визначають стиль написання програм (способи організації обчислень; питання структурування програми тощо). Парадигма програмування не визначається однозначно мовою програмування; усі сучасні мови програмування у тій чи іншій мірі допускають використання різних парадигм.

Мова C++ успадкувала від мови C парадигму *процедурного програмування*, згідно з якою програма – це послідовність процедур (*функцій* у C++). Функція – це поіменована послідовність інструкцій, які необхідно виконати. Будь-яку функцію можна викликати з довільної точки програми, включаючи інші функції або ж її саму (*рекурсивний* виклик функції).

З моменту винаходу мову C++ періодично оновлювали та доповнювали новими засобами. На початку 1990 року було сформовано об'єднаний комітет ISO/ANSI (International Standards Organization / American National Standards Institute) для стандартизації мови C++. Цей комітет 25 січня 1994 року прийняв перший проект стандарту C++, в який було включено усі засоби, уперше визначені Б. Страуструпом, і додано деякі нові.

Незабаром після цього відбулася подія, що змусила розширити існуючий стандарт. Мова йде про створену Олександром Степановим стандартну бібліотеку шаблонів (Standard Template Library, STL), яка значно розширила можливості C++. Однак долучен-

ня STL істотно сповільнило процес стандартизації C++. Кінцевий результат роботи комітету – ISO/ANSI-стандарт мови C++ вийшов у світ 1998 року (цю специфікацію ще називають Standard C++).

Існує безліч систем програмування, які підтримують Standard C++, з яких слід виокремити Visual C++ корпорації Microsoft. Інтегроване середовище розробки Microsoft Visual Studio підтримує дві різні версії мови C++: за стандартом ISO/ANSI, і нову версію – C++/CLI (Common Language Infrastructure), розроблену Microsoft і затверджену у стандарті ECMA (European Association for Standardizing Information and Computer Systems). Ці дві версії C++ взаємно доповнюють одна одну і відіграють різні ролі.

Версія мови ISO/ANSI C++ призначена для розробки високопродуктивних застосувань, які компілюються у "рідний" (native) код процесора. Версію мови C++/CLI розроблено спеціально для виконання на платформі .NET Framework.

У першій частині навчального посібника розглянуто парадигму процедурного програмування ISO / ANSI-стандарту мови C++. Посібник призначено для студентів, які вивчають програмування на базі C++. У зв'язку з цим наведено велику кількість прикладів, які демонструють використання базових конструкцій мови. Приклади підібрано так, щоб покращити навички студентів у застосуванні *типових алгоритмів* опрацювання скалярних величин, скінченних послідовностей, рядів, цілих чисел, масивів, рядків тощо.

Для студентів, що спеціалізуються у галузі теорії оптимальних процесів, корисним буде ознайомлення з п'ятим розділом посібника, в якому висвітлено питання програмування жадібного методу та методу динамічного програмування для розв'язування типових задач комбінаторної оптимізації.

Усі необхідні теоретичні положення з елементарної, дискретної чи вищої математики у конспективній формі наведено у посібнику. Нумерацію формул, рисунків, прикладів і таблиць подано у межах розділу. Під час посилання на ці елементи з інших розділів зліва через крапку вказано номер розділу, в якому розміщено елемент.

Приклади протестовано автором в інтегрованому середовищі розробки Visual Studio 2010.

1. ШВИДКИЙ СТАРТ

План викладу матеріалу:

1. Перші програми на C++.
2. Підготовка програми C++ до виконання.
3. Директиви передпроцесора.
4. Введення/виведення даних.
5. Консольні застосування Microsoft Visual C++.

Ключові терміни розділу

- *Логічна структура програми*
- *Фізична структура програми*
- *Лексеми та сутності*
- *Інструкції та директиви*
- *Блоки; заголовки блоків*
- *Визначення сутностей*
- *Сутності-об'єкти*
- *Сутності-суб'єкти*
- *Коментарі*
- *Консольна програма*
- *Визначення функцій*
- *Головна функція*
- *Стандартний потік виведення*
- *Стандартний потік введення*
- *Оператор виведення даних*
- *Оператор витягання даних*
- *Простори назв*
- *Маніпулятор* `endl`
- *Файл специфікацій*
- *Файл реалізацій*
- *Бібліотека C-функцій*
- *Бібліотека класів C++*
- *Передпроцесор*
- *Об'єктний модуль*
- *Виконавчий модуль*
- *Директиви передпроцесора*
- *Компілювання програми*
- *Компонування програми*
- *Заголовки у новому стилі*
- *Заголовки у старому стилі*
- *Маніпулятори потоку*
- *Типи маніпуляторів*
- *Консольне застосування C++*
- *Віконне застосування C++*
- *Каркаси класів (VCL, Framework)*
- *Середовище виконання (CLR)*

1.1. Перші програми на C++

Під час вивчення мови програмування найбільші труднощі полягають у тому, що усі елементи та сутності мови є взаємозв'язаними, отож не вдається розглянути довільний елемент чи сутність ізольовано та незалежно від інших елементів / сутностей.

Іншими словами, послідовний (або лінійний) стиль викладу матеріалу тут не підходить, оскільки майже постійно необхідно

опиратися на речі, які детально описано далі за текстом. Тобто доцільно або подавати відповідні посилання у тексті, що є малопродуктивним, або коротко описувати необхідні речі.

Метою цього параграфа є короткий опис деяких елементів і сутностей, які траплятимуться майже у кожній програмі. Щодо них автор не заглиблюватиметься у деталі, оскільки опис таких елементів і сутностей наведено у наступних параграфах цього розділу та подальших розділах посібника.

У зв'язку з цим необхідно зазначити: уважний читач у тексті може відшукати певні повторення фрагментів тестів (можливо з певними модифікаціями); автор не вважає це великим огріхом, а навпаки – певним прийомом, що полегшує вивчення мови програмування.

Окрім цього, чимало читачів усе одно переглядатиме посібник "*по діагоналі*", виловлюючи необхідну їм інформацію для виконання індивідуальних завдань, написання курсових тощо.

Термін "*програма*" має багато аспектів (чи точок зору на термін), отож і трактування цього терміна залежить від контексту обговорення (чи вивчення). У цьому розділі термін "*програма*" нас цікавитиме з точки зору фізичної та логічної структури *консольної* програми (чи консольного застосування).

Фізична структура консольної програми (або *набору файлів* визначеного типу та призначення) в інтегрованому середовищі Microsoft Visual Studio 2010 буде об'єктом вивчення у наступному параграфі цього розділу. У цьому ж параграфі обмежимося розглядом логічної структури консольної програми.

Логічна структура програми – це набір визначень та оголошень *сутностей* програми, який дає змогу реалізувати *алгоритм* розв'язку певної задачі. Сутності формують за допомогою *лексем* (синоніми: елементи мови; елементарні частини мови тощо).

Найпростіше розібратися з *лексемами* – це ідентифікатори, ключові слова, літерали та роздільники (дужки, знаки операцій (оператори) та різноманітні невидимі символи пропусків). Здебільшого читачам зрозумілі ці терміни, оскільки вони простежуються в усіх мовах програмування (детальніше див. розділ 2).

Оскільки лексеми – це елементарні частини мови, то з них формують визначення та оголошення *сутностей* програми. Питання: що таке *сутність*? Відповідь: усе, що не є лексемами!

Так, у жартівливій формі, на початку вивчення мови можна відповісти на це дуже складне питання, оскільки термін "*сутність*" теж має дуже багато аспектів. Щоб не заглиблюватися у ці аспекти, а головне – не заплутати читача, автор коротко і дуже популярно спробує відповісти на це питання.

На початку вивчення С++ здебільшого читачі не розрізняють термінів "*визначення сутності*" та "*оголошення сутності*", вважаючи їх синонімами. Однак це зовсім не так (детальніше див. розділ 2). Оскільки у програмах найчастіше вживають "*визначення сутностей*", то у цьому розділі переважно говоритимемо про нього.

Визначення та оголошення сутностей реалізують за допомогою *інструкцій* та *директив препроцесора* (формально: інструкція чи директива – це сутності, які дають змогу визначити чи оголосити інші сутності програми). Ознакою інструкції слугує *крапка з комою*, яка розміщена у кінці інструкції. Приклади інструкцій:

```
double a, b, c, p;
p = (a+b+c)/2;
S = sqrt(p*(p-a)*(p-b)*(p-c));
```

Іноді однієї інструкції для визначення сутності замало! Тоді усі інструкції, які її визначають, об'єднують у *блок* – послідовність інструкцій, обмежену фігурними дужками: відкриваючою ("{") і закриваючою ("}"). Блок – це інструкція (тобто блок у програмі може стояти на будь-якому місці, де може стояти інструкція).

Перед деякими блоками мають стояти *заголовки* – сутності, за допомогою яких визначають *типи* інших *сутностей*. Заголовки дають змогу компіляторові відрізнити одні сутності від інших. Наприклад: *функція*, *клас*, *структура* мають різні заголовки.

Приклад визначення *функції*:

```
int sign(double x)
{ return x<0? -1 : (x>0? 1 : 0); }
```

У цьому визначенні заголовком слугує `int sign(float x)`, а фігурні дужки обмежують блок (або *тіло* функції). Хоча блок

містить тільки одну інструкцію, однак фігурні дужки тут обов'язкові, оскільки вони *обмежують* функцію у програмі. Заголовок подібний до інструкції, однак ставити після нього крапку з комою не можна!

Принагідно дещо скажемо про функції. Функція об'єднує послідовність інструкцій під певною назвою і може повертати *одне значення* певного типу. Функція, зазвичай, отримує певні дані (*аргументи*) для обробки. У наведеному вище фрагменті програми визначена функція `sign`, що має тип `int` (функція повертає значення знаку числа (±1) чи нуль для нульового значення аргументу).

Програма складається з однієї або декількох функцій, серед яких має бути *головна функція* з назвою `main`. Якщо програма складається з однієї функції, то це має бути головна функція.

Для кращого розуміння логіки програми у тексті програми варто використовувати *пояснення* (чи *коментарі*). Чимало програмістів через півроку-рік не можуть одразу ж пояснити власну програму; добре продумані коментарі полегшують читання та розуміння як чужих, так і власних програм. Компілятор коментарі ігнорує!

У C++ розрізняють два типи коментарів: *багаторядковий*:

```
/* Текст, який може розташовуватися
   на декількох рядках */
```

та *однорядковий*:

```
// Текст після рисок до кінця рядка – коментар.
```

Попереднє визначення функції з коментарями:

```
/* Визначення функції sign, яке складається із
   заголовка і блоку (тіла функції) */
int sign(double x) // Заголовок функції
// Фігурні дужки обмежують блок (тіло функції)
{ return x<0? -1 : (x>0? 1 : 0); }
```

Однак ніщо не заважає записати визначення і так:

```
// Визначення функції sign, яке складається із
// заголовка і блоку (тіла функції)
int sign(double x) // Заголовок функції
// Фігурні дужки обмежують блок (тіло функції)
{ return x<0? -1 : (x>0? 1 : 0); }
```


Отже, поняття сутності починає потроху прояснятися – ми ще зовсім не зрозуміли змісту сутностей, однак уже знаємо, що є такі сутності, як *функція*, *клас*, *структура*. Застосовують також інші сутності – *об'єднання*, *перелік*, *файл* тощо.

Отже, заголовки стосуються *складних* сутностей, визначення яких базується на блокові. Тепер перейдемо до сутностей, які визначають однією інструкцією. Іноді сутності поділяють на *сутності-об'єкти* і *сутності-суб'єкти* (чи на *об'єкти* і *суб'єкти*).

До *об'єктів* зачислимо сутності, для яких компілятор виокремлюватиме осередки пам'яті комп'ютера, в яких зберігатимуться їхні значення. Це можуть бути константи, змінні, масиви, екземпляри структур чи класів тощо. Інструкції, які реалізують визначення (чи оголошення) *сутностей-об'єктів*, називатимемо інструкціями *опису*.

В інструкції опису

```
double x, y=5.3;
```

визначено дві змінні з *назвами* (ідентифікаторами) `x` та `y`, що мають тип `double` (раціональний тип подвійної точності). Слово `double` – це *ключове* (зарезервоване) слово. Відповідно до цього опису компілятор виокремить у пам'яті комп'ютера два осередки по 8 байтів (один з них він ототожнюватиме зі змінною `x`, а інший – зі змінною `y`).

Окрім цього, в осередку пам'яті, що ототожнюється зі змінною `y`, буде розміщено її початкове значення `5.3` (тобто водночас із визначенням змінної відбувається її *ініціалізація*). Початкове значення змінної `x` залежатиме від контексту, в якому розташована інструкція опису (це буде `0`, якщо `x` – глобальна чи статична змінна; або довільне (нефіксоване) значення, якщо `x` – локальна змінна). У процесі виконання програми значення змінних (тобто значення, які розміщені у відповідних осередках пам'яті) можуть змінюватися.

Позначка (чи *модифікатор*) `const` у визначенні змінної вказує на те, що значення цієї змінної у програмі змінювати не можна (за цим строго стежитиме компілятор). Таку змінну називають *константою*. Під час визначення константи обов'язково має бути ініціалізація.

Приклади визначення констант:

```
const double pi = 3.1415;
const char plus = '+'; // Константа-символ (char)
```

До суб'єктів належатимуть сутності, для яких компілятор формує послідовності *машинних команд*. Інструкції, які реалізують визначення *сутностей-суб'єктів*, називатимемо інструкціями *дії*. Оскільки більшість інструкцій програми є власне інструкціями дії, то, зазвичай, слово "*дія*" опускають.

У програмі найчастіше трапляється *інструкція виразу*, яка містить вираз, за яким стоїть крапка з комою, наприклад:

```
a=b*3+c; d++; x--; y+=x; // 1 - Чотири інструкції виразу
```

Цей же фрагмент програми можна записати за допомогою однієї інструкції виразу, яка складається з послідовності виразів, розділених комою:

```
a=b*3+c, d++, x--, y+=x; // 2 - Одна інструкція виразу
```

Обидва фрагменти компілюються майже однаково та у підсумку дають одні і ті ж самі результати, однак фрагмент 2 є кориснішим за фрагмент 1, оскільки послідовність виразів компілятор інтерпретує як *єдиний* вираз. Отже, фрагмент 2 дає змогу розмістити послідовність виразів там, де, згідно із синтаксисом, можна розміщувати тільки один вираз.

На початку цього параграфа ми декларували про наміри описати логічну структуру *консольної* програми. Отже, ми уже дещо знаємо про визначення та оголошення *сутностей*, однак ще нічого не знаємо про те, як їх об'єднати в єдиний *набір* (логічну структуру). На це запитання ми отримаємо відповідь під час розробки першої програми. Залишилося ще з'ясувати, що таке *консоль*.

Програми, які працюють в операційних системах Windows, прийнято називати *додатками* (або *застосуваннями*). Інтерфейс користувача у таких застосуваннях має вигляд *форми*, на якій розміщені елементи керування (поля, індикатори, списки, кнопки тощо) для введення / відображення інформації та керування роботою додатка.

Інтегроване середовище Microsoft Visual Studio 2010 дає змогу створювати і простіші, так звані *консольні*, застосування (програми), в яких для введення / виведення даних використовують ***консоль*** (об'єднання клавіатури та екрана дисплея). За домовленістю, введення даних у консольному режимі здійснюється з клавіатури, а виведення – на екран дисплея. Такі застосування дуже зручні під час початкового вивчення мови – дають змогу ознайомитися з базовими сутностями, не відволікаючи на деталі організації інтерфейсу користувача.

Хоча консольне застосування створюють у Microsoft Visual Studio 2010, однак його виконання компілятор *імітує* як виконання в операційній системі MS DOS (це робиться з метою сумісності з попередніми версіями мови C++ та для сумісності з мовою C).

Як відомо, у MS DOS використовують кодову таблицю CP866, а у Windows – CP1251, в яких коди *кириличних* символів мають *різне* кодування. Повідомлення (або рядок символів), сформоване у <u>тексті програми</u> кириличними символами за допомогою текстового редактора коду Microsoft Visual Studio 2010, у пам'яті комп'ютера зберігається у кодах CP1251.

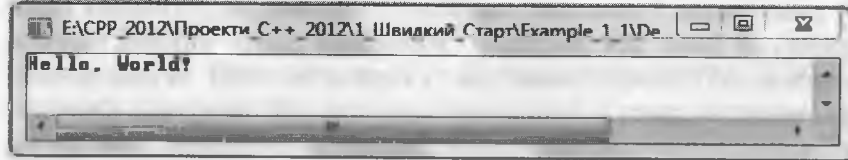
Під час виведення на консоль компілятор інтерпретує збережені символи, як символи у кодах CP866. Для латинських літер та спеціальних символів це не має жодного значення (в обох кодових таблицях вони мають однакові коди), однак під час виведення на консоль кириличних символів отримуємо "*абракадабру*" – послідовність незрозумілих символів псевдографіки. Отож повідомлення консольних застосувань, зазвичай, записуватимемо латинськими літерами. Як з цим впоратися, розповімо у четвертому розділі.

Приклад 1. Розробити найпростішу програму виведення на екран "*класичного*"[1] повідомлення-привітання.

```
#include <iostream>
void main()
{ std::cout<< "Hello, World!"; char h; std::cin>>h; }
```

[1] Майже в усіх книжках по C++ подану таку програму (автор не бажає порушувати цієї традиції, яка бере початок від Б. Страуструпа – самого творця мови C++).

Результати *тестування* програми:



Записане у тексті програми повідомлення (або рядок символів) "Hello, World!" виводиться на екран консолі. Рядок символів, розміщений безпосередньо у тексті програми, називають *літералом*.

У першому рядку розміщена *директива* передпроцесора

```
#include <lostream>
```

Завдяки їй підключається стандартна бібліотека введення/виведення iostream, без якої компілятор не зрозуміє, що назва cout позначає стандартний *потік виведення*, а назва cin – стандартний *потік введення*.

У цій програмі потік cout використовують для виведення на екран рядка символів "Hello, World!" (лапки тільки обмежують рядок; на екрані лапки не відображаються), а потік cin – для організації *паузи* – програма очікуватиме введення довільного символу для змінної h (визначення char h).

Без цієї паузи неможливо було б прочитати повідомлення на консолі, оскільки воно дуже швидко зникає. Надалі паузу в програмі організовуватимемо без додаткових пояснень. В окремих випадках інструкцію для паузи необхідно *повторювати двічі* (це зв'язано з деякими тонкими нюансами під час обміну рядками між консоллю та пам'яттю комп'ютера).

Як уже зазначено, програма містить одну або декілька функцій, серед яких має бути *головна функція* (main). Наша програма складається тільки з головної функції. Оскільки функція не повертає жодних значень, то вона має тип void.

Функція main може повертати значення процесу (зазвичай, це операційна система), який її запускав. Здебільшого в операційних

системах значення 0 свідчитиме про успішне завершення програми; інші значення свідчитимуть про відповідні помилки.

Якщо виникає потреба, щоб функція main повертала відповідне значення процесу, який її запускав, то їй необхідно присвоїти тип int (тобто заголовок має виглядати так: int main()). У деяких джерелах наполегливо радять для функції main завжди використовувати тип int (це, на думку авторів, полегшує процес тестування програми). Оскільки програми у цьому посібнику доволі прості та прозорі, то автор на цьому питанні не зациклюватиметься: в одних випадках використовуватиме void, а в інших – int.

Тепер пояснимо використання у програмі виразу std::, що містить оператор *розширення області видимості* (::). Однак усе це зробимо поступово.

Стандарт ISO/ANSI узаконив поняття *простору назв* (namespace), який дає змогу *локалізувати* (обмежити) *область видимості* назв різноманітних елементів мови С++, оголошених та визначених у цьому просторі. Загальний формат завдання простору назв:

```
namespace назва
{
  // Оголошення і визначення
}
```

Наприклад:

```
namespace MyNS { int count; … }
```

На назви, уведені всередині простору назв, можуть прямо посилатися інструкції цього ж простору. Поза простором назв доступ до необхідної назви можна отримати двома способами. По-перше, можна використати операцію розширення області видимості, наприклад:

```
MyNS::count=12;
```

По-друге, можна використати інструкцію using (інструкцію використання простору назв), яка розширює поточну область видимості за рахунок простору назв, заданого цією інструкцією, наприклад:

```
using namespace MyNS;
…; count=12; // назва з простору назв MyNS
```

Прикарпатський національний університет імені Василя Стефаника код 02125266 НАУКОВА БІБЛІОТЕКА Інв. №

До появи конструкції namespace стандартні функції оголошували в *єдиному* (безіменному) глобальному просторі назв. Стандарт ISO/ANSI зобов'язує розробників компіляторів оголошувати стандартні функції у просторі назв std. Отже, у нашій програмі ми використали вираз std:: для того, щоб скористатися назвами об'єктів (потоків) cout та cin з простору назв std. Зручніше користуватися другим (його інколи називають "лінивим"), який полягає у використанні інструкції using.

У рядку програми можна записувати декілька інструкцій. Однак, з погляду простоти читання та розуміння тексту програми, цим не треба зловживати. Рекомендують виокремлювати інструкції у групи, використовуючи для цього відступи і коментарі.

Орієнтуючись на усе вищезазначене, запишемо нашу першу *програму* у такому вигляді:

```
#include <iostream>       // Декларація підключення
                          // бібліотеки
using namespace std;      // Інструкція підключення
                          // простору назв
int main()                // Заголовок головної функції
{
 cout<< "Hello, World!"; // Виведення повідомлення
 char h; cin>>h;          // Організація паузи
 return 0;                // Успішне завершення програми
}
```

Отже, у програмі для виведення рядків використовують *стандартний потік виведення* cout. Подвійний знак "*менше*" (<<) позначає оператор *виведення* даних у вихідний потік (за домовленістю – на екран дисплея). Оператор виведення використовують також для виведення окремих символів, чисел, значень змінних тощо.

Під час формування потоку виведення можна використовувати декілька операторів виведення у потік, наприклад:

```
cout<< "Chyslo " << 3.14159 << " zaokruglue PI";
```

Після виконаня інструкції на екрані отримаємо таке:

```
Chyslo 3.14159 zaokruglue PI
```


Іноді доводиться розміщувати дані на декількох рядках дисплея. Для переміщення курсора виведення на початок наступного рядка дисплея, у рядку символів розміщують *символ нового рядка* "\n". Якщо ж немає рядка символів для виведення, то "\n" можна просто розмістити усередині апострофів. Наприклад, наступна інструкція виводить числа 9, 10 і 11, кожне у своєму рядку:

```
cout <<'\n' << 9 << '\n' << 10 << '\n' << 11;
```

Для пересування курсора на початок наступного рядка можна також використовувати *модифікатор* потоку endl (кінець рядка). Попередній оператор можна записати й так:

```
cout <<endl << 9 << endl << 10 << endl << 11;
```

Для отримання даних, які набрані за допомогою клавіатури, використовують стандартний вхідний потік cin. При цьому вказують одну або декілька змінних, яким присвоюватимуть введені значення за допомогою операторів *витягання* даних з потоку (>>).

Для виокремлення даних у вхідному потоці (чисел, символів, рядків) використовують символи: пропуску, табуляції або повернення каретки (натискання клавіші Enter). Рядки / символи вводять, відповідно, без лапок / апострофів.

Приклад 2. Розробити програму виведення на екран повідомлення-запрошення на введення користувачем його прізвища та імені; отримати відповідь від користувача. Використати отримані дані для формування привітання користувача.

```
#include <iostream>
using namespace std;
void main() { char c[20]; // Визначення масиву символів
  cout<< "Enter your name and surname!\n";
  cin>>c;       // Введення даних користувачем
  cout<< "Hello, mr. "<< c<<"!\n";
  cin>>c; cin>>c;  // Організація паузи
}
```

Додатковий коментар № 1. Для зберігання даних, які вводитиме користувач, зарезервовано *масив* символів (char c[20]), який може максимально умістити 19 символів (1 байт зарезервовано для службового символу "\0" – ознаки закінчення рядка).

Результати *тестування* програми:

```
E:\CPP_2012\Проекти_С++_2012\1_Швидкий_Старт\Example-1-_
Enter your name and surname!
Вася Пупкін
Hello, mr. Вася!
```

Додатковий коментар № 2. Унаслідок тестування програми маємо дві новини: добру і погану. Почнемо з доброї: компілятор без проблем зчитує кириличні літери і відтворює їх знову на екрані (окрім букви "і", яку вводять з латинської розкладки клавіатури). Пояснення просте: компілятор отримані з консолі літери зберігає у пам'яті комп'ютера у кодовій таблиці CP866; під час виведення на екран також інтерпретує ці літери як коди CP866.

Новина погана: компілятор введену інформацію сприйняв не повністю (втрачено прізвище `Пупкін`)! Якщо вдуматися, то і тут можна знайти пояснення. Воно полягає у такому: оператор *витягання* даних з потоку (`>>`) вважає черговим значенням послідовність літер до пропуску (тобто `Вася`). Отже, у цьому випадку використовувати оператор (`>>`) не варто. У класі, який описує об'єкт `cin`, є спеціальний метод `getline`, який дає змогу вводити рядки з пропусками (ознакою закінчення цього рядка є натискання клавіші `Enter`). Наведемо фрагмент програми, в якому треба зробити виправлення:

```
...
  cout<< "Enter your name and surname!\n";
  cin.getline(c,20);    // Тут треба записати так!
...
```

Результати *тестування* виправленої програми:

```
E:\CPP_2012\Проекти_С++_2012\1_Швидкий_Старт\Example-1-_
Enter your name and surname!
Вася Пупкін
Hello, mr. Вася Пупкін!
```


1.2. Підготовка програми С++ до виконання

Програма на С++, як *фізична структура*, складається з множини *модулів*. Принцип модульності дуже важливий для побудови надійних і легко змінюваних програм (застосувань), адже будь-які зміни в окремому модулі не зачіпають решти модулів. Розрізняють *вихідні* (початкові), *об'єктні* та *виконавчі* модулі.

Текст програми мовою С++ може міститися в одному чи декількох *вихідних* модулях, його можна набирати у будь-якому текстовому редакторі. Текст вихідного модуля розділяють на *файл специфікацій* (файл з розширенням `.h`), який містить оголошення сутностей (класів, функцій тощо), і *файл реалізації* (файл з розширенням `.cpp`), який містить визначення цих сутностей.

Увага! Учбові програми, зазвичай, розміщують тільки в *одному* файлі реалізації (у ньому можуть бути й оголошення сутностей).

Оголошення (*declaration*) "розказують" компіляторові про тип і назву сутності, проте деталі в оголошеннях відсутні. Визначення (*definition*) забезпечує компілятор деталями. Наприклад: для *об'єкта* визначення – це те місце у програмі, де компілятор виділяє йому пам'ять; для *функції* визначення – це заголовок функції разом з тілом функції. Оголошення сутності, якщо воно наявне, має *завжди передувати* його визначенню.

Зазвичай, файли специфікацій містять *інтерфейсні частини* модулів – ті деталі, які мають бути доступними іншим модулям (директиви передпроцесора, оголошення функцій і класів, визначення глобальних констант і змінних тощо). *Файл специфікацій* ще називають *заголовним* файлом (від *header*). Файли реалізації містять визначення констант, змінних, методів класів, функцій тощо.

Вихідні модулі програм використовують різноманітні *стандартні функції*, які розміщені у спеціальних *бібліотечних файлах* компілятора. Існує два види бібліотек стандартних функцій: *бібліотека С-функцій* (підтримується *усіма* компіляторами С і С++) і *бібліотека класів* С++ (підтримується *тільки* компіляторами С++).

Для кожної стандартної функції, яку використовуватиме програма, у вихідному модулі треба попередньо вказувати її прототип, необхідний компіляторові для коректної роботи з пам'яттю.

Для правильного оголошення прототипів функцій з бібліотечних файлів компілятор C++ надає власні файли специфікацій, які й містять коректні прототипи цих функцій (та багато іншого).

Процес створення *виконавчого* модуля складається з трьох етапів: опрацювання директив передпроцесора, компілювання та компонування програми (рис. 1).

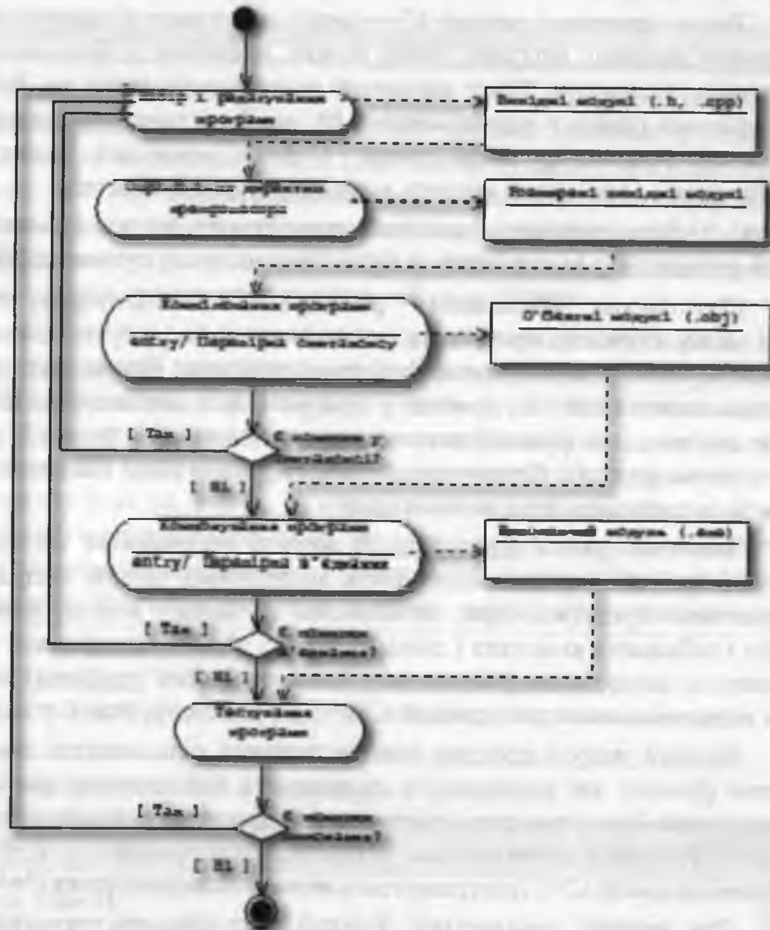


Рис. 1. Схема підготовки програми Visual C++ до виконання

На першому етапі працює *передпроцесор* (*preprocessor*) – програма, яка перед компілюванням програми виконує опрацювання *директив* (див. наступний параграф), розміщених у вихідних модулях.

Програма будь-який рядок, що починається символом #, сприймає як директиву передпроцесора. Перед символом # і після нього можуть стояти пропуски. Крапку з комою після директиви передпроцесора *не ставлять* (директива займає *тільки* один рядок)! Директиви передпроцесора, зазвичай, розміщують на початку файлу, хоча їх можна записувати у будь-якому місці програми.

Передпроцесор під час роботи:

- включає у текст програми файли специфікацій компілятора, які містять прототипи стандартних функцій, оголошення класів, визначення символічних констант тощо;
- виконує підстановку макророзширень замість назв макросів та їхніх аргументів;
- виконує умовну компіляцію.

Увага! Запуск передпроцесора відбувається *автоматично* під час компілювання програми. Результат роботи передпроцесора – розширені вихідні модулі, що є внутрішніми об'єктами (недоступними програмістові), з якими працює компілятор.

На другому етапі працює *компілятор* (*compiler*) – програма, яка переводить (транслює) кожний розширений вихідний модуль у машинний код. Результатом цього компілювання є набір *об'єктних модулів* (файлів з розширенням `.obj`) - для кожного розширеного вихідного модуля створюється власний об'єктний модуль.

Під час компілювання розширеного вихідного модуля компілятор може виявити синтаксичні помилки, що не даватиме йому змоги побудувати об'єктний модуль. Ці помилки необхідно виправити у редакторі коду та повторити компілювання модуля.

На третьому етапі працює *компонувальник* (*linker*) – програма, яка з'єднує коди об'єктних модулів з кодами стандартних функцій зі спеціальних бібліотечних файлів, а також з'єднує всі об'єктні модулі в єдине ціле – *виконавчий модуль* (файл з розширенням `.exe`).

Під час компонування програми компонувальник може виявити помилки з'єднання. Здебільшого це означає, що посилання на стандартні функції визначено помилково, або неправильно задані (чи взагалі відсутні) шляхи до файлів специфікацій тощо.

У цих випадках необхідно уточнити посилання на стандартні функції і/або правильно задати шляхи до файлів специфікацій компілятора. Після цього необхідно повторити компілювання та компонування програми.

Результати роботи виконавчого модуля перевіряють на *тестових* прикладах (тестування програми - це своєрідна наука), які повинні враховувати можливі варіанти для вхідних даних:

- значення даних належать допустимим діапазонам змін;
- значення даних розміщені на межах діапазонів змін;
- значення даних заборонені (перебувають поза межами діапазонів змін).

Під час тестування програми можуть виникати *помилки виконання*, які ініціюють аварійне завершення роботи програми. Існують також і *логічні помилки*, внаслідок яких програма видає неправильні результати, проте такі помилки не спричиняють аварійного завершення її роботи.

Якщо хоча б один тестовий приклад виконуватиметься некоректно, то необхідно визначити помилку, виправити вихідний модуль, здійснити компілювання та компонування програми і повторити тестові приклади.

Під час тестування програми доволі часто використовують *програму налагодження* (*debugger*), яка дає змогу реалізувати виконавчий модуль крок за кроком (трасування програми).

Програміст може самостійно підготувати програму C++ до виконання за допомогою командного рядка, активізуючи почергово компілятор командного рядка C++ та компонувальник.

Проте здебільшого програмісти для підготовки програм C++ до тестування та виконання використовують *інтегроване середовище розробки* (*Integrated Development Environment*, *IDE*), яке містить редактор коду, передпроцесор, компілятор, компонувальник, програму налагодження та інші інструменти.

1.3. Директиви передпроцесора

У програмах C++ найчастіше зустрічається директива `#include`, яка долучає до вихідних модулів файли специфікацій компілятора та інші текстові файли. Ця директива має два формати:

```
#include <заголовок_специфікацій>
#include "специфікація_файлу"
```

Перший формат використовують з метою долучення до тексту програми копій файлів специфікацій компілятора, а другий - з метою доповнення програми копіями текстових файлів, підготовлених програмістом. Наприклад:

```
#include "c:\func\func_cpp\MyFunc.h"
```

Якщо ж у лапках вказано тільки назву файлу, наприклад:

```
#include "MyUnit.h"
```

то передпроцесор шукає файл, переглядаючи каталоги у такій послідовності:

- каталог, який містить файл з цією директивою;
- каталоги, які містять файли, долучені раніше директивами `#include`;
- поточний каталог;
- каталоги, які задають опцією компілятора `/I`;
- каталоги, які задає змінна оточення `INCLUDE`.

Розрізняють два *стилі* вказівки `заголовка_специфікацій` (просто `заголовка`) у директиві `#include`: *старий* та *новий* стиль (уведено стандартом ISO/ANSI). У старому стилі `заголовок` відповідає назві відповідного файлу специфікацій, наприклад:

```
#include <math.h> // Математичні функції
#include <iostream.h> // Введення / виведення даних
```

Наведемо найуживаніші файли специфікацій С-функцій:

`ctype.h`	Робота з символами	`stdio.h`	Введення/виведення
`float.h`	Межі раціональних значень	`stdlib.h`	Різні змішані оголошення
`limits.h`	Межі значень типів	`string.h`	Робота з рядками
`math.h`	Математичні функції	`time.h`	Дата/час
`stddef.h`	Різні константи	`wchar.h`	Двобайтові символи

Наведемо найуживаніші файли специфікацій класів С++:

complex.h	Клас комплексних чисел
exception.h	Клас виняткових ситуацій
iomanip.h	Маніпулятори введення/виведення
ios.h	Класи введення/виведення нижнього рівня
iostream.h	Стандартні класи введення/виведення
istream.h	Класи роботи із вхідними потоками
numeric.h	Універсальні операції над числами
ostream.h	Класи роботи із вихідними потоками
sstream.h	Класи роботи із потоками-рядками
stdexcept.h	Клас стандартних виняткових ситуацій
cstring.h	Стандартний клас опрацювання рядків

У *новому* стилі заголовок_специфікацій - це абстракція (деяка *стандартна назва*), за якою компілятор відшукує необхідні файли. Ці заголовки компілятор опрацьовує так, як це йому необхідно (тобто заголовок може перетворитися у назву файлу, проте це є зовсім не обов'язковим).

Оскільки нові заголовки є стандартними назвами, для них не вказують розширення .h, а тільки заголовок у кутових дужках. Окрім цього, до назви файлу специфікацій С-функцій додають префікс с. Попередній приклад у новому стилі виглядає так:

```
#include <cmath>       // Математичні функції
#include <iostream>  // Введення та виведення даних
using namespace std; // Підключення простору назв
```

Отже, щоб компілятор міг правильно опрацьовувати заголовки у новому стилі, у програмі необхідно вказати простір назв std, в якому оголошені усі стандартні функції.

Старі назви файлів специфікацій класів С++ стандарт більше не підтримує, а стосовно заголовків С-функцій цього зроблено не було (через підтримку сумісності із С). Насправді ж у розробників компіляторів немає приводу відхрещуватися від успадкованого програмного забезпечення, отож вважатимемо, що старі заголовкові файли у деяких компіляторах ще можуть траплятися.

Якщо оцінити вищесказане з практичної точки зору, то ситуація з заголовками у Microsoft Visual C++ 2010 така:

- здебільшого заголовки С++ у старому стилі (зокрема, <iostream.h>) компілятор не підтримує;
- заголовки С++ у новому стилі (наприклад, <iostream>) наділені переважно тими ж можливостями, що й відповідні старі заголовки, однак їхня сукупність міститься у просторі назв std;
- як і раніше, підтримуються заголовки С (наприклад, <stdio.h>), розташовані поза простором назв std;
- заголовки С у новому стилі (наприклад, <cstdio>) наділені переважно тими ж можливостями, що й відповідні старі заголовки, однак їхня сукупність міститься у просторі назв std.

Ситуація на перший погляд видається дуже заплутаною, проте, насправді, її можна легко розтлумачити. Найбільші труднощі полягають у тому, щоб зорієнтуватися у заголовках для *рядків*:

- <string.h> – заголовок у старому стилі для функцій, що працюють з рядками типу char*;
- <cstring> – заголовок у новому стилі для функцій, що працюють з рядками типу char*;
- <cstring.h> / <bstring.h> – заголовок у старому стилі стандартного *класу* опрацювання рядків;
- <string> – заголовок у новому стилі стандартного *класу* опрацювання рядків.

Розглянемо тепер інші директиви. Директива визначення *макросу* має такий вигляд:

```
#define назва_макросу(параметри) макророзширення
```

Елемент назва_макросу задає певну назву, замість якої під час передпроцесорної обробки програми підставляється макророзширення. Елемент (параметри) – не обов'язковий! Якщо ж він все таки наявний, то замість параметрів у макророзширення попередньо підставляють відповідні аргументи. Директива скасування визначення макросу має вигляд:

```
#undef назва_макросу
```

Область дії макросу починається з місця визначення і закінчується його скасуванням директивою #undef чи закінченням файлу. Після заміни макросу передпроцесор знову переглядає програму (можна застосовувати вкладені макроси). Приклади макросів:

```
#define N 50
#define pow2(x) x*x
#define Real lohg double
…
Real x;        // розширення до lohg double x;
double ar[N]; // розширення до double ar[50];
y=pow2(a);     // розширення до y=(a)*(a);
…
```

Як бачимо з цих прикладів, макроси без параметрів вводять у програму символічні константи та скорочення, а макроси з параметрами – еквівалентні функціям. Виклик функції зв'язаний з витратами часу і затягує виконання програми. З іншого боку, макрос розширюється в усіх місцях тексту, де використовують його виклик. Якщо таких місць у програмі багато, то це збільшує розмір тексту і, відповідно, розмір модуля, що виконується. Отож функції скорочують обсяг виконавчого модуля, а макроси – скорочують швидкість виконання програми.

Недоліком макросів є відсутність вбудованого контролю узгодження типів аргументів і формальних параметрів. Однак найбільшим недоліком макросів є поява побічних ефектів, якщо аргументом макросу є деякий вираз. У зв'язку з вищесказаним, макроси у коді програми застосовувати не рекомендують!

Умовна компіляція дає змогу програмістові керувати виконанням директив передпроцесора і компілюванням програмного коду. Залежно від істинності умови, деякий фрагмент коду може компілюватися, а може не компілюватися. Це доволі часто використовують для доручення / вилучення з коду програми операторів налагодження, для попередження повторних долучень у код програми заголовкових файлів та їхніх циклічних посилань один на інший тощо.

Умовна директива передпроцесора зазвичай обчислює значення деякої умови (цілочисельного константного виразу) і, залежно від результату, здійснює або не здійснює компіляцію. В умовах можна використовуватися ідентифікатори та звичайні арифметичні чи логічні оператори (окрім операторів перетворення типів, оператора sizeof і констант переліченого типу). Умовна компіляція здійснюється директивами передпроцесора #if, #ifdef, #ifndef, #elif, #else. Розглянемо кожну директиву:

Директива	Опис
#if умова // Фрагмент_коду #endif	Фрагмент_коду компілюватиметься, якщо значення умови не дорівнюватиме нулю
#ifdef назва // Фрагмент_коду #endif	Фрагмент_коду компілюватиметься, якщо назву раніше визначено директивою #define
#ifndef назва // Фрагмент_коду #endif	Фрагмент_коду компілюватиметься, якщо назву не визначено директивою #define, або її визначення скасовано директивою #undef
#elif і #else	Ці директиви у поєднанні з директивами #if, #ifdef і #ifndef дають змогу будувати умовну компіляцію з множинним вибором. Директива #else передує коду, який повинен компілюватися, якщо попередня умова не виконана, а директива #elif передує новій умові

Приклад умовної компіляції з множинним вибором:

```
#if умова_1
 // Фрагмент_коду_1
#elif умова_2
 // Фрагмент_коду_2
#else
 // Фрагмент_коду_3
#endif
```

`Фрагмент_коду_1` компілюватиметься, якщо виконується `умова_1`; `фрагмент_коду_2` компілюватиметься, якщо виконується `умова_2`; `фрагмент_коду_3` компілюватиметься, якщо не буде виконано жодної з попередніх умов.

Якщо переглянути будь-який стандартний заголовний файл С++ (нехай `access_2k.h`), то можна зауважити, що першими директивами у ньому є дві директиви вигляду:

```
#ifndef __Access_2K_h__
#define __Access_2K_h__
```

А завершується заголовний файл директивою

```
#endif // __Access_2K_h__
```

Така організація заголовного файлу дає змогу унеможливити *зациклювання* і *повторне компілювання* під час виконання директив `#include`, які долучають заголовні файли до різних модулів програми. Коли у застосування *вперше* долучити файл `access_2k.h`, то виконуються перші дві директиви, вказані вище, та ідентифікатор `__Access_2K_h__` стає визначеним, а текст файлу `access_2k.h` компілюється. Якщо ж унаслідок виконання `#include` такий файл буде долучено у застосування вдруге, то виявиться, що ідентифікатор `__Access_2K_h__` уже визначено і повторного компілювання файлу `access_2k.h` не буде. Цей спосіб називають використанням *охоронців долучення* (`include guards`).

Стандарт ISO/ANSI дає змогу також унеможливити *зациклювання* і *повторне компілювання* заголовних файлів директивою

```
#pragma once // Файл має компілюватися один раз
```

Цю директиву розташовують на початку заголовного файлу. У програмі в одних файлах можна простежити `include guards`, а в інших – `#pragma once`. Їх навіть можуть поєднати в одному файлі.

Інший формат директиви

```
#pragma назва_опції
```

задає певний режим роботи передпроцесора. Список можливих опцій, які задають режими роботи передпроцесора, залежить від компілятора і є доволі великим. Знайомство з окремими опціями здійснюватимемо за необхідністю під час викладу матеріалу.

1.4. Введення / виведення даних

Будь-яка програма обмінюється інформацією із зовнішніми пристроями комп'ютера, які використовують для введення даних у програму і виведення результатів. Характер використання засобів введення/виведення мови визначають такі чинники:

- *рівень абстрагування* від деталей взаємодії із пристроями;
- *стиль програмування*, що підтримує систему введення/виведення;
- *принципи керування* пристроями та файловою системою.

Можна виокремити такі *рівні абстрагування*:

- *високий* (рівень потоків і об'єктів типу "*файл*");
- *середній* (рівень консолі: клавіатура та екран дисплею);
- *низький* (рівень операційної системи під час роботи з двійковими файлами та портами введення/виведення).

З позиції *стилю програмування* у мові С++ можна виокремити *об'єктно-орієнтовану* систему введення/виведення і три успадковані *процедурно-орієнтовані* системи мови С: потокові засоби стандарту *ANSI C*, систему типу *UNIX* та засоби введення/виведення низького рівня.

Базові об'єктно-орієнтовані засоби введення / виведення, які забезпечують роботу з потоками (`streams`), оголошені у заголовку `iostream`. Під час підключення заголовку `fstream` стають доступні об'єктно-орієнтовані засоби обміну даними між програмою та файлами через потоки. У заголовку `sstream` оголошено засоби для обміну даними з областю пам'яті, яку розглядають як масив символів або як рядок типу `string`.

Процедурно-орієнтовані засоби високого рівня реалізовані за допомогою бібліотеки стандартного введення/виведення *ANSI C* і стають доступними під час підключення файлу `stdio.h`.

У багатьох реалізаціях С++ процедурно-орієнтовані засоби роботи з консоллю доступні під час підключення файлу `conio.h`. Засоби низького рівня введення/виведення оголошені в файлах `bios.h`, `direct.h`, `dirent.h`, `dos.h`, `io.h` та інших.

У цьому параграфі розглянемо *стандартні засоби високорівневого* введення/виведення, які базуються на двох абстракціях:

файлу і *потоку*. Ці абстрактні поняття базуються на *принципах керування* пристроями і файловою системою, визначених у C++.

Файл (*File*) – поіменована сукупність даних, які містяться у довготривалій пам'яті комп'ютера чи зовнішніх пристроїв і мають визначені атрибути (характеристики). Правила найменування файлів визначають за правилами файлової системи. Розрізняють *текстові* та *бінарні* (двійкові) файли.

Файл, який розглядають як послідовність *рядків символів*, розділених символами пропуску, називають *текстовим*. Окрім власне символу *пропуску* (`' '`), символами пропуску вважають спеціальні символи: нової сторінки (`'\f'`), нового рядка (`'\n'`), повернення каретки (`'\r'`), горизонтальної (`'\t'`) і вертикальної (`'\v'`) табуляції. Послідовність символів `'\r'` і `'\n'` розділяє файл на *лінії* (файлові рядки).

Рядками символів у текстових файлах вважають послідовності символів, які не є символами пропуску. Під час введення ці рядки інтерпретують засоби введення/виведення як дані певного типу, представлені у деякому форматі.

Файл, який розглядають як послідовність байтів (чи символів), називають *бінарним* (або *двійковим*). Цей файл має *початок* (перед першим байтом) і *кінець* (після останнього байта).

Файлами вважають не тільки файли на дисках, а й будь-які пристрої (файлові), з якими можна здійснювати операції введення/виведення. Такими файлами є *клавіатура*, *дисплей*, *модем*, *принтер* тощо.

Потік (*Stream*) – це абстрактний канал зв'язку, який утворюється в програмі для обміну даними з файлів.

Під час роботи з потоками і файлами розрізняють *буферизоване* і *небуферизоване* (без використання буфера) введення/виведення. *Буфер* (*buffer*) – це область оперативної пам'яті, яку використовують засоби введення/виведення для тимчасового збереження даних, якими обмінюється програма та зовнішні пристрої.

Виведення даних у потік з буфера зумовлює до виведення цих даних у відповідний файл тільки після заповнення буфера. Виведення даних у небуферизований потік зумовлює до негайного

виведення даних у файл. Використання буфера дає змогу пришвидшити роботу потоку шляхом *поблочного* обміну даними.

За напрямом передачі даних розрізняють такі потоки:

- *вхідні* (або потоки введення, `input stream`), з яких витягають (читають) дані для змінних програми;
- *вихідні* (або потоки виведення, `output stream`), в які вставляють (записують) значення із програми;
- *двоспрямовані* (або потоки введення/виведення) дають змогу витягати дані з потоку і вставляти їх у потік.

Вхідний потік не може зв'язуватися з файлом, який передбачає лише запис (наприклад, принтер чи монітор). Вихідний потік не може зв'язуватися з файлом, що має атрибут "тільки для читання".

За *способом утворення* розрізняють такі потоки:

- потоки, які утворюються компілятором *автоматично* і зв'язуються зі стандартними системними пристроями введення/виведення (*стандартні потоки* введення/виведення);
- потоки, які утворюють *явно* для обміну даними із *файлом*;
- потоки, які утворюють *явно* для обміну даними із *рядком* в оперативній пам'яті.

Унаслідок підключення заголовка `iostream` під час компілювання програми автоматично утворюються такі стандартні потоки:

- `cin` – об'єкт класу `istream` (стандартного потоку введення), який застосовують для *буферизованого введення* даних зі стандартного пристрою (за домовленістю – з клавіатури);
- `cout` – об'єкт класу `ostream` (стандартного потоку виведення), який застосовують для *буферизованого виведення* даних на стандартний пристрій (за домовленістю – на екран дисплея);
- `cerr` – об'єкт класу `ostream`, який застосовують для стандартного *небуферизованого виведення* повідомлень про *помилки* (за домовленістю – на екран дисплея);
- `clog` – об'єкт класу `ostream`, який застосовують для стандартного *буферизованого виведення* повідомлень про *помилки* (за домовленістю – на екран дисплея).

Обмін даними через стандартні потоки, зазвичай, називають *консольним* обміном. Зв'язки стандартних потоків з пристроями можна змінити на рівні операційної системи або у програмі. Потоки для роботи з іншими файловими пристроями необхідно у програмі створювати явно.

Механізм *перевантаження операторів* у класах забезпечує зручний запис операторів введення / виведення даних для потоків. Оператор зсуву вправо (>>) перевантажений для позначення *введення* даних з потоку, його називають оператором "витягнути (прочитати) з потоку".

Оператор зсуву вліво (<<) перевантажений для позначення *виведення* даних у потік, його називають оператором "вставити (записати) у потік". Про ці оператори ми достатньо сказали у першому параграфі цього розділу. Проте дещо додамо.

Перевантаження операцій читання / запису для потоків реалізовано так, що ці оператори розпізнають тип свого правого операнда і, за встановленими *правилами форматування*, перетворюють значення цього операнда. Розглянемо ці правила форматування:

- рядки символів і символи виводять без лапок / апострофів;
- числа виводять у десятковій системі числення;
- знак у додатних чисел не виводять;
- у цілих числах виводять тільки значущі цифри;
- дійсні числа виводять зі збереженням до 6-ти значущих цифр і вказуванням рухомої крапки.

Можна керувати форматом даних під час введення/виведення за допомогою *прапорців форматування* потоку, які оголошені у класі ios (див. вбудовану довідку). Однак для форматування найчастіше використовують *маніпулятори* (вище розглянуто endl) – спеціальні функції, які дають змогу змінити стан потоку. Відмінність маніпуляторів від звичайних функцій полягає у тому, що їхні назви використовують як правий операнд в операторах читання / запису для потоків.

Розрізняють маніпулятори *без параметрів* (оголошені у заголовку iostream) і маніпулятори з параметрами (оголошені у заголовку iomanip).

Маніпулятори без параметрів:

Маніпулятор	*Дії у потоці*
dec	Представлення числа у 10-ій системі числення
hex	Представлення числа у 16-ій системі числення
oct	Представлення числа у 8-ій системі числення
endl	Вставка символу нового рядка та очистка буфера
ends	Вставка у потік символу закінчення рядка
flush	Очистка буфера вихідного потоку
ws	Отримання та ігнорування символів пропуску
showbase	Вставка ознаки системи числення
noshowbase	Видалення ознаки системи числення
skipws	Пропуск символів пропуску під час введення
noskipws	Відміна пропуску символів пропуску
uppercase	Використання символів верхнього регістра під час виведення чисел
nouppercase	Відміна використання символів верхнього регістра
internal	Вставка заповнювачів між знаком і модулем виведеного числа
left / right	Вставка заповнювачів після значення / перед значенням у полі виведення
fixed	Формат dddd.dd для раціональних чисел
scientific	Формат d.ddddd e dd для раціональних чисел
boolalpha	Виведення даних bool у вигляді true / false
nobooialfa	Виведення даних типу bool, як цілих чисел

Маніпулятори з параметрами:

Маніпулятор	*Дії у потоці*
setiosflags/resetiosflags(long fflags)	Встановлення/відміна прапорців форматування
setbase(int b)	Встановлення системи числення
setfill(int cf)	Встановлення символу-заповнювача
setprecision(int p)	Встановлення точності під час виведення раціональних чисел
setw(int w)	Встановлення ширини поля виведення

1.5. Консольні застосування Microsoft Visual C++

1.5.1. Загальні положення

Як уже відзначено у передмові, інтегроване середовище розробки Microsoft Visual Studio 2010 підтримує дві різні версії мови C++: за стандартом ISO/ANSI і нову версію – C++/CLI, які взаємно доповнюють одна іншу та відіграють різні ролі.

Версія мови ISO/ANSI C++ призначена для розробки високопродуктивних застосувань, які компілюються у "*рідний*" (*native*) код процесора. Версія мови C++/CLI розроблена спеціально для виконання на платформі .NET Framework.

Платформу .NET Framework сформовано зі *спільного середовища виконання коду* (*Common Language Runtime*, *CLR*) і *бібліотеки класів* Framework .NET (*Framework Class Library*, *FCL*).

Спільне середовище виконання (надалі CLR) – програмна система, яка керує виконанням програмного коду, забезпечує переносимість програм між різними платформами, підтримує програмування багатьма мовами і гарантує безпеку даних. Головна роль CLR полягає у тому, щоб виявляти та завантажувати типи .NET і керувати ними відповідно до команд програми. CLR бере на себе усю роботу нижчого рівня: автоматичне керування пам'яттю, міжмовну взаємодію, розгортання двійкових бібліотек тощо.

Для застосувань Windows, орієнтованих на CLR, як базу побудови графічного інтерфейсу користувача (graphical user interface, GUI) використовують каркас Windows Forms з FCL.

Програму C++, яку виконують під керуванням CLR, називають *керованим кодом* C++, оскільки дані і код цієї програми знаходяться під контролем CLR.

Програму C++, що виконують поза CLR, називають *некерованим* або"*рідним*" *кодом* C++ (програма компілюється безпосередньо у рідний код процесора). Програміст у таких програмах має самостійно піклуватися про очищення динамічної пам'яті. Програма може складатися частково з керованого C++ і частково – з "рідного" коду.

Для одержання "рідного" коду можна використовувати як консольні застосування, так Windows-застосування з графічним інтерфейсом користувача на базі бібліотеки класів MFC.

На рис. 2 проілюстровано головні варіанти вибору застосувань C++ у Microsoft Visual Studio 2010.

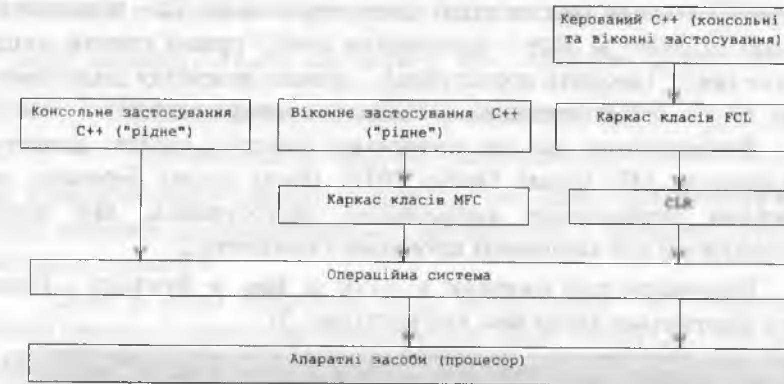


Рис. 2. Вибір головних застосувань C++ у MS Visual Studio 2010

Для упорядкування множини файлів, утворених у процесі підготовки виконавчого модуля, у MS Visual Studio 2010 використовують *проекти* та *рішення*. Множина файлів зберігається у *папці проекту*, а назва цієї папки ототожнюється з *назвою проекту*. Папка проекту містить також інші папки, які використовують для збереження результатів *компілювання* та *компонування* проекту.

Рішення (*solution*) – це папка, в якій зібрано всю інформацію про один чи декілька проектів, причому папки проектів вкладені у папку рішення. Рішення надає механізм об'єднання усіх програм та інших ресурсів, які представляють розв'язання певної проблеми, що стосується обробки даних. Наприклад, система введення замовлень для деякої бізнес-операції може складатися з декількох програм, кожна з яких є проектом усередині єдиного рішення.

Для невеликих програм (зокрема, учбових) кожен проект, зазвичай, стосується окремого рішення. Під час створення проекту нове рішення створюється автоматично, якщо тільки проект не додають до існуючого рішення.

1.5.2. Створення консольних програм ISO/ANSI C++

Наведемо алгоритм створення проекту "рідного" консольного застосування у MS Visual Studio 2010. Для спрощення опису роботи з інтегрованим середовищем використовуватимемо такі позначення: ➢ – вибір команди меню; ⇩ – перехід на список; ⬀ – виокремлення (активізація) елементу списку; ❑ – натиснення кнопки; ❑ `ЛКМ` / ❑ `ПКМ` – натиснення лівої / правої кнопки миші; `поле:= текст` (вводить користувач); … ознака відкриття діалогового вікна; ☑ / ⦿– вибір незалежного / залежного перемикача.

Вважатимемо, що ми починаємо роботу з самого початку, відкриваючи MS Visual Studio 2010. Після цього беремось до створення *порожнього* консольного застосування, яке потім використаємо для виконання програми з прикладу 2.

Виконаємо такі команди: ➢ `File` ➢ `New` ➢ `Project` … Після цього відкриється вікно `New Project` (рис. 3).

Рис. 3. Вигляд вікна `New Project`

Далі виконаємо таке:

⇩ `Installed Template :` ⬀ `Visual C++` ⬀ `Empty project`

```
Name:= Example_2     // Назва / папка проекту
Location:= E:\CPP_2012\Проекти_С++_2012\1_Швидкий_Старт\
// Каталог обираємо через навігацію за допомогою ❑Browse
```

☑ (зняти позначку) `Create directory for solution` ❑ `Ok`.

Увага! Знявши позначку незалежного перемикача

`Create directory for Solution,`

ми утворили папку проекту `Example_2` без папки рішення (файли опису рішення створюватимуться автоматично і зберігатимуться у цій папці проекту).

Щоб у вікні редактора коду переглядати чи створювати файли проекту, необхідно активізувати вікно `Solution Explorer` (рис. 4), виконавши для цього такі дії:

➢ `View` ➢ `Solution Explorer`

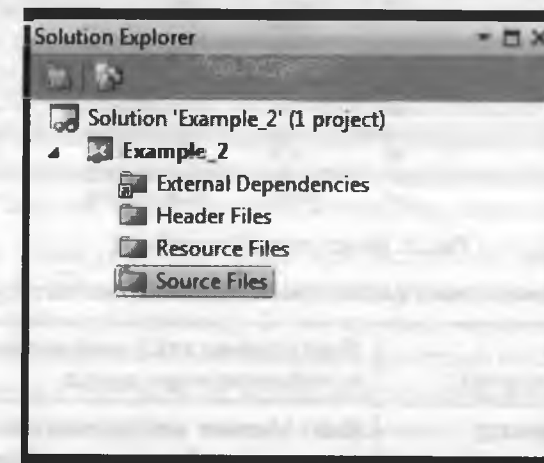


Рис. 4. Вигляд вікна `Solution Explorer`

Тепер необхідно ввести у проект єдиний базовий файл реалізації (назвемо його `Приклад_2`).

У вікні `Solution Explorer` виконаємо такі дії:

`⬀ Source Files ❑ ПКМ ❑ Add ❑ New Item ...`

Після цього відкриється вікно `Add New Item`, в якому виконаємо таке:

`⇩ Installed Template : ⬀ Visual C++  ⬀ C++ File (.cpp)`
`Name:= Приклад_2  /* Назва файлу */  ❑ Add`

Унаслідок цього у текстовому редакторі інтегрованого середовища MS Visual Studio 2010 відкриється порожня область, в якій необхідно буде вручну набрати текст програми прикладу 2. <u>На деякий час</u> залишимо файл `Приклад_2.cpp` *порожнім*.

Збережемо результати нашої роботи:

`➢ File ➢ Save All.`

Якщо заглянути у папку `Example_2` (рис. 5), то побачимо початкові файли нашого проекту / рішення, які створив компілятор.

Рис. 5. Вміст папки `Example_2`

Коротка характеристика файлів проекту консольного застосування:

Файл проекту (`Example_2.vcproj`)	Файл (формат `xml`), який містить детальну інформацію про проект
Файл опцій проекту (`Example_2.vcproj.user`)	Файл (формат `xml`) *користувача міграції* (після міграції проекту з Visual Studio 2008 цей файл містить дані, які були перетворені з деякого VSPROPS-файлу

Файл опцій проекту (`Example_2.vcproj.filters`)	Файл (формат `xml`), який містить значення *фільтрів*, що вказують, де розмістити той чи інший файл проекту
Файл рішення (`Example_2.sln`)	Текстовий файл, який містить інформацію про проекти, що входять у рішення
Файл опцій рішення (`Example_2.suo`)	Текстовий файл, який містить інформацію про опції рішення, вибрані користувачем (за відсутності – опції за домовленістю)
Файл реалізації програми (`Example_2.cpp`)	Текстовий файл, який зберігає код усієї програми чи певної частини програми
Файл бази даних перегляду (`Example_2.sdf`)	Двійковий файл, який підтримує функції перегляду та навігації

Здебільшого ці файли мають службовий характер і надто вникати у них немає потреби. Важливе значення для програміста мають тільки ті файли, які відображаються у вікні `Solution Explorer` (у нашому прикладі це файл `Example_2.cpp`).

Файли, які відображаються у вікні `Solution Explorer`, програміст може редагувати чи доповнювати на власний розсуд. Файли, які не відображаються у цьому вікні, програміст змінити прямо не може. Усі зміни у ці файли вносить компілятор, після того як програміст виконає команди меню, встановить певні опції тощо.

Файл `Example_2.cpp` поки що містить 0 байтів (файл є порожнім). В області набору текстового редактора вручну наберемо текст програми прикладу 2 (початковий варіант програми на с. 19; виправлення початкового варіанта програми на с. 20).

Запустимо програму на виконання за допомогою натиснення кнопки `Start Debugging` (або за допомогою клавіші `F5`). Після цього автоматично запускається процес <u>створення</u> виконавчого модуля (передпроцесорна обробка, компілювання та компонування) та його <u>виконання</u>. Результати роботи цієї програми можна побачити на с. 20.

Після виконання прикладу 2 у папці `Example_2` можна побачити нову вкладену папку `Debug`, яка містить *об'єктний* (`Example_2.obj`) та *виконавчий* (`Example_2.exe`) модулі проекту, а також багато інших допоміжних файлів, які використовують компілятор та компонувальник для оптимізації своєї роботи.

Якщо створюють проект разом з рішенням (встановлена позначка незалежного перемикача

`Create directory for solution`,

то згодом можна *додати* до рішення додаткові проекти.

Під час підготовки виконавчого модуля, можна установити різноманітні опції проекту, виконавши команди

➢ `Project` ➢ `Example_2 Properties …` .

Ці опції визначають, як обробляють програму на стадіях компілювання та компонування. Набір опцій, який породжує конкретну версію виконавчого модуля, називають *конфігурацією*.

Visual C++ 2010 автоматично створює дві конфігурації для побудови версій виконавчого модуля (`Debug` і `Release`). Конфігурація `Debug`, окрім коду, містить додаткову інформацію, яка допомагає у налагодженні програми (з метою пошуку помилки можна виконувати код програми крок за кроком, перевіряючи значення даних, з якими працює програма).

Після налагодження програми і впевненості у тому, що вона працює правильно, програму перебудовують у робочій конфігурації. Для цього виконують команди:

➢ `Build` ➢ `Configuration Manager`

⇩ `Active solution configuration`

⇗ `Release` ❑ `Close`

➢ `Build` ➢ `Rebuild solution`.

Після цього у папці `Example_2` утворюється нова папка `Release`.

Робоча конфігурація (`Release`) містить оптимізований компілятором, машинний код без жодної додаткової інформації. Наприклад, файл `Example_2.exe` у конфігурації `Debug` займає `53760` байтів, а у конфігурації `Release` – `11776` байтів (відчуйте різницю).

1.5.3. Особливості виконання програм у CLR

Середовище CLR є реалізацією *Common Language Infrastructure* (інфраструктура спільної мови) – специфікації, створеної Microsoft і підтриманої ECMA (European Association for Standardizing Information and Computer Systems, Європейська асоціація зі стандартизації інформаційних та обчислювальних систем).

Код, розроблений для виконання у середовищі CLR, називають *керованим* (*managed code*). Програма, написана на .Net-мові, і ресурси, використані у ній, *керуються* винятково середовищем CLR (звідси і назва – керований код).

Керований код оформляється у вигляді *складеного модуля* (*assembly*), який містить *скомпільований код* і *метадані* (детальний опис складеного модуля й *усіх* типів, які використано у ньому). Частину метаданих, які описують складений модуль, називають *маніфестом*. Складений модуль зберігається у спеціальному двійковому PE-файлі (*portable executable*) з розширенням `.exe` чи `.dll`. Складений модуль може міститися і у декількох файлах.

Складені модулі містять метадані та код на *проміжній мові Microsoft* (*Microsoft Intermediate Language*, *MSIL* чи просто *IL*). Програма на мові IL є текстовим зображенням двійкового коду (IL та складений модуль співвідносяться, як асемблер і машинний код). Ця програма є *незалежною* від мови програмування та процесора.

Зазвичай, програмісти використовують мови програмування високого рівня (Visual Basic, Visual C# , Visual C++ тощо). Компілятори цих мов створюють IL-код. Цей IL-код можна написати і на асемблері IL. Для перегляду програми на мові IL Microsoft надає *дизасемблер* IL (файл `ildasm.exe`), який інколи активізують так:

➢`Start` ➢`MS.NET Framework SDK v6.0` ➢`Tools` ➢`IL Disassembler`

Базовими компонентами CLR є *віртуальна система виконання* (*Virtual Execution System*, *VES*) і *система метаданих* (*Metadata System*). VES реалізовано у бібліотеці `mscoree.dll` (міститься у папці `C:\Windows\System32`); вона підтримує завантаження і виконання складених модулів. Система метаданих описує типи .NET. Компілятори використовують метадані для створення типів у їхніх мовах, а VES – для керування типами під час виконання.

Код, який виконує безпосередньо операційна система і не потребує каркасу .NET, називають *некерованим*. Мови Visual Basic і Visual C# дають змогу створювати тільки керований код, на відміну від Visual C++, яка дає змогу створювати керований і некерований коди.

Під час компілювання програми у некерований код цілковито зникає інформація про *внутрішню будову* програми, оскільки це не потрібно для її виконання. Під час компілювання у складений модуль цієї інформації не втрачають, оскільки опис внутрішньої будову програми, починаючи від локальних змінних і закінчуючи класами та просторами назв, зберігається у *метаданих*.

Для виконання складеного модуля VES *транслює* (тобто компілює та компонує) його у "рідний" (native) машинний код *обчислювального вузла* (апаратна платформа + операційна система).

З цією метою до VES під'єднано *компілятор часу виконання* (*Just-In-Time Compiler*), який здійснює трансляцію складеного модуля у машинний код саме під час виконання програми. Для конкретної комбінації апаратної платформи (Intel, Pocket PC тощо) та операційної системи (Windows, Linux тощо) має існувати власна реалізація цього компілятора – це головний недолік технології .NET.

Процес підготовки програми на .Net-мові до виконання виглядає так (компілятори мов містяться у Visual Studio .NET):

1. Створюють вихідний текст програми Net-мовою.
2. Компілятор мови перевіряє синтаксис тексту програми.
3. Компілятор мови компілює програму в складений модуль.
4. За допомогою компілятора Just-In-Time складений модуль *за потреби* транслюється у машинний код процесора (just-in-time трансляція) і виконується.

Під час just-in-time трансляції кожен метод компілюється лише тоді, коли його вперше активізують. Далі цей метод розміщується в оперативній пам'яті для того, щоб його можна було використати повторно без компілювання.

Іноді необхідно звернутися до програмних та апаратних ресурсів, які можуть не відображатися у .NET. З метою розв'язання цієї проблеми розробники .NET створили технологію виконання

спільного коду в рамках компілятора *Manager Extension for C++* (*MC++*). Цей компілятор дає змогу в одному файлі комбінувати як керований, так і некерований код.

Усі мови платформи .NET використовують ідентичну графічну оболонку, однаковий механізм опрацювання виняткових ситуацій, однаковий механізм створення форм і безліч усього іншого. Ще одна перевага CLR полягає в тому, що усі .NET-мови використовують однакові засоби налагодження програм.

Іншою важливою перевагою CLR є виконання багатьох програм в одному процесі, що забезпечується механізмом поділу процесу на віртуальні частини, які називають *доменами програми*.

Операційна система Windows ізолює різні програми одну від іншої, використовуючи різні процеси. Недоліком такої моделі є значне використання пам'яті. Використання пам'яті не є важливим чинником для автономних систем, якими послуговується один користувач, однак є визначальним чинником для серверів, які обслуговують водночас тисячі користувачів.

У деяких випадках (наприклад, під час розробки веб-сайту на основі технології ASP.NET) CLR не створює новий процес для кожного користувача, а створює лише один процес з доменами програм для кожного користувача. Домени програм є безпечними з погляду доступу, як і процеси, оскільки вони створюють обмеження, які керовані програми не можуть порушити.

Зауважимо, що ресурси, які виділяє керований код, вивільняються автоматично за допомогою програми – "збирача сміття", яку налічує VES. Програміст виділяє пам'ять, проте не займається її вивільненням – збирач сміття вивільняє її автоматично.

Точні алгоритми, які використовує збирач сміття, унеможливлюють втрати пам'яті. Недоліком цього механізму вивільнення ресурсів є те, що під час збирання сміття у деякому процесі все інше виконання у цьому процесі моментально припиняється. Однак збирання сміття трапляється порівняно зрідка, отож воно не надто й впливає на швидкодію.

Ще однією перевагою .NET є те, що CLR – незалежна від мови програмування і слугує лише синтаксичним засобом для генерування складеного модуля.

1.5.4. Бібліотека класів платформи **.NET**

Бібліотека класів платформи .NET (FCL) надає об'єктно-орієнтований каркас класів, на базі якого керовані програми розробляють і виконують. Під час створення програми на базі платформи .NET Framework використовують різні технології (API, MFC, ATL, COM та інші) через бібліотеку FCL.

Каркас класів .NET Framework надає також шаблони для розробки різних типів застосувань: консольні програми, віконні програми, сервіси, веб-сайти, веб-сервіси тощо.

Бібліотека класів .NET міститься у багатьох складених модулях (з розширенням `.dll`), проте *головний* складений модуль, який містить *базові класи*, розташовано у файлі `mscorlib.dll`. Список усіх складених модулів бібліотеки класів .NET можна побачити за допомогою *оглядача об'єктів* (*Object Browser*). Цей оглядач можна відобразити на екрані (рис. 6), виконавши такі дії:

➢ `View` ➢ `Object Browser`

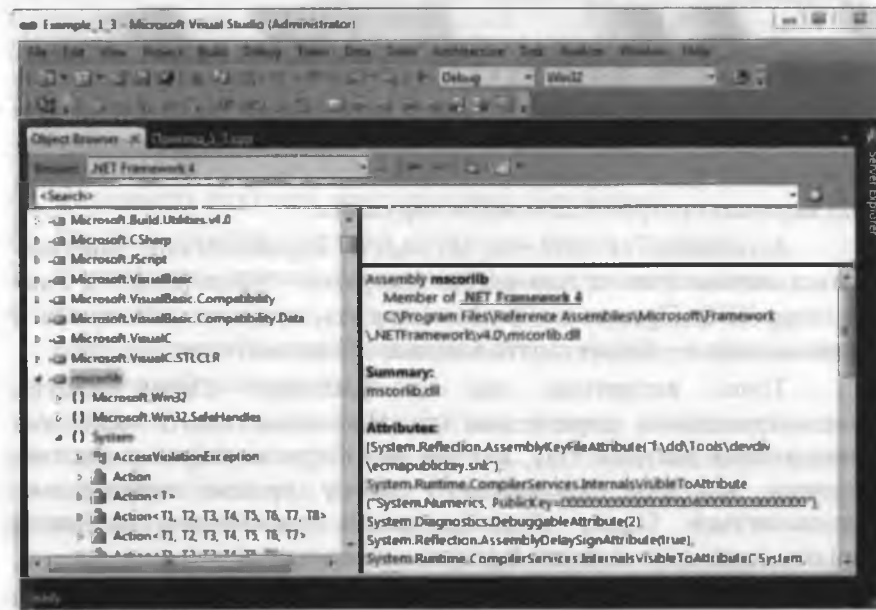


Рис. 6. Вікно `Object Browser`

Бібліотека класів Framework .NET містить понад 7 000 типів (класів, структур, інтерфейсів, перелічених типів і делегатів), які поділено між *просторами назв*, кожен з яких відповідає за служби з певної області.

Найбільше просторів назв можна побачити у *головному* складеному модулі бібліотеки FCL (`mscorlib`), в якому серед інших розміщено найважливіший простір назв `System`. У цьому просторі назв визначено класи, що забезпечують найважливіші функції .NET-мов (не вдасться створити жодної програми без використання цього простору назв).

На рис. 6 у списку лівого підвікна виокремлено назву `mscorlib`, що дає змогу у правому нижньому підвікні одержати докладні відомості про цей елемент списку.

Простори назв забезпечують ієрархію класів, отож допускають функціонування двох різних класів з однаковими назвами, які знаходяться у різних просторах назв. Отже, простір назв – це не що інше, як область дії типів.

У просторі назв `System` бібліотеки `mscorlib` міститься клас `Console`, який призначено для виконання програм у режимі консолі. Він містить методи для організації введення/виведення даних у стандартні потоки введення/виведення.

Назва простору є частиною докладної назви об'єкта, що має загалом синтаксис `namespace::typename` (у цьому випадку `System::Console`). З метою уникнення вживання назви простору у програмі треба її попередньо ввести за допомогою `using`.

Клас `Console`, як і решта класів бібліотеки класів Framework .NET, містить надзвичайно багато методів і властивостей. Довідку про синтаксис та короткий опис застосування цих методів і властивостей також можна одержати у вікні `Object Browser`.

З цієї метою необхідно у списку лівого підвікна виокремити назву `Console`, що даватиме змогу у правому верхньому підвікні одержати список методів і властивостей цього класу. Якщо далі у цьому підвікні виокремити деякий метод чи деяку властивість, то у правому нижньому підвікні одержимо докладні відомості про синтаксис та застосування методу чи властивості.

На рис. 7 виокремлено назву класу `Console` та назву методу цього класу `ReadLine`, який витягає (читає) з консолі *рядок символів* до найближчого символу переходу на новий рядок (натиснення клавіші `Enter`). Метод `Read` – читає з консолі один символ.

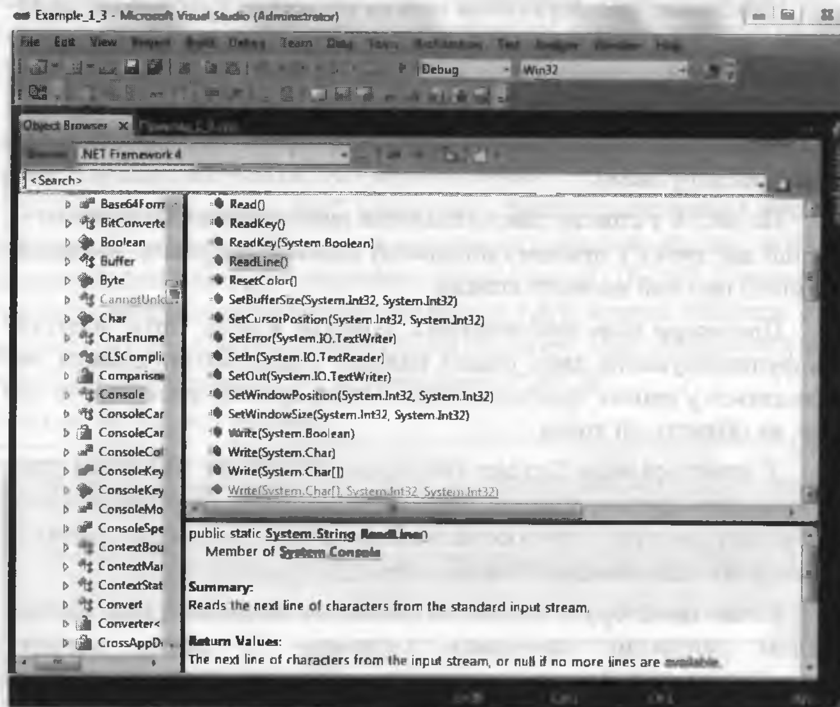


Рис. 7. Відображення методів класу `Console`

Клас `System::Console` надає два методи виведення даних у режимі консолі, перевантажених для різних типів даних:

- `Writeline` – виводить на екран рядок символів, доповнюючи його у кінці символами переходу на новий рядок і переведення каретки; метод коректно виводить кириличні літери (окрім літери "і"), зображені символами з розширеного набору;
- `Write` – робить те саме, що й `Writeline`, однак без доповнення рядка символом переходу на новий рядок.

1.5.5. Створення керованих консольних програм

Наведемо алгоритм створення проекту *керованого* консольного застосування у MS Visual Studio 2010:

Виконаємо такі команди: ➢ `File` ➢ `New` ➢ `Project` … Після цього відкриється вікно `New Project`.

Далі виконаємо таке:

```
⇩ Installed Template : ⬀ CLR  ⬀ CLR Empty project
Name:= Example_3    // Назва / папка проекту
Location:= E:\CPP_2012\Проекти_C++_2012\1_Швидкий_Старт\
// Каталог обираємо через навігацію за допомогою ❑Browse
```

☑ (зняти позначку) `Create directory for solution` ❑ `Ok`.

Тепер необхідно доповнити проект єдиним базовим файлом реалізації (назвемо його `Приклад_3`). У вікні `Solution Explorer` виконаємо такі дії:

```
⬀ Source Files ❑ ПКМ ❑ Add ❑ New Item …
```

Після цього відкриється вікно `Add New Item`, в якому виконаємо таке:

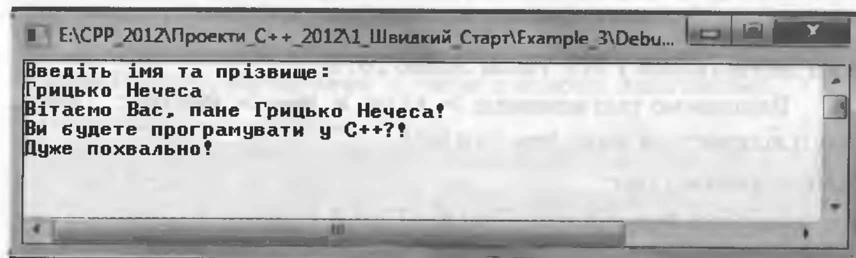
```
⇩ Installed Template : ⬀ Visual C++  ⬀ C++ File (.cpp)
Name:= Приклад_3  /* Назва файлу */   ❑ Add
```

У файлі `Приклад_3.cpp` запишемо діалог з користувачем:

```
using namespace System;
int main()
{
 String ^st;   // Так описують посилання на рядок
 Console::WriteLine (L"Введіть імя та прізвище: ");
 st= Console::ReadLine();   // Введення рядка з консолі
 Console::WriteLine (L"Вітаємо Вас, пане {0}!", st);
 Console::WriteLine (L"Ви будете програмувати у С++?!");
 Console::WriteLine (L"Дуже похвально!");
 st= Console::ReadLine(); // Пауза
 return 0;
}
```

Додатковий коментар. Під час формування рядка `"Вітаємо Вас, пане {0}!"` замість `{0}` буде підставлено значення рядка `st`.

Результати тестування програми:



Якщо зазирнути у папку `Example_3`, то побачимо файли проекту/рішення, які знайомі нам з попереднього прикладу. Нові файли з'явилися у папках `Debug/Release`, оскільки виконавчий модуль, яким керує CLR, складається з одного чи декількох складених модулів, кожен з яких є об'єднанням коду і *метаданих*.

? Запитання для самоперевірки

1. Що таке консоль?
2. Що підключає до програми директива передпроцесора `#include`?
3. Який коментар можна вставити на місці довільного пропуску?
4. Як задати коментар від певної позиції до кінця рядка?
5. Що таке "стандартний потік виведення" і як його позначають?
6. Що таке "стандартний потік введення" і як його позначають?
7. Як позначають оператор вставки даних у потік?
8. Як позначають оператор витягання даних з потоку?
9. Як працює метод `ReadLine` класу `Console`?
10. Як працює метод `Read` класу `Console`?
11. Як працює метод `WriteLine` класу `Console`?
12. Як працює метод `Write` класу `Console`?

Завдання для програмування

Завдання 1. Скласти *некеровану* консольну програму, яка анкетує користувача (3-4 запитання) і робить деякі тривіальні висновки. Вивести дані про автора програми.

Завдання 2. Скласти *керовану* консольну програму, яка анкетує користувача (3-4 запитання) і робить деякі тривіальні висновки. Вивести дані про автора програми.

2. РОБОТА З ДАНИМИ

План викладу матеріалу:

1. Символи та лексеми.
2. Вбудовані типи даних.
3. Визначення змінних.
4. Конструювання виразів.
5. Перетворення типів даних.
6. Приклади розробки програм.
7. Початкове знайомство з функціями.
8. Робота з даними у С++/CLI

Ключові терміни розділу

- *Лексеми, ідентифікатори*
- *Поняття типу даного*
- *Логічний тип і тип* `void`
- *Визначення та оголошення змінних*
- *Види та пріоритет операцій*
- *Перетворення типів даних*
- *Визначення та оголошення функції*
- *Параметри, список параметрів*
- *Передача аргументів за значеннями*
- *Ключові слова, літерали*
- *Цілі та раціональні типи*
- *Символьні константи*
- *Рядки символів, перелік даних*
- *Вирази, обчислення виразів*
- *Функція, головна функція*
- *Активізація функції*
- *Аргументи*
- *Використання посилань*

2.1. Символи та лексеми

Довільний текст будь-якою мовою містить чотири базові елементи: символи, слова, словосполучення та речення. Подібні елементи містить і мова програмування, тільки слова називають *лексемами*, словосполучення – *виразами*, а речення – *інструкціями*.

Набір символів, які використовують для створення тексту, називають *алфавітом*. Алфавіт мови С++ містить:

- символи *літер*: `A`, …, `Z`, `a`, …, `z` і символ підкреслення ("_"), який також вважають літерою;
- арабські *цифри*: `0`, `1`, …, `9`;
- *спеціальні* символи: `+ - * / < > = | & ! \ ~ ' @ # $ % ^ ? _ : ; , . ( ) [ ] { } "`
- невидимі символи *пропусків* (звичайний пропуск, табуляція, перехід на новий рядок).

Із символів алфавіту формують *лексеми* (або елементарні конструкції мови) – мінімальні одиниці мови, які мають самостійний смисл. До лексем у C++ належать:

- *ідентифікатори* (послідовності літер і цифр довільної довжини, які починають літерою);
- *ключові* (чи *зарезервовані*) слова;
- *літерали* (величини, задані власне у коді програми);
- *оператори* (чи знаки операцій);
- *роздільники* (різноманітні дужки, крапка, кома тощо).

Межі лексем визначають за іншими лексемами, такими як роздільники чи оператори.

Використання символів *кирилиці* допускають у коментарях, рядках символів, назвах файлів (якщо це дозволяє операційна система).

Ідентифікатор – це *назва* деякої *сутності* програми (константи, змінної, типу даних, функції тощо). Назви вводять, щоб відрізнити (ідентифікувати) одні *сутності* від інших та оперувати з ними. В ідентифікаторі може бути символ підкреслення, оскільки його вважають літерою. Великі і малі літери вважають *різними* символами. Отож, наприклад, ідентифікатори `Suma` і `suma` є *різними* і співвідносяться з різними сутностями програми.

Деякі слова мови (*ключові*) зарезервовані для службових цілей (не можуть бути ідентифікаторами). Перелік ключових слів:

```
asm         double      new         switch
auto        else        operator    template
break       enum        private     this
case        extern      protected   throw
catch       float       public      try
char        for         register    typedef
class       friend      return      union
const       goto        short       unsigned
continue    if          signed      virtual
default     inline      sizeof      void
delete      int         static      volatile
do          long        struct      while
```


Редактор коду C++ зарезервовані слова виділяє спеціальним шрифтом і/або кольором, щоб можна було одразу ж виявити помилки під час набору та використання цих слів.

Літерал – це константа, тип і значення якої визначають за її зовнішнім виглядом безпосередньо у коді програми. У необхідних випадках пам'ять літералу виділяється автоматично, але може не виділятися і взагалі (наприклад для літералу у константному виразі). Оскільки вигляд літералу залежить від його *типу*, то про них детальніше у наступному параграфі.

Оператор (чи знак операцій) – це один чи декілька символів, які визначають дії над операндом (чи операндами). Всередині оператора недопустимо використовувати пропуски. Оператори, залежно від кількості операндів, ділять на унарні, бінарні та тернарні. Один і той же спеціальний символ, залежно від контексту, може визначати різні оператори. Оскільки оператори використовують під час формування *виразів*, то про оператори детально поговоримо у параграфі 2.4.

2.2. Вбудовані типи даних

Програма, зазвичай, оперує з *сутностями* (змінними, константами, масивами, екземплярами класів чи структур, функціями тощо), які попередньо необхідно *визначати*, щоб повідомити компіляторові:

- назву (ідентифікатор) сутності;
- розмір і спосіб виокремлення осередків пам'яті для зберігання значення (чи сукупності значень) сутності;
- допустимі дії, які можна виконувати із сутністю;
- можливу ініціалізацію сутності (завдання початкового значення чи сукупності початкових значень).

Поруч з терміном "*визначення* сутності" використовують термін "*оголошення* сутності". Це не одне і те саме!

Оголошення та визначення – це частина коду програми, з якої компілятор отримує дані про сутність, представлену своїм ідентифікатором.

Оголошення (*declaration*) "повідомляють" компіляторові про тип і назву сутності, однак деталі в оголошеннях відсутні (тобто

стверджується тільки факт *існування* сутності в програмі). Оголошення має передувати першому використанню сутності, оскільки на базі тієї інформації, яка наявна в оголошенні, компілятор перевіряє правильність використання сутності у відповідному контексті програми.

Визначення (*definition*) забезпечує компілятор деталями. Для *об'єкта* (змінної, константи, масиву, екземпляра класу чи структур тощо) під час визначення компілятор виокремлює осередок пам'яті та, можливо, заповнює його початковими даними. Для *функції* та *шаблону функції* визначення – це заголовок функції чи шаблону разом з тілом функції / шаблону. Оголошення сутності, якщо воно наявне, має *завжди передувати* його визначенню.

Звичайно, уся ця інформація звучить дещо заплутано, сприйняти її одразу важко. Тому до тлумачення термінів "оголошення" та "визначення" ми ще неодноразово повертатимемося під час вивчення конкретних сутностей.

Отже, під *типом даних* (чи просто *типом*) розуміють множину допустимих *значень* даних і множину допустимих *операцій* над ними. Водночас тип даних визначає і розмір пам'яті, необхідної для зберігання цих значень.

У C++ типи даних ділять на *вбудовані* (синоніми: *базові*, *фундаментальні* або *стандартні*) та *похідні*. *Базові типи*: символьний (`char`), символьний двобайтовий (`wchar_t`), цілочисловий (`int`), раціональний (`float`), раціональний подвійної точності (`double`), логічний (`bool`) і тип "без значень" (`void`). Раціональний тип ще називають типом з рухомою крапкою, а логічний тип – булівським.

Цілий тип `char` найзручніший для збереження та обробки числових *кодів* символів на комп'ютері. Зазвичай, значення цього типу займають один байт. Залежно від платформи, `char` інтерпретують як знакове чи беззнакове ціле. Використовуючи `unsigned char` (беззнакове ціле), отримують програми, які без проблем можна використовувати на різних платформах.

Тип даних `wchar_t` (*wide char type*, розширений символьний тип) призначений для роботи з *широкими символами*, під час зберігання яких використовують понад 8 біт. Стандарт C++ залишає семантику широких символів на конкретну реалізацію.

Поширена помилка – вважати `wchar_t` тотожним UNICODE, хоча у багатьох реалізаціях це справді так. У компіляторі Microsoft Visual C++ змінні цього типу зберігають 2-байтові числові коди від 0 до 65 535 (кодування символів UNICODE згідно з UTF-16).

У C++ перед цілим типом даних (`int`) дозволено використовувати *модифікатори типів*: `short` (короткий), `long` (довгий), `signed` (зі знаком) і `unsigned` (без знака).

Перед символьним типом даних (`char`) дозволено використовувати *модифікатори* `signed` і `unsigned`, а перед раціональним типом подвійної точності (`double`) – *модифікатор* `long`.

Замість комбінацій `unsigned int`, `short int` і `long int`, дозволено використовувати слова `unsigned`, `short` і `long`, відповідно. За відсутності у записі цілого типу модифікатора `unsigned/signed` за домовленістю вважають `signed`.

Обсяг осередків пам'яті, який виділяють різним цілим типам стандартом ISO/ANSI C++ не лімітовано. Зазначено тільки, що `1=sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)`.

Можливою є ситуація, коли деякі цілі типи у *конкретній реалізації мови* C++ (платформі C++) вимагають однакового обсягу пам'яті для даних. Окрім того, обсяг пам'яті для типів даних може змінюватися від однієї платформи до іншої, і це треба враховувати під час перенесення програми на іншу платформу.

У стандарті ISO/ANSI C++, на відміну від деяких застарілих версій C++, *логічний* тип `bool` реалізовано як окремий тип, а не як псевдонім цілого. Однак це не заважає (за необхідності) у логічних виразах використовувати цілі значення замість булівських (`true`, `false`). Щодо цього значення `0` (нуль) розцінюють як `false`, а будь-яке ненульове значення – як `true`.

Базовими типами даних для представлення раціональних чисел є типи `float` і `double`. Перший з них вимагає для розміщення даних 32 біти, а другий – 64. До типу `double` можна застосовувати модифікатор `long`, однак для компілятора Microsoft Visual C++ це нічого не значить. Діапазони можливих значень даних і витрати пам'яті для базових типів та їхніх комбінацій з модифікаторами для компілятора Microsoft Visual C++ наведено у табл. 1.

Таблиця 1. Допустимі комбінації типів даних

Тип	К-сть байт	Діапазон значень
`char`	1	Від −128 до 127
`unsigned char`	1	Від 0 до 255
`signed char`	1	Аналогічно `char`
`wchar_t`	2	Від 0 до 65 535
`int`	4	Від – 2 147 483 648 до 2 147 483 647
`unsigned [int]`	4	Від 0 до 4 294 967 295
`signed int`	4	Аналогічно `int`
`short [int]`	2	Від –32 768 до 32 767
`unsigned short [int]`	2	Від 0 до 65 535
`signed short [int]`	2	Аналогічно `short [int]`
`long [int]`	4	Аналогічно `int`
`unsigned long [int]`	4	Аналогічно `unsigned [int]`
`signed long [int]`		Аналогічно `signed int`
`float`	4	Від $3{,}4\ 10^{-38}$ до $3{,}4\ 10^{38}$ (за модулем)
`double`	8	Від $1{,}7\ 10^{-308}$ до $1{,}7\ 10^{308}$ (за модулем)
`long double`	8	Аналогічно `double`
`bool`	1	`true` чи `false`

Увага! Квадратні дужки під час опису синтаксичних конструкцій (за винятком опису *масивів*) означають, що елемент синтаксису в цих дужках у програмах можна або вказувати, або не вказувати (на вибір програміста).

Заголовний файл `cfloat` містить межі діапазонів для раціональних типів, а `climits` – для цілих типів. Також можна використовувати операцію `sizeof` для визначення розміру будь-якого типу (у байтах).

Кожна платформа, зазвичай, визначає деякий розширений набір типів даних. Зокрема, компілятор GNU C++ використовує тип `long long` (дуже довгий цілий), який не є стандартним типом мови C++, отож він може бути відсутнім в інших реалізаціях мови чи називатися по-іншому (у компіляторі Microsoft Visual C++ цей тип, окрім `long long`, ще має псевдонім `_int64`). Діапазон типу:

`signed long long`: від `-9223372036854775807`
до `9223372036854775807`;
`unsigned long long`: від `0` до `18446744073709551615`.

Похідні типи даних містять вказівники або посилання на певні базові типи даних, масиви базових типів даних, типи функцій, класи, структури, об'єднання чи перелік даних. Ці типи вважаються похідними, оскільки, наприклад, класи, структури, об'єднання можуть містити в собі об'єкти різних типів. Визначення і застосування похідних типів даних розглянемо у наступних розділах.

Тип `void` (порожній) синтаксично є базовим типом. Однак використовувати його можна тільки як частину похідного типу (об'єктів типу `void` не існує). Його використовують як вказівник, що функція не повертає значення, чи як базовий тип для вказівників на об'єкти довільного типу.

```
void y()   // функція y не повертає значення
void* ptr; // вказівник на об'єкт довільного типу
```

Після того, як ми ознайомилися з типами даних, можна уточнити поняття *літералу* (див. попередній параграф). Розрізняють *числові* літерали (або просто *числа*), *символьні* літерали і *рядки символів* (або просто *рядки*). Числа ділять на *цілі* та *раціональні*.

У мові C++ цілі числа можна записувати як у *десятковій* системі числення (послідовність десяткових цифр, яка починається не з `0`, якщо це не саме число нуль), так і у *вісімковій* (послідовність вісімкових цифр, яка починається з `0`) чи *шістнадцятковій* (послідовність шістнадцяткових цифр, яка починається з `0x`). Уточнити тип числового літералу можна за допомогою суфіксів.

Суфікс `L` (або `l`) означає, що число інтерпретується як довге (`long`). Суфікс `U` (або `u`) явно вказує на число типу `unsigned`. Можна комбінувати обидва суфікси у довільному порядку, як-от `976Lu`.

Символьні літерали – це набір з одного або двох символів, обмежений апострофами. Односимвольний літерал (наприклад: `'A'`, `'$'`) займає у пам'яті осередок обсягом 1 байт і має тип `char`.

Літерал з префіксом `L` (наприклад: `L'A'`, `L'$'`) займає у пам'яті осередок обсягом 2 байти і має тип `wchar_t`.

Увага! Замість терміна "*односимвольний літерал*" часто вживають термін "*символ*".

Двосимвольний літерал (наприклад: `'bd'`) займає у пам'яті осередок обсягом 2 байти і має тип `short`, причому перший символ розміщується у байті з меншою адресою.

Ознакою раціонального числа (або числа з рухомою крапкою) є наявність у записі числового літералу десяткової крапки або символу `e` (або `E`), наприклад: `2.35e-4` (тобто $2{,}35 \cdot 10^{-4}$); `-2.45`; `-44.` (або `-44.0`); `.03` (або `0.03`); `.09e+18`; `67.E4`.

За домовленістю, якщо раціональне число не має суфіксів уточнення, то воно має тип `double`. За наявності суфікса `f` (або `F`) число має тип `float`, `l` (або `L`) – тип `long double`.

У табл. 2 наведено приклади літералів усіх допустимих типів.

Таблиця 2. Приклади літералів

Тип	Приклади літералів
`char`, `unsigned char` чи `signed char`	`'D'`, `'a'`, `'*'`
`wchar_t`	`L'D'`, `L'a'`, `L'*'`
`int` (десяткові)	`-23`, `8695`
`int` (вісімкові)	`-023`, `08695`
`int` (шістнадцяткові)	`-0x23`, `0x8695`, `0X134`, `-0X3F`
`unsigned int`	`25U`, `0565u`, `0x99A6u`
`long`	`-879L`, `0237563l`, `0x1CAL`
`unsigned long`	`879UL`, `999992345ul`, `0x1Aul`
`float`	`3.14F`, `-3e8F`, `1F`, `.1f`, `1.2e-5f`
`double`	`3.14`, `-3e8`, `1.`, `.1`, `1.2e-5`
`long double`	`3.143E7L`, `1.2e-5l`

Окреме положення займають *символи керування* (або *escape-*символи), які в своєму зображенні містять два чи більше символів, причому першим символом є символ оберненої косої риски. Символи керування використовують для представлення:

1) кодів, які *не мають графічного зображення* (звуковий сигнал (`'\a'`); повернення назад зі знищенням символу зліва (`'\b'`); перехід на нову сторінку (`'\f'`); перехід на новий рядок (`'\n'`); повернення каретки (`'\r'`); табуляція горизонтальна (`'\t'`) і вертикальна (`'\v'`) тощо;
2) символу, що *має графічне зображення* і не підпадає під попередній пункт (наприклад, `'\q'` еквівалентно `'q'`, однак `'\a'` – не еквівалентно `'a'`); передусім це стосується символів, які беруть участь у формуванні символьних літералів і рядків: апострофа (`'\''`), зворотної косої риски (`'\\'`), лапок (`'\"'`);
3) будь-якого символу за допомогою його вісімкового чи шістнадцяткового *числового коду*, наприклад: `'\0007'` або `'\0x07'` еквівалентно `'\a'`, а `'\0x5C'` – `'\\'`.

Рядок символів – послідовність символів, обмежена з обох боків лапками, наприклад `"Це рядок символів"`. У пам'яті рядок символів зберігається як *масив символів*, який завершується *нульовим* символом `'\0'`. Порожній рядок (зображення `""`) містить тільки нульовий символ.

Перед рядком, що містить широкі символи, необхідно розміщувати префікс `L` (наприклад, `L"Це рядок широких символів"`).

Рядок, який містить широкі символи, інколи називають *широким рядком*.

У рядку символів можна використовувати символи керування, наприклад `"\"Ім\'я\" \t\tАдреса \nПийвода\t\tКиїв"` на екрані виглядатиме так

```
"Ім'я"          Адреса
Пийвода         Київ
```

Детальніше про роботу з рядками символів буде описано у розділі 4. До цього часу ми використовуватимемо рядки-літерали тільки для виведення підказок у діалогах з користувачами.

2.3. Визначення змінних

Змінна – це ідентифікатор, що позначає певний осередок пам'яті, в якому зберігається значення деякого типу[1] (значення змінної). Це значення, зазвичай, може змінюватися під час виконання програми. Перед використанням змінну описують за допомогою такої інструкції:

```
[клас_пам'яті] [const] тип список_змінних;
```

де `список_змінних` – послідовність ідентифікаторів, розділених комами. У списку може бути і одна змінна! Необов'язкові елементи опису `[клас_пам'яті]` і `[const]` називають модифікаторами (їх розглядатимемо згодом). Приклад визначення двох змінних раціонального типу:

```
double p3, a;
```

Змінні можна описувати у будь-яких місцях коду. Головне, щоб змінну було описано до моменту її першого використання у програмі. Здебільшого *опис* змінної відповідає її *визначенню* (винятком є тільки *зовнішні змінні*[2]). Під час визначення деякі чи усі змінні можна *ініціалізувати* початковими значеннями. Ініціалізацію можна записувати за допомогою оператора присвоювання (`= літерал`) чи круглих дужок (`(літерал)`), наприклад:

```
int i1 = 10, i2(57);
```

Форма ініціалізації для компілятора не має жодного значення, однак з точки зору зрозумілості коду програми автор схиляється до оператора присвоювання, оскільки круглі дужки воленс-ноленс спрямовують наші міркування до сприйняття `i2(57)` як виклику функції, хоча це не так!

Для ініціалізації можна також використовувати довільні вирази, що містять визначені раніше константи і змінні. Наприклад:

```
int x = 1, y = 4 * x;
```

[1] В осередку пам'яті зберігаються *двійкові* значення; їхню приналежність до певного типу (тобто: як їх інтерпретувати і використовувати) встановлює компілятор на базі відповідного визначення.

[2] Змінні з класом пам'яті `extern`.

Для змінних ініціалізація необов'язкова, проте наполегливо рекомендована (існує дуже мало причин введення змінної без її ініціалізації). Визначення змінних може бути окремою інструкцією програми чи знаходитися усередині інших, складних за структурою інструкцій, як, наприклад, в інструкції циклу:

```
for (int i = 1; i <= 25; i++) x[i]=i;
```

Модифікатор `const` у визначенні змінної вказує на те, що значення цієї змінної у програмі змінювати не можна (за цим строго стежитиме компілятор). Змінну з модифікатором `const` називають *поіменованою константою* (чи просто *константою*).

Приклад визначення констант:

```
const float pi = 3.1415;
const char plus = '+';
```

Значення константи не може змінюватися після ініціалізації, отож вона *обов'язково* має бути ініціалізованою під час визначення. Значенням константи може бути константний вираз, що містить попередньо визначені константи, наприклад:

```
const float pi2=2*pi; // Подвоєне число пі
const float K=pi/180; // Градуси у радіани
```

Спроба будь-де у програмі змінити значення константи зумовить до помилки компіляції з отриманням відповідного повідомлення.

Оскільки рядки символів є масивом, то присвоювати значення-рядок деякій змінній-масиву можна тільки під час визначення цієї змінної (з одночасною ініціалізацією), наприклад:

```
char s[] = "Це рядок, [] - ознака масиву";
```

Кожна змінна має три важливі *характеристики*:

- *область дії* ідентифікатора змінної – частина програми, в якій його використовують для доступу до відповідного осередку пам'яті;
- *час існування* змінної (може бути *постійним* чи *тимчасовим*);
- *область видимості* ідентифікатора змінної – частина програми, в якій можливий *звичайний* доступ (без `::`) до осередку пам'яті.

Ці показники між собою взаємозв'язані і суттєво впливають на використання змінної у програмі.

Якщо змінну визначено у блоці, то її називають *локальною*; *область дії* – від точки визначення до кінця блоку, включаючи усі внутрішні блоки. Нагадаємо, що *блок* – це фрагмент коду, обмежений фігурними дужками (`{` і `}`). *Час існування* локальних змінних є *тимчасовим* (протягом виконання відповідного блоку).

Якщо змінну визначено поза будь-яким блоком і поза будь-якою функцією, то її називають *глобальною*; *область дії* – файл, в якому її визначено, від точки визначення до кінця файлу. *Час існування* глобальних змінних є *постійним* (протягом виконання програми).

Відповідно до змінної область її дії також називають *локальною* чи *глобальною*.

Назви *функцій* мають глобальну область дії (винятки для функцій – членів класу). Як тільки функцію визначено (або оголошено через прототип функції), її можна активізувати інша функція, яка розташована у програмі *після* її визначення чи оголошення. У С++ немає локальних функцій: не можна визначати чи оголошувати функцію всередині іншої функції.

Коли функція визначає локальну змінну з тією ж назвою, що у глобальної змінної, то локальна змінна має пріоритет у тілі функції (глобальна змінна стає тимчасово недоступною). Цей принцип називають *пріоритетом назв* (*name precedence*) або *приховуванням назв* (*name hiding*). Цей же принцип діє і для локальної змінної: вона є недоступною у тих внутрішніх блоках, у яких визначено змінні з аналогічними назвами.

Отже, у ситуаціях, за яких реалізується принцип пріоритету назв, область *дії* ідентифікатора не співпадає з його областю *видимості* (хоча в інших ситуаціях це одне і те ж).

Справді, у цих ситуаціях змінна (локальна чи глобальна), зовнішня до певного вкладеного блоку, є *недоступною* у ньому, хоча блок і входить в область дії цієї змінної. Однак звернутися до такої невидимої глобальної змінної все ж таки можна, використавши оператор доступу (`::`). Принагідно зауважимо, що оператор доступу (`::`) не можна застосовувати до локальних змінних.

У С++ є чотири модифікатори класу пам'яті змінної (`auto`, `register`, `static` і `extern`), які задають спосіб виділення осередків пам'яті для зберігання значень змінної і тим самим впливають на її область дії та час існування.

Осередок пам'яті для *автоматичної* змінної (`auto`) компілятор виділяє перед входом у блок, в якому її визначено. Ця змінна активується тільки під час роботи блоку і знищується після виходу з блоку. *Автоматична* змінна – це *локальна* змінна, яка має локальну область дії і тимчасовий час існування.

До автоматичних змінних належать і *параметри* функції, область дії яких – тіло функції. Якщо модифікатор класу пам'яті локальної змінної не задано, то за домовленістю вважається `auto`. Ключове слово `auto` використовують тільки під час визначення локальних змінних (зазвичай, його опускають). Визначення

```
auto float x, y;
```

є еквівалентним такому

```
float x, y;
```

Регістрова змінна (`register`) відрізняється від автоматичної змінної тільки тим, що вона має зберігатися у регістрі, який забезпечує швидкий доступ до неї. У випадку відсутності вільних регістрів ця змінна стає автоматичною. Ключове слово `register` використовують тільки під час визначення локальних змінних.

Осередок пам'яті для *статичної* змінної (`static`) виділяється тільки один раз, коли програма починає виконуватися (у момент визначення змінної), і знищується тільки під час завершення роботи програми.

Локальні змінні, оголошені з ключовим словом `static`, відомі тільки у тому блоці, в якому їх визначено. Однак, на відміну від автоматичних змінних, локальні змінні `static` зберігають свої значення протягом усього часу виконання програми. При кожному наступному зверненні до цього блока локальні змінні містять ті значення, які вони мали при попередньому звертанні. Наприклад:

```
{ static int i = 1;   ...  i++; }
```

Ініціалізація змінної i відбудеться тільки один раз за час виконання програми (при першому зверненні до блоку змінна i ініціалізується одиницею). На час завершення першого виконання блоку її значенням буде 2. Це значення збережеться до наступного звернення до блоку. Після закінчення повторного виконання блоку значення i дорівнюватиме 3 і т. д. Отже, статична змінна здатна зберігати інформацію між зверненнями до блоку (це дає змогу організувати лічильник кількості виконання блоку в програмі).

Модифікатор класу пам'яті `extern` використовують для оголошення *зовнішніх* змінних, які використовують у всіх файлах програми, де це оголошення наявне. Ключове слово `extern` вказує на те, що змінну тільки *оголошено*, а її визначення знаходиться в іншому місці програми (далі за текстом, або навіть в іншому файлі). Незалежно від місця оголошення, *зовнішня* змінна є глобальною змінною та має постійний час життя.

Наприклад, нехай у файлі `U1.cpp` визначено *глобальну* змінну a (`int a = 5;`). Якщо в іншому файлі `U2.cpp` оголошено змінну a (`extern int a;`), то компілятор зрозуміє, що мова йде про ту ж саму змінну a. І обидва файли можуть з нею працювати (навіть немає необхідності зв'язувати ці файли директивою `#include`).

Узагальнимо матеріал про характеристики змінних (табл. 3).

Таблиця 3. Характеристики змінних

Місце визначення	Клас пам'яті (за домовленістю)	Область дії	Час існування
Поза функцією	Не задано (`static`)	Глобальна	Постійний
У функції	Не задано (`auto`)	Локальна	Тимчасовий
Поза функцією (заборонено)	`auto / register`		
У функції	`auto`	Локальна	Тимчасовий
У функції	`register`	Локальна	Тимчасовий
Поза функцією	`static`	Глобальна	Постійний
У функції	`static`	Локальна	Постійний
Поза функцією	`extern`	Глобальна	Постійний
У функції	`extern`	Глобальна	Постійний

Якщо під час визначення статичної чи глобальної змінної немає явної ініціалізації, то за домовленістю змінна ініціалізується *нульовим* значенням відповідного типу (логічні, цілі, раціональні типи) або *порожнім* значенням (для символів чи рядків). Автоматичні та регістрові змінні за домовленістю не ініціалізується.

Отже, *оголошення* змінних (ознакою слугує слово `extern`) може виникати у *багатьох* місцях програми, однак *визначення* будь-якої змінної можливе тільки в *одному* місці програми.

Увага! Ініціалізація в *оголошенні* змінної неприпустима, оскільки при цьому оголошення анулюється (`extern` ігнорується).

Для кращого розуміння матеріалу розглянемо такий приклад:

```
double a;          // 1 – глобальна змінна a (a=0.0)
void main () {
 double b;         // 2 – локальна змінна b (b=?)
 extern double c;  // 3 – оголошення зовнішньої змінної c
 static int d;     // 4 – локальна статична змінна d (d=0)
 a=2.3;            // 5 – глобальна змінна a (a=2.3)
 int a;            // 6 – локальна змінна a (a=?)
…
 a=5;              // 7 – локальна змінна a (a=5)
 ::a=9.2;          // 8 – глобальна змінна a (a=9.2)
…
{
 double c=6.5;     // 9 – визначення змінної c (c=6.5)
```

Змінна a, визначена поза блоком, є *глобальною* змінною: осередок пам'яті для неї виділяється у *сегменті даних* один раз на початку програми; час життя– постійний; область дії –програма; область видимості –програма, окрім частини блоку, починаючи з 6-го коментаря до кінця блоку, де існує локальна змінна a.

Змінні b і d – *локальні*; області дії та видимості – блок, однак розміщення у пам'яті та час життя у них різний:

- осередок пам'яті для b виділяється у *стекові* при вході у блок і звільняється під час виходу з нього; час життя – тимчасовий;
- осередок пам'яті для d виділяється у *сегменті даних* один раз на початку програми; час життя – постійний.

Користувач може вводити у програму *синоніми* для типів даних. Визначення синонімів типів можна записувати у різних місцях коду. Місце визначення впливає на область дії синонімів типів так само, як і у випадку визначення змінних. Синтаксис визначення синонімів типів:

```
typedef визначення_типу ідентифікатор;
```

де `ідентифікатор` – назва синоніма типу, а `визначення_типу` – опис відповідного типу даних. Наприклад, інструкція

```
typedef unsigned char Symbol;
```

визначає синонім `Symbol` для типу даних `unsigned char`. Надалі на цей синонім можна посилатися під час визначення змінних, наприклад:

```
Symbol Ccyr='p';
```

Метою введення короткого синоніма, зазвичай, є мнемонічне позначення застосування типу даних у програмі.

Інколи виникає потреба у змінних, які приймають обмежений набір цілих значень; посилатися на ці значення зручно за допомогою певних позначок (днів тижня, місяців року тощо). Для цих випадків існує ще один спосіб визначення цілочислових констант – використання *типу переліку даних* (`enum`). Наприклад, інструкція

```
enum color { red, yellow, green } mycolor;
```

визначає тип переліку `color` і змінну цього типу `mycolor`, що може приймати *тільки* константні значення `red`, `yellow` чи `green`. Ці значення надалі використовують як константи для присвоювання значень змінній `mycolor` чи для перевірки її значень, наприклад:

```
mycolor=green; … if(mycolor==green) mycolor=yellow;
```

Цим константам відповідають цілі значення, обумовлені їх місцем у списку оголошення: `red = 0, yellow = 1, green = 2`. Ці значення можна змінити, якщо ініціалізувати константи явно. Наприклад, оголошення

```
enum col { blue, yellow = 3, green = blue+1} mc;
```

зумовить до того, що значення констант будуть такими: `blue = 0`, `yellow = 3`, `green = 1`. Маючи визначення переліку, можна визначити іншу змінну цього ж типу, наприклад:

```
enum col mc2;     // або col mc2;
```


2.4. Конструювання виразів

Оператори та *вирази* задають певну послідовність дій, однак не є закінченими інструкціями мови. *Прості вирази* містять знак оператора та операнди, наприклад `3.14+x`. Оператори можуть мати один, два або три операнди. Відповідно, розрізняють *унарні*, *бінарні* та *тернарні* оператори. Операндами можуть бути *літерали*, *константи*, *змінні*, *виклики функцій* і *вирази*.

Символи `'+'`, `'-'`, `'*'` і `'/'` – знаки *бінарних* арифметичних операторів додавання, віднімання, множення і ділення відповідно. Якщо в оператора ділення обидва операнди є *цілими* числами, то результатом оператора є *ціле* число, а в інших випадках – результатом є раціональне число. Наприклад, `13/2=6`, а `13.0/2=6.5`. Для цілих чисел визначений оператор `%` – знаходження остачі (чи залишку) від ділення). Наприклад, `14%3=2`.

Прикладом *унарного* арифметичного оператора є оператор *зміни знака числа*, який позначають символом "мінус", що стоїть перед одним операндом, як-от `-x+y`. Як бачимо, смисл оператора залежить від контексту. Смисл оператора залежить також від типу його операндів: `'+'` у виразі `a+b` означає додавання раціональних чисел, якщо операнди мають тип `float`, проте це буде додавання цілих чисел, якщо вони мають тип `int`.

Виклик функції – це вказівка назви функції, за якою у круглих дужках записують список аргументів. Наприклад: `F(x+c, y)`.

Вираз – це послідовність операторів, операндів і круглих дужок, яка задає процес отримання результату певного типу. *Найпростішими* виразами є *константи*, *змінні* та *виклики функцій*.

Оператор присвоювання визначають так:

```
lvalue = rvalue
```

де `lvalue` (*left value*, ліве значення) – змінна чи інший елемент мови, що надсилають на адресу пам'яті; `rvalue` (*right value*, праве значення) – деякий вираз. Оператор виконується так:

- отримання результату обчислення виразу `rvalue`;
- за потреби результат перетворюється до типу даних `lvalue`;
- обчислене (і перетворене) значення заноситься у ділянку памяті, на яку посилається `lvalue`.

Зауважимо, що у C++ визначено *оператор присвоювання*, а не *інструкцію* присвоювання, як у деяких мовах. Отож присвоювання можна побачити у несподіваному контексті, наприклад

```
y = sqrt(x=5+b).
```

Порядок обчислення виразу визначають через розміщення круглих дужок, операторів та *пріоритетності* операторів. Оператори з найвищим пріоритетом виконують першими. Якщо у виразі є декілька операторів одного і того ж пріоритету, то діє правило: унарні оператори та оператор присвоювання - *правоасоціативні*, а всі інші – *лівоасоціативні*, тобто: `a = b = c` означає `a = (b = c)`, `a+b-c` означає `(a+b)-c`.

Перелік і пріоритетність операторів наведено у табл. 4 (чим вищий пріоритет, тим менший його номер у першому стовпці таблиці). У сумнівних випадках необхідно застосовувати круглі дужки.

Таблиця 4. Перелік і пріоритетність операцій

Пріоритет	Знаки операторів	Назва оператора
1	*2*	*3*
1	`::`	Розширення області видимості

2	`.`
	`->`
	`[]`
	`()`
	`sizeof`
		Вибір елемента (члена) за назвою
		Вибір елемента (члена) за вказівником
		Індексація
		Виклик функції / конструювання значення
		Розмір типу чи виразу у байтах

3	`++`
	`--`
	`~`
	`!`
	`+`
	`-`
	`&`
	`*`
	`new`
	`delete`
	`(назва_типу)`
		Інкремент (постфіксний або префіксний)
		Декремент (постфіксний або префіксний)
		Інверсія (порозрядне НЕ)
		Заперечення (логічне НЕ)
		Унарний плюс
		Унарний мінус
		Адреса об'єкта
		Рознайменування (непряма адресація)
		Виділення пам'яті
		Звільнення пам'яті
		Перетворення типу

Закінчення табл. 4

1	2	3

4	`*`
	`/`
	`%`
		Множення
		Ділення
		Залишок від цілочислового ділення

5	`+` `-`	Додавання Віднімання
6	`<<` `>>`	Зсування бітів вліво Зсування бітів вправо

7	`<`
	`>`
	`<=`
	`>=`
		Менше
		Більше
		Менше або рівне
		Більше або рівне

8	`==` `!=`	Рівне Не рівне
9	`&`	Порозрядне (побітове) І
10	`^`	Порозрядне додавання за модулем 2
11	`|`	Порозрядне АБО
12	`&&`	Логічне І
13	`||`	Логічне АБО
14	`?:`	Арифметичний `if` (або оператор умови)
15	=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	Присвоювання (просте і складне)
16	`,`	Послідовність виразів

За визначенням `++x` означає `x+=1`, що, своєю чергою означає `x=x+1`. Значення `++x` є нове (тобто збільшене на `1`) значення `x`. Аналогічно, зменшення на `1` виражається оператором `--`. Оператори `++` і `--` можуть бути і як префіксні, і як постфіксні. Наприклад:

```
x=5; y=++x;    // префіксний оператор ++ викликає збільшення x
               //  перед використанням, тобто x=6; y=6
x=5; y=x++;    // постфіксний оператор ++ викликає збільшення x
               // після використання, тобто y=5; x=6
```

Важливою властивістю оператора присвоювання є те, що його можна поєднувати з більшістю бінарних операцій. Наприклад, `x *= 3` означає `x = x*3`.

Порозрядні (побітові) логічні оператори `&`, `|`, `^`, `~`, `>>` і `<<` застосовують до *цілих*, тобто до змінних типу `char`, `short`, `int`, `long unsigned` та їхніх аналогів, результати теж цілі.

Не плутайте побітові логічні оператори з логічними операторами: `&&`, `||`, `!`. Останні повертають `false` чи `true`; їх, зазвичай, використовують для перевірки умови в операторах `if`, `while` чи `for`.

Знаки `?` і `:` визначають *операцію умови* (арифметичний `if`). Наприклад, результатом виразу `(a>=0) ? a : -a` буде *модуль* `a`.

Явне перетворення типу дає значення одного типу для цього значення іншого типу. Наприклад:

```
float x = (float)4;
```

перед присвоюванням перетворить ціле значення `4` до раціонального значення `4.0`. Розрізняють два способи запису явного перетворення типу: *традиційний* `(float)4` та *функціональний* – `float(4)`. Функціональний запис не можна застосовувати для типів, які не мають простої назви. Наприклад, щоб перетворити значення до типу вказівника, треба використати традиційний запис

```
char* p = (char*)0777;
```

Оператор `sizeof(назва_типу)` чи `sizeof вираз` – обчислює розмір типу або результату виразу в байтах. Оператор `,` (кома) задає послідовність виразів. Наприклад, вираз `(a=4, x=a+5, i=7)` складається з трьох простих виразів, а `y=x++` еквівалентне `y=(t=x, x+=1, t)`, де `t` – змінна того ж типу, що й `x`.

Під час обчислення алгебраїчних виразів доволі часто доводиться використовувати математичні функції ($\sqrt{x}$, e^x, $\ln x$ тощо). Стандартна математична бібліотека C++ містить значну кількість математичних функцій. Щоб застосувати ці функції у власній програмі, необхідно підключити заголовний файл `cmath`:

```
#include <cmath>
```

Функцію від одного аргументу `x` викликають, наприклад, так: `sin(x)`. Аргумент `x` може бути довільним числом, змінною чи

виразом. Функція повертає значення, яке можна вивести на екран, присвоїти іншій змінній чи використати у виразі, наприклад:

```
y = sin(x) + 2; cout << sqrt(2) << endl;
```

Перелік найуживаніших базових математичних функцій наведено у табл. 5. Усі перелічені у таблиці функції працюють з аргументами типу `double` і повертають значення функції типу `double`.

Таблиця 5. Базові математичні функції

Функція	Опис
Заокруглення	
`round(x)`	Заокруглює число за правилами арифметики, тобто `round(1.5)==2; round(-1.5)==-2`
`floor(x)`	Заокруглює число вниз (“до підлоги”), тобто `floor(1.5)==1; floor(-1.5)==-2`
`ceil(x)`	Заокруглює число уверх (“до стелі”), тобто `ceil(1.5)==2; ceil(-1.5)==-1`
`trunc(x)`	Заокруглює число в сторону нуля (відкидання дробової частини), тобто `trunc(1.5)==1; trunc(-1.5)==-1`
`fabs(x)`	Модуль (абсолютна величина) `x`
Корені, степені, логарифми	
`sqrt(x)`	Корінь квадратний; повертає $\sqrt{x}$ $(x \geq 0)$
`pow(a, b)`	Піднесення до степеня; повертає a^b $(a > 0)$
`exp(x)`	Експонента; повертає e^x
`log(x)`	Натуральний логарифм; повертає $\ln x$ $(x > 0)$
`log10(x)`	Десятковий логарифм; повертає $\lg x$ $(x > 0)$
Тригонометричні функції	
`acos(x) / asin(x)`	Арккосинус / арксинус (значення у радіанах)
`atan(x)`	Арктангенс (значення у радіанах)
`cos(x) / sin(x)`	Косинус / синус кута `x` (`x` заданий у радіанах)
`tan(x)`	Тангенс кута `x` (`x` заданий у радіанах)

2.5. Перетворення типів даних

У C++ розрізняють явне та неявне перетворення типів даних. Неявне перетворення виконує компілятор C++, а явне – програміст.

В арифметичному виразі, який містить елементи різних типів, компілятор у процесі обчислення автоматично здійснює приховане (*неявне* чи *автоматичне*) *перетворення* типів даних за принципом: якщо оператор має операнди різних типів, то тип операнда "молодшого" типу (менш точнішого) зводиться до типу операнда "старшого" типу (більш точнішого).

Наприклад, якщо в операторі беруть участь коротке ціле і довге ціле, то коротке зводиться до довгого; якщо беруть участь цілий і раціональний операнди, то цілий зводиться до раціонального і т. д. Після подібного зведення типів обидва операнди будуть одного типу. І результат виконання оператора матиме цей же тип.

В оператора присвоювання відбувається зведення типу результату правої частини до типу лівої частини. Якщо тип лівої частини "молодший", ніж тип результату, можливою є втрата точності чи отримання неправильного результату загалом.

Розглянемо приклади неявного перетворення типів. У результаті виконання таких інструкцій:

```
double x = 6.4, y = 4; int a = x * y;
```

змінна `a` отримає значення `25`, хоча справжнє значення дорівнюватиме `25.6`. Значення `25.6` насправді буде отримано під час множення `x*y`, однак згодом дробову частину буде відкинуто, оскільки `a` – ціла змінна.

Результатом виконання інструкцій:

```
int m = 1, n = 2; double x = m / n;
```

буде значення `x=0`. Оскільки `m` і `n` – цілі змінні, то ділення `m/n` дає цілочисловий результат з відкиданням дробової частини. Унаслідок же виконання подібних інструкцій:

```
int m = 1; double n = 2; double x = m / n;
```

отримаємо правильний результат `x = 0.5`, оскільки операнд `n` має тип `double`, то тип іншого, цілого, операнда `m` буде теж зведено до `double` і результат ділення матиме тип `double`.

Ще один приклад, що дає зовсім неочікуваний результат:

```
double a = 300, b = 200; short c = a * b;
```

Якщо реалізувати цей приклад, то побачимо, що змінна `c` отримає значення `-5536` замість очікуваного `60 000`. Справа у тім, що змінна типу `short` може зберігати значення, які не перевищують `32 767`. Оскільки вираз правої частини оператора присвоювання дорівнює `60 000`, то його перетворення до типу `short` дає зовсім неочікуване значення.

Значення логічного типу (`bool`) автоматично перетворюються у цілі числа 0 (`false`) чи 1 (`true`) під час їхнього використання у виразах цілого типу.

Якщо в операторі арифметичної операції хоча б один з операндів є числом, то найпростішим способом явного перетворення цілого типу даних до раціонального є використання *крапки* у зображенні числа. Розглянемо приклад:

```
int m = 9; double x = m / 2;     // x = 4!?
```

Оскільки цей результат нас, очевидно, не задовольнятиме, то після незначної модифікації (зображення `2` у вигляді `2.0`) одержимо бажаний результат:

```
int m = 9; double x = m / 2.0;   // x = 4.5!!
```

Отже, неявне перетворення типів даних не завжди дає очікуваний результат. Це можна виправити, застосувавши операцію *явного перетворення* типів даних. Наприклад, один з попередніх прикладів можна записати у такому вигляді:

```
int m = 1, n = 2; double x = (double) m / n;
```

У цьому випадку змінна `m`, до якої застосовується оператор явного перетворення типів даних, розглядається як величина типу `double`. Тоді і змінна `n` неявно зводиться до типу `double`, отож оператор ділення здійснюється вже не з цілими, а з раціональними числами. Результат виходить правильним (значення `0.5`).

У попередньому прикладі під час застосування оператора явного перетворення типів даних було використано *традиційний* спосіб запису цього оператора, за яким назву типу обмежують круглими дужками.

Цей же приклад можна переписати з використанням *функціонального* способу відображення оператора явного перетворення типів даних, за яким круглими дужками обмежують сам вираз, тобто

```
int m=1, n=2; double x=double(m)/n;
```

Між цими двома способами немає відчутних відмінностей, просто дужки розставляють по-різному. Часто ці способи явного перетворення типів даних називають перетвореннями типів у *старому* стилі (або у стилі мови С).

Стандарт ISO/ANSI C++ задає також і нові оператори явного перетворення типів даних (або перетворення типів у стилі C++): `const_cast`; `dynamic_cast`; `reinterpret_cast`; `static_cast`. У кожного оператора є власна "вузька" спеціалізація:

- `const_cast` застосовують для відкидання константності об'єкта;
- `dynamic_cast` призначений для виконання безпечного понижуючого перетворення типів даних;
- `reinterpret_cast` призначений для низькорівневих перетворень типів даних, залежних від реалізації;
- `static_cast` використовують для явного перетворення типів у "традиційному" розумінні (заміна операторів у стилі мови С).

Синтаксис, правила використання та приклади застосування операторів `const_cast`; `dynamic_cast` та `reinterpret_cast` наводитимемо під час вивчення об'єктно-орієнтованої парадигми програмування у C++. Синтаксис оператора `static_cast`:

```
static_cast<тип_даних>(вираз)
```

Попередній приклад можна переписати так:

```
int m=1, n=2; double x=static_cast<double>(m)/n;
```

Застосування операторів перетворення типів даних у старому стилі залишається цілком законним, однак оператори у новому стилі мають певні переваги: їх легко знайти в коді; вузькоспеціалізоване призначення кожного оператора перетворення типів дає змогу компілятору діагностувати помилки їхнього використання.

У добротній програмі на C++ перетворення типів даних використовують зрідка, однак цілковито відмовлятися від них також не варто.

2.6. Приклади розробки програм

Для того, щоб комп'ютер міг допомогти у розв'язанні певної практичної задачі, необхідно розробити програму, яка базується на *алгоритмові* розв'язку задачі. Власне програма складається з певної послідовності команд (чи інструкцій), записаних на спеціальній мові програмування (у нашому випадку – на мові C++).

Програмування – це процес розробки (створення) програми, який, зазвичай, складається з такої послідовності кроків (чи етапів):

1) специфікація програми;
2) розробка алгоритму розв'язку задачі;
3) кодування (запис алгоритму на мові програмування);
4) налагодження програми;
5) тестування програми.

Специфікація програми – це послідовність таких дій:

- формулювання (постановка) задачі, яку необхідно розв'язати;
- опис вхідних даних (склад, вигляд, обмеження на допустимі значення, способи введення тощо);
- опис результатів розв'язку задачі (склад, вигляд, способи виведення тощо);
- опис поведінки програми в особливих випадках (наприклад, під час введення неправильних даних) тощо.

У цьому посібнику постановкою задачі слугує формулювання *прикладу / завдання*. Під час розв'язування прикладу описові вхідних даних відповідає фрагмент тексту з позначкою *На вході*, а описові результатів розв'язку задачі – з позначкою *На виході*. Опис поведінки програми в особливих випадках, залежно від конкретної ситуації, можна зустріти у будь-якому з фрагментів.

На етапі *розробки алгоритму* необхідно визначити послідовність дій, які необхідно виконати для отримання результату. Якщо задачу можна розв'язати декількома способами, то програміст, опираючись на деякий критерій (наприклад, швидкість роботи програми), обирає найбільш відповідний алгоритм. Результатом етапу розробки алгоритму є його докладний словесний опис, блок-схема тощо.

У цьому посібнику приклади, зазвичай, є відображенням певних алгоритмів чи стандартних прийомів програмування, які описують у конкретному параграфі чи пункті. Це означає, що алгоритм чи стандартний прийом описують поза межами прикладу.

Однак, за потреби уточнення (чи опису в окремих випадках) алгоритму / стандартного прийому, під час розв'язування прикладу формується фрагмент тексту з позначкою *Попередні міркування*. У цьому фрагменті інколи можна відшукати і міркування щодо формулювання обмежень на вхідні дані та результати тощо.

Після того як складено алгоритм розв'язування задачі, його записують обраною мовою програмування (*кодують*). У результаті одержують *вихідну програму*.

Налагодження – це процес пошуку та усунення помилок у вихідній програмі. Помилки в програмі поділяють на дві групи: *синтаксичні* (помилки в тексті) і *алгоритмічні* (чи логічні). Синтаксичні помилки виявляє компілятор; усунути їх досвідчений програміст може доволі легко. Логічні помилки компілятор не виявляє; їх повинен виявити та усунути програміст.

Після налагодження програми здійснюють її запуск за різних *наборів* вхідних даних (тобто виконують етап *тестування* програми). Цей етап є підсумком виконання усіх попередніх етапів і слугує підтвердженням (або запереченням) їхньої правильності.

Після цього етапу іноді доводиться переглядати сам підхід до розв'язування задачі і повертатися до першого етапу з метою повторного виконання всіх етапів з оглядом на набутий досвід.

Упевнившись, що у заданому діапазоні вхідних даних програма функціонує правильно, програміст (чи відповідна фірма) передає розроблену програму для експлуатації (тобто для розв'язування заданого класу задач).

Формування тестів та методика їхнього застосування сьогодні є доволі складною та цікавою областю комп'ютерної науки, яка, на жаль, виходить за межі посібника. Зазначимо тільки, що під час розробки великих програм (програмних комплексів) програмують одні люди (відділи, фірми тощо), а їхні програми тестують зовсім інші (відділи, фірми тощо).

Тестування простих програм у посібнику реалізовуватимемо спочатку шляхом введення користувачем деякого набору даних (тесту) через консоль. Потім використовуватимемо набори даних, сформовані з псевдовипадкових чисел, які розміщуватимемо у масиві чи файлі.

Аналогічні способи тестування повинні використовувати і студенти під час виконання ними індивідуальних завдань. У багатьох прикладах тести мають містити спеціально підібрані *вхідні* дані разом з *результатами*, які отримуватимуть за цими даними.

Наприклад, під час розробки програми розв'язування нелінійних рівнянь необхідно, щоб розв'язки класичних рівнянь, які видає програма, <u>співпадали</u> з розв'язками, які отримано з відповідних аналітичних формул розв'язків цих рівнянь.

Результати можна отримати унаслідок певних спрощень уручну, за допомогою Excel чи будь-якого математичного пакета. У деяких випадках використовують відомі (отримані та перевірені раніше) результати з літератури чи Інтернету тощо.

Незалежно від способів формування тестів, у <u>будь-якій</u> програмі в посібнику для реалізації етапу тестування використовуватимемо *цикл перебирання* вхідних даних (у протилежному випадку програму необхідно було б запускати кожен раз для чергової порції вхідних даних). Розглянемо коротко дві *конструкції циклу*, які при цьому використовуватимемо. Конструкція циклу з передумовою:

```
while (вираз) інструкція_циклу;
```

Замість `інструкції_циклу` може стояти `блок`. Спочатку програма обчислює значення `виразу`, який може бути цілого чи логічного типу. Якщо його значення не дорівнює нулю чи є істинним (`!=0 / true`), то виконується `інструкція_циклу`. Якщо ж значення `виразу` дорівнює нулю чи є хибним (`=0 / false`), то керування передається інструкції, розміщеній одразу ж за циклом.

Конструкція циклу з постумовою:

```
do інструкція_циклу while(вираз);
```

Замість `інструкції_циклу` може стояти блок. Спочатку виконується `інструкція_циклу`, а потім обчислюється значення `виразу`, який може бути цілого чи логічного типу. Якщо його значення

не дорівнює нулю чи є істинним (`!=0` / `true`), то `інструкція_циклу` виконується знову і т. д. Якщо ж значення виразу дорівнює нулю чи є хибним (`=0` / `false`), то керування передається інструкції, розміщеній одразу за конструкцією циклу.

Також під час організації циклів використовуватимемо *оператор послідовного обчислення* (оператор "кома"), який вказує на необхідність обчислення деякої послідовності виразів, розділених комами. Значенням цієї послідовності вважають значення *останнього* виразу цієї послідовності. Наприклад, після виконання фрагмента програми

```
y = 15, x = (y-=5, 50/y);
```

змінна `x` набуде значення 5. Оператор "кома" можна уявити собі як інструкцію "зробити одне, потім друге і т. д.".

Отже, під час тестування необхідно виконати перебирання певної послідовності значень, причому кожне значення у процесі розв'язування задачі використовуватимемо тільки один раз. Одноразове використання конкретного значення на певній ітерації (кроці) циклу вказує на те, що заводити масив немає підстав.

Оскільки конкретні значення задіяні тільки на певному кроці циклу, то їх і вводитимемо з консолі на відповідній ітерації циклу. Зазвичай, кількість кроків циклічного перебирання вхідних даних залежатиме від користувача і не може бути відомою наперед, отож на закінчення тестування указуватиме спеціально обумовлена *ознака закінчення* – певне число (певні числа) чи комбінація чисел.

Для організації тестувань необхідно виконати таке:

1) видати користувачеві на консоль запрошення на введення чергового вхідного значення;
2) отримати від користувача чергове вхідне значення;
3) якщо чергове вхідне значення не співпадає з *ознакою закінчення*, то обробити його; інакше – вийти з циклу перебирання вхідних значень.

Послідовність цих кроків у C++ реалізують надзвичайно просто та елегантно – за допомогою виразу в конструкції `while`.

Пояснимо це на такому прикладі. Нехай деякий алгоритм визначено тільки для *додатних* чисел. У цьому випадку ознакою закінчення тестування може слугувати довільне *недодатне* число:

```
while (cout<<"Enter x (x>0):", cin>>x, x>0)
{інструкції_реалізації_алгоритму};
```

У виразі конструкції `while` розміщена послідовність трьох виразів: `cout` (запрошення на введення чергового додатного значення `x`); `cin` (введення чергового значення `x`); перевірка умови продовження обробки `x` (`x>0`).

Як відомо, значенням послідовності виразів є значення останнього виразу (`x>0`). Якщо умова (`x>0`) справджується, то виконаються `інструкції_реалізації_алгоритму`; у протилежному випадку відбудеться вихід з циклу.

Приклад 1. У режимі консолі скласти програму обчислення функції $y = \ln^2(\sin x + 2) + \sqrt{4 - x^2}$. Значення аргументу `x` отримувати від користувача.

➢ *Попередні міркування.* У програмі необхідно передбачити перевірку виконання умови $4 - x^2 \geq 0$ (або $|x| \leq 2$).

На вході (`Enter x (|x|<=2):`). Цей рядок забезпечує введення аргументу `x` з проміжку $[-2; 2]$. Ознакою завершення тестування слугуватиме введене значення `x` з поза меж проміжку $[-2; 2]$.

На виході. Для заданого `x` одержуємо значення функції (`y=`).

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//---------------------------------------------------
void main() { float x, y;
 while(cout<<"Enter x (|x|<=2): ", cin>>x, 4-x*x>=0)
   y=log(sin(x)+2),
   cout<<"  y="<<y*y+sqrt(4-x*x)<<endl;
  cin>>x; // Пауза
}
```

Додатковий коментар. Зверніть увагу на те, що `інструкцією_циклу` слугує інструкція виразу, яка складається з послідовності двох виразів, розділених комою (виникає тавтологія щодо терміна *вираз*, однак інакше сказати не можна). Це дає змогу відмовитися від фігурних дужок.

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2_1\Debug\Example_2_1.exe
Enter x (|x|<=2): -1.5
  y=1.32288
Enter x (|x|<=2): -.3
  y=2.26174
Enter x (|x|<=2): 0
  y=2.48045
Enter x (|x|<=2): .4
  y=2.71832
Enter x (|x|<=2): 1
  y=2.82266
Enter x (|x|<=2): 1.5
  y=2.52799
Enter x (|x|<=2): 2
  y=1.14044
Enter x (|x|<=2): 5
```

Якщо області допустимих значень вхідних параметрів співпадають з множиною дійсних чисел, то скористатися попереднім прийомом закінчення тестування уже не вдасться, оскільки будь-яке значення окремого параметра чи будь-яка комбінація значень параметрів є допустимою для обробки алгоритмом.

За цих умов діють так: деяке значення окремого параметра чи деяку комбінацію значень параметрів "призначають на роль" *ознаки завершення* тестування, однак і цю ознаку <u>також</u> обробляють алгоритмом. У цьому випадку для організації циклу перебирання значень тесту необхідно використовувати конструкцію do … `while`.

Для обчислення формул, що мають декілька гілок, використовуватимемо оператор умови:

`(умова)? вираз_1 : вираз_2`

Якщо умова справджується, то виконуватиметься `вираз_1`, інакше – `вираз_2`. На місці `виразу_1 / виразу_2` може стояти послідовність виразів чи інший оператор умови (у цих випадках для розуміння логіки програми доцільно ставити круглі дужки).

Приклад 2. Обчислити значення функції $y = \begin{cases} x^2 - 2a \cdot x, & x < a; \\ 2, & x = a; \\ 3x - a, & x > a. \end{cases}$

➢ *Попередні міркування*. Оскільки області допустимих значень вхідних параметрів a та x співпадають з множиною дійсних чисел, то *ознакою завершення* тестування вважатимемо значення `a==-99`. Про цю *ознаку завершення* тестування користувача необхідно попередити перед початком тестування.

На вході (`Enter a, x:`). Цей рядок забезпечує введення параметрів a та x. Ознакою завершення тестування слугуватиме `a==-99`.

На виході. Для заданих a та x одержуємо значення функції (y=).

Програма:

```
#include <iostream>
using namespace std;
//---------------------------------------------------
void  main() {  float a, x, y;
 cout<<"If a==-99, then exit!"<<endl;
 do
 { cout<<"Enter a, x: ";  cin>>a>>x;
   y=(x<a)? x*x-2*a*x : (x==a)? 2 : 3*x-a;
   cout<<"  y="<<y<<endl;
 } while (a!=-99);
 cin>>x; // Пауза
}
```

Додатковий коментар. Якщо у тілі циклу замість послідовності інструкцій виразів використати одну інструкцію виразу, що складатиметься з послідовності виразів, то можна відмовитися від фігурних дужок. В операторі умови для розуміння логіки програми варто також поставити круглі дужки. Одержимо такий фрагмент:

```
do
   cout<<"Enter a, x: ", cin>>a>>x,
   y=(x<a)? x*x-2*a*x : ((x==a)? 2 : 3*x-a),
   cout<<"  y="<<y<<endl;
while (a!=-99);
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2\Debug\Example...
If a==-99, then exit!
Enter a, x: 4 2
  y=-12
Enter a, x: -5 -5
  y=2
Enter a, x: -2 2
  y=8
Enter a, x: -99 3
  y=108
```

Приклад 3. У трикутнику відомі усі сторони. Визначити периметр і площу трикутника, а також радіуси кіл: вписаного у трикутник та описаного навколо нього.

➢ *Попередні міркування*. Нехай значення сторін трикутника відображають параметри a, b та c. На ці параметри накладають такі обмеження:

$$0 < a < b + c;$$

$$0 < b < a + c;$$

$$0 < c < a + b.$$

Умови додатності значень параметрів можна використати для формування умови продовження тестування: `a>0 && b>0 && c>0` (якщо хоча б одне зі значень параметрів a, b чи c буде недодатним, то тестування завершиться). Інші обмеження перевірятимемо під час виконання алгоритму – при порушенні видаватимемо відповідне повідомлення (`These values are incorrect`).

Для обчислення площі трикутника використаємо формулу Герона

$$S = \sqrt{p(p-a)(p-b)(p-c)}\,,$$

де $p = \dfrac{a+b+c}{2}$ – півпериметр трикутника.

Після цього визначимо периметр трикутника (P), а також радіуси кіл, вписаного у трикутник (r) та описаного навколо нього (R):

$$P = 2p;\ r = \frac{S}{p};\ R = \frac{a \cdot b \cdot c}{4S}.$$

На вході (`Enter a,b,c:`). Цей рядок забезпечує введення додатних значень параметрів `a`, `b` та `c` – сторін трикутника. Якщо хоча б одне зі значень параметрів `a`, `b` чи `c` буде недодатним, то це слугуватиме ознакою завершення тестування.

На виході. Для заданих значень параметрів `a`, `b` та `c` одержуємо значення периметра (`P=`) та площі (`S=`) трикутника, а також радіуси кіл: вписаного у трикутник (`r=`) та описаного навколо нього (`R=`).

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//-------------------------------------------------------
void  main() { float a, b, c, p, S;
  cout<<"Enter a,b,c: a<b+c; b<a+c; c<a+b"<<endl;
  while (cout<<"Enter a,b,c:", cin>>a>>b>>c,
                                 a>0 && b>0 && c>0)
    (a<b+c && b<a+c && c<a+b)?
        ( p=(a+b+c)/2,
          S=sqrt(p*(p-a)*(p-b)*(p-c)),
          cout<<"  P="<<p*2<<"  S="<<S<<
          "  r="<<S/p<<"  R="<<.25*a*b*c/S<<endl )
       : cout<<"These values are incorrect"<<endl;
  cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2_3\Debug\Example_2...
Enter a,b,c: a<b+c; b<a+c; c<a+b
Enter a,b,c:3 4 5
  P=12  S=6  r=1  R=2.5
Enter a,b,c:5 8 10
  P=23  S=19.81  r=1.72261  R=5.04795
Enter a,b,c:3 4 8
  These values are incorrect
Enter a,b,c:2 0 5
```

2.7. Початкове знайомство з функціями

Якщо деяку послідовність інструкцій у програмі необхідно виконати декілька разів, то доцільно цю послідовність оформити у вигляді *функції*, а згодом активізувати її для виконання у потрібних місцях.

Функція об'єднує послідовність інструкцій під певною назвою і може повертати *одне значення* певного типу. Функції, зазвичай, передають певні дані (*аргументи*) для обробки.

Програма складається з однієї або декількох функцій, серед яких має бути *головна функція* з назвою `main`. Якщо програма складається з однієї функції, то це має бути головна функція.

З функцією зв'язані три поняття: *визначення* функції, *оголошення* функції (чи *прототип* функції) і *виклик* функції.

Якщо у програмі з'являється виклик *функції*, то керування передається цій функції. Після виконання функції керування програмою передається у *точку повернення*, яка умовно розміщена зразу ж після виклику функції.

Визначення функції містить зі *заголовок* і *тіло* функції та має таку форму запису:

```
// Заголовок функції
[тип_результату] назва_функції([список_параметрів])
// Тіло_функції (блок)
{
  // Визначення та інструкції
}
```

Тут `назва_функції` – ідентифікатор, `тип_результату` – *тип* значення, яке повертає функція. `Тип_результату` може бути довільним, окрім масиву та функції (проте може бути вказівником на масив чи функцію).

Якщо `тип_результату` не задано, то за домовленістю вважають тип `int`. Якщо функція не повертає значення, то необхідно вказати тип `void`.

`Список_параметрів` – послідовність ідентифікаторів, які розділяють комами. Для кожного параметра необхідно вказати його *тип*. `Список_параметрів` може бути порожнім (круглі дужки при цьому зберігаються).

`Тіло_функції` – послідовність визначень та інструкцій у фігурних дужках, які реалізують певний алгоритм. Серед інструкцій тіла функції, зазвичай, є принаймні одна інструкція повернення:

```
return вираз;
```

Ця інструкція забезпечує повернення значення `виразу` в точку виклику і передачу керування у точку повернення. У тілі функції може бути декілька інструкцій `return` або не бути жодної (у цьому випадку передача керування у точку повернення відбувається після виконання останньої інструкції тіла цієї функції). Інструкція

```
return;
```

тільки забезпечує передачу керування у точку повернення без передачі значення.

Прототип функції (або *оголошення* функції) необхідно вказувати до першого виклику функції для того, щоб компілятор зміг виконати перевірку відповідності типів параметрів та аргументів.

Прототип функції ідентичний заголовку функції; тільки наприкінці прототипу записують крапку з комою. Назви параметрів у прототипі компілятору не потрібні (головне для нього – *типи параметрів*), отож, зазвичай, їх опускають.

Якщо визначення функції передує першому викликові цієї функції у програмі, то прототип функції не задають (у ньому немає потреби). Прототип функції використовують у випадках, коли першому викликові функції передує визначення цієї функції, або виклики функції та визначення функції розташовані у різних файлах.

Виклик функції є одним із найпростіших виразів

```
назва_функції([список_аргументів])
```

У `списку_аргументів`, зазвичай, вказують *константи*, *літерали*, *змінні* та *вирази*. Кількість аргументів має *співпадати* з кількістю параметрів, а типи аргументів мають *відповідати* типам параметрів. За потреби тип значення аргументу автоматично перетворюється на тип параметра, якщо це можливо.

Під час виклику функції аргументи *підставляються* замість параметрів (аргументи *передаються* функції). Є декілька *механіз-*

мів (способів) такої передачі. Під час виклику функції ці способи можуть бути різними для різних параметрів.

Найуживанішим є механізм *передачі аргументів* у функцію *за значенням*. Він діє, якщо опис відповідного параметра складається з типу та назви змінної-параметра, без застосування символів "*","[ ]" або "&".

Механізм передачі аргументу в функцію за значенням передбачає послідовне виконання таких кроків:

1. Під час компіляції функції для параметра виділяється ділянка пам'яті у стекові згідно з його типом (цей параметр є внутрішнім об'єктом функції). Інколи параметри, які отримують аргументи *за значенням*, називають *внутрішніми* параметрами.
2. Обчислюється *значення аргументу-виразу* і результат заноситься у відповідну ділянку пам'яті, виокремлену параметрові. Значення змінної або константи/літерала, яка є аргументом, просто копіюється у цю ділянку пам'яті.
3. У тілі функції виконується обчислення (у тому числі з використанням внутрішніх параметрів). Будь-яка зміна значення внутрішнього параметра не впливає на відповідний аргумент.
4. Після завершення роботи функції ділянки пам'яті, виокремлені внутрішнім параметрам, звільняються.

Приклад 4. Обчислити значення величини

$$z = (\mathrm{sign}(a) + \mathrm{sign}(y))\cdot \mathrm{sign}(a + y),$$

де $\mathrm{sign}(x)$ повертає 1, якщо $x > 0$; −1, якщо $x < 0$; 0, якщо $x = 0$.

➢ *Попередні міркування*. Оскільки області допустимих значень вхідних параметрів a та y співпадають з множиною дійсних чисел, то *ознакою завершення* тестування вважатимемо значення `a==-99`. Про цю *ознаку завершення* тестування користувача необхідно попередити перед початком тестування. Очевидно, що обчислення величини $\mathrm{sign}(x)$ необхідно оформити у вигляді функції C++.

На вході (`Enter a, y:`). Цей рядок забезпечує введення параметрів a та y. Ознакою завершення тестування слугуватиме `a==-99`.

На виході. Для заданих a та y одержуємо значення величини *z* (`z=`).

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------------
int sign(float); // Прототип функції sign
//--------------------------------------------------
void main() { float a, y; int z;
 cout<<"If a==-99, then exit!"<<endl;
 do
    cout<<"Enter a, y: ", cin>>a>>y,
    z = (sign(a)+sign(y))*sign(a+y),
    cout<<"  z="<<z<<endl;
 while (a!=-99);
 cin>>a; // Пауза
}
//--------------------------------------------------
// Визначення функції sign
int sign(float x)
{ return x<0? -1 : (x>0? 1 : 0); }
```

Додатковий коментар. У цій програмі прототип функції застосовано тільки з навчально-методичною метою. Якщо визначення функції розмістити перед головною функцією, то потреба у прототипі відпаде, а програма стане компактнішою:

```
#include <iostream>
using namespace std;
//--------------------------------------------------
// Визначення функції sign
int sign(float x) {return x<0? -1 : (x>0? 1 : 0);}
//--------------------------------------------------
void  main() { float a, y; int z;
 cout<<"If a==-99, then exit!"<<endl;
 do
   cout<<"Enter a, y: ", cin>>a>>y,
   z = (sign(a)+sign(y))*sign(a+y),
   cout<<"  z="<<z<<endl;
 while (a!=-99);
cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2-4\Debug\Exa...
If a==-99, then exit!
Enter a, y: 3 0
  z=1
Enter a, y: -2 -4.5
  z=2
Enter a, y: -2 4
  z=0
Enter a, y: 3.4 6.5
  z=2
Enter a, y: -99 3
  z=0
```

Для реалізації деякого алгоритму функція, окрім параметрів, може потребувати допоміжних змінних, які визначають або у тілі функції, або ззовні функції.

Змінні, визначені у тілі функції, називають *локальними* змінними. Їх компілятор створює на час виконання функції та розміщує у стекові разом з адресою повернення функції. Ці змінні є доступними тільки для функції, в якій їх визначено. Різні функції можуть визначати локальні змінні з однаковими назвами.

Змінні, описані поза функціями, називають *глобальними змінними*. Глобальні змінні запам'ятовуються у сегменті даних програми і займають пам'ять незалежно від того, потрібні вони чи ні. Час існування глобальних змінних співпадає з часом існування програми. Глобальні змінні можна оголошувати у будь-якому місці програми, проте поза функціями, у тому числі і `main`. Глобальні змінні є загальнодоступними для усіх функцій.

Глобальні змінні для передачі даних між функціями бажано не використовувати - це *поганий стиль* програмування! Треба прагнути до того, щоб функції були максимально незалежними, а їхній інтерфейс повністю визначався заголовком чи прототипом.

Модифікатор `static`, зазвичай, використовують для визначення локальних *статичних змінних*, які зберігають свої значення між викликами функцій, наприклад:

```
void AnyFun() { static int i=1;  …  }
```


Для статичної змінної `i` виділяється постійна область пам'яті у сегменті даних, яка ініціалізується одиницею тільки під час першого виклику функції. Далі змінну `i` можна використовувати, змінюючи її значення, однак після нового виклику `AnyFun()`, значення `i` не буде повторно ініціалізуватися, а зберігатиме своє останнє значення. Змінну `i`, зокрема, можна використати для підрахунку кількості викликів функції `AnyFun()`.

Приклад 5. Використання різних змінних у функції.

➤ *Попередні міркування*. Програма відображає зміну значень змінних різних типів унаслідок виконання функції `f`. Значення глобальної змінної `j` зменшуються на `1` під час виконання циклу `while` у функції `f`. У цьому ж циклі локальна статична змінна `n`, локальна змінна функції `m` і локальна змінна циклу `p` збільшуються на `1`.

На вході – нічого. У головній функції двічі викликається `f`.

На виході. У таблиці отримуємо значення змінних (`j m n p`).

```
#include <iostream>
using namespace std;
//-----------------------------------------------
int j=7; // глобальна змінна
//-----------------------------------------------
void f(int k)    // k - параметр
{
 int m=0;         // m - локальна змінна
 static int n=0; // n - статична змінна
 cout << "j m n p" << endl <<endl;  // Виведення
заголовку
 while (k--)
 {
   int p=0;       // p - локальна змінна блоку
   cout << j-- << ' ' << m++ << ' ';
   cout << n++ << ' '<< p++ << ' ' <<endl;
 }
 cout<<endl;
}
//-----------------------------------------------
void main( )
{ char a; f(2),f(3); cin>>a; }
```

Результати роботи програми:

```
E:\CPP_2012\Проекти_С++_2012\2_Типи_Даних\Example_2_5\Debug\Exampl...
j m n p
7 0 0 0
6 1 1 0
j m n p
5 0 2 0
4 1 3 0
3 2 4 0
```

◄

Отже, під час передачі аргументу за значенням у стеку створюється його копія, отож функція працює з цією копією. Зміни копії не впливають на значення оригіналу в точці виклику. Недоліком передачі аргументів за значенням є те, що під час передавання великих об'ємів даних створення їхніх копій зумовлює до значних витрат часу та пам'яті комп'ютера.

Інший механізм зв'язку параметрів і аргументів базується на передачі у функцію адреси аргументу (або ж коротше: передачі аргументу *за адресою*).

У випадку передачі функції аргументу *за адресою* у стек заноситься *адреса* аргументу, а функція здійснює прямий доступ до даних в осередку пам'яті за цією адресою і може їх *змінити*.

Передачу аргументу *за адресою* реалізують за допомогою *посилання* (у цьому випадку часто вживають уточнений термін *передача аргументу за посиланням*) або за допомогою *вказівника* (*передача аргументу за вказівником*).

Передача аргументу за адресою виключає витрати на копіювання великих об'ємів даних, однак такий виклик послаблює захищеність даних, оскільки функція може їх змінити.

У цьому параграфі надалі зосередимося на вивченні механізму передачі аргументів *за посиланням.* Параметр, який отримує аргумент за посиланням, – це *псевдонім* відповідного аргументу. Щоб вказати, що параметр функції отримує аргумент за посиланням, після типу параметра у заголовку (і у прототипі) функції ставлять символ амперсанда (&).

Наприклад, опис

```
int& count  // або int &count
```

у заголовку функції можна читати як "count є посиланням на ціле". Під час виклику функції досить для відповідного аргументу вказати *назву* деякої цілої змінної і вона буде передана за посиланням. Можливість зміни об'єкта функцією є найважливішим наслідком передачі аргументу за посиланням. Наприклад:

```
void IncPosition(int& xPos, int& yPos)
{xPos++, yPos++; }
…
int x=20; int y=40;
IncPosition(x, y); // Тепер x = 21, a y = 41
```

Увага! Виклик функції під час передачі аргументів за значенням і за посиланням виглядає однаково.

Стандартно функція через інструкцію `return` може повернути тільки одне значення. Аргументи, які передаються за адресою, можуть забезпечити повернення (зміну) декількох значень.

Приклад 6. Скласти функцію визначення коренів квадратного рівняння $ax^2 + bx + c = 0$ $(a \neq 0)$. Контроль правильності введення коефіцієнта a здійснювати у головній функції.

➢ *Попередні міркування*. Вимогу $a \neq 0$ можна використати для формування умови продовження тестування: `a!=0` (якщо значення параметра a буде нулем, то тестування завершиться). Для визначення коренів квадратного рівняння передусім необхідно обчислити *дискримінант* $d = b^2 - 4a \cdot c$. Якщо $d \geq 0$, то існують два *дійсні* корені, які визначають згідно з формулою $x_{1,2} = \dfrac{-b \pm \sqrt{d}}{2a}$.

Якщо $d < 0$, то існують два *комплексні* корені, які визначають згідно з формулою $x_{1,2} = \dfrac{-b \pm i\sqrt{(-d)}}{2a}$.

На вході (`Enter a,b,c:`). Цей рядок забезпечує введення коефіцієнтів квадратного рівняння. Якщо перше значення з цієї трійки дорівнюватиме нулю, то це означатиме завершення тестування.

На виході. Для заданих значень параметрів a, b та c одержуємо дійсні (Real:) чи комплексні (Complex:) корені квадратного рівняння. У випадку дійсних коренів далі просто отримуємо значення першого (x1=) і другого (x2=) коренів. У випадку комплексних коренів далі отримуємо значення *дійсної* частини цих коренів (Re=) та модуль *уявної* частини цих коренів (Im=).

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//-----------------------------------------------------
bool Sq(double a, double b, double c,
        double &x1, double &x2)
// Визначення коренів квадратного рівняння;
// a, b, c - вхідні параметри (коефіцієнти рівняння).
// Передумова: a<>0 (функція не контролює).
// Постумова. Якщо дискримінант d<0, то функція поверне
// false; x1 міститиме дійсну частину, а x2 - модуль
// уявної частини комплексних коренів.
// Якщо дискримінант d>=0, то функція поверне true;
// x1, x2 міститимуть дійсні корені рівняння.
//-----------------------------------------------------
{ double d=b*b-4*a*c; bool oz=true;
  (d<0)? (oz=false, x1=-b/(2*a), x2=sqrt(-d)/(2*a))
      : (x1=(-b-sqrt(d))/(2*a), x2=(-b+sqrt(d))/(2*a));
  return oz;
}
//-----------------------------------------------------
void  main() { double a, b, c, x1, x2;
 cout<<"If a==0, then exit!"<<endl;
 while (cout<<"Enter a,b,c:  ", cin>>a>>b>>c, a!=0)
  Sq(a, b, c, x1, x2)?
     cout<<"Real: "<<"  x1="<<x1 <<"  x2="<<x2<<endl
   : cout<<"Complex: "<<" Re="<<x1 <<" Im="<<x2<<endl;
 cin>>a; // Пауза
}
```

Додатковий коментар. Функція реалізує певний алгоритм. Відповідно до доброго стилю програмування, у коментарях після заголовка циклу прийнято коротко описувати алгоритм, вхідні та вихідні параметри. Обмеження, які накладають на вхідні параметри та особливості застосування алгоритму коментують у фрагменті тексту з позначкою *Передумова*, а результати роботи – у фрагменті з позначкою *Постумова*.

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2_6\Debug\Examp...
If a==0, then exit!
Enter a,b,c:   2 -3 5
Complex:    Re=0.75  Im=1.39194
Enter a,b,c:   4 7 -3
Real:    x1=-2.10611  x2=0.356107
Enter a,b,c:   1 -10 25
Real:   x1=5  x2=5
Enter a,b,c:   0 4 5
```

Аргумент, який передають у функцію за адресою, ця функція може змінити. Іноді така зміна є *неприпустимою*. То як же діяти, якщо треба передати аргумент за адресою і водночас уберегти його від зміни? Для цього під час визначення відповідного параметра необхідно вказати модифікатор const, наприклад

```
void SomeFunction(const MyStruct& s)
{
   // Тіло функції
}
… MyStruct x; … SomeFunction(x); …
```

Тепер можна спокійно передавати за посиланням об'єкт x, не турбуючись, що його змінить функція. Об'єкт x є константою тільки усередині функції SomeFunction. Його можна змінювати як до, так і після виклику цієї функції.

Рекомендують указувати const перед усіма параметрами, зміну значень яких у функції не передбачають. Це полегшує налагодження великих програм, оскільки за заголовком функції можна визначити, які величини у ній змінюються, а які – ні. Окрім цього, параметрові типу &const можна передавати константу.

Наведемо ще декілька прикладів, в яких необхідно сформувати і протестувати деякі функції мовою C++.

Приклад 7 (*модифікація* прикладу 1). Оформити функцію С++ для обчислення $y = f(x) = \ln^2(\sin x + 2) + \sqrt{4 - x^2}$.

➢ *Попередні міркування*. Область визначення цієї функції: $|x| \le 2$ (або має виконуватися умова $4 - x^2 \ge 0$). Функція С++ матиме тип `bool`. Якщо значення `x` <u>задовольнятиме</u> вимозі $4 - x^2 \ge 0$, то для нього обчислимо відповідне значення `y` (*вихідного* параметру), а функції присвоїмо значення `true`. Якщо ж параметр `x` <u>не задовольнятиме</u> вимозі $4 - x^2 \ge 0$, то жодних обчислень не виконуватимемо, а функції присвоїмо значення `false`. Оскільки функція може обробляти будь-яке значення *вхідного* параметра `x`, то для завершення тестування у головній функції необхідно сформувати спеціальну ознаку, про яку має знати користувач.

На вході (`Enter x:`). Цей рядок забезпечує введення параметра `x` Ознакою завершення тестування слугуватиме `x==-99`.

На виході. Для заданого `x` одержуємо або значення функції (`y=`), або повідомлення про неможливість здійснення обчислень.

Програма:

```
#include <iostream>
#include <cmatb>
using namespace std;
//-------------------------------------------------------
bool f(float x, float& y)
// Обчислення значень функції f(x); x - вхідний параметр.
// Область визначення функції f (ОВФ): 4-x*x>=0.
// Передумова: x набуває довільних значень.
// Постумова. Якщо x  задовольняє ОВФ, то функція f
// поверне true, а вихідний параметр y отримає
// відповідне значення функції f.
// Якщо x  не задовольняє ОВФ, то f поверне false.
//-------------------------------------------------------
{ float a=4-x*x, b; bool oz=true;
  (a<0) ? oz=false : (b=log(sin(x)+2), y=b*b+sqrt(a));
  return oz;
}
```



```
//-------------------------------------------------------
void main()
{
  float x, y;
  cout<<"If x==-99, then exit!"<<endl;
  do
    cout<<"Enter x: ", cin>>x,
    f(x, y)? cout<<"  y="<<y<<endl
         : cout <<"The value x is incorrect!"<<endl;
  while (x!=-99);
  cin>>x; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\2_Типи_Даних\Example_2_7\Debug\Example_2_7.exe
If x==-99, then exit!
Enter x: -1.4
  y=1.42849
Enter x: 0
  y=2.48045
Enter x: 1
  y=2.82266
Enter x: 1.8
  y=2.05957
Enter x: 3
The value x is incorrect!
Enter x: -4
The value x is incorrect!
Enter x: -99
The value x is incorrect!
```

Приклад 8 (*модифікація* прикладу 2). Оформити функцію С++ для обчислення значень $y = f(a,x) = \begin{cases} x^2 - 2a \cdot x, & x < a; \\ 2, & x = a; \\ 3x - a, & x > a. \end{cases}$

➢ *Попередні міркування*. Оскільки області допустимих значень вхідних параметрів `a` та `x` співпадають з множиною дійсних чисел, то *ознакою завершення* тестування вважатимемо значення `a==-99`. Про цю *ознаку завершення* тестування користувача необхідно попередити перед початком тестування.

На вході (`Enter a, x:`). Цей рядок забезпечує введення параметрів а та х. Ознакою завершення тестування слугуватиме `a==-99`.

На виході. Для заданих а та х одержуємо значення функції (y=).

Програма:

```
#include <iostream>
using namespace std;
//-------------------------------------------------------
float f(float a, float x)
// Обчислення значень функції f(a,x).
// Передумова: а та х набувають довільних значень.
// Постумова. Функція повертає f(a,x).
//-------------------------------------------------------
{ return x<a? x*x-2*a*x : (x==a? 2 : 3*x-a); }
//-------------------------------------------------------
void main()
{
  float a, x;
  cout<<"If a==-99, then exit!"<<endl;
  do
     cout<<"Enter a, x: ",  cin>>a>>x,
     cout<<"  y="<<f(a,x)<<endl;
  while (a!=-99);
  cin>>x; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\2_Типи_Даних\Example_2_8\Debug\Ex...
If a==-99, then exit!
Enter a, x: 3 6
  y=15
Enter a, x: 7 -8
  y=176
Enter a, x: 9 9
  y=2
Enter a, x: -99 3
  y=108
```


Приклад 9 (*модифікація* прикладу 3). У трикутнику відомі усі сторони. Оформити функцію С++ для визначення периметра і площі трикутника, а також радіусів кіл: вписаного у трикутник та описаного навколо нього.

➢ *Попередні міркування*. Нехай значення сторін трикутника відображають параметри *a*, *b* та *c*. На ці параметри накладають такі обмеження:

$$0 < a < b + c;$$

$$0 < b < a + c;$$

$$0 < c < a + b.$$

Умови додатності значень параметрів можна використати для формування умови продовження тестування у головній функції: `a>0 && b>0 && c>0` (якщо хоча б одне зі значень параметрів *a*, *b* чи *c* буде недодатним, то тестування завершиться). Інші обмеження перевірятимемо у тілі функції `Tryc` під час виконання алгоритму.

Якщо умови `a<b+c`, `b<a+c`, `c<a+b` виконуватимуться, то функція поверне `true`, обчислить відповідні характеристики трикутника і розмістить їх у вихідних параметрах: `P` – периметр; `S` – площа; `r` – радіус вписаного кола: `R` – радіус описаного кола.

Якщо умови `a<b+c`, `b<a+c`, `c<a+b` не виконуватимуться, то функція поверне `false`.

Формули обчислення, зазначених характеристик трикутника, можна відшукати у прикладі 3.

На вході (`Enter a,b,c:`). Цей рядок забезпечує введення *додатних* значень параметрів `a`, `b` та `c` – сторін трикутника. Якщо хоча б одне зі значень параметрів `a`, `b` чи `c` буде недодатним, то це слугуватиме ознакою завершення тестування.

На виході. Якщо значення функції `Tryc` дорівнюватиме `true`, то для заданих значень параметрів `a`, `b` та `c` одержуємо значення периметра (`P=`) та площі (`S=`) трикутника, а також радіуси кіл: вписаного у трикутник (`r=`) та описаного навколо нього (`R=`). Якщо значення функції `Tryc` дорівнюватиме `false`, то отримаємо повідомлення: `These values are incorrect`.

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//-------------------------------------------------------
bool Tryc(double a, double b, double c,
        double &P, double &S, double &r,double &R)
// Визначення характеристик трикутника;
// a, b, c - вхідні параметри (сторони трикутника).
// Передумова: a<b+c,  b<a+c,  c<a+b (1).
// Постумова. Якщо виконується (1), то функція поверне
// true, обчислить відповідні характеристики трикутника
// і розмістить їх у вихідних параметрах: P - периметр;
// S - площа; r - радіус вписаного кола; R - радіус
// описаного кола.
// Якщо  (1) не виконується, то функція поверне false.
//-------------------------------------------------------
{ bool oz=true; double p;
   (a<b+c && b<a+c && c<a+b)?
      ( p=(a+b+c)/2, S=sqrt(p*(p-a)*(p-b)*(p-c)),
        P=p*2, r=S/p, R=.25*a*b*c/S )
     :   oz=false;
   return oz;
}
//-------------------------------------------------------
void  main()
{
 double a, b, c, P, S, r, R;
 cout<<"Enter a,b,c: a<b+c; b<a+c; c<a+b"<<endl;
 while (cout<<"Enter a,b,c:", cin>>a>>b>>c,
        a>0 && b>0 && c>0)
  Tryc(a,b,c,P,S,r,R)?
      cout<<"  P="<<P<<"  S="<<S
      <<"  r="<<r<<"  R="<<R<<endl
    :  cout<<"These values are incorrect"<<endl;
 cin>>a; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2_9\Debug\Example_2_9.exe
Enter a,b,c: a<b+c; b<a+c; c<a+b
Enter a,b,c:3 4 5
  P=12  S=6  r=1  R=2.5
Enter a,b,c:5 7 9
  P=21  S=17.4123  r=1.65831  R=4.52267
Enter a,b,c:2 3 6
These values are incorrect
Enter a,b,c:3 0 5
```

Приклад 10. Оформити функцію С++ для визначення функції

$$y = f(a,b,c,x) = \begin{cases} c \cdot a^2 + \sqrt{bx - a}, & x < a; \\ \ln(cx - a) + b \cdot \cos(a \cdot x), & a \le x \le b; \\ \cos(a + b \cdot x) + \sqrt{a + bx}, & x > b. \end{cases}$$

де $a \le 0{,}5$; $b \ge 6$; $c \ge -5$.

➢ *Попередні міркування*. Обмеження, які накладають на параметри a, b та c функції $f(a,b,c,x)$ можна використати для формування умови продовження тестування у головній функції: `a<=0.5 && b>=6 && c>=-5` (якщо хоча б одне зі значень параметрів a, b чи c не задовольнятиме початковій умові, то тестування завершиться).
Аргумент x функції $f(a,b,c,x)$ може набувати довільних значень.
Обчислення функції $f(a,b,c,x)$ може здійснюватися на одній із трьох гілок і на кожній гілці необхідно перевіряти відповідну умову, яка забезпечуватиме правильність обчислення функції квадратного кореня чи натурального логарифма. Ці умови перевірятимемо у тілі функції `f` під час виконання алгоритму.
Якщо відповідна умова на гілці (під час виклику функції потраплятимемо на деяку конкретну гілку) <u>виконуватиметься</u>, то функція `f` поверне значення `true`, а вихідний параметр `y` отримає значення функції $f(a,b,c,x)$.
Якщо відповідна умова на гілці не <u>виконуватиметься</u>, то функція `f` поверне значення `false`.

На вході (a,b,c,x:). Цей рядок забезпечує введення значень a, b, c та x – параметрів функції. Якщо хоча б одне зі значень параметрів a, b чи c не відповідатиме початковим умовам (a<=0.5 && b>=6 && c>=-5), то це слугуватиме ознакою завершення тестування.

На виході. Якщо умова, що забезпечує правильність обчислення функції квадратного кореня чи натурального логарифма, на відповідній гілці функції виконується (значення функції f дорівнюватиме true), то на екран виведеться значення функції $f(a,b,c,x)$ (позначка y=). Якщо ця умова не виконується, то на екран виведеться повідомлення: These values are incorrect.

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//--------------------------------------------------
bool f(double a, double b, double c,
        double x, double &y)
// Обчислення значень функції y = f(a,b,c,x);
// a, b, c, x - вхідні параметри.
// Передумова: a, b, c, x - довільні.
// Постумова. Обчислення функції здійснюється на одній
// із трьох гілок і на кожній гілці необхідно переві-
// ряти умову, яка забезпечуватиме обчислення функції
// квадратного кореня чи натурального логарифма.
// Якщо умова виконується, то функція поверне true,
// а вихідний параметр y отримає значення f(a,b,c,x).
// Якщо умова не виконується, то функція поверне false.
//--------------------------------------------------
{ bool oz=true; double t;
  x<a? (t=b*x-a, t>=0? y=c*a*a+sqrt(t) : oz=false)
  : (x<=b?
      (t=c*x-a, t>0? y=b*cos(a*x)+log(t) : oz=false)
    : (t=b*x+a, t>=0? y=cos(a+b*x)+sqrt(t)
                     : oz=false));
  return oz;
}
//--------------------------------------------------
```



```
void main()
{
 double a, b, c, x, y;
 cout<<"Enter a,b,c: a<=0.5 && b>=6 && c>=-5!"<<endl;
 while (cout<<"a,b,c,x:", cin>>a>>b>>c>>x,
                          a<=0.5 && b>=6 && c>=-5)
  f(a,b,c,x,y)?  cout<<"  y="<<y<<endl
              : cout<<"These values are incorrect"<<endl;
 cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2-10\Debug\...
Enter a,b,c: a<=0.5 && b>=6 && c>=-5!
a,b,c,x:0.5 7 1 0.25
  y=1.36803
a,b,c,x:0.5 7 1 2
  y=4.18758
a,b,c,x:0 7 1 4
  y=8.38629
a,b,c,x:0 7 -3 4
These values are incorrect
a,b,c,x:3 2 1 4
```

Під час передачі аргументу за вказівником відповідний параметр очевидно, має бути *вказівником* на тип цього об'єкта. Наведемо приклад функції SwapVal, яка обмінює значення двох змінних:

```
void SwapVal(double *x, double *y)
{ double t;  t = *x; *x = *y;  *y = t; }
… double a=5, b=35; … SwapVal(&a, &b); …
```

Передача аргументу функції за допомогою вказівника принципово не відрізняється від передачі за посиланням, однак з точки зору синтаксису – це зайвий головний біль, адже весь час треба пам'ятати про операцію рознайменування.

За домовленістю аргументи будь-якого типу, за винятком масиву та функції, передаються у функцію за значенням.

2.8. Робота з даними у С++/CLI

2.8.1. Спільна система типів платформи .NET

Спільна система типів (*Common Types System*, *CTS*) платформи .NET – це формальна специфікація, яка визначає, як будь-який тип необхідно описувати для його сприйняття VES.

Спільна система типів визначає синтаксичні конструкції, які може підтримувати чи не підтримувати конкретна мова програмування. Наприклад, перевантаження операцій підтримується у Visual С++ і Visual C# і не підтримується у Visual Basic. Набір типів CTS значно перевищує кількість типів реальної мови.

У CLR забезпечено ефективну взаємодію програмних модулів, створених на різних .NET-мовах. Ця взаємодія забезпечується тим, що всі .NET-мови повинні задовольняти певному *набору правил*, які обмежують типи даних і складові системи компіляції та виконання деякою групою компонентів, які надає CLR.

Цей набір правил визначено у *спільній специфікації алгоритмічних мов* (*Common Language Specification*, *CLS*). CLS є підмножиною CTS і містить *мінімальний* набір стандартів, який мають підтримувати усі компілятори для .NET-мов.

Тип (*type*) – це визначення (формат чи "креслення"), за яким створюється екземпляр значення. FCL містить множину типів, яку розділяють на *типи-значення* (*value types*) і типи-посилання (*reference types*). Кожний тип є *класом*, що володіє методами форматування, перетворення типів тощо. Базовим класом усіх типів є клас `Object`, визначений у просторі назв `System` (або ж простіше, `System::Object`).

Типи-значення (*розмірні* типи) є контейнерами двійкових значень, які інтерпретують згідно з їхнім форматом. Усі типи-значення походять від базового типу `System::ValueType` і діляться на три категорії: *вбудовані* (*примітивні*) типи, тип *перелічення* (`Enum`) і тип *користувача*. Вбудовані типи – це типи, які перевизначать у конкретній мові програмування.

Тип `ValueType` є прямим спадкоємцем `Object`. Будь-який тип, похідний від `ValueType`, є *структурою*, отож типи-значення

ще називають *структурними* типами. Значення структурних типів компілятор розміщує у стекові.

Типи-посилання містять *посилання* (адреси) на дані, розміщені у динамічній пам'яті комп'ютера (*купі*). Їх розділяють на *інтерфейси*, *вказівники* та типи, які самі себе описують (*масиви* та *класи*). Усі типи-посилання є нащадками класу `System::Object`.

Типи можуть налічувати *члени* (*members*), які можуть бути *полями* (*fields*) чи *методами* (*methods*). *Властивості* (*properties*) і *події* (*events*) є спеціальними типами методів. Поля і методи можуть належати усьому типу чи окремому *екземпляру* (*instance*).

Доступ до поля чи виклик методу, які належать усьому типу, можна здійснити навіть у випадку, якщо цей тип не має жодного екземпляра. Такі члени типу (поля/методи) називають *статичними*. Доступ до поля чи методу *екземпляра типу* можна одержати, тільки за вказівки цього екземпляра.

Вкажемо на відповідність типів FCL та типів МС++:

Назва типу в FCL	Назва у МС++
`System::Byte`	`unsigned char`
`System::SByte`	`signed char / char`
`System::Int16`	`short`
`System::Int32`	`int / long`
`System::Int64`	`long long / _int64`
`System::UInt16`	`unsigned short`
`System::UInt32`	`unsigned int / unsigned long`
`System::UInt64`	`unsigned long long / unsigned _int64`
`System::Single`	`float`
`System::Double`	`double / long double`
`System::Char`	`_wchar_t`
`System::Decimal`	`Decimal`
`System::Boolean`	`bool`

Тип `System::Decimal` використовують для відображення точних десяткових значень величин астрономічного масштабу.

Існує відповідність між типами FCL та типами MC++, оскільки власне типи FCL використовують під час перетворення значень одного типу до значень іншого типу під час використання методів класу `System::Convert` (рис. 1).

Рис. 1. Деякі методи класу `System::Convert`

У бібліотеці класів платформи .NET визначено ще декілька типів-значень: `System::DateTime` для представлення часу; `System::TimeSpan` для представлення проміжків часу; `System::Guid` для представлення глобально унікальних ідентифікаторів – Globally Unique Identifier (GUI) тощо. Насправді вони є *визначеними користувачем* типами (тобто розробниками платформи Framework .NET), а не є складовою частиною типів-значень у середовищі CLR.

2.8.2. *Форматування під час виведення результатів*

У CLR передбачено *механізм форматування*, який дає змогу точно керувати відображенням результатів на екрані консолі за допомогою методів `Console::Write()` та `Console::WriteLine()`. Для керування форматуванням числових даних необхідно використовувати таку форму цих методів:

```
Write / WriteLine("Рядок_форматування", arg0, argl, ... , argN);
```

Елемент `рядок_форматування` містить дві частини: *постійну* і *змінну*. Постійна частини – це друковані літери, які відображаються так, як є, а змінна частина складається з *позначок підстановок*, які мають таку загальну форму запису:

```
{номер_аргументу [, ширина: формат]}
```

Тут елемент `номер_аргументу` визначає порядковий номер аргументу (нумерація починається з нуля: `arg0, argl, ... , argN`). За допомогою елемента `ширина` вказують *мінімальну* ширину поля виведення, а `формат` задає тип форматування. За необхідності значення `ширини` може бути перевищене, щоб вивести увесь результат.

Якщо під час виконання методу `WriteLine()` чи `Write()` у `рядку_форматування` з'являється *позначка підстановки*, замість неї компілятор підставляє і відображає аргумент, що відповідає заданому `номеру_аргументу`. Елементи `ширина` і `формат` зазначати не обов'язково; у цьому випадку позначка підстановки `{0}` означатиме `arg0`, `{1}` – `arg1` і т. д. Аргументи, які зв'язані з позначками підстановок, можуть бути виразами (див. прикл. 11).

Приклад 11. Створити керовану консольну програму, яка у табличній формі відображатиме результати обчислення квадратів і кубів натуральних чисел від 1 до 10.

```
using namespace System;
void main()
{
  Console::WriteLine("\tЧисло\tКвадрат\tКуб");
  for(int i=1; i<10; i++)
     Console::WriteLine("\t{0}\t{1}\t{2}",i,i*i,i*i*i);
  Console::ReadLine(); // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2_11\Debug\Ex...
Число    Квадрат  Куб
1        1        1
2        4        8
3        9        27
4        16       64
5        25       125
6        36       216
7        49       343
8        64       512
9        81       729
```

Для зображення формату використовують рядок `Axx`, де `A` – одно символьна специфікація формату значення, а `xx` одно- чи двозначне число, яке задає точність виведення значення.

Специфікацією ширини поля виведення є ціле число; значення буде вирівняно за правою межею поля, якщо ширина додатна або за лівою межею поля, якщо ширина від'ємна. Якщо значення займає менше символів, ніж це вказано у ширині, то компілятор доповнює поле пропусками. Якщо значення займає більше символів, ніж це вказано у ширині, то ширину компілятор до уваги не бере.

Специфікація формату значення:

- `f` або `F` – результат форматується як раціональне число;
- `e` або `E` – результат форматується в експонентному зображенні;
- `p` або `P` – результат форматується як відсоток числа;
- `n` або `N` – результат форматується як раціональне число з роздільниками розрядів;
- `c` або `C` – результат форматується як сума у місцевій валюті;
- `d` або `D` – результат форматується як десяткове число;
- `g` або `G` – результат форматується як раціональне число або в експонентному зображенні;
- `x` або `X` – результат форматується у шістнадцятковому зображенні.

Використання елементів форматування наведено у прикладі 12.

Приклад 12. Створити керовану консольну програму для відображення результатів форматування за допомогою різних елементів форматування.

```
using namespace System;
void main()
{
 // Форматування виведення цілих чисел
 int m1 = 12345, m2 = 67890;
 Console::WriteLine("m1={0, 6}, m2={1, 4}", m1, m2);
 Console::WriteLine("m1 у форматі 10:d={0, 10:d}", m1);
 Console::WriteLine("m1 у форматі 10:x={0,10:x2}",m1);
 // Форматування виведення раціональних чисел
 double D = 1234.56789;
 Console::WriteLine("D у форматі 10:f3={0,10:f3}",D);
 float F = 1234.56789f;
 Console::WriteLine("F у форматі 10:e2={0,10:e2}",F);
 Console::WriteLine("F у форматі 10:e2={0,10:e3}",F);
 Console::WriteLine("F у форматі 10:g2={0,10:g2}",F);
 Console::ReadLine();// Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example_2_12\Debug\Example_2_12.exe
m1 =  12345, m2 = 67890
m1 у форматі 10:d =      12345
m1 у форматі 10:x =       3039
D у форматі 10:f3=  1234,568
F у форматі 10:e2= 1,23e+003
F у форматі 10:e2=1,235e+003
F у форматі 10:g2=   1,2e+03
```

Під час введення даних за допомогою клавіатури консолі необхідно пам'ятати, що внаслідок цього введення отримаємо окремі символи чи рядки символів. Далі їх необхідно перетворити до значень потрібного типу за допомогою методів класу `System::Convert`.

2.8.3. Приклади розробки керованих консольних програм

Приклад 13 (*аналог* прикладу 1). Скласти керовану консольну програму обчислення функції $y = \ln^2(\sin x + 2) + \sqrt{4 - x^2}$. Значення аргументу x отримувати від користувача.

➢ *Попередні міркування.* У програмі необхідно передбачити перевірку виконання умови $4 - x^2 \geq 0$ (або $|x| \leq 2$).

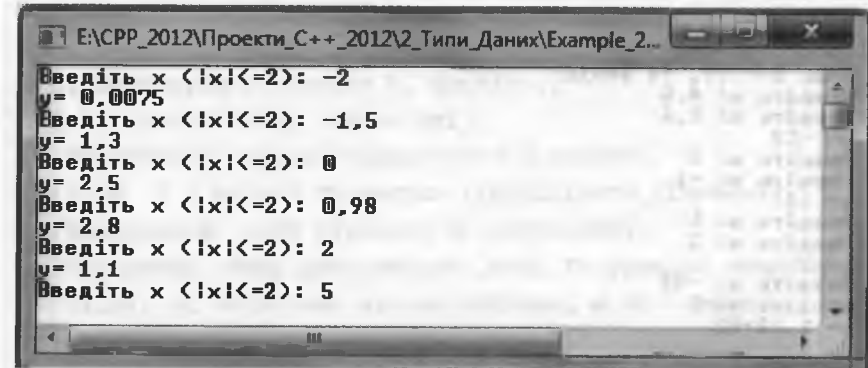
На вході (`Введіть x (|x|<=2):`). Цей рядок забезпечує введення аргументу x з проміжку $[-2; 2]$. Ознакою завершення тестування слугуватиме введене значення x з поза меж проміжку $[-2; 2]$.

На виході. Для заданого x одержуємо значення функції (y=).
Програма:

```
using namespace System;
//-------------------------------------------------
void main()
{
 float x, y;
 while ( Console::Write("Введіть x (|x|<=2): "),
         x=Convert::ToSingle(Console::ReadLine()),
         4-x*x>=0
       )
         y=Math::Log(Math::Sin(x)+2),
         Console::WriteLine("y= {0,-10:G2} ",
                     y*y+Math::Sqrt(4-x*x));
 Console::ReadLine(); // Пауза
}
```

Додатковий коментар № 1. Як бачимо, організація тестування керованої консольної програми цілковито відповідає некерованій консольній програмі прикладу 1. Очевидно, що оператор виведення даних у потік замінив метод `Console::Write`, а оператор витягання даних – `Console::ReadLine`. Окрім цього, на програміста покладено обов'язок керування явним перетворенням типів даних. Зазначимо також, що простір назв `System` містить клас `Math`, статичні методи якого відповідають математичним функціям.

Результати тестування програми:



Додатковий коментар № 2. Для розділення цілої та дробової частин чисел під час обміну даними з консоллю за допомогою методів `Console` використовують *кому* (крапку у потоках `iostream`). ≺

Приклад 14 (*аналог* прикладу 2). Скласти керовану консольну програму обчислення значення функції $y = \begin{cases} x^2 - 2a \cdot x, & x < a; \\ 2, & x = a; \\ 3x - a, & x > a. \end{cases}$

➢ *На вході* (`Введіть a:` і `Введіть x:`). Ці рядки забезпечують уведення параметрів a та x. Ознакою завершення буде `a==-99`.

На виході. Для заданих a та x одержуємо значення функції (y=).
Програма:

```
using namespace System;
void main() { float a, x, y;
 Console::WriteLine("Якщо a==-99, то вихід!");
 do Console::Write("Введіть a: "),
    a=Convert::ToSingle(Console::ReadLine()),
    Console::Write("Введіть x: "),
    x=Convert::ToSingle(Console::ReadLine()),
    y=(x<a)? x*x-2*a*x : (x==a)? 2 : 3*x-a,
    Console::WriteLine("y= {0,-10:G2} ", y);
 while (a!=-99);      Console::ReadLine(); // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\2_Типи_Даних\Example_2_14...
Якщо a==-99, то вихід!
Введіть a: 4,5
Введіть x: 2,6
y= -17
Введіть a: 5
Введіть x: -1
y= 11
Введіть a: 5
Введіть x: 5
y= 2
Введіть a: -99
Введіть x: 5
y= 1,1E+02
```

Приклад 15 (*аналог* прикладу 6). У керованій консольній програмі скласти функцію визначення коренів квадратного рівняння $ax^2 + bx + c = 0 \quad (a \neq 0)$. Контроль правильності введення коефіцієнта a здійснювати у головній функції.

➢ *Попередні міркування.* Вимогу $a \neq 0$ використаємо для умови продовження тестування. Корені залежать від $d = b^2 - 4a \cdot c$. Якщо $d \geq 0$, то існують два *дійсні* корені $x_{1,2} = \frac{-b \pm \sqrt{d}}{2a}$. Якщо $d < 0$, то існують два *комплексні* корені $x_{1,2} = \frac{-b \pm i\sqrt{(-d)}}{2a}$.

На вході (`Введіть a:`, `Введіть b:`, `Введіть c:`). Ці рядки забезпечують уведення коефіцієнтів квадратного рівняння. Якщо перше значення з цієї трійки дорівнюватиме нулю, то це означатиме завершення тестування.

На виході. Для заданих значень параметрів `a`, `b` та `c` одержуємо дійсні чи комплексні корені квадратного рівняння. У випадку дійсних коренів далі просто отримуємо значення першого (`x1=`) і другого (`x2=`) коренів. У випадку комплексних коренів далі отримуємо значення *дійсної* частини цих коренів (`x1=`) та модуль *уявної* частини цих коренів (`x2=`).

Програма:

```
using namespace System;
//--------------------------------------------------------
bool Sq(double a, double b, double c,
         double &x1, double &x2)
// Визначення коренів квадратного рівняння;
// a, b, c - вхідні параметри (коефіцієнти рівняння).
// Передумова: a<>0 (функція не контролює).
// Постумова. Якщо дискримінант d<0, то функція поверне
// false, x1 містититиме дійсну частину, а x2 - модуль
// уявної частини комплексних коренів.
// Якщо дискримінант d>=0, то функція поверне true,
// x1 та x2 міститимуть дійсні корені рівняння.
//--------------------------------------------------------
{
 double d=b*b-4*a*c; bool oz=true;
 (d<0)?
      (oz=false, x1=-b/(2*a), x2=Math::Sqrt(-d)/(2*a))
   : (  x1=(-b-Math::Sqrt(d))/(2*a),
         x2=(-b+Math::Sqrt(d))/(2*a)  );
  return oz;
}
//--------------------------------------------------------
void main()
{
 double a, b, c, x1, x2;
 Console::WriteLine("Якщо a==0, то вихід!");
 while(
        Console::Write("Введіть a: "),
        a=Convert::ToSingle(Console::ReadLine()),
        Console::Write("Введіть b: "),
         b=Convert::ToSingle(Console::ReadLine()),
         Console::Write("Введіть c: "),
         c=Convert::ToSingle(Console::ReadLine()), a!=0 )
```

```
  Sq(a, b, c, x1, x2)?
   Console::WriteLine
      ("Дійсні: x1= {0,-10:G2} x2= {1,-10:G2} ", x1,x2)
     :
   Console::WriteLine
      ("Комплексні: x1= {0,-10:G2} x2= {1,-10:G2} ",
        x1,x2);
 Console::ReadLine(); // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\2_Типи_Даних\Example-2_14\D...
Якщо a==0, то вихід!
Введіть a: 2,5
Введіть b: 1
Введіть c: 4
Комплексні: x1= -0,2      x2= 1,2
Введіть a: 1
Введіть b: -10
Введіть c: 25
Дійсні: x1= 5       x2= 5
Введіть a: 3,4
Введіть b: 7,2
Введіть c: 2,7
Дійсні: x1= -1,6      x2= -0,49
Введіть a: 0
Введіть b: 1
Введіть c: 2
```

? Запитання для самоперевірки

1. Що таке ідентифікатор?
2. Які символи входять в алфавіт мови C++?
3. З якою метою використовують ключові слова?
4. Що розуміють під типом даних?
5. Що повідомляється компіляторові при визначенні змінних/констант?
6. Зазначте та охарактеризуйте цілі типи даних.
7. Зазначте та охарактеризуйте раціональні типи даних.
8. Що таке логічний тип і тип `void`?

9. Як представляються числа у мові C++?
10. Як представляються окремі символи та рядки символів у мові C++?
11. Що таке вираз? Вкажіть правила обчислення виразу.
12. Наведіть приклади операцій з однаковим пріоритетом.
13. Яка різниця між префіксним і постфіксним інкрементом?
14. Яка різниця між префіксним і постфіксним декрементом?
15. Наведіть приклади складного присвоювання.
16. Наведіть приклади порозрядних логічних операцій.
17. Що таке логічне І/АБО?
18. Наведіть приклади унарних операцій.
19. Що таке автоматичне перетворення типів даних?
20. Що таке явне перетворення типів даних?
21. Що таке заголовок функції та прототип функції?
22. Як визначити функцію?
23. Де функція може отримати значення?
24. Коротко охарактеризуйте передачу аргументів за значеннями.
25. Коротко охарактеризуйте передачу аргументів за посиланнями.
26. Що таке Common Language Runtime?
27. Що таке Framework Class Library?
28. Що таке Common Language Specification?
29. Що таке Intermediate Language?
30. З якою метою використовують компілятор часу виконання?
31. Що таке керований код?
32. Що таке загальна система типів?
33. Що таке система метаданих?

Завдання для програмування

Увага! Під час виконання завдань 1 – 8 створити *некеровану* (незалежну від CLR) консольну програму.

Завдання 1. Скласти програму обчислення функції (функцію обрати згідно з номером студента у списку студентів підгрупи). Визначити область допустимих значень функції та на її базі організувати цикл тестування програми:

1. $y = \cos^3(2x) - \log_2(x^2 - 2)$.
2. $y = \sin^4(3x) - 2e^{x-2} + \sqrt[4]{x-3}$.

3. $y=\cos^2(2x)-\sin x+\ln(4-x^2)$.

4. $y=\sin^3(3x)-\cos x+\sqrt{9-x^2}$.

5. $y=\sqrt{x^2-4}-2\log_5(x^2+1)$.

6. $y=\sqrt{x^3-3x}+2\sin x$.

7. $y=(x^2-2x)^3-4\log_3(x^3-4)$.

8. $y=\sqrt[3]{x^4-3x^3}-2\sin x$.

9. $y=2^{\ln(x^3+2x)}-4\cos x$.

10. $y=3^{x^3-2x}+4\log_4(x^2-9)$.

11. $y=\sin^{\frac{2}{5}}(3x)+\cos x+\sqrt{9-x^2}$.

12. $y=\sin^{\frac{4}{7}}(3x)+2\cos x+\ln(4-x^2)$.

13. $y=\sqrt[4]{x^5-3x^3}+2\sin x$.

14. $y=\sqrt[4]{x^5-5x^3}-4x^{\frac{2}{3}}$.

15. $y=\sqrt{x^3-5x}+10\sin x$.

Завдання 2. Скласти програму обчислення функції, яка містить декілька гілок (функцію обрати згідно з номером студента у списку студентів підгрупи). Сформувати умову продовження тестування. Під час виконання алгоритму перевірте умови, які забезпечуватимуть можливість застосування математичних функцій.

№	Формула для y	Параметри
1.	$y=\begin{cases} abx\cos^2 zx, & x<3{,}5a; \\ (a+bx)^2-\ln(zx); & 3{,}5a\le x\le b; \\ \sqrt{a+bx-zx^2}, & x>b. \end{cases}$	$\lvert a\rvert \le 0{,}4$; $b>2{,}5$; $z=e^{2x}$.

№	Формула для y	Параметри
2.	$y=\begin{cases} \sin(bm+\cos nx), & \lvert bm\rvert>n^2; \\ \cos(bm-\sin x), & \lvert bm\rvert<n^2; \\ \sqrt{e^{\lvert\ln x\rvert}+\sqrt{\lvert bmx\rvert}}, & \lvert bm\rvert=n^2. \end{cases}$	$b<-1{,}6$; $m>0{,}9$; $\lvert n\rvert\le 1{,}4$.
3.	$y=\begin{cases} a\sin^2 x+b\cos zx, & x<-\ln(a); \\ a^b-\cos^3(a+zx), & -\ln(a)<x\le b; \\ \sqrt{2{,}5a^3+(b-zx^2)^p}, & x>b. \end{cases}$	$\lvert a\rvert\le 0{,}2$; $\lvert b\rvert\ge 0{,}5$; $z=e^{ax}$
4.	$y=\begin{cases} \sin(e^{a+b})+x^2, & e^{a+b}>e^c; \\ \mathrm{arctg}(abc)+\sqrt[3]{x}, & e^{a+b}=e^c; \\ \cos(\sqrt{\lvert x+abc\rvert}), & e^{a+b}<e^c. \end{cases}$	$a<-4{,}2$; $b\ge 5.3$; $c\ge 1{,}5$.
5.	$y=\begin{cases} 2{,}8\sin^2 ax-bx^3z, & x<a; \\ z\cos(ax+b)^2+\ln(z), & a\le x\le b^2; \\ e^{2{,}5ax}+zabx, & x>b^2. \end{cases}$	$a\le -5$; $b\ge 2{,}5$; $z=\ln\lvert bx^3\rvert$.
6.	$y=\begin{cases} xe^a+e^{\lvert bc\rvert}, & \lvert 1-b^2\rvert=a+c; \\ \sin^2 ax+\cos bc, & \lvert 1-b^2\rvert>a+c; \\ \sqrt{ab^4+\sqrt{cx}}, & \lvert 1-b^2\rvert<a+c. \end{cases}$	$\lvert a\rvert\ge 3$; $b\le -0{,}7$; $c\ge 2{,}5$.

№	Формула для *y*	Параметри
7.	$y=\begin{cases}\ln\lvert mx+n\rvert, & k^2>m+n;\\ e^{\ln\lvert mx-n\rvert}, & k^2=m+n;\\ \sqrt[3]{k^2+\cos^2 x}, & k^2<m+n.\end{cases}$	$\lvert k\rvert\le 3{,}1$; $\lvert m\rvert>5{,}5$; $n\ge -0{,}5$.
8.	$y=\begin{cases}a\sin^2 x+b\cos(zx+a), & x<a^3;\\ (a+bx)^2-\sin(a+zx), & a^3\le x\le b;\\ \sqrt{(\sin(a+bx+z)-x)}, & x>b.\end{cases}$	$a\le -2{,}2$; $b\ge 7{,}2$; $z=e^x$.
9.	$y=\begin{cases}\sqrt[3]{b^2+\sqrt{\lvert x+c\rvert}}, & \lg a+\lg b<\lg c;\\ \cos(x-a+b-c), & \lg a+\lg b=\lg c;\\ \sin(x+a-b+c), & \lg a+\lg b>\lg c.\end{cases}$	$a\ge 101$; $b\le 9$; $c\ge 1112$.
10.	$y=\begin{cases}e^{ax}-3{,}5\cos^2(z+bx), & x\le a;\\ a+\ln\lvert a+bx\rvert-2x, & a<x\le b^{3,5};\\ a+\cos^{3,5}(a+bxz), & x>b^{3,5}.\end{cases}$	$a\le -1$; $b\ge 3{,}4$; $z=\mathrm{tg}\,bx$.
11.	$y=\begin{cases}\ln(\lg\lvert kx+mn\rvert), & 3k>\lvert m+n\rvert;\\ \sin kmx+\sqrt{\lvert nx\rvert}, & 3k=\lvert m+n\rvert;\\ e^{k\cos x}+e^{m+n}, & 3k<\lvert m+n\rvert.\end{cases}$	$k\ge 4$; $m\le -14{,}7$; $n\ge -0{,}7$.
12.	$y=\begin{cases}x^2e^{2k}+\ln\lvert rx\rvert, & \cos k=\cos rs;\\ \sqrt[3]{x^2}+\lvert\sqrt{k+rsx}\rvert, & \cos k>\cos rs;\\ \mathrm{arctg}(kx+rs), & \cos k<\cos rs.\end{cases}$	$k\ge 1{,}33$; $r\le 0{,}85$; $s\ge 3{,}5$.
13.	$y=\begin{cases}2{,}5b^2+ax-4{,}5\cos xz, & x\le 5a;\\ (a^2-5.4x)^3+\ln(xz), & x>b;\\ \sqrt{6{,}5b^2+(a-x^3z)}, & 5a<x\le b.\end{cases}$	$a\le 0{,}5$; $b\ge 4{,}5$; $z=e^{ax}$.
14.	$y=\begin{cases}\sqrt{ax-\cos^2 b^3x+5{,}1c^2}, & \lvert 1-b^2\rvert=a+c;\\ e^{0{,}007x}+\ln\lvert b^5\cos sx\rvert, & \lvert 1-b^2\rvert>a+c;\\ \cos^2 b^3x^2+\ln\lvert bx-a^2\rvert, & \lvert 1-b^2\rvert<a+c.\end{cases}$	$a\ge 3{,}5$; $b\le -0{,}73$; $c\ge 2{,}5$.
15.	$y=\begin{cases}3{,}5\sin(bx+z)-e^{3{,}5a}, & x\le a;\\ \ln(a+b^3x)+a, & a<x\le b^{2,5};\\ \cos^2(a^b+xz)+a^2, & x>b^{2,5}.\end{cases}$	$a\le 0{,}2$; $b\ge 0{,}5$; $z=e^{2{,}5ax}$.

Завдання 3. Скласти програму визначення характеристик геометричних об'єктів (задачу обрати згідно з номером студента у списку студентів підгрупи). Сформувати умову продовження тестування. Під час виконання алгоритму перевіряти умови, які забезпечуватимуть правильність застосування відповідних математичних формул:

1. Задано катети прямокутного трикутника. Визначити гіпотенузу і кути трикутника.
2. Відома гіпотенуза і гострий кут прямокутного трикутника. Визначити периметр і площу трикутника, а також радіус кола, вписаного у цей трикутник.
3. Відома діагональ квадрата. Обчислити площу і периметр квадрата.
4. Відома діагональ прямокутника і кут між діагоналлю та більшою стороною. Обчислити площу та периметр прямокутника.
5. Трикутник заданий величинами усіх своїх сторін. Визначити усі кути трикутника.

6. Тіло має форму паралелепіпеда з висотою h. Прямокутник в основі має діагональ d. Відомо, що діагоналі основи перетинаються під кутом α. Визначити об'єм тіла і площу повної поверхні.
7. У трикутнику відомі один з катетів і площа. Визначити гіпотенузу, інший катет та гострі кути.
8. Відома площа квадрата. Обчислити сторону і діагональ квадрата, а також площу круга, описаного навколо квадрата.
9. У рівнобедреному трикутнику відома основа і кут при ній. Визначити площу трикутника і величину бічної сторони.
10. Задано катет і гіпотенузу прямокутного трикутника. Обчислити його площу і периметр, а також радіус кола, вписаного у цей трикутник.
11. Відомі площа і периметр прямокутного трикутника. Визначити усі сторони трикутника.
12. Відома діагональ ромба. Обчислити його площу і периметр.
13. Відомі довжина діагоналей прямокутника і кут між ними. Обчислити площу і периметр прямокутника.
14. У прямокутному трикутнику відомі катет і площа. Обчислити периметр трикутника, а також радіус кола, описаного навколо нього.
15. Відомо значення периметра рівностороннього трикутника. Обчислити сторону трикутника та його площу, а також радіус кола, описаного навколо нього.

Завдання 4. Виконати **завдання 1** з обов'язковим введенням функції C++ для обчислення заданої математичної функції.

Завдання 5. Виконати **завдання 2** з обов'язковим введенням функції C++ для обчислення заданої функції з трьома гілками.

Завдання 6. Виконати **завдання 3** з обов'язковим введенням функції C++ для визначення характеристик геометричних об'єктів.

Завдання 7. Оформити функцію C++ для розв'язування таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи):

1. Визначити кількість двоцифрових натуральних чисел, які при діленні на 3 дають в остачі a (a = 0, 1, 2).
2. Визначити кількість двоцифрових натуральних чисел, які при діленні на 4 дають в остачі a (a = 0, 1, …, 3).
3. Визначити кількість двоцифрових натуральних чисел, які при діленні на 5 дають в остачі a (a = 0, 1, …, 4).
4. Визначити кількість двоцифрових натуральних чисел, які при діленні на 7 дають в остачі a (a = 0, 1, …, 6).

5. Визначити кількість трицифрових натуральних чисел, які при діленні на 3 дають в остачі a (a = 0, 1, 2).
6. Визначити кількість трицифрових натуральних чисел, які при діленні на 4 дають в остачі a (a = 0, 1, …, 3).
7. Визначити кількість трицифрових натуральних чисел, які при діленні на 5 дають в остачі a (a = 0, 1, …, 4).
8. Визначити кількість трицифрових натуральних чисел, які при діленні на 7 дають в остачі a (a = 0, 1, …, 6).
9. Визначити суму двоцифрових натуральних чисел, які при діленні на 3 дають в остачі a (a = 0, 1, 2).
10. Визначити суму двоцифрових натуральних чисел, які при діленні на 4 дають в остачі a (a = 0, 1, …, 3).
11. Визначити суму двоцифрових натуральних чисел, які при діленні на 5 дають в остачі a (a = 0, 1, …, 4).
12. Визначити суму двоцифрових натуральних чисел, які при діленні на 7 дають в остачі a (a = 0, 1, …, 6).
13. Визначити суму трицифрових натуральних чисел, які при діленні на 3 дають в остачі a (a = 0, 1, 2).
14. Визначити суму трицифрових натуральних чисел, які при діленні на 4 дають в остачі a (a = 0, 1, …, 3).
15. Визначити суму трицифрових натуральних чисел, які при діленні на 5 дають в остачі a (a = 0, 1, …, 4).

Завдання 8. Оформити функцію C++ для розв'язування таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи).

1. Визначити число, якщо 22% його дорівнює a.
2. Визначити число, якщо 65% його дорівнює a.
3. Визначити число, якщо 35% його дорівнює a.
4. Обчислити 16% від числа a.
5. Обчислити 26,25% від числа a.
6. Обчислити 36,75% від числа a.
7. Порода містить 32% мінералу, в якому є a % золота. Який відсоток золота в породі?
8. Порода містить 45% мінералу, в якому є a % золота. Який відсоток золота в породі?
9. Скільки грамів золота міститься у злитку масою 2 кг, якщо його вміст становить a %?

10. Скільки грамів золота міститься у злитку масою 4,2 кг, якщо його вміст становить *a* %?
11. Турист пройшов 2/5 шляху за *a* год. За скільки годин він пройде решту шляху?
12. Мати старша від сина у 4 рази. Разом їм *a* років. Скільки років синові?
13. Мати старша від дочки удвічі. Разом їм *a* років. Скільки років дочці?
14. Батько старший від сина в *a* разів. Скільки років синові, якщо батько старший на 20 років?
15. Батько старший від сина у 5 разів. Скільки років синові, якщо батько старший на *a* років?

Завдання 9. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 1**.

Завдання 10. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 2**.

Завдання 11. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 3**.

Завдання 12. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 4**.

Завдання 13. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 5**.

Завдання 14. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 6**.

Завдання 15. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 7**.

Завдання 16. Створити *керовану* (залежну від CLR) консольну програму для виконання **завдання 8**.

3. ГАЛУЖЕННЯ І ЦИКЛИ

План викладу матеріалу:

1. Опис конструкцій та інструкцій C++.
2. Конструкції галуження та вибору.
3. Конструкції циклу.
4. Програми з простим повторенням.
5. Обчислення із заданою точністю.
6. Рекурсивні та ітераційні процеси.
7. Програмування задач цілочислової арифметики.

Ключові терміни розділу

- *Базові конструкції*
- *Інструкція виразу*
- *Позначена інструкція*
- *Блок*
- *Конструкція галуження*
- *Конструкція циклу* `while`
- *Конструкція циклу* `for`
- *Інструкція* `break`
- *Ітераційні процеси*
- *Порожня інструкція*
- *Інструкція* `return`
- *Інструкція* `goto`
- *Локалізація об'єктів у блоці*
- *Конструкція вибору*
- *Конструкція циклу* `do … while`
- *Вкладені цикли*
- *Інструкція* `continue`
- *Рекурсивні процеси*

3.1. Опис конструкцій та інструкцій C++

Відомо, що будь-яку програму C++ можна записати (закодувати) тільки за допомогою трьох базових конструкцій: *послідовності інструкцій*; *галуження* (вибору) та *циклу*.

Паралельно з терміном *конструкція* використовують термін *інструкція* (тобто вживають терміни: інструкція галуження, інструкція вибору, інструкція циклу). Видається, що термін *конструкція* є зайвим. Однак це не так, оскільки при цьому фіксується той факт, що *конструкція* – це складна інструкція із певною зафіксованою структурою.

Оскільки будь-яку конструкцію циклу можна записати за допомогою конструкцій послідовності інструкцій та галуження (це ми продемонструємо згодом), то її зачисляють до переліку базових конструкцій тільки з мотивів зручності та спрощення програмування. З цих же мотивів, поруч з базовими конструкціями, у мові

C++ функціонують ще додаткові прості інструкції, які значно спрощують процес кодування програми. Вичерпний перелік конструкцій та інструкцій мови C++ відображено на рис. 1.

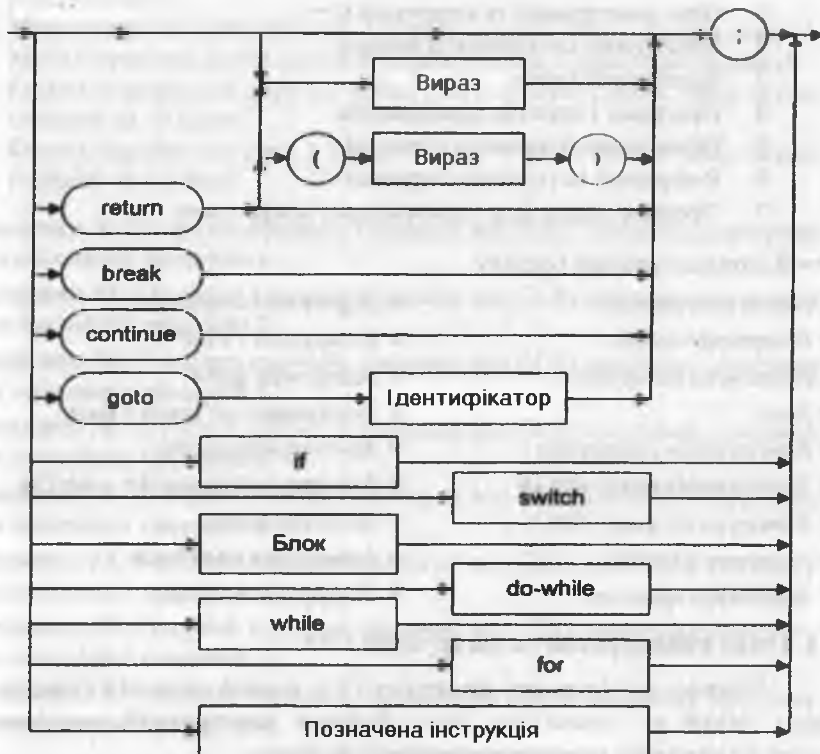


Рис. 1. Перелік конструкцій та інструкцій C++

Найпростішою формою інструкції є *порожня інструкція* (;), яка містить тільки крапку з комою. Ця інструкція не виконує жодних дій. Однак вона корисна у випадках, коли синтаксис вимагає наявності інструкції, а за логікою кодування програми інструкції не потрібно.

У програмі найчастіше трапляється *інструкція виразу*, яка містить вираз, за яким стоїть крапка з комою, або кома, або інший контекст. Наприклад:

```
a=b*3+c; d++; x--, y+=x;
```


Інструкція `return` має дві форми:

```
return;
return вираз;
```
або еквівалентно `return(вираз);`

Перша форма забезпечує передачу керування з блоку поточної функції типу `void` у *точку повернення*. Ця точка відповідає початку інструкції, що розташована одразу ж після виклику функції.

Друга форма не тільки забезпечує передачу керування у точку повернення, але й повертає значення `виразу` в точку виклику функції. Отже, цю форму застосовують у блоці функції, яка повертає значення. Очевидно, що тип `виразу` в інструкції `return` і тип функції мають співпадати.

У блоці функції може бути декілька інструкцій `return` або не бути жодної (передача керування у точку повернення відбувається після виконання останньої інструкції блоку цієї функції).

Увага! Відсутність інструкції `return` засвідчує поганий стиль програмування.

Інструкції `break` (*припинити*) і `continue` (*продовжити*) використовують тільки у конструкціях вибору та циклу, де їх опишемо.

Будь-яку інструкцію у блоці функції можна позначити, поставивши перед нею через дві крапки ідентифікатор, наприклад:

```
m1: x=2*t-1;
```

Ідентифікатор, який розміщено перед інструкцією, називають *позначкою* (mark), а саму інструкцію – *позначеною*. Областю видимості позначки є блок поточної функції, а власне позначка слугує об'єктом тільки для інструкції `goto`. А це означає, що з будь-якої точки блоку поточної функції можна здійснювати безумовну передачу керування на позначену інструкцію за допомогою інструкції

```
goto позначка;
```

Для нашого прикладу ця інструкція має вигляд

```
goto m1;
```

Увага! Використання інструкції `goto` засвідчує поганий стиль програмування. Зазвичай, допускають використання `goto` для виходу з набору вкладених циклів під час виявлення помилок (`break` дає змогу вийти тільки з одного циклу).

Блок (або складений оператор) – це послідовність *визначень* об'єктів (констант, змінних, екземплярів класів тощо) та *інструкцій*, обмежена з обох боків фігурними дужками, наприклад:

```
{ int a, b=1; // визначення об'єктів (змінних a, b)
  a+= b; b++; // інструкції виразів
  int x=5;
  …
  -- x;
}
```

Для будь-якого об'єкта важливими характеристиками є область дії та час життя. *Областю дії* об'єкта називають ту частину програми, в якій цим об'єктом можна користуватися.

Часом життя об'єкта називають проміжок часу, протягом якого значення цього об'єкта є доступним у деякій частині програми. Час життя об'єкта може бути настільки коротким, як час виконання інструкцій блоку, або настільки ж довгим, як час виконання усієї програми. Детально про ці характеристики об'єктів ми говорили у розділі 2 під час вивчення змінних.

Будь-які об'єкти, визначені у блоці, мають силу (область дії та час життя) тільки у цьому блоці. Іншими словами, блок *локалізує* об'єкти, які в ньому визначені. Визначення об'єктів та інструкцій у блоці можуть чергуватися, однак визначення конкретного об'єкта має передувати його першому використанню в деякій інструкції.

В усіх місцях програми, де, згідно з синтаксисом, допускається *інструкція*, можна записати блок, а у будь-якому блоці можна локалізувати змінні. Наприклад, синтаксис найпростішої конструкції галуження:

```
if (вираз) інструкція
```

Конкретний вигляд цієї конструкції може бути таким:

```
if (x>0) { int i=2; …; cout<<i; }
```


3.2. Конструкції галуження та вибору

Синтаксичну діаграму конструкції галуження (інструкції `if`) наведено на рис. 2.

Рис. 2. Синтаксична діаграма конструкції галуження

Конструкція галуження має дві форми (довгу і коротку):

```
if (вираз) Інструкція_1; else Інструкція_2;
if (вираз) Інструкція_1;
```

Незалежно від форми, програма спочатку обчислює значення `виразу`, який може бути двох типів: цілого чи логічного. Якщо його значення не дорівнює нулю чи є істинним (`!=0` / `true`), то виконується `Інструкція_1`. Після цього керування передається інструкції, яка розміщена одразу ж за конструкцією галуження.

Якщо ж значення `виразу` дорівнює нулю чи є хибним (`=0` / `false`), то виконується `Інструкція_2` для довгої форми, чи *порожня* інструкція – для короткої форми. Після цього керування переходить інструкції, яка розміщена одразу за конструкцією галуження.

Приклад 1. Скласти фрагмент програми обчислення $z = \max(x, y)$.

➢ Фрагмент програми:

```
if (x>y) z=x; else z=y;
```

Використавши тернарний умовний оператор, цей приклад можна закодувати ще й так:

```
z = (x>y)? x : y;
```
⊲

На місці `Інструкції_1` / `Інструкції_2` у свою чергу може стояти інша інструкція `if` (тобто інструкції `if` можуть вкладатися одна в іншу). У цьому випадку діє просте *правило*: кожне `else` зв'язується з найближчим `if`, записаним зліва від нього.

Приклад 2. Скласти програму обчислення $z = \max(a, b, c)$.

➢ *Попередні міркування*. Якщо число a більше за числа b і c, то a – максимальне (найбільше) число, у протилежному випадку найбільшим числом буде число b або число c. Відповідний фрагмент програми має вигляд:

```
if(a>b && a>c) z = a;
  else if(b>c) z=b; else z=c;
```

Фрагмент `else z=c` зв'язується з умовою `if(b>c)`. Використавши тернарний умовний оператор, матимемо таке:

```
z=(a>b && a>c)? a : b>c? b : c;
```

Складемо програму, орієнтуючись на останній варіант.

На вході (`Enter a b c`). Кожен рядок є окремим тестом, що містить трійку цілих чисел. Ознакою закінчення вхідних даних є рядок, який на початку містить 0 (нуль).

На виході (`Result`). Для кожної трійки цілих чисел одержуємо z.

Програма:

```
#include <iostream>
using namespace std;
//---------------------------------------------------
void  main() { int a, b, c, z;
  while (cout<<"Enter a b c:", cin>>a>>b>>c, a)
   {  z=(a>b && a>c)? a : b>c? b : c;
     cout<<"  Result: z="<<z<< endl;  }
  cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_If_Ex2\Example_2\Debug\Example_2.exe
Enter a b c: 2 3 8
  Result: z=8
Enter a b c: 5 6 1
  Result: z=6
Enter a b c: 4 1 2
  Result: z=4
Enter a b c: 3 3 3
  Result: z=3
Enter a b c: 4 5 5
  Result: z=5
Enter a b c: 0
```

◄

Приклад 3. Скласти програму обчислення функції

$$y = \begin{cases} -1, & \text{якщо } x < -2; \\ x+1, & \text{якщо } -2 \le x < 0; \\ x^2+1, & \text{якщо } 0 \le x \le 1; \\ 3x-1, & \text{якщо } x > 1. \end{cases}$$

➢ *Попередні міркування*. Для кожної умови функції запишемо окрему інструкцію `if` (*перший варіант*). Оскільки істинною є тільки одна умова, то присвоєння значення y відбудеться лише раз.

На вході (`Enter x`). Кожен рядок є окремим тестом, що містить раціональне число. Ознакою завершення тестування є число –99.

На виході (`Result`). Для кожного числа одержуємо y.

Програма:

```
#include <iostream>
using namespace std;
//----------------------------------------------------
void main () { float x, y;
 do
   {  cout << "Enter x:"; cin >> x;
      if (x<-2)          y=-1;
      if (x>=-2 && x<0)  y=x+1;
      if (x>=0  && x<=1) y=x*x+1;
      if (x>1)           y=x+1;
      cout << "  Result: y= " << y << endl;
   } while (x!=-99);
  cin>> x; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_If\Example_3\Debug\Example_3.exe
Enter x:-4.5
  Result: y= -1
Enter x:-1.7
  Result: y= -0.7
Enter x:0.5
  Result: y= 1.25
Enter x:5
  Result: y= 6
Enter x:-99
  Result: y= -1
```

Замість чотирьох інструкцій if можна записати тільки одну складну інструкцію if (*другий варіант*):

```
…
if (x<-2) y=-1;
   else if (x<0) y=x+1;
           else if (x<=1) y=x*x+1; else y=x+1;
…
```

Перевірка належності аргументу *x* наступному проміжку виконується тільки у випадку, якщо *x* не входить у попередній проміжок. Програма стала компактнішою та ефективнішою, проте менш наочною. Який же варіант є кращим?

У сучасній ієрархії критеріїв якості програм на першому місці стоїть надійність і простота модифікації, а ефективність і компактність відходять на другий план. Отже, якщо немає спеціальних вимог щодо швидкодії, рекомендують обирати найпростіший варіант (у нашому випадку – перший).

Отже, якщо інструкція if реалізує вибір між двома альтернативами, то немає жодних проблем з простотою та наочністю коду програми. Збільшення кількості альтернатив вимагає збільшення кількості інструкцій if або значного ускладнення логіки програми.

У деяких випадках певним компромісом між цими граничними варіантами буде використання *конструкції вибору*, синтаксичну діаграму якої наведено на рис. 3.

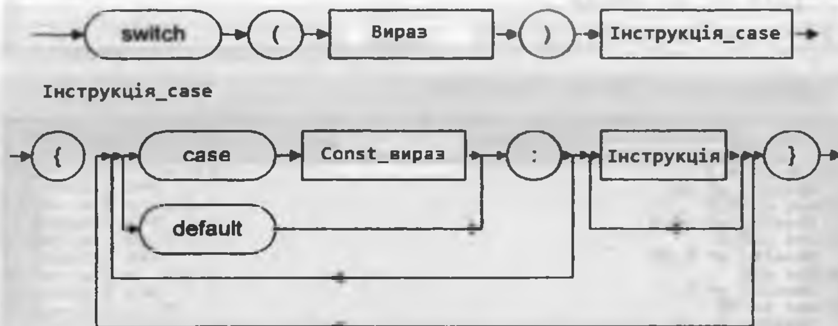


Рис. 3. Синтаксична діаграма конструкції вибору

У цій конструкції програма передусім обчислює *значення виразу*, вказаного після слова switch (назвемо його switch_значенням), яке може бути тільки *цілого* чи *символьного* типу.

Потім програма *послідовно* переглядає префікси case (*випадок*), обчислює const_вирази, а отримані значення порівнює зі switch_значенням. Якщо для деякого case ці результати збіглися, то керування передається інструкції, яка розміщена за цим case.

Якщо *жодного* співпадіння не відбулося, а у конструкції вибору *зазначено* слово default (*за домовленістю*), то керування передається інструкції, яка розміщена за цим словом.

Якщо *жодного* співпадіння не відбулося, а у конструкції вибору *не зазначено* слово default, то керування передається інструкції, яка розміщена одразу ж за конструкцією вибору.

Після передачі керування інструкції за префіксом case виконується ця та усі наступні інструкції (незалежно від значень відповідних const_виразів) до завершення Інструкції_case (до закриваючої фігурної дужки), якщо тільки цю послідовність виконання інструкцій не буде змінено інструкціями break або goto.

Дія інструкції break за цих умов полягає у передачі керування інструкції, яка розміщена одразу ж за конструкцією вибору. Пояснимо це на прикладі такої конструкції вибору:

```
switch (вираз)
{ case значення_1: Інструкція_1; break;
  case значення_2: Інструкція_2; break;
 ...
  case значення_N: Інструкція_N; break;
  default: Інструкція_default;
}
```

Якщо значення виразу = значенню_1, то виконується Інструкція_1 і виходимо зі switch. Якщо ж значення виразу = значенню_2, то виконується Інструкція_2; виходимо зі switch і т. д. Якщо ж значення виразу не збігається з жодним зі значень (значення_1, …, значення_N), то виконується Інструкція_default.

Найчастіше конструкції вибору застосовують разом з *переліком констант* (enum).

Приклад 4. Проаналізувати значення кольору.

➢ *На вході* (Enter i). Кожен рядок є окремим тестом, що містить невід'ємне ціле число. Числа 0; 1; 2 і 3 відповідають кольорам з переліку color, якщо його переглядати зліва направо. Ознакою завершення тестування слугує довільне від'ємне число.

На виході. Якщо введене число відповідає кольору, то одержуємо характеристику кольору (гарячий – Hot, холодний – Cool). У протилежному випадку виводимо повідомлення про помилку (Error).

Програма:

```
#include <iostream>
using namespace std;
void main ()
{ enum color { red, yellow, green, blue }; int i;
  while ( cout<<"Enter i: ", cin>>i, i>=0)
   { switch (i)
      {  case red:
         case yellow: cout << "Hot!"<<endl; break;
         case green:
         case blue: cout << "Cool!"<<endl; break;
         default: cout << "Error!"<<endl;
      }
   }
  cin>>i; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_If\Example_4\Debug\...
Enter i: 0
Hot!
Enter i: 1
Hot!
Enter i: 2
Cool!
Enter i: 3
Cool!
Enter i: 4
Error!
Enter i: -5
```


Використання конструкції вибору є простим і наочним способом програмування галужень у випадках, коли маємо понад дві альтернативи. Однак цією конструкцією неможливо скористатися *прямо*, якщо необхідно перевіряти *комбінації* декількох умов. Розглянемо такий приклад.

Приклад 5. Значення змінної w залежить від значень змінних x, y, z:
1) якщо x<-2 і y<2, то для z<0 змінна w=-2, а для z>=0 змінна w=-3;
2) якщо x<-2 і y>=2, то для z<0 змінна w=2, а для z>=0 змінна w=3;
3) якщо x>=-2 і y<2, то незалежно від z змінна w=20;
4) якщо x>=-2 і y>=2, то незалежно від z змінна w=30.

Скласти фрагмент програми присвоєння значень змінній w.

➢ Найпростіший та найзрозуміліший варіант розв'язку полягає у використанні *послідовності* інструкцій if:

```
// ініціалізація x, y, z
if (x<-2 && y<2 && z<0) w=-2;
if (x<-2 && y<2 && z>=0) w=-3;
if (x<-2 && y>=2 && z<0) w=2;
if (x<-2 && y>=2 && z>=0) w=3;
if (x>=-2 && y<2) w=20;
if (x>=-2 && y>=2) w=30;
...
```

Як бачимо, цей варіант цілковито повторює умови прикладу засобами C++. Компактнішим, однак менш зрозумілим, є варіант розв'язку, за якого використовують *вкладеність* інструкцій if:

```
// ініціалізація x, y, z
if (x<-2)
   if (y<2)
      if (z<0) w=-2; else w=-3;
   else
      if (z<0) w=2; else w=3;
else if (y<2) w=20; else w=30;
...
```

Неможливість використання конструкції вибору для перевірки *комбінації* декількох умов можна легко обійти, якщо скористатися *змінною стану* програми, значення якої відображають певні *події* під час роботи програми, зокрема, виконання чи невиконання деяких логічних умов.

У попередньому прикладі маємо три умови: `x<-2`, `y<2` і `z<0`. Згадаємо, що значення *істинності* умови (`true`) відповідає цілому числу 1, а значення *хибності* умови (`false`) – числу 0. Це дає змогу ввести *змінну стану* програми `t`, яка набуватиме значень, що залежатимуть від комбінацій *істинностей* / *хибностей* усіх трьох умов.

У вираз, який ініціалізуватиме змінну стану програми `t`, кожна умова входитиме зі своєю вагою: зовнішня умова (`x<-2`) матиме найбільшу вагу 4; внутрішня умова (`z<0`) – найменшу вагу 1, а проміжна умова (`y<2`) – вагу 2, тобто:

`t = 4·(x<-2)+2·(y<2)+(z<0).`

Така побудова правої частини `t` дає змогу цілими числами від 0 до 7 однозначно *ідентифікувати* будь-яку можливу комбінацію умов попереднього прикладу:

1) якщо `t=4·1+2·1+1 = 7`, то `w=-2`; якщо `t=4·1+2·1+0 = 6` то `w=-3`;
2) якщо `t=4·1+2·0+1 = 5`, то `w=2`; якщо `t=4·1+2·0+0 = 4`, то `w=3`;
3) якщо `t=4·0+2·1+1 = 3` або `t=4·0+2·1+0 = 2`, то `w=20`;
4) якщо `t=4·0+2·0+1 = 1` або `t=4·0+2·0+0 = 0`, то `w=30`.

Змінну стану `t` для програмування складних галужень не обов'язково завжди

використовувати з конструкцією вибору. Якщо для будь-якого значення `t` результат інструкції, що виконуватиметься, наперед можна обчислити / визначити, то для реалізації галужень можна задіяти одновимірний масив, індекси якого відповідатимуть значенням `t`, а елементи масиву – відповідним результатам. Для попереднього прикладу матимемо таке:

```
// ініціалізація x,y,z
int v[]={30, 30, 20, 20, 3, 2, -3, -2};
int t=4*(x<-2)+2*(y<2)+(z<0);  w=v[t];
…
```


3.3. Конструкції циклу

Для організації повторення виконання деякої послідовності дій використовують конструкції циклів: `while`, `do … while` і `for`. Перші дві конструкції ще називають циклами типу `while`.

Конструкцію циклу `while` називають *циклом з передумовою*, оскільки перевірка умови продовження циклу здійснюється *перед* виконанням `інструкції_циклу`. Цю конструкцію циклу використовують у випадках, коли кількість повторення циклу наперед невідома і можливим є випадок, за якого `інструкція_циклу` не виконається жодного разу. Конструкція циклу з передумовою:

```
while (вираз) інструкція_циклу;
```

Замість `інструкції_циклу` може стояти `блок`. Роботу цього циклу пояснимо за допомогою такої послідовності інструкцій:

```
Cycle: if(вираз)
{ інструкція_циклу;
  goto Cycle;
}
```

Отже, спочатку обчислюється значення `виразу`, який може бути цілого чи логічного типу. Якщо його значення не дорівнює нулю чи є істинним (`!=0` / `true`), то виконується `інструкція_циклу`. В іншому випадку керування передається інструкції, розміщеній одразу ж за циклом.

Отже, `вираз` є умовою *продовження* циклу. Одним із результатів виконання `інструкції_циклу` має бути зміна значення змінних, які входять у `вираз`. У протилежному випадку цикл може виконуватися нескінченну кількість разів. Якщо значення `виразу` на початку виконання циклу дорівнює нулю чи є хибним (`=0` / `false`), то `інструкція_циклу` не виконається жодного разу.

Приклад 6. Скласти фрагмент програми обчислення *n*!

➤

```
int n, f=1, i=1;
 // Введення величини n
 // Змінна f отримає значення n!
while(i<n) { i++; f*=i;}
```

◄

Розглянемо деякі спеціальні випадки. Замість інструкції_-циклу може стояти порожня інструкція, наприклад:

```
while (oznaka(x)<0);
```

У цьому випадку уся робота циклу відбувається за рахунок виконання умов циклу і не вимагає інших інструкцій (oznaka(x) – функція, яка повертає цілі значення; як тільки вона поверне невід'ємне значення – робота циклу завершиться). Очевидно, що цей цикл слугує для організації певного *таймера*.

Зазвичай, замість інструкції_циклу стоїть блок, який дає змогу циклічно виконувати групу інструкцій. Серед інструкцій блоку можуть зустрічатися інструкції break і continue. Інструкція break передає керування інструкції, розміщеній одразу за циклом while. Інструкція continue дає змогу пропустити інструкції, які розміщені після неї у блоці, та почати нову ітерацію циклу. Принцип виконання інструкцій break і continue уточнимо за допомогою позначок та відповідних коментарів з goto:

<table>
<tr>
<td><pre>while (вираз)
 { інструкція;
    …
    break; // goto After;
    інструкція;
    …
 }
After: інструкція;</pre></td>
<td><pre>while (вираз)
 { інструкція;
    …
    continue; // goto End;
    інструкція;
    …
    End: ;
 }</pre></td>
</tr>
</table>

У цьому випадку можливим є навмисне створення "псевдо-нескінченного" циклу, наприклад:

```
while (1)
 {  інструкція;
    …
    break; // goto After;
    інструкція;
    …
 }
After: інструкція;
```

Вихід з цього циклу можливий тільки за допомогою break.

Конструкцію циклу do … while називають *циклом з пост-умовою*, оскільки перевірка умови продовження циклу здійснюється *після* виконання інструкції_циклу. Цю конструкцію циклу використовують у випадках, коли кількість повторення циклу наперед невідома, а інструкція_циклу має виконатися хоча б один раз. Конструкція циклу з постумовою:

```
do інструкція_циклу while(вираз);
```

Замість інструкції_циклу може стояти блок. Роботу цього циклу пояснимо за допомогою такої послідовності інструкцій:

```
Cycle: інструкція_циклу;
if(вираз) goto Cycle;
```

Отже, спочатку виконується інструкція_циклу, а потім обчислюється значення виразу, який може бути цілого чи логічного типу. Якщо його значення не дорівнює нулю чи є істинним (!=0 / true), то інструкція_циклу виконується знову і т. д. Якщо ж значення виразу дорівнює нулю чи є хибним (=0 / false), то керування передається інструкції, розміщеній одразу за конструкцією циклу.

Отже, вираз є умовою *продовження* циклу. Одним із результатів виконання інструкції_циклу має бути зміна значення змінних, які налічує вираз. У протилежному випадку цикл може виконуватися нескінченну кількість разів.

Якщо замість інструкції_циклу стоїть блок, то серед інструкцій блоку можуть траплятися інструкції break і continue, дія яких аналогічна дії цих інструкцій у блоці конструкції while.

Приклад 7. Використовуючи цикл з постумовою, обчислити *n*!

➤

```
int n, f=1, i=0;
 // Введення величини n
 // Змінна f отримає значення n!
do {i++; f*=i;} while(i<n);
```

◄

Увага! Хоча кількість повторення циклу у прикладах 6 і 7 наперед відома, ми використали цикли типу `while` з демонстраційною метою (алгоритм обчислення *n*! є очевидним).

Конструкцію циклу `for` називають *циклом з параметром* і використовують у випадках, коли наперед *відома* точна *кількість повторення* циклу.

Змінні, які змінюють свої значення у блоці циклу і беруть участь у перевірці умов продовження циклу, називають *параметрами циклу*. Цілочислові параметри циклу, які змінюються з постійним кроком, називають *лічильниками циклу*. Зазвичай, у циклі використовують один параметр. Конструкція циклу з параметром:

```
for (вираз_1; вираз_2; вираз_3) інструкція_циклу;
```

Замість `інструкції_циклу` може стояти блок. Роботу цього циклу пояснимо за допомогою такої послідовності інструкцій:

```
вираз_1;
Cycle: if(вираз_2)
{ інструкція_циклу; вираз_3; goto Cycle;  }
```

`Вираз_1` задає ініціалізацію *параметрів циклу* та інших величин, що використовуються у циклі; `вираз_2` – задає умову продовження циклу; `вираз_3` – задає збільшення / зменшення параметрів циклу. Один з `виразів` чи усі `вирази` можуть не вказувати у `for`. Відсутність `виразу_2` еквівалентна умові `if(1)`; інші не вказані `вирази` просто пропускаються під час трансляції програми.

Приклад 8. Використовуючи цикл з параметром, обчислити *n*!

➤

```
int n;
 // Введення величини n
int f=1; // Змінна f отримає значення n!
for (int i=1; i<=n; i++) f*=i;
```

Детальніше роботу цього циклу `for` можна пояснити так:

```
Begin: int i=1; // i - параметр циклу
Cycle: if(i<=n) { f*=i; i++; goto Cycle; }
After: інструкція;
```

`Вираз_1 (int i=1)` задає визначення параметра циклу – змінної `i`, яка має силу тільки у циклі (від `Begin` до початку `After`). ⮜

Якщо у попередньому прикладі необхідно одержати значення параметра циклу `i` після виходу із циклу, то наступний фрагмент програми (`Фрагмент_1`)

```
// Фрагмент_1
int f=1; // Змінна f отримає значення n!
// Параметр циклу i - внутрішня змінна циклу
for (int i=1; i<=n; i++) f*=i;
cout << "i="<<i<<endl; // Помилка
…
```

спричинить помилку трансляції, якщо змінну `i` визначено тільки у цьому контексті. У цьому фрагменті параметр циклу `i` – це *внутрішня* змінна циклу.

Якщо ж змінну `i` визначено також у деякому контексті, що охоплює `Фрагмент_1`, то виведеться значення `i` з цього контексту, однак у жодному разі це не буде значення параметра циклу `i` після виходу із циклу.

Для одержання значення параметра циклу `i` після виходу із циклу необхідно застосувати фрагмент програми (`Фрагмент_2`), в якому параметр циклу `i` буде *зовнішньою* змінною циклу:

```
// Фрагмент_2
int f=1; // Змінна f отримає значення n!
int i=1; // Параметр циклу i - зовнішня змінна циклу
for (; i<=n; i++) f*=i;
cout << "i="<<i<<endl; // Результат: i = n+1
…
```

Очевидно, що змінна `f`, яка отримує результат обчислення `n!`, може бути тільки *зовнішньою* змінною циклу.

В усіх трьох циклах на місці `інструкції_циклу` можна записати іншу конструкцію циклу (отримуємо *вкладені цикли*). Інструкція `break` забезпечує вихід тільки з того блоку, де він розміщений.

Під час використання вкладених блоків варто будь-які *визначення* об'єктів виносити за межі зовнішнього циклу (найпершого за порядком запису у програмі).

3.4. Програми з простим повторенням

3.4.1. Цикли з покроковим уведенням даних

Розглянемо задачі, для розв'язання яких необхідно виконати перебирання певної послідовності значень, причому кожне значення у процесі розв'язування задачі використовуватимемо тільки один раз.

Одноразове використання конкретного значення на певній ітерації (на певному кроці) циклу вказує на те, що заводити масиви для цих задач немає змісту.

Оскільки конкретні значення задіяні тільки на певному кроці циклу, то їх і вводитимемо з консолі на відповідному кроці (ітерації) циклу. Кількість кроків циклічної обробки даних може бути відомою наперед (застосовують цикл `for`), або визначатися за допомогою спеціально обумовленої *ознаки* – певного числа чи комбінації чисел (застосовують цикли типу `while`).

Таку схему циклічного опрацювання даних з їхнім покроковим введенням ми застосовуємо уже з перших прикладів для *організації тестування програм* (у цьому розділі її застосовано у попередніх прикладах 2 – 4).

Приклад 9. Серед 5-ти введених чисел полічити кількість нульових.

➢ *На вході* (k=). Кожен рядок є окремим тестом, що містить довільне ціле число. Кількість чисел, які необхідно ввести, жорстко зафіксовано у програмі.
На виході (n=). Отримуємо кількість уведених 0 (нулів).

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------
void main() { int n=0, k;
  for (int i=1; i<=5; i++)
  {  cout << " k= "; cin>>k;
     if(k==0) n++;
  }
  cout<< "n = " << n<< endl; cin>> n; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_8\Debug\Example_8.exe
k= 5
k= 0
k= -1
k= 4
k= 0
n = 2
```

Увага! Програму прикладу 9 наведено тільки з навчально-методичних міркувань. Для більшої універсальності цієї програми необхідно у діалоговому режимі отримати від користувача кількість чисел, які перевірятимуть на 0.

Приклад 9 (*модифікований*). Серед введених чисел полічити кількість нульових. Кількість чисел, які перевірятимуть на 0, отримати від користувача.

➢ *Попередні міркування*. Оскільки кількість чисел (змінна `m`) – це натуральне число, то у програмі передбачено перевірку правильності введення `m` за допомогою циклу `do … while`.
На вході. Рядок (`Enter m>0:`) забезпечує введення `m` (кількості чисел). Він повторюватиметься доти, доки користувач вводитиме неправильні дані. Кожен наступний рядок (k=) є окремим тестом, що містить довільне ціле число.
На виході (n=). Отримуємо кількість уведених 0 (нулів).

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------
void main()
{ int n=0, k, m;
  do {cout<< "Enter m>0: "; cin>>m;} while (m<=0);
  for (int i=1; i<=m; i++)
    {cout << " k= "; cin>>k; if(k==0) n++;}
  cout<< "n = " << n<< endl; cin>> n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_8\Debug\Example_8.exe
Enter m>0: -4
Enter m>0: 0
Enter m>0: 3
 k= -5
 k= 0
 k= 2
n = 1
```

Приклад 10. Пари додатних раціональних чисел вводять з клавіатури. Обчислити добуток кожної пари і суму всіх чисел.

➢ *На вході* (`Enter a b`). Кожен рядок є окремим тестом і містить раціональні числа `a` і `b`, які *водночас* не дорівнюють нулю (недодатні числа *ігноруватимуться*). Ознакою завершення введення даних є рядок, в якому *обидва* числа дорівнюють 0.

На виході. Для кожної пари *додатних* раціональних чисел, одержуємо значення їхнього добутку (`a*b=`), а наприкінці (`s=`) – суму усіх чисел, які формували відповідні добутки.

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------------
void main() { float a, b, s=0;
  while (cout<<"Enter a b: ", cin>>a>>b, a||b)
    if(a>0 && b>0) {cout<<" a*b="<<a*b<<endl; s+=a+b;}
  cout<<"s="<<s<<endl;  cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_10\Debug\Example_10.exe
Enter a b: 1.2 2.1
 a*b=2.52
Enter a b: 1 -3.4
Enter a b: 2.5 3.5
 a*b=8.75
Enter a b: 0 0
s=9.3
```


3.4.2. Тестування програми за допомогою випадкових чисел

У попередніх прикладах *тестування* програми, зазвичай, відбувалося на певній підмножині чисел, які вводив користувач за допомогою клавіатури. Цей спосіб є недосконалим, оскільки вимагає значних затрат часу (насамперед під час введення значної кількості елементів масиву). Окрім цього, під час тестування числа мають бути наперед неочікуваними (*випадковими*).

Користувачеві із винятково психологічних причин дуже важко забезпечити випадковість чисел для тестування. Як засвідчують численні експерименти, користувач, зазвичай, вводить *цілі* числа із діапазону від –20 до 20. Отож одним зі способів тестування програми є її *тестування* на певній множині *випадкових* чисел.

З цією метою на комп'ютері використовують *псевдовипадкові* числа – числа, отримані за допомогою деякої формули, яка імітує значення випадкової величини. Рано чи пізно деякі послідовності цих чисел починають повторюватися (виникають періоди), звідси й назва – *псевдовипадкові* числа. У С++ для одержання псевдовипадкових чисел використовують функцію `rand()` з прототипом:

```
int rand(void);
```

Вона генерує псевдовипадкове ціле число на проміжку значень від 0 до `RAND_MAX`. Величина `RAND_MAX` є константою, яка залежить від реалізації мови С++, однак найчастіше дорівнює 32 767.

Функція `rand()` генерує псевдовипадкові числа за допомогою складного алгоритму, який базується на параметрі `seed` (зерно). Тобто псевдовипадкові числа залежать від значення, яке має `seed` у момент виклику `rand()`. За домовленістю компілятор встановлює `seed = 1` (тобто послідовність чисел хоч і буде псевдовипадковою, проте завжди однаковою). А це не те, що нам потрібно.

Виправити ситуацію допомагає функція `srand()`, яка має такий прототип:

```
void srand (unsigned int seed);
```

Функція `srand` встановлює `seed` рівним значенню параметра, з яким її викликали. І послідовність псевдовипадкових чисел теж буде іншою. Питання: як зробити випадковим `seed`, адже від нього все залежить?

Типова відповідь на це питання: використати функцію time, яка має такий прототип:

```
time_t time(time_t * timer);
```

Ця функція повертає кількість секунд, що пройшли з 1 січня 1970 року. Значення цієї функції передають у функцію srand() (водночас виконується неявне приведення типу), і у програмі Кожен раз генеруватимуться усе нові та унікальні послідовності псевдовипадкових чисел.

Для задіювання функцій rand() і srand() потрібно підключити заголовковий файл <cstdlib>, а для використання time() – файл <ctime>.

Для одержання псевдовипадкових цілих чисел на довільному проміжку $[a; b]$ необхідно задіяти таку послідовність інструкцій:

```
float scale = rand()/float(RAND_MAX); // Одержання
// псевдовипадкового числа на проміжку [0; 1]
int isc = int(a+scale*(b-a) + 0.5); // Перетворення
// (масштабування) цього числа до числа з про-
// міжку [a; b] з подальшим заокругленням до
// цілого значення
```

Приклад 11. На довільному проміжку $[a; b]$ випадковим чином отримати 10 цілих чисел і встановити, скільки серед них додатних чисел, від'ємних чисел і числа 0.

➤ *Попередні міркування*. Оскільки числа з проміжку $[a; b]$ кожен раз будуть іншими, то їх обов'язково треба виводити на екран для подальшого аналізу (*протоколювати*). У програмі необхідно також передбачити перевірку умови a<b (тільки за умов її істинності програма працюватиме).

На вході (Enter a b). Кожен такий рядок є окремим тестом, що задає межі проміжку. Ознакою завершення введення даних є рядок, в якому *обидва* числа дорівнюють 0.

На виході. Для кожної послідовності 10 випадкових чисел з проміжку $[a; b]$ одержуємо кількість додатних чисел (kp=), від'ємних чисел (kn=) і числа 0 (kz=).

Програма:

```
#include<iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
//-----------------------------------------------------
int main() { int a,b, isc, kn, kp, kz;  float scale;
  while (cout<<"Enter a b: ", cin>>a>>b, a||b)
  { if (a<b)
    {
     kn=0, kp=0, kz=0;
     cout << "10 random numbers ["<<a<<"; "<<b<<"]:\n";
     srand(time(NULL));
     for(int i=0; i<10; i++)
     { scale = rand()/float(RAND_MAX);
       isc= int(a+scale*(b-a) + 0.5);
       (isc<0)? kn++ : ((isc>0)? kp++ : kz++);
       cout << isc  << " ";
     }
     cout << endl << "kn=" << kn << "; kz=" << kz;
     cout << ";  kp=" << kp << endl;
    }
  }
 cin>>a;
 return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Random\Debug\Random.exe
Enter a b: -20 20
10 random numbers [-20; 20]:
-6 11 -10 8 3 -19 -14 -9 15 -15
kn=6;  kz=0;  kp=4
Enter a b: 1 15
10 random numbers [1; 15]:
5 6 13 15 15 11 7 15 14 8
kn=0;  kz=0;  kp=10
Enter a b: 7 5
Enter a b: -50 50
10 random numbers [-50; 50]:
-16 5 -36 4 30 -45 -37 0 -44 -26
kn=6;  kz=1;  kp=3
Enter a b: 0 0
```

3.4.3. Цикли з покроковим виведенням даних

Розглянемо задачі, в яких йтиме мова про обчислення і виведення на консоль певної послідовності значень. Розглянемо декілька прикладів.

Приклад 12. Протабулювати значення функції $y=\sqrt{x^2+1}\cdot\cos x$ на проміжку $[a;\, b]$ з кроком h.

➢ *Попередні міркування*. У програмі необхідно передбачити перевірку виконання умов `a<b` та `h>0`. Цикл перебирання аргументів є надзвичайно простим. Значна частина програми "відповідає" за виведення результатів у табличній формі.

На вході `(a=,b=(a<b),h=(h>0))`. Цей рядок забезпечує уведення меж проміжку $[a;\, b]$ і кроку `h`. Оскільки користувач попереджений явно про обмеження на ці величини, то за умови порушення цих обмежень програма автоматично їх виправлятиме.

На виході. У табличній формі одержуємо значення аргументів і відповідних щодо них значень функції.

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
void main() { float a,b,h,x,y;
  cout<<"a=,b=(a<b),h=(h>0):"; cin>>a>>b>>h;
  if(a>b) {float r=a; a=b; b=r;} // Забезпечення a<b
  if(h<0)h=-h;                   // Забезпечення h>0
  cout << " --------------------------- " << endl;
  cout << " |    x        |       y      | " << endl;
  cout << " --------------------------- " << endl;
  for(x=a; x<=b; x+=h)
   { y=sqrt(x*x+1)*cos(x);
     cout << " |  "  << fixed << x << " |  " ;
     cout  << fixed << y <<"   |  " << endl;
     cout << " -------------------------" << endl;
   }
  cin>> a; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_12\Debug\Example_12.exe
a=,b=(a<b),h=(b>0):1 5 0.5
 -------------------------
 |  x         |   y       |
 -------------------------
 |  1.000000  |  0.764103  |
 -------------------------
 |  1.500000  |  0.127523  |
 -------------------------
 |  2.000000  |  -0.930533  |
 -------------------------
 |  2.500000  |  -2.157145  |
 -------------------------
 |  3.000000  |  -3.130631  |
 -------------------------
 |  3.500000  |  -3.408754  |
 -------------------------
 |  4.000000  |  -2.695042  |
 -------------------------
 |  4.500000  |  -0.971721  |
 -------------------------
 |  5.000000  |  1.446399  |
 -------------------------
```

Очевидно, що виведення деякої послідовності значень у табличній формі може стосуватися не тільки значень функції.

Приклад 13. Сформувати кодову таблицю CP866 (MS DOS).

➢ *На вході* – нічого.

На виході. У табличній формі одержуємо значення пар вигляду `код: символ CP866` (по 10 пар у рядку).

Програма:

```
 #include <iostream>
 using namespace std;
 //-------------------------------------------------------
 void main () { int i=1; // Лічильник кількості пар
   for (int s=1; s<=255;  s++)
    {
        cout << s<< ": "<<char(s) <<"  ";
        // Виведення по 10 пар у рядку
        i++; if (i==11) { i=1; cout <<endl; }
    }
   cin>> i; // Пауза
 }
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Char_Table\Debug\Char_Table.exe
1: ☺  2: ☻  3: ♥  4: ♦  5: ♣  6: ♠  7:   8:   9:                10:

 14: ♫  15: ☼  16: ►  17: ◄  18: ↕  19: ‼  20: ¶
21: §  22: ▬  23: ↨  24: ↑  25: ↓  26: →  27: ←  28: ∟  29: ↔  30: ▲
31: ▼  32:    33: !  34: "  35: #  36: $  37: %  38: &  39: '  40: (
41: )  42: *  43: +  44: ,  45: -  46: .  47: /  48: 0  49: 1  50: 2
51: 3  52: 4  53: 5  54: 6  55: 7  56: 8  57: 9  58: :  59: ;  60: <
61: =  62: >  63: ?  64: @  65: A  66: B  67: C  68: D  69: E  70: F
71: G  72: H  73: I  74: J  75: K  76: L  77: M  78: N  79: O  80: P
81: Q  82: R  83: S  84: T  85: U  86: V  87: W  88: X  89: Y  90: Z
91: [  92: \  93: ]  94: ^  95: _  96: `  97: a  98: b  99: c  100: d
101: e  102: f  103: g  104: h  105: i  106: j  107: k  108: l  109: m  110: n

111: o  112: p  113: q  114: r  115: s  116: t  117: u  118: v  119: w  120: x

121: y  122: z  123: {  124: |  125: }  126: ~  127: ⌂  128: А  129: Б  130: В

131: Г  132: Д  133: Е  134: Ж  135: З  136: И  137: Й  138: К  139: Л  140: М

141: Н  142: О  143: П  144: Р  145: С  146: Т  147: У  148: Ф  149: Х  150: Ц

151: Ч  152: ■  153: ■  154: Ъ  155: Ы  156: Ь  157: Э  158: Ю  159: Я  160: а

161: б  162: в  163: г  164: д  165: е  166: ж  167: з  168: и  169: й  170: к

171: л  172: м  173: н  174: о  175: п  176: ░  177: ▒  178: ▓  179: │  180: ┤

181: ╡  182: ╢  183: ╖  184: ╕  185: ╣  186: ║  187: ╗  188: ╝  189: ╜  190: ╛

191: ┐  192: └  193: ┴  194: ┬  195: ├  196: ─  197: ┼  198: ╞  199: ╟  200: ╚

201: ╔  202: ╩  203: ╦  204: ╠  205: ═  206: ╬  207: ╧  208: ╨  209: ╤  210: ╥

211: ╙  212: ╘  213: ╒  214: ╓  215: ╫  216: ╪  217: ┘  218: ┌  219: █  220: ▄

221: ▌  222: ▐  223: ▀  224: р  225: с  226: т  227: у  228: ф  229: х  230: ц

231: ч  232: ■  233: щ  234: ъ  235: ы  236: ь  237: э  238: ■  239: я  240: Ё

241: ё  242: Є  243: є  244: Ї  245: ї  246: Ў  247: ў  248: °  249: ∙  250:

251: √  252: №  253: ¤  254: ■  255:
```

Приклад 14. Сформувати таблицю кодів CP1251 (для порівняння з результатами попереднього прикладу), починаючи з коду 101.

➢ *На вході* – нічого.

На виході. У табличній формі одержуємо значення пар вигляду `код: символ ANSI` (по 10 пар у рядку).

Програма:

```
#include <iostream>
#include <windows.h> // Підключення CharToOemA
using namespace std;
void main () { int i=1; char t[1];
 for (int s=101; s<=255;  s++)
 {t[0]=char(s); CharToOemA(t,t); //Перетворення до ASCII
  cout<<s<<": "<<t[0]<<" ";   i++;
  if (i==11) { i=1; cout <<endl; }
 } /* for */   cin>> i; /* Пауза*/   }
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_14\Debug\Example_14.exe
101: e  102: f  103: g  104: h  105: i  106: j  107: k  108: l  109: m  110: n

111: o  112: p  113: q  114: r  115: s  116: t  117: u  118: v  119: w  120: x

121: y  122: z  123: {  124: |  125: }  126: ~  127: ⌂  128: _  129: _  130: '

131: _  132: "  133: :  134: ┼  135: ╪  136: _  137: %  138: _  139: <  140: _

141: _  142: _  143:    144: _  145: '  146: '  147: "  148: "  149:    150: -
151: -  152: _  153: Т  154: _  155: >  156: _  157: _  158: _  159: _  160:

161: Ў  162: ў  163: _  164: ¤  165: _  166: |  167: §  168: Ё  169: c  170: Є

171: <  172: ┐  173: -  174: R  175: Ї  176: °  177: +  178: _  179: _  180: _

181: ч  182: ¶  183: ·  184: ё  185: №  186: є  187: >  188: _  189: _  190: _

191: ї  192: А  193: Б  194: В  195: Г  196: Д  197: Е  198: Ж  199: З  200: И

201: Й  202: К  203: Л  204: М  205: Н  206: О  207: П  208: Р  209: С  210: Т

211: У  212: Ф  213: Х  214: Ц  215: Ч  216: Ш  217: ■  218: Ъ  219: Ы  220: Ь

221: Э  222: Ю  223: Я  224: а  225: б  226: в  227: г  228: д  229: е  230: ж

231: з  232: и  233: й  234: к  235: л  236: м  237: н  238: о  239: п  240: р

241: с  242: т  243: у  244: ф  245: х  246: ц  247: ч  248: ш  249: щ  250: ъ

251: ы  252: ь  253: э  254: ю  255: я
```

Приклад 15. Послідовність визначають за таким рекурентним співвідношенням:

$$a_n = \begin{cases} 1, & n = 0; \\ n \cdot a_{n-1} + 2n - 3, & n > 0. \end{cases}$$

Дано m ($m > 5$). Вивести на консоль a_0, a_1, ..., a_{n-1}, де n – номер першого члена послідовності, для якого виконується умова $a_n > m$.

➢ *Попередні міркування*. Оскільки у рекурентному співвідношенні фігурують цілі числа, то члени послідовності також будуть цілими числами. Число m за цих обставин можна вважати натуральним числом (у програмі необхідно передбачити перевірку правильності введення `m` за допомогою циклу `do … while`). Кількість членів послідовності, які виведуть на консоль, наперед передбачити неможливо, отож масив використати не вдасться. Однак він тут і не потрібен: для обчислення членів послідовності достатньо однієї простої змінної *a*. Обчислення рекурентної формули задають такою інструкцією присвоєння: $a = n \cdot a + 2n - 3$.

На вході. Рядок `(Enter m>5)` забезпечує введення `m` (обмеження кількості членів послідовності). Він повторюватиметься доти, доки користувач вводитиме неправильні дані.

На виході (`a0=   a1=   …`). Послідовно одержуємо члени послідовності (по 4 в одному рядку).

Програма:

```
#include <iostream>
using namespace std;
void main () { int n, a, m, i=1;
  do {cout<< "Enter m>5: "; cin>>m;} while (m<=5);
  n=0; a=1;  // Значення a(0)
  do { cout<< "a"<<n<<"="<<a<<"  "; // Виведення a(n-1)
      // Виведення по 4 члени послідовності у рядку
      i++; if (i==5) {i=1; cout<<endl; }
      n++; a=n*a+2*n-3; // Значення a(n)
    } while (a<=m);
 cin>>i; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_15\Debug\Example_15.exe
Enter m>5: 5
Enter m>5: 250
a0=1  a1=0  a2=1  a3=6
a4=29  a5=152
```

◄

Приклад 16. Записати фрагмент програми обчислення n-го числа Фібоначчі. Ці числа визначають за рекурентною формулою:

`F(0)=0; F(1)=1; F(i)=F(i-1)+F(i-2)`, якщо `i ≥ 2`.

➢ *Попередні міркування*. Для обчислення чергового члена послідовності потрібні значення уже *двох* попередніх членів.

Фрагмент програми:

```
int a, b, u, i, n;   // a=F(i-2), b=F(i-1), u=F(i)
cout<<"Enter n >=0: ";
if(!n) u=0; if(n==1) u=1;
for(a=0, b=1, i=1; i<n; i++, u=a+b, a=b, b=u);
cout<<" F"<<n<<"="<<u<<endl;
…
```

◄

3.4.4. Обчислення скінченних сум і добутків

Під час обчислення сум зі скінченною чи нескінченною кількістю доданків перед початком циклу обов'язково необхідно присвоїти значення 0 (нуль) змінній, в якій накопичуватиметься сума цих доданків.

Аналогічно, під час обчислення добутків зі скінченною чи нескінченною кількістю множників перед початком циклу обов'язково необхідно присвоїти значення 1 (один) змінній, в якій накопичуватиметься добуток цих множників (див. приклади обчислення `n!`).

Приклад 17. Скласти програму обчислення $s=\sum_{i=1}^{n} i^2$.

➢ *Попередні міркування*. Приклад надзвичайно простий, однак демонструє чудову можливість С++: накопичення суми можна реалізувати тільки за допомогою *заголовку* циклу. Також варто звернути увагу на інструкцію виразу `s+=i*i++`, яка дає змогу накопичувати суму і водночас збільшувати лічильник циклу на 1.

На вході (`Enter n>0`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (кількості доданків). Ознакою завершення введення даних є рядок, в якому введене значення *не є додатним*.

На виході. Для кожного `n` одержуємо відповідну суму (`s1=  s2= …`).

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------------------
void main()
{ int i, s, n;  // Накопичення суми у змінній s
  while (cout<<"Enter n>0: ", cin>>n, n>0)
   {
    for (i=1, s=0; i<=n; s+=i*i++); // Накопичення суми
    cout<<" s"<<n<<"="<<s<<endl;
   }
  cin>>s; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_17\Debug\Example_17.exe
Enter n>0: 1
 s1=1
Enter n>0: 2
 s2=5
Enter n>0: 3
 s3=14
Enter n>0: 4
 s4=30
Enter n>0: 5
 s5=55
Enter n>0: 6
 s6=91
Enter n>0: -4
```

Приклад 18. Скласти програму обчислення $s = \sum_{i=1}^{n} \frac{i^2+2}{(i-3)(i-5)}$.

➢ *Попередні міркування.* Під час накопичення суми необхідно відкидати доданки, в яких знаменник дорівнюватиме нулю. Це доданки, які проявляються для $i = 3$ та $i = 5$. Здебільшого користувачі без особливих зусиль пропонують такий цикл для накопичення суми:

```
for (i=1, s=0; i<=n; i++)
   if (i!=3 && i!=5) s+=float(i*i+2)/(i*i-8*i+15);
```

Цей варіант є дуже простим і зрозумілим. Однак і в цьому прикладі для накопичення суми можна використати тільки заголовок циклу:

```
for(i=1, s=0; i<=n;  s=(i!=3 && i!=5)?
           s+float(i*i+2)/(i*i-8*i+15) : s, i++);
```

У програмі реалізовано обидва варіанти.

На вході (`Enter n>0`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (кількості доданків). Ознакою завершення введення даних є рядок, в якому введене значення `n` *не є додатним*.

На виході. Для кожного `n` одержуємо відповідну суму для першого (`s1=`) та другого варіанта (`s2=`) обчислення суми. За результатами тестування можна впевнитися, що значення цих сум співпадають (тобто є незалежними від способу організації циклу з накопичення суми).

Програма:

```
#include <iostream>
using namespace std;
//-----------------------------------------------------------
void main()
{ int i, n; float s;  // Накопичення суми у змінній s
  while (cout<<"Enter n>0: ", cin>>n, n>0)
   {for (i=1, s=0; i<=n; i++)
      if (i!=3 && i!=5) s+=float(i*i+2)/(i*i-8*i+15);
    cout<<" s1"<<"="<<s;
    for(i=1, s=0; i<=n; s=(i!=3 && i!=5)?
             s+float(i*i+2)/(i*i-8*i+15) : s, i++);
    cout<<"  s2"<<"="<<s<<endl;
   }
  cin>>s; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_18\Debug\Example_18.exe
Enter n>0: 2
 s1=2.375  s2=2.375
Enter n>0: 3
 s1=2.375  s2=2.375
Enter n>0: 4
 s1=-15.625  s2=-15.625
Enter n>0: 5
 s1=-15.625  s2=-15.625
Enter n>0: 45
 s1=66.5802  s2=66.5802
Enter n>0: 8
 s1=7.81667  s2=7.81667
Enter n>0: 98
 s1=126.43  s2=126.43
Enter n>0: -5
```

Приклад 19. Скласти фрагмент програми обчислення величини

$$z = \prod_{i=-3}^{8} \frac{(i+2)(i-5)}{(i+1)(i-3)}.$$

➢

```
int i, n; float z;  // Накопичення добутку у змінній z
for(i=-3, z=1; i<=8;  z=(i!=-1 && i!=3)?
      z*float(i*i-3*i-10)/(i*i-2*i-3) : z, i++);
```

Приклад 20. Скласти програму обчислення $s = \sum_{i=-2}^{n} \frac{i+1}{i} \prod_{k=1}^{i+3} \frac{k}{k+1}$.

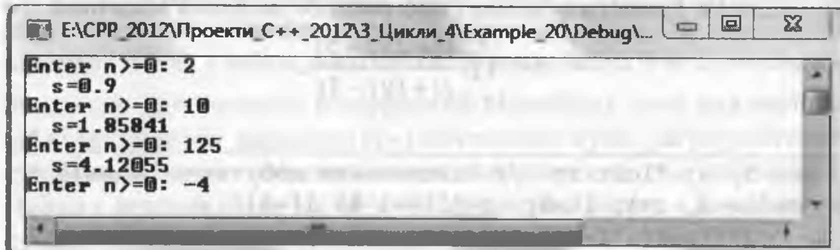
➢ *Попередні міркування*. Добуток $z = \prod_{k=1}^{i+3} \frac{k}{k+1}$ є множником доданка, обчисленого у внутрішньому циклі. Оскільки k цілого типу, то для отримання $\frac{k}{k+1}$ необхідно чисельник перетворити у float.

На вході (Enter n>0). Кожен такий рядок є окремим тестом, що задає значення змінної n (верхня межа індексу суми). Ознакою завершення тестування є рядок, в якому значення n *не є додатним*.

На виході. Для кожного n одержуємо відповідну суму (s=).

```
#include <iostream>
using namespace std;
//-------------------------------------------------
void main()
{ int i, k, n; float s, z; // Накопичення суми в s
                           // Накопичення добутку в z
  while (cout<<"Enter n>0: ", cin>>n, n>0)
  { for (i=-2, s=0; i<=n; i++)
      if (i!=0)
       { for (k=z=1; k<=i+3; z*= float(k)/(k+1), k++);
         s+= (i+1)*z/i;   }
    cout<<"  s"<<"="<<s<<endl;
  }
  cin>>s; // Пауза
}
```

Результати тестування програми:



3.4.5. Найпростіші рекурентні співвідношення

Розглянемо деяку послідовність, яка складається з n елементів (чи членів): $a_1, a_2, ..., a_n$ (це *скінченна* послідовність). Якщо ж кількість елементів послідовності є нескінченною, то її, відповідно, називають *нескінченною* ($a_1, a_2, ...$). Виходячи з конкретної предметної області, нумерацію елементів послідовності можуть починати і з *нуля* ($a_0, a_1, ..., a_n$ чи $a_0, a_1, ...$). Розглянемо тільки *скінченні* послідовності.

Якщо існує функція *f*, що $a_i = a(i) = f(i)$, то кажуть, що послідовність визначена за допомогою *загального члена*. Прикладами можуть слугувати арифметична та геометрична прогресії, в яких відомі формули загального члена.

Однак, зазвичай, визначити формулу загального члена послідовності дуже важко (якщо навіть у принципі це можливо). Найчастіше значення елемента послідовності визначають за допомогою інших елементів цієї ж послідовності (*рекурентно*).

Найпростішим рекурентним співвідношенням (чи рекурентним рівнянням) називають деяку функцію, яка визначає значення *поточного* (чергового) елемента послідовності через одне або декілька значень *попередніх* елементів цієї ж послідовності; причому відомими є значення відповідної кількості початкових елементів.

Нехай u_i позначає *поточний* елемент послідовності; тоді $u_{i-1}, u_{i-2}, ...$ позначатимуть *попередні* елементи цієї ж послідовності. Індекс i пробігає скінченну / нескінченну підмножину натуральних чисел (1; 2; …) чи невід'ємних цілих чисел (0; 1; 2; …).

Згідно з означенням, *рекурентне співвідношення*, яке визначатиме *поточне* значення елемента послідовності через значення *попереднього* елемента, загалом набуде вигляду:

$$u_i = \begin{cases} c = \text{const}, & i = 0; \\ g(u_{i-1}), & i = \overline{1; n}, \end{cases} \qquad u_i = \begin{cases} c = \text{const}, & i = 1; \\ g(u_{i-1}), & i = \overline{2; n}, \end{cases} \qquad (1)$$

де g – деяка функція.

Незважаючи на наявність індексів у формулах (1), заводити масив для зберігання усіх членів послідовності, зазвичай, не потрібно. Це стосується задач, в яких необхідно обчислити кінцеве значення послідовності u_n або використовувати поточні значення u_i тільки на одній ітерації циклу.

За цих умов для обчислення членів послідовності достатньо однієї простої змінної u. Обчислення рекурентних формул (1) задають інструкцією присвоєння `u=g(u)`. Змінна `u` лівої частини цієї інструкції позначає *поточний* елемент послідовності u_i, а змінна `u` правої частини – *попередній* елемент u_{i-1}. Формулу (1) у C++ можна реалізувати, використовуючи тільки заголовок циклу `for`:

```
for (u=c, i=0; i<n; i++, u=g(u));
```

або

```
for (u=c, i=1; i<n; i++, u=g(u));
```

Згідно з означенням, *рекурентне співвідношення*, яке визначатиме *поточне* значення елемента послідовності через значення двох *попередніх* елементів, загалом набуде вигляду:

$$u_i = \begin{cases} c1 = \text{const}, & i = 0; \\ c2 = \text{const}, & i = 1; \\ h(u_{i-2}, u_{i-1}), & i = \overline{2; n}, \end{cases} \qquad u_i = \begin{cases} c1 = \text{const}, & i = 1; \\ c2 = \text{const}, & i = 2; \\ h(u_{i-2}, u_{i-1}), & i = \overline{3; n}, \end{cases} \qquad (2)$$

де h – деяка функція. Аналогічно визначають *рекурентні співвідношення* і для більшої кількості початкових значень.

З метою реалізації формул (2) вводять дві допоміжні змінні: `a` – для зберігання значень u_{i-2} та `b` – для u_{i-1}:

```
if(n==0) u=c1; if(n==1) u=c2;
for(a=c1, b=c2, i=1; i<n; i++, u=h(a, b), a=b, b=u);
```

або

```
if(n==1) u=c1; if(n==2) u=c2;
for(a=c1, b=c2, i=2; i<n; i++, u=h(a, b), a=b, b=u);
```


Доволі часто формула (2) має такий вигляд:

$$u_i = \begin{cases} c = \text{const}, & i = \overline{0; 1}, \\ h(u_{i-2}, u_{i-1}), & i = \overline{2; n}, \end{cases} \qquad u_i = \begin{cases} c = \text{const}, & i = \overline{1; 2}, \\ h(u_{i-2}, u_{i-1}), & i = \overline{3; n}, \end{cases} \qquad (3)$$

де h – деяка функція.

З метою реалізації формул (3) вводять дві допоміжні змінні: `a` – для зберігання значень u_{i-2} та `b` – для u_{i-1}:

```
for(u=a=b=c, i=1; i<n; i++, u=h(a, b), a=b, b=u);
```

або

```
for(u=a=b=c, i=2; i<n; i++, u=h(a, b), a=b, b=u);
```

З рекурентними співвідношеннями ми мали справу у прикладах 15, 16 і під час розгляду конструкцій циклу (базовим прикладом слугувало обчислення $n!$). Якщо не брати до уваги $n = 0$, то обчислення чисел Фібоначчі визначає рекурентне співвідношення

$$u_i = \begin{cases} 1, & i = \overline{1; 2}, \\ u_{i-2} + u_{i-1}, & i = \overline{3; n}, \end{cases} \qquad (4)$$

яке можна реалізувати такою інструкцією циклу:

```
for (u=a=b=1, i=2; i<n; i++, u=a + b, a=b, b=u);
```

Оскільки $u_i = i! = i \cdot (i-)! = i \cdot u_{i-1}$; $0! = 1! = 1$, то рекурентне співвідношення визначення факторіала виглядатиме так:

$$u_i = \begin{cases} 1, & i = \overline{0; 1}, \\ i \cdot u_{i-1}, & i = \overline{2; n}. \end{cases} \qquad (5)$$

Співвідношення (5) можна реалізувати такою інструкцією:

```
for (i=u=1; i<n; u*=++i);
```

Співвідношення (4), (5) початково визначають через рекурентні співвідношення, однак у багатьох випадках рекурентна природа математичного об'єкту є *прихованою*.

Приклад 21. Скласти фрагмент програми обчислення величини $k = \sqrt{3 + \sqrt{3 + \ldots + \sqrt{3}}}$ (всього n коренів).

➢ *Попередні міркування*. Фактично змінна k відображає останній член такої числової послідовності:

$$a_1 = \sqrt{3}\,;\; a_2 = \sqrt{3+\sqrt{3}}\,;\; \ldots;\; k = a_n = \sqrt{3+\sqrt{3+\ldots+\sqrt{3}}}\,.$$

Для цієї послідовності маємо таке рекурентне співвідношення:

$$u_i = \begin{cases} 0, & i = 0; \\ \sqrt{u_{i-1}+3}, & i = \overline{1;n}. \end{cases}$$

Фрагмент програми:

```
int i, n; float u;
// Введення значення n (n>=1)
for(i=u=0; i<n; i++, u= sqrt(u+3));
…
```

◄

Як відомо, члени скінченної послідовності нумеруються від одиниці до n ($a_1,\ a_2, \ldots, a_n$) або з нуля до n ($a_0,\ a_1, \ldots, a_n$). Інколи послідовні члени відповідного рекурентного співвідношення *доцільно* нумерувати у зворотному порядку (наприклад: $u_n, \ldots, u_1$). Така нумерація допомагає "розгадати" рекурентне співвідношення (типовими прикладами слугують гіллясті дроби). Для реалізації цих співвідношень використовують цикли, параметри яких набувають послідовних *спадних* значень.

Приклад 22. Скласти фрагменти програм обчислення величин:

а) $t = \cos(2+\cos(4+\ldots+\cos(2(n-1)+\cos(2n))))$;

б) $$v = \cfrac{1}{2+\cfrac{1}{4+\cfrac{1}{\ddots\, 98+\cfrac{1}{100}}}};$$

в) $$w = \cfrac{x}{x^3+\cfrac{2}{x^3+\cfrac{4}{\ddots\, x^3+\cfrac{2^n}{x^3}}}}.$$

➢ а) *Попередні міркування*. Фактично змінна t відображає останній член такої числової послідовності:

$$a_1 = \cos(2n);\; a_2 = \cos(2(n-1)+\cos(2n));\; \ldots;\,.$$

$$t = a_n = \cos(2+\cos(4+\ldots+\cos(2(n-1)+\cos(2n))))\,.$$

Якщо ввести такі позначення:

$$a_1 = u_n = \cos(2n);\; a_2 = u_{n-1} = \cos(2(n-1)+u_n);\; \ldots;\; t = a_n = u_1,$$

то легко "розгадати" і записати таке рекурентне співвідношення:

$$u_i = \begin{cases} 0, & i = n+1; \\ \cos(2i+u_{i+1}), & i = \overline{n;1}. \end{cases}$$

Фрагмент програми:

```
int i, n; float u;
// Введення значення n (n>=1)
for(i=n+1, u=0; i>1; i--, u= cos(2*i+u));
…
```

б) *Попередні міркування*. Фактично змінна v відображає останній член такої числової послідовності:

$$a_1 = \frac{1}{100};\; a_2 = \cfrac{1}{98+\cfrac{1}{100}};\; a_3 = \cfrac{1}{96+\cfrac{1}{98+\cfrac{1}{100}}};\; \ldots;\; a_{50} = v\,.$$

Якщо ввести такі позначення:

$$a_1 = u_{100} = \frac{1}{100};\, a_2 = u_{98} = \frac{1}{98+u_{100}};\, a_3 = u_{96} = \frac{1}{96+u_{98}};\ldots;\, a_{50} = u_2\,,$$

то легко "розгадати" і записати таке рекурентне співвідношення:

$$u_i = \begin{cases} 0{,}01, \; i = 100; \\ 1/(i+u_{i+2}), \; i = 98, 96, \ldots, 4, 2. \end{cases}$$

Фрагмент програми:

```
int i, n; float u;
// Введення значення n
for(i=100, u=0.01; i>2; i-=2, u= 1(i+u));
…
```

в) *Попередні міркування*. Якщо $y = x^3$; $z = y + \cfrac{2}{y + \cfrac{4}{y + \cfrac{\dots}{y + \cfrac{2^n}{y}}}}$, то $w = \frac{x}{z}$.

Змінна z відображає, фактично, останній член такої числової послідовності:

$$a_1 = y + \frac{2^n}{y} = u_n;\ a_2 = y + \frac{2^{n-1}}{u_n} = u_{n-1};\ \dots;\ a_n = z = y + \frac{2^1}{u_2} = u_1.$$

Для зберігання степенів числа 2 використовуватимемо змінну m. Ця змінна відіграватиме також і роль параметра циклу.

Фрагмент програми:

```
int i, n, m; float x, y, u, w;
// Введення значень n (n>=1) та x
for(i=0, m=1; i<n; i++, m*=2); // m=2^n
y=x*x*x;
for(u=y; m>1; u= y+m/u, m/=2);
w=x/u;
…
```

◄

Доволі часто у практиці програмування виникає необхідність обчислення сум, в яких кожен доданок містить різноманітні комбінації факторіалів і степенів, наприклад:

$$s = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \dots + (-1)^{n+1} \cdot \frac{x^{2n-1}}{(2n-1)!}.$$

У цих сумах кожен черговий доданок є часткою двох потенційно *великих* чисел: для $x > 1$ степінь зростає дуже швидко, а факторіал взагалі є найшвидше зростаючою математичною функцією. Під час ділення дуже великих чисел можлива втрата точності, за якою частка не міститиме жодної точної цифри.

З метою ефективного обчислення сум такого вигляду використовують підхід, який базується на *рекурентній формулі* обчислення доданків.

З метою визначення рекурентної формули спочатку необхідно визначити *рекурентний множник*. Це дасть змогу позбавитися вкладеного циклу для обчислення факторіала та умови перевірки на парність / непарність показника степеня (n + 1).

Представимо нашу суму $s = \sum_{i=1}^{n} \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)!}$ у вигляді $s = \sum_{i=1}^{n} u_i$, де $u_i = \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)!}$. Рекурентний множник R – це співвідношення двох послідовних членів суми, тобто: $u_2 = R \cdot u_1$; $u_3 = R \cdot u_2$; $\dots u_n = R \cdot u_{n-1}$. Отже, $R = u_i / u_{i-1}$.

У нашому випадку матимемо таке:

$$R = \frac{(-1)^{i+1} x^{2i-1}}{(2i-1)!} : \frac{(-1)^i x^{2i-3}}{(2i-3)!} = -\frac{x^2}{(2i-1)(2i-2)}.$$

Отже, одержали таке рекурентне співвідношення:

$$u_i = \begin{cases} x, \ i = 1; \\ -\dfrac{x^2}{2 \cdot (2i-1)(i-1)} \cdot u_{i-1}, \ i = \overline{2;\ n}. \end{cases} \qquad (6)$$

Фрагмент програми, який реалізує співвідношення (6):

```
int i, n; float x, y, u, s;
// Введення значень n (n>1) та x
y=-x*x/2; // Постійне значення – обчислено поза циклом!
for(i=1, s=u=x; i<n; i++, u*=y/(2*i*i-3*i+1), s+=u);
…
```

3.5. Обчислення із заданою точністю

3.5.1. Обчислення числових рядів

Розглянемо числову послідовність, задану загальним членом $\{a_i\}$ $(i = 1;2;...)$, і формально із її елементів утворимо суму вигляду:

$$a_1 + a_2 + ... + a_i + ... = \sum_{i=1}^{\infty} a_i \; . \qquad (7)$$

Суму (7) називають *числовим рядом*, або просто *рядом*. Числа $a_1, a_2, ..., a_i, ...$ називають *елементами*, або *членами* ряду.

Суми перших елементів ряду

$$S_1 = a_1,\ S_2 = a_1 + a_2,\ ...,\ S_n = a_1 + a_2 + ... + a_n = \sum_{i=1}^{n} a_i \qquad (8)$$

називають *частковими сумами* ряду. Ці суми $S_1, S_2, ..., S_n, ...$ утворюють числову послідовність $\{S_i\}$ $(i = 1;2;...)$, границю якої (скінченну чи нескінченну) називають *сумою* ряду S ($S = \lim\limits_{i \to \infty} S_i$).

Формально $S = a_1 + a_2 + ... + a_i + ... = \sum\limits_{i=1}^{\infty} a_i$. Якщо границя S – скінченна, то ряд (7) називають *збіжним*, а інакше – *розбіжним*.

Теорема 1 (*необхідна умова збіжності*). Якщо ряд (7) збіжний, то його загальний елемент прямує до нуля (тобто $\lim\limits_{i \to \infty} a_i = 0$).

З теореми 1 випливає таке:

- якщо $\lim\limits_{i \to \infty} a_i \neq 0$, то ряд (7) є розбіжним;
- якщо $\lim\limits_{i \to \infty} a_i = 0$, то ряд (7) може бути збіжним, або розбіжним.

Наприклад, ряд $1 + \frac{1}{2} + \frac{1}{3} ... + \frac{1}{n} + ...$ (його називають *гармонічним*), є розбіжним рядом, хоча його доданки прямують до нуля.

Отже, в постановці задачі на обчислення рядів можуть фігурувати як збіжні, так і розбіжні ряди.

У випадку *збіжного* ряду, зазвичай, необхідно обчислити суму ряду S із заданою точністю ε $(\varepsilon > 0)$. Вважають, що необхідна точність досягнута, якщо черговий член виявиться *за модулем* меншим, ніж ε, – цей член і всі наступні можна не враховувати. Цей підхід базується на теоремі 1, однак ряд повинен при цьому апріорі бути збіжним!

У випадку *коректної* постановки задачі (заданий в умові задачі ряд – збіжний) – *все буде добре* (хоча процес обчислення може тривати *дуже і дуже довго*)! Однак є багато чинників, які призводять до некоректної постановки задачі: ряд, який автор задачі апріорі вважає збіжним, чомусь розбігається – виникає зациклення і т. д. Здебільшого чинники базуються на тому, що автор задачі глибоко не вникав у теорію рядів, чи, можливо, і не підозрює, що така теорія існує взагалі. Деякі чинники спричинені елементарними описками тощо. Як бути? Можна запропонувати таке:

1) вводити величину максимальної кількості ітерацій, яка слугуватиме індикатором для аварійного виходу з циклу;
2) дослідити ряд з математичної точки зору (це важко, однак сприяє інтелектуальному розвиткові особистості), використовуючи *достатні умови* збіжності рядів;
3) використати обидва перелічені підходи.

Достатніх умов збіжності рядів є дуже багато. Нагадаємо деякі з них.

Теорема 2 (*ознака Д'Аламбера*). Нехай ряд (7) містить тільки *додатні* елементи та існує границя $\lim\limits_{i \to \infty} \frac{a_{i+1}}{a_i} = l$. Якщо $l < 1$, то ряд (7) є збіжним, і розбіжним у випадку $l > 1$.

Увага! Якщо $l = 1$, то ряд може бути збіжним або розбіжним, отож необхідно використовувати інші ознаки збіжності.

Теорема 3 (*ознака Коші*). Нехай ряд (7) містить тільки *додатні* елементи та існує границя $\lim\limits_{i \to \infty} \sqrt[i]{a_i} = L$. Якщо $L < 1$, то ряд (7) є збіжним, і розбіжним у випадку $L > 1$.

Увага! Якщо $L = 1$, то ряд може бути збіжним або розбіжним, отож необхідно використовувати інші ознаки збіжності.

Приклад 23. Скласти фрагмент програми обчислення суми ряду $S=\sum_{i=1}^{\infty}\frac{1}{2^i+7^i}$ із заданою точністю $\varepsilon\ (\varepsilon>0)$.

➢ *Попередні міркування.* Використавши ознаку Д'Аламбера, отримаємо $\lim_{i\to\infty}\frac{2^i+7^i}{2^{i+1}+7^{i+1}}=\lim_{i\to\infty}\frac{(2/7)^i+1}{2\cdot(2/7)^i+7}=\frac{1}{7}<1$ (ряд збіжний).

Для накопичення степенів 2^i використовуватимемо змінну *a*, а для степенів 7^i – змінну *b* (для $i=1$: $a=2, b=7$).

Фрагмент програми (варіант 1 – використання while):

```
double a, b, u, s, eps= 1.e-6;
s=0, a=2, b=7, u= 1.0/9; // Початкові значення
while (u>eps) { s+=u, a*=2, b*=7, u= 1.0/(a+b); };
…
```

Фрагмент програми (варіант 2 – використання for):

```
double a, b, u, s, eps= 1.e-6;
for(s=0, a=2, b=7, u= 1.0/9; u>eps;
              s+=u, a*=2, b*=7, u=1.0/(a+b));
…
```

Ряд (7) з довільними членами (як додатними, так і від'ємними), називають *абсолютно збіжним*, якщо збігається ряд $\sum_{i=1}^{\infty}|a_i|$.

Якщо (7) – абсолютно збіжний, то він водночас є й збіжним.

Розглянемо *знакопереміжні* ряди:

$$b_1-b_2+\ldots+(-1)^{i+1}b_i+\ldots=\sum_{i=1}^{\infty}(-1)^{i+1}b_i\,, \qquad (9)$$

де $\forall i\in N: b_i>0$.

Теорема 4 (*ознака Лейбніца*). Якщо елементи (9) монотонно спадають за абсолютною величиною і $\lim_{i\to\infty}b_i=0$, то ряд (9) є збіжним.

Приклад 24. Скласти програму обчислення суми *знакопереміжного* ряду $S=\sum_{i=1}^{\infty}\frac{(-1)^i\cdot\sqrt{i}}{i+100}$ із заданою точністю $\varepsilon\ (\varepsilon>0)$.

➢ *Попередні міркування.* Використавши ознаку Лейбніца, легко переконатися, що заданий в умові задачі ряд збіжний. *Рекурентне* співвідношення тут заводити не потрібно. Для відображення знака доданка заведемо змінну k.

На вході (Enter eps>0). Кожен такий рядок є окремим тестом, що задає точність обчислення суми ряду. Ознакою завершення введення даних є рядок, в якому введене значення *не є додатним*.

На виході. Для кожного eps одержуємо відповідне значення суми ряду (s) та значення кількості ітерацій (i=), які були затрачені.

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//-----------------------------------------------------
void main() { double u, s, eps; int i, k;
 while (cout<<"Enter eps>0: ", cin>>eps, eps>0)
 { for (s=0, i=1, k=-1, u=-1./101; fabs(u)>eps;
   s+=u, i++, k*=-1, u=k* sqrt(double(i))/(i+100));
   cout<<"  s="<<s<<"  i="<<i-1<<endl;
 }
 cin>>s; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_24\Debug\...
Enter eps>0: 0.1
  s=0  i=0
Enter eps>0: 0.01
  s=0  i=0
Enter eps>0: 0.001
  s=-0.00428909  i=999799
Enter eps>0: 0.0001
  s=-0.00383909  i=99999799
Enter eps>0: 0.00001
  s=-0.00379988  i=2147483647
Enter eps>0:
```

Додатковий коментар. Проаналізувавши результати тестування, можна зробити висновок, що хоча заданий в умові задачі ряд збігається, однак ця збіжність надзвичайно *повільна*. Наприклад, щоб досягнути точності $\varepsilon = 10^{-4}$, необхідно здійснити 99 999 799 ітерацій, а за точності $\varepsilon = 10^{-5}$ досягаємо верхньої межі (значення 2 147 483 647) розрядної сітки для цілої змінної `i` (тип `int`).

З метою уникнення переповнень розрядної сітки визначимо змінну `i` за допомогою типу `double`, тобто у програмі зробимо таку заміну:

```
double u, s, eps, i;
```

Хоча ми отримали суму ряду з необхідною точністю, однак затратили при цьому "астрономічну" кількість ітерацій; не кажучи про час, протягом якого це все обчислювали (пропонуємо на базі цієї простої програми перевірити потужність власного комп'ютера). ≺

Попередній приклад добре ілюструє положення про те, що під час обчислення рядів необхідно бути дуже уважним, дружити з теорією, всебічно аналізувати результати, багато експериментувати тощо.

Розглянемо постановку задач на *розбіжні* ряди з додатними членами. Оскільки часткові суми цих рядів весь час збільшуються, то в умові задачі, зазвичай, пропонують здійснювати обчислення доти, доки значення поточної суми ряду не перевищуватиме деякого фіксованого наперед значення.

Приклад 25. Нехай маємо ряд $\sum_{i=1}^{\infty} i \cdot 2^i$. Скласти програму, яка визначатиме *найменшу* кількість початкових членів ряду, за яких відповідна часткова сума ряду уперше перевищить число $E\ (E > 0)$.

➢ *Попередні міркування.* Використавши ознаку Д'Аламбера, отримаємо $\lim\limits_{i \to \infty} \dfrac{(i+1) \cdot 2^{i+1}}{i \cdot 2^i} = 2 \cdot \lim\limits_{i \to \infty} \dfrac{i+1}{i} = 2 > 1$ (ряд розбіжний).

Рекурентне співвідношення заведемо тільки для обчислення 2^i.

На вході (`Enter E>0`). Кожен такий рядок є окремим тестом, що задає число $E\ (E > 0)$, яке обмежує зверху суму ряду. Ознакою завершення тестування є рядок, в якому значення *не є додатним*.

На виході. Для кожного `E` одержуємо значення часткової суми ряду (`s=`), яка уперше перевищить `E`, і значення кількості доданків цієї суми (`i=`).

Програма:

```
#include <iostream>
#include <cmath>
uslng namespace std;
//------------------------------------------------------
void main()
{ double u, s, E, i;
  while (cout<<"Enter E>0: ", cin>>E, E>0)
  {
    for(s=u=2, i=1;          //Ініціалізація параметрів
            fabs(s)<=E;      //Перевірка умови
                 i++, u*=2, s+=u*i); // Алгоритм
    cout<<"  s="<<s<<"  i="<<i<<endl;
  }
  cin>>s; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_25\Debug\Example_25.exe
Enter E>0: 1
  s=2  i=1
Enter E>0: 3
  s=10  i=2
Enter E>0: 11
  s=34  i=3
Enter E>0: 36
  s=98  i=4
Enter E>0: 900
  s=1538  i=7
Enter E>0: 3000
  s=3586  i=8
Enter E>0: -4
```

≺

3.5.2. *Обчислення степеневих рядів*

Ряд

$$f_1(x)+f_2(x)+\ldots+f_i(x)+\ldots=\sum_{i=1}^{\infty}f_i(x), \quad (10)$$

в якому членами ряду є функції $f_i(x)$ від аргументу x, називають *функціональним* рядом. При $x=x_0$ ряд (10) перетворюється на числовий ряд:

$$f_1(x_0)+f_2(x_0)+\ldots+f_i(x_0)+\ldots=\sum_{i=1}^{\infty}f_i(x_0). \quad (11)$$

Якщо числовий ряд (11) збігається (розбігається), то кажуть, що при $x=x_0$ збігається (розбігається) функціональний ряд (10).

Усі значення аргументу x, за яких ряд (10) збіжний, називають *областю збіжності* функціонального ряду. В області збіжності існує границя часткових сум функціонального ряду

$$\lim_{n\to\infty}S_n(x)=S(x),$$

де функції $S_n(x)=\sum_{i=1}^{n}f_i(x)$ – часткові суми, а $S(x)$ – сума ряду.

Ряд $r_n(x)=f_{n+1}(x)+f_{n+2}(x)+\ldots$ називають *залишком* функціонального ряду (10). В області збіжності функціонального ряду виконується формула

$$S(x)=S_n(x)+r_n(x),$$

де $\lim\limits_{n\to\infty}r_n(x)=0$.

Функціональний ряд

$$a_0+a_1x+a_2x^2+\ldots+a_ix^i+\ldots=\sum_{i=0}^{\infty}a_ix^i \quad (12)$$

називають *степеневим* рядом, де $f_i(x)=a_ix^i$ – загальний член, числа $a_0, a_1, a_2, \ldots, a_i, \ldots$ – коефіцієнти степеневого ряду.

Розглядають і узагальнений степеневий ряд

$$a_0+a_1(x-c)+a_2(x-c)^2+\ldots+a_i(x-c)^i+\ldots=\sum_{i=1}^{\infty}a_i(x-c)^i. \quad (13)$$

Якщо у (13) візьмемо $x-c=y$, то одержимо ряд типу (12), отож властивості ряду (12) неважко перефразувати і для ряду (13).

Теорема 5 (*ознака Абеля*). Якщо степеневий ряд (12):

1) збігається при $x=x_0$, то він абсолютно збігається для будь-якого x, що задовольняє нерівність $|x|<|x_0|$;

2) якщо ряд (12) розбігається при $x=x_1$, то він розбігається за всіх значень x, що задовольняють нерівність $|x|>|x_1|$.

Інтервалом збіжності степеневого ряду (12) називають інтервал $|x|<R$, в усіх внутрішніх точках якого ряд збігається абсолютно, а для всіх точок $|x|>R$ ряд є розбіжним; при цьому число $R>0$ називають *радіусом збіжності* степеневого ряду.

Для узагальненого степеневого ряду (13) інтервал збіжності $(c-R;\ c+R)$ має центр симетрії у точці $x=c$.

Увага! На кінцях інтервалу збіжності (тобто в точках $x=\pm R$) ряд (12) може як збігатись, так і розбігатись. Це питання потребує спеціального дослідження у кожному випадку. Аналогічно і для (13): нічого не можна стверджувати про збіжність у точках $x=c\pm R$.

Для визначення радіуса збіжності степеневого ряду можна скористатися ознакою Д'Аламбера, тоді

$$R=\frac{1}{l}=\lim_{i\to\infty}\left|\frac{a_i}{a_{i+1}}\right|, \quad (14)$$

або ознакою Коші, тоді

$$R=\lim_{i\to\infty}\frac{1}{\sqrt[i]{|a_i|}}. \quad (15)$$

Формули, що подають функцію $f(x)$ у вигляді степеневих рядів, мають вигляд:

$$f(x)=f(0)+\frac{f'(0)}{1!}x+\frac{f''(0)}{2!}x^2+\ldots+\frac{f^{(i)}(0)}{i!}x^i+\ldots \qquad (16)$$

або

$$f(x)=f(c)+\frac{f'(c)}{1!}(x-c)+\frac{f''(c)}{2!}(x-c)^2+\ldots+\frac{f^{(i)}(c)(x-c)^i}{i!}+\ldots \qquad (17)$$

Ряд у правій частині формули (16) називають рядом Маклорена, а у формули (17) – рядом Тейлора. Кажуть, що ряд Маклорена дає розвинення функції у ряд поблизу точки $x = 0$, а ряд Тейлора – поблизу точки $x = c$. Чим ближче x до точки розвинення функції $f(x)$ у ряд, тим меншою кількістю членів ряду буде досягнуто більшої точності під час обчислення $f(x)$.

Наведемо розклад у ряд Маклорена деяких відомих функцій:

$$y=e^x=\sum_{i=0}^{\infty}\frac{x^i}{i!};\ |x|<\infty,$$

$$y=\cos x=\sum_{i=0}^{\infty}\frac{(-1)^i\cdot x^{2i}}{(2i)!};\ |x|<\infty,$$

$$y=\sin x=\sum_{i=0}^{\infty}\frac{(-1)^i\cdot x^{2i+1}}{(2i+1)!};\ |x|<\infty.$$

$$y=\ln(1-x)=-\sum_{i=0}^{\infty}\frac{x^i}{i};\ -1\le x<1.$$

$$y=\ln\frac{x+1}{1-x}=2\sum_{i=0}^{\infty}\frac{x^{2i+1}}{2i+1};\ |x|<1.$$

$$y=\mathrm{arctg}\,x=\frac{\pi}{2}+\sum_{i=0}^{\infty}\frac{(-1)^{i+1}}{(2i+1)\cdot x^{2i+1}};|x|>1.$$

$$y=\mathrm{arcctg}\,x=\frac{\pi}{2}+\sum_{i=0}^{\infty}\frac{(-1)^{i+1}\cdot x^{2i+1}}{2i+1};\ |x|\le 1.$$

Для розвинення функції у степеневий ряд, зазвичай, використовують *рекурентні співвідношення*, техніку виявлення яких продемонстровано у пункті 3.4.5 під час обчислення скінченної суми $s=\sum_{i=1}^{n}\frac{(-1)^{i+1}x^{2i-1}}{(2i-1)!}$ (див. формулу 6).

Приклад 26. Написати програму обчислення функції $\mathrm{Ch}\,x$ (косинус гіперболічний) з точністю $\varepsilon>0$ за допомогою розкладу в степеневий ряд за формулою: $y=1+\frac{x^2}{2!}+\frac{x^4}{4!}+\ldots+\frac{x^{2i}}{(2i)!}+\ldots$

➢ *Попередні міркування*. Заданий в умові задачі, ряд є збіжним при $|x|<\infty$. Для обчислення ряду скористаємося рекурентною формулою $u_i=u_{i-1}\cdot R$, де R – рекурентний множник:

$$R=\frac{u_i}{u_{i-1}}=\frac{x^{2i}}{(2i)!}\cdot\frac{(2i-2)!}{x^{2i-2}}=\frac{x^2}{2i(2i-1)}.$$

Отже, одержали таке рекурентне співвідношення:

$$u_i=\begin{cases}1,\ i=0;\\ \dfrac{x^2}{2\cdot i\cdot(2i-1)}\cdot u_{i-1},\ i=\overline{1;\,n}.\end{cases} \qquad (18)$$

Формулу (18) у програмі реалізує функція ch_my(x). Значення цієї функції під час тестування порівнюватимемо зі значеннями функції cosh(x) стандартної бібліотеки.

Тестування здійснюватимемо через табулювання обидвох функцій (ch_my, cosh) на проміжку $[a;\,b]$ з кроком h. У програмі необхідно передбачити перевірку виконання умов $a<b$ та $h>0$.

Цикл перебирання аргументів у головній функції `main` є надзвичайно простим. Значна частина `main` "відповідає" за виведення результатів у табличній формі.

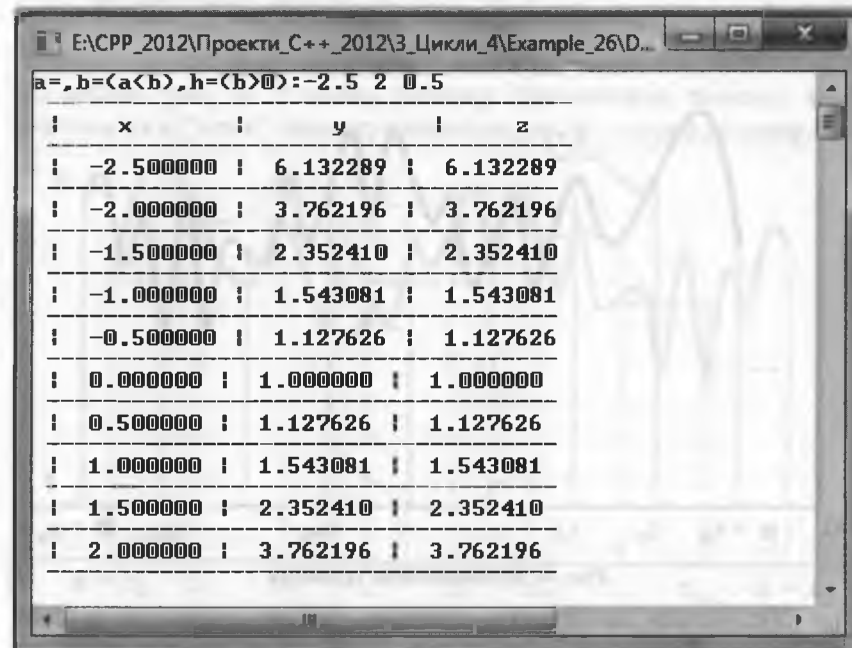
На вході (a=,b=(a<b),h=(h>0)). Цей рядок забезпечує уведення меж проміжку $[a; b]$ і кроку h. Оскільки користувач попереджений явно про обмеження на ці величини, то за порушення обмежень програма автоматично їх виправлятиме.

На виході. У табличній формі одержуємо значення аргументів (стовпець x) і відповідних щодо них значень функцій *ch_my* (стовпець y) і *cosh* (стовпець z).

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//-----------------------------------------------------------
double ch_my(double x)
{const double eps=0.0000001; // Точність обчислень
 int i; double y, u, s;
 y=x*x/2; // Постійне значення - обчислено поза циклом!
 for(i=0, s=u=1; fabs(u)>eps; i++, u*=y/(2*i*i-i), s+=u);
 return s;
}
void main () { double x,y,z,a,b,h;
 cout<<"a=,b=(a<b),h=(h>0):"; cin>>a>>b>>h;
 if(a>b) { double r=a; a=b; b=r; } // Забезпечення a<b
 if(h<0) h=-h;                     // Забезпечення h>0
 cout<<" -------------------------------------"<<endl;
 cout<<" |     x      |      y      |     z    "<<endl;
 cout<<" -------------------------------------"<<endl;
 for(x=a; x<=b; x+=h)
   { y=ch_my(x); z=cosh(x);
     cout<< " | " << fixed << x << " | ";
     cout<< fixed << y << " | ";
     cout<< fixed << z << endl;
     cout<<"-------------------------------------\n";
   }  cin>> x; // Пауза
}
```


Результати тестування програми:


E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_26\D...

a=,b=(a<b),h=(b>0):-2.5 2 0.5

x	y	z
-2.500000	6.132289	6.132289
-2.000000	3.762196	3.762196
-1.500000	2.352410	2.352410
-1.000000	1.543081	1.543081
-0.500000	1.127626	1.127626
0.000000	1.000000	1.000000
0.500000	1.127626	1.127626
1.000000	1.543081	1.543081
1.500000	2.352410	2.352410
2.000000	3.762196	3.762196

3.5.3. Обчислення визначених інтегралів

У багатьох наукових і практичних застосуваннях виникає необхідність обчислення визначеного інтеграла $\int_a^b f(x)dx$. Якщо для неперервної на проміжку $[a;b]$ підінтегральної функції $f(x)$ можна знайти первісну функцію $F(x)$, то визначений інтеграл легко обчислити за формулою Ньютона–Лейбніца

$$\int_a^b f(x)dx = F(b) - F(a).$$

Однак у багатьох випадках не вдається знайти первісну функцію або вона є надзвичайно складною для обчислення. У цих випадках застосовують методи числового інтегрування, які базуються

на тому, що значення визначеного інтеграла дорівнює площі криволінійної трапеції, розміщеної поміж лінією графіка функції та віссю абсцис (рис. 4).

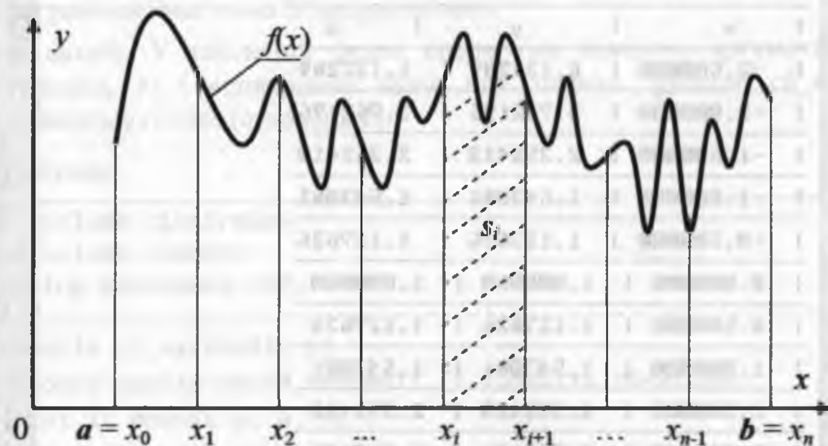


Рис. 4. Криволінійна трапеція

Усі методи на початку, зазвичай, роблять одне і те ж саме: розбивають проміжок інтегрування $[a;b]$ на скінченну кількість *рівних* відрізків (нехай на n відрізків) та замінюють площу S криволінійної трапеції на суму площ малих криволінійних трапецій s_i, тобто $S = \sum_{i=0}^{n-1} s_i$, причому на кожному проміжку $[x_i; x_{i+1}], i = \overline{0; n-1}$ функцію $f(x)$ замінюють наближеною функцією $f_i(x), i = \overline{0; n-1}$.

Отже, визначений інтеграл обчислюють наближено з певною похибкою $R(x)$:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} s_i + R(x). \quad (19)$$

Залежно від способу задання $f_i(x)$, існують різні методи обчислення визначених інтегралів. Розглянемо найвідоміші методи числового інтегрування: *прямокутників*, *трапецій* та *Сімпсона*.

За методом *прямокутників* $f(x)$ на кожному проміжку $h = x_{i+1} - x_i$ ($i = \overline{0; n-1}$) замінюють прямою лінією, паралельною осі абсцис (рис. 5). У цьому випадку криволінійна трапеція заміняється на n "*лівих*" прямокутників (на рис. 2 – штрихові лінії).

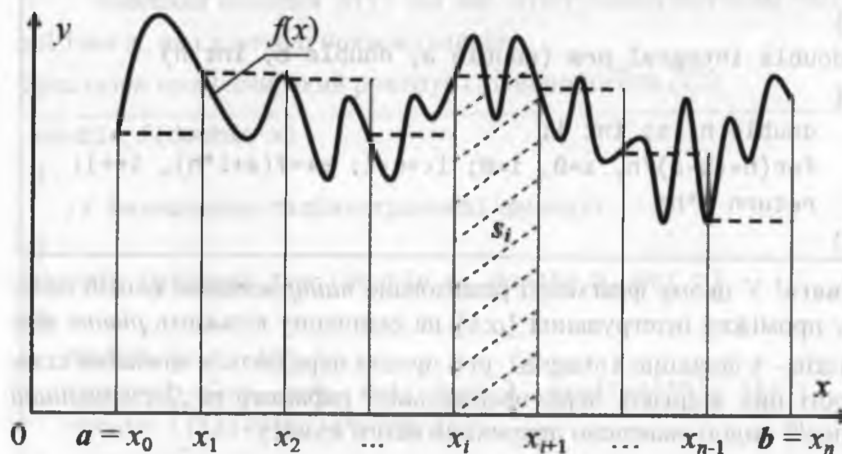


Рис. 5. Заміна криволінійної трапеції "*лівими*" прямокутниками

Ширина кожного "*лівого*" (надалі опускатимемо) прямокутника $h = x_{i+1} - x_i$, а довжина $f(x_i)$, де $i = \overline{0; n-1}$. Площа усіх прямокутників (а, отже, і наближене значення визначеного інтеграла):

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} s_i = \sum_{i=0}^{n-1} (x_{i+1} - x_i) f(x_i) = h \cdot \sum_{i=0}^{n-1} f(x_i). \quad (20)$$

Значення похибки $R(x)$ під час інтегрування методом прямокутників визначаємо за формулою $R(x) = \frac{h^2(b-a)}{12} \cdot \max_{t\in[a;\, b]} |f''(t)|$.

Щоби похибка не перевищувала задану точність обчислень ε $(\varepsilon > 0)$, крок інтегрування необхідно обирати з умови

$$h \le \sqrt{\frac{12 \cdot \varepsilon}{(b-a) \cdot \max_{t\in[a;\, b]} |f''(t)|}}. \quad (21)$$

Фрагмент програми, який реалізує співвідношення (20):

```
double f(double x)
{
  // Визначення підінтегральної функції
}
double integral_prm (double a, double b, int n)
{
  double h, s; int i;
  for(h=(b-a)/n, s=0, i=0; i<=n-1; s+=f(a+i*h), i++);
  return s*h;
}
```

Увага! У цьому фрагменті реалізовано *найпростіший* спосіб поділу проміжку інтегрування $[a;b]$ на скінченну кількість *рівних* відрізків – у функцію `integral_prm` просто передається значення кількості цих відрізків через формальний параметр `n`. *Досконаліший* спосіб поділу наведено наприкінці цього пункту.

За методом *трапецій* функцію $f(x)$ на кожному проміжку $h = x_{i+1} - x_i$ ($i = \overline{0;\, n-1}$) замінюють прямою лінією, яка проходить через точки $(x_i;\, f(x_i))$ і $(x_{i+1};\, f(x_{i+1}))$. На кожному проміжку $h = x_{i+1} - x_i$ утворюється трапеція площею:

$$s_i = \frac{f(x_{i+1}) + f(x_i)}{2} \cdot (x_{i+1} - x_i) = h \cdot \frac{f(x_{i+1}) + f(x_i)}{2}.$$

Загальна площа усіх трапецій (а отже і наближене значення визначеного інтеграла):

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} s_i = \frac{h}{2} \cdot \sum_{i=0}^{n-1} \big(f(x_i) + f(x_{i+1})\big) =$$

$$= \frac{h}{2} \cdot \left(f(x_0) + f(x_n) + 2 \cdot \sum_{i=1}^{n-1} \big(f(x_i)\big) \right).$$

Остаточний вигляд формули інтегрування методом трапецій:

$$\int_a^b f(x)dx \approx \frac{h}{2} \cdot \left(f(a) + f(b) + 2 \cdot \sum_{i=1}^{n-1} f(x_i) \right). \qquad (22)$$

Значення похибки $R(x)$ під час інтегрування методом трапецій таке ж, як і у методі прямокутників.

Фрагмент програми, який реалізує співвідношення (22):

```
double f(double x)
{
  // Визначення підінтегральної функції
}
double integral_trp (double a, double b, int n)
{
  double h, s; int i;
  for(h=(b-a)/n, s=0, i=1; i<=n-1; s+=f(a+i*h), i++);
  return (f(a)+f(b)+2*s)*h/2;
}
```

З метою наближеного обчислення визначеного інтеграла за методом Сімпсона підінтегральну функцію $f(x)$ замінюють на відрізки квадратичних парабол, проведених через кінці кожних трьох сусідніх ординат значень функції: $f(x_0)$, $f(x_1)$, $f(x_2)$; $f(x_1)$, $f(x_2)$, $f(x_3)$; ..., $f(x_{n-2})$, $f(x_{n-1})$, $f(x_n)$. При цьому весь проміжок інтегрування $[a;b]$ розбивають на парну кількість n відрізків $[x_i - h;\, x_i + h]$.

Наведемо без доведення остаточний вигляд формули інтегрування методом Сімпсона:

$$S = \frac{h}{3} \cdot \left(f(a) + f(b) + 4 \cdot \sum_{i=1}^{n-1} f(x_i) + 2 \cdot \sum_{j=2}^{n-2} f(x_j) \right), \qquad (23)$$

де i – непарні цілі; j – парні цілі.

Значення похибки $R(x)$ під час інтегрування методом Сімпсона визначаємо за формулою $R(x) = \frac{h^4(b-a)}{180} \cdot \max_{t \in [a;\, b]} \left| f^{(4)}(t) \right|$.

Щоби похибка не перевищувала задану точність обчислень $\varepsilon\ (\varepsilon > 0)$, крок інтегрування необхідно обирати з умови

$$h \le \sqrt[4]{\frac{180 \cdot \varepsilon}{(b-a) \cdot \max\limits_{t \in [a;\, b]} \left| f^{(4)}(t) \right|}}. \quad (24)$$

Фрагмент програми, який реалізує співвідношення (23):

```
double f(double x)
{    /* Визначення підінтегральної функції */ }
double integral_Smp (double a, double b, int n)
{
  double h, y, s1, s2; int i;
  for(h=(b-a)/n, s1=s2=0, i=1; i<=n-1;
     y=f(a+i*h), (i%2)? s1+=y : s2+=y, i++);
  return (f(a)+f(b)+4*s1+2*s2)*b/3;
}
```

Отже, в усіх фрагментах коду С++, які реалізують формули числового інтегрування, використано *найпростіший* спосіб поділу проміжку інтегрування $[a;b]$ на скінченну кількість *рівних* відрізків – у відповідну функцію передається значення кількості цих відрізків через формальний параметр n. Очевидно, що такий підхід не витримує жодної критики.

Формули (21), (24) дають змогу автоматизувати процес визначення величини h на базі заданої наперед точності обчислень $\varepsilon\ (\varepsilon > 0)$. Однак це не зовсім проста задача, оскільки необхідно також визначати $\max\limits_{t \in [a;\, b]} \left| f''(t) \right|$ чи $\max\limits_{t \in [a;\, b]} \left| f^{(4)}(t) \right|$.

"Витонченіші" та професійні реалізації формул числового інтегрування базуються на *принципі Рунге*, за яким точність обчислень $\varepsilon\ (\varepsilon > 0)$ вважають досягнутою, якщо виконується умова

$$\left| S_{2n} - S_n \right| < \varepsilon, \quad (25)$$

де S_n – наближене значення визначеного інтеграла під час розбиття проміжку інтегрування $[a;b]$ на n рівних відрізків, а S_{2n} – на $2 \cdot n$ рівних відрізків. Умова (25) задає "грубу" (чи узагальнену) оцінку точності, яку для кожного методу уточнюють (про це ітиме мова згодом).

Отже, незалежно від методу інтегрування, з метою досягнення певної точності задають деяке *початкове* значення n, обчислюють і порівнюють S_n і S_{2n}. За принципом Рунге точності ε досягнуто, якщо виконується (25). У цьому випадку обчислення *зупиняють*. Якщо ж умова (25) не виконується, то обчислюють S_{4n} і порівнюють його з S_{2n} і т. д.

Виконаємо певні дослідження. Розглянемо метод *прямокутників* (подібні дослідження без проблем можна здійснити і для інших методів числового інтегрування). Нехай $[0; 10]$ – проміжок інтегрування, а $n = 10$. Тоді для визначення S_{10} буде обчислено значення підінтегральної функції $f(x)$ у точках 0; 1; 2; ...; 9, а для визначення S_{20} – у точках 0, 0,5; 1; 1,5; ...; 9; 9,5.

Отже, друга множина точок (для S_{20}) містить усі точки першої (для S_{10}). Ця ж ситуація повторюватиметься під час кожного наступного *подвоєння* кількості точок x_i: щоразу тільки половина точок буде новою, і тільки для них потрібно обчислювати $f(x_i)$, а для "старої" половини можна використати значення, обчислені на попередній ітерації.

Загалом для визначення S_n (n – *початкове* значення кількості рівних відрізків проміжку $[a;b]$) необхідно обчислити

$f(x)$ у точках множини $B_0 = \{a + i \cdot h \mid i = \overline{0; n-1}\}$, а для S_{2n} – у точках множини $B_1 = B_0 \cup \{a + \frac{h}{2} + i \cdot h \mid i = \overline{0; n-1}\}$.

Якщо продовжити наші дослідження, то для визначення S_{40} необхідно обчислити значення $f(x)$ у точках 0; 0,25; 0,5; 0,75; 1; 1,25; 1,5; 1,75; ...; 9; 9,25; 9,5; 9,75. Тут необхідно обчислити 20 нових значень $f(x)$ у точках: 0,25; 0,75; 1,25; 1,75; ...; 9,25; 9,75.

Отже, для визначення S_{4n} необхідно обчислити значення $f(x)$ у точках множини $B_2 = B_1 \cup \{a + \frac{h}{4} + i \cdot \frac{h}{2} \mid i = \overline{0; n-1}\}$, для S_{8n} – у точках множини $B_3 = B_2 \cup \{a + \frac{h}{8} + i \cdot \frac{h}{4} \mid i = \overline{0; n-1}\}$ і т. д.

Загалом для визначення $S_{2^k n}$ (домовимося, що $S_{2^0 n} \equiv S_n$) необхідно обчислити $f(x)$ – у точках множини

$$B_k = B_{k-1} \cup \{a + \frac{h}{2^k} + i \cdot \frac{h}{2^{k-1}} \mid i = \overline{0; n-1}\};\ k \in \mathrm{Z},\ k \geq 1.$$

Тепер можна надати змістовну інтерпретацію множинам точок $B_k (k \geq 0)$, у точках (елементах) яких обчислюють $f(x)$:

B_0 – початкова множина (містить n точок);

B_1 – множина після першого подвоєння (містить $2n$ точок);

B_2 – множина після другого подвоєння (містить $4n$ точок);

...

B_k – множина після k-го подвоєння (містить $2^k \cdot n$ точок).

Наближене значення інтеграла після k-го подвоєння:

$$S_{2^k n} = \frac{h}{2^k} \cdot \sum_{x_i \in B_k} f(x_i);\ k = 0, 1, 2, ... \quad (26)$$

Для методів прямокутників і трапецій уточнимо умову (25) завершення обчислень. Нехай $v = S_{2^k n}$, $w = S_{2^{k+1} n}$. Вважатимемо, що після $(k + 1)$-го подвоєння залишковий член $R_{2^{k+1} n} < \varepsilon$.

Тоді $\frac{R_{2^k n}}{R_{2^{k+1} n}} = \frac{h^2}{2^{2k}} : \frac{h^2}{2^{2k+2}} = 4$; $R_{2^k n} = 4 \cdot R_{2^{k+1} n}$. Оскільки досягнуто заданої точності, то маємо таке: $v + R_{2^k n} = w + R_{2^{k+1} n}$;

$v + 4 \cdot R_{2^{k+1} n} = w + R_{2^{k+1} n}$; $R_{2^{k+1} n} = \frac{|w - v|}{3} < \varepsilon$.

Фрагмент програми реалізації методу прямокутників з автоматичним подвоєнням кількості точок:

```
double f(double x) { /* Визначення функції */ }
double integral_prm2 (double a, double b, double eps)
{ double h, v, w, s=0.;
  // v - наближення інтеграла після k-го поділу
  // w - наближення інтеграла після (k+1)-го поділу
  // s - накопичення суми значень функції
  int i, k=0, n=10;  h=(b-a)/n;
  // k - номер поділу; для уникнення зациклення: k < 10
  for(i=0; i<=n-1; s+=f(a+i*h), i++);
  v=0;
  w = h*s;     // Тут w - початкове наближення інтеграла
  while (fabs(v - w)/3>=eps && k<10)
   { k++; v=w;
     for(i=0; i<=n-1; s+=f(a+h/2+i*h), i++);
     h=h/2; n*=2;
     w = h*s; // w - наступне наближення інтеграла
   }
}
```

Обмежимося фрагментом коду C++, оскільки реалізація методу і в цьому випадку є *недосконалою*, адже існує жорсткий зв'язок функції `integral_prm2` з функцією `f`.

Щоби функція `integral_prm2` стала універсальною, необхідно їй передавати назву (адресу осередку пам'яті) довільної підінтегральної функції (до цього питання повернемося згодом).

3.5.4. Розв'язування рівнянь з однією змінною

Розглянемо рівняння

$$f(x) = 0. \quad (27)$$

Якщо $f(x)$ – алгебраїчний багаточлен, то рівняння (27) називають *алгебраїчним*; якщо $f(x)$ містить якісь спеціальні математичні функції (наприклад, $\cos x$, $\sin x$, $\lg x$ тощо), то рівняння (27) називають *трансцендентним*.

Значення змінної x^*, за якого виконується $f(x^*) = 0$, називають *нулем* функції $f(x)$, чи *розв'язком* рівняння (27). Рівняння може мати один чи декілька розв'язків, або не мати жодного.

Проміжок $[a; b]$, на якому є один і тільки один розв'язок x^* рівняння (27), називають *проміжком ізоляції*, а сам розв'язок є *ізольованим* на цьому проміжку.

Для переважної кількості рівнянь не існує аналітичних формул визначення розв'язків. Отож виникає необхідність застосування наближених числових методів розв'язування рівнянь, які, зазвичай, складаються з двох етапів:

1) *визначення проміжків ізоляції* з метою обчислення наближених значень розв'язків рівняння (тобто тут визначають *початкове* (чи *нульове*) наближення);
2) *уточнення* наближеного значення розв'язку до значення із заданою точністю.

Для визначення проміжків ізоляції, зазвичай, використовують таку теорему.

Теорема 6. Якщо функція $f(x)$ є неперервною на проміжку $[a; b]$ і на його кінцях набуває значення протилежних знаків (тобто $f(a)\cdot f(b) < 0$), то всередині цього проміжку існує хоча б один розв'язок рівняння (27). Якщо, окрім цього, перша похідна $f'(x)$ на цьому проміжку зберігає знак, то цей розв'язок буде єдиним.

На підставі цієї теореми зрозуміло, що для визначення усіх проміжків ізоляції достатньо побудувати графік функції або обчислити таблицю її значень. Цим ми далі не займатимемося, а займемося тільки методами уточнення наближеного розв'язку.

Розглянемо найпростіший метод уточнення наближеного розв'язку рівняння – метод *ділення проміжку ізоляції навпіл*. Нехай $f(x)$ є неперервною функцією на проміжку $[a; b]$ і $f(a)\cdot f(b) < 0$. Тоді з теореми 6 випливає, що розв'язок x^* рівняння (27) знаходиться на проміжку $[a; b]$.

Для обчислення наближеного значення розв'язку рівняння виконаємо такі дії. Поділимо проміжок $[a; b]$ навпіл на два проміжки: $[a; \frac{a+b}{2}]$ і $[\frac{a+b}{2}; b]$. Якщо $f\left(\frac{a+b}{2}\right) = 0$, то $x^* = \frac{a+b}{2}$ є точним розв'язком (обчислення припиняємо). Зазвичай, це не так.

Якщо $f\left(\frac{a+b}{2}\right) \neq 0$, то обираємо той проміжок з половин, на кінцях якого $f(x)$ має протилежні знаки. Цей звужений проміжок (позначимо його $[a_1; b_1]$) знову ділимо навпіл і виконуємо подібні дії і т. д. У результаті одержуємо на деякому кроці або точний розв'язок рівняння (27), або нескінченну послідовність вкладених проміжків $[a; b] \supset [a_1; b_1] \supset [a_2; b_2] \supset \ldots \supset [a_i; b_i] \supset \ldots$, таких, що

$$f(a_i)f(b_i) < 0 \quad (i = 1, 2, \ldots), \quad (28)$$

$$b_i - a_i = \frac{b-a}{2^i} \quad (i = 1, 2, \ldots). \quad (29)$$

Оскільки ліві кінці $a_1 \leq a_2 \leq \ldots \leq a_i \leq \ldots$ утворюють монотонну неспадну обмежену послідовність, а праві кінці $b_1 \geq \ldots \geq b_i \geq \ldots$ – монотонну незростаючу обмежену послідовність, то, згідно з (29), одержимо таке:

$$\lim_{i\to\infty}(b_i - a_i) = \lim_{i\to\infty}\frac{b-a}{2^i}; \ \lim_{i\to\infty}(b_i - a_i) = 0; \ \lim_{i\to\infty} b_i = \lim_{i\to\infty} a_i = x^*.$$

Якщо у нерівності (28) перейти до границі при $i \to \infty$, то внаслідок неперервності $f(x)$ одержимо $(f(x^*))^2 \leq 0$. Звідси $f(x^*) = 0$, тобто x^* є розв'язком рівняння (27), причому:

$$0 \leq x^* - a_i \leq \frac{b-a}{2^i} \quad (i = 1, 2, \ldots).$$

Якщо процес поділу зупинити на деякому n-му кроці, то за наближене значення розв'язку можна обрати значення

$$x_n = \frac{a_n + b_n}{2}.$$

Очевидно, що абсолютна похибка (рис. 6) буде такою:

$$\varepsilon = |x_n - x^*| \le \frac{b-a}{2^{n+1}}.$$

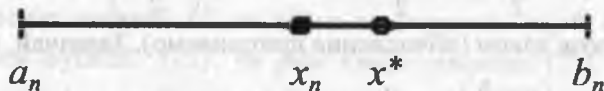


Рис. 6. Геометрична інтерпретація наближення розв'язку

Фрагмент програми, який реалізує метод ділення проміжку ізоляції навпіл:

```
double f(double x)
{
  // Визначення підінтегральної функції
}
double binary_division (double a, double b, double eps)
{
  double x;
  do
   {
    x=(a+b)/2;
    if(!f(x)) return x;
    (f(a)*f(x)<0)? b=x : a=x;
   } while (fabs(b-a)>=eps);
  return (a+b)/2;
}
```

Реалізація методу є *недосконалою*, оскільки існує жорсткий зв'язок функції `binary_division` з функцією `f` (до цього питання повернемося згодом).

3.6. Рекурсивні та ітераційні процеси

3.6.1. Загальні методи конструювання алгоритмів

У мовах C і C++ функції можуть викликати самих себе. Цей процес називають *рекурсією*, а функцію, яка сама себе викликає, – *рекурсивною*. Коли функція викликає сама себе, у системному стекові компілятор виокремлює осередки пам'яті для нових *локальних* змінних і *аргументів*, а код функції виконується із самого початку.

До рекурсії належить також і такий спосіб роботи з даними, за яким функція викликатиме сама себе через ланцюжок викликів інших функцій. Отже, *рекурсивний* процес полягає у *багатократному виконанні* певних дій через повторні виклики функції.

До цього часу ми використовували *ітераційні* процеси, в яких також деякі дії багатократно виконувалися, однак без повторних викликів функцій. Обчислені значення при цьому зберігалися в одній чи декількох локальних змінних. Зокрема, ітераційні процеси використовувалися для реалізації рекурентних співвідношень (див. пункти 3.4.5, 3.5.1, 3.5.2).

Рекурсія є потужним інструментом (чи засобом) конструювання алгоритмів. Для деяких загальних методів конструювання алгоритмів рекурсивний підхід є "*природним*" підходом, який найповніше відображає суть методу. Для глибшого розуміння необхідно детальніше обговорити ці загальні методи.

Загальні методи конструювання (чи розробки) алгоритмів – це узагальнені підходи та принципи розробки алгоритмів, які можна застосувати до найрізноманітніших задач із різних галузей людської діяльності.

Дослідження загальних методів конструювання алгоритмів має два важливі аспекти. По-перше, загальний метод – це своєрідна узагальнена "*інструкція*" про те, як створити новий алгоритм. Відомими є тисячі алгоритмів, однак існує дуже мало методів проектування алгоритмів. По-друге, загальні методи допомагають класифікувати та організовувати алгоритми, що сприяє їхньому розумінню та правильному використанню.

Загальні методи розробки алгоритмів:

Базові	Похідні (оптимізація)
Метод грубої сили Частковий випадок: вичерпне перебирання	“Жадібний” метод
Пошук з поверненням	Метод гілок і меж
Зменшення розміру задачі	Метод послідовного поліпшення розв’язків
Декомпозиція	Динамічне програмування
Перетворення	

Увага! Розв’язки деяких задач базуються на декількох методах.

Метод *грубої сили* – прямолінійний підхід до розв’язування задачі, що базується тільки на постановці задачі та на поняттях, які у ній визначено. Приклади (розділ 4): сортування вибором, сортування методом “бульбашки”, послідовний пошук тощо.

Зазначимо, що розв’язки значної кількості задач, які ми досі розглядали, базувалися на методі грубої сили.

Вичерпне перебирання – це застосування методу грубої сили до розв’язування задачі пошуку елемента у скінченній множині із заданою властивістю. Зазвичай, це пошук серед комбінаторних об’єктів, таких як комбінації, перестановки чи підмножини.

“*Жадібний*” метод (надалі лапки опускатимемо) – це модифікація методу вичерпного перебирання до розв’язування деяких задач *оптимізації*, за якого на кожному кроці визначається розв’язок, який є локально оптимальним, кінцевим і задовольняє усім обмеженням задачі. Із реалізацією деяких жадібних алгоритмів на С++ можна ознайомитися у розділі 5.

Увага! Жадібний метод коректно розв’язує тільки *певні* задачі оптимізації, однак не є загальним методом розв’язування цих задач.

Пошук з поверненням – це покращена версія вичерпного перебирання, що базується на побудові спеціального *дерева простору станів* (*вершини* – часткові розв’язки; *гілки* – переходи до вершин-нащадків, які є розширенням часткових розв’язків), в якому відкидаються *неперспективні* вершини (разом з гілками, які з них виходять) і здійснюється повернення до батьківської вершини.

Метод *гілок і меж* – це удосконалена версія пошуку з поверненням для задач *оптимізації*. Для кожної вершини у дереві простору станів метод обчислює *межу* (оцінку величини функції мети в усіх нащадках цієї вершини), яка існує для відкидання неперспективних вершин та визначає порядок побудови дерева (вершина з найкращою межею досліджується раніше за інших).

Метод *зменшення розміру задачі* – це підхід, що використовує зв’язок між розв’язком задачі заданого розміру і розв’язком цієї ж задачі меншого розміру. Якщо цей зв’язок встановлено, то він може використовуватися або “*зверху вниз*” (рекурсивно), або “*знизу вгору*” (ітеративно, без рекурсії). Виокремлюють три різновиди методу зменшення розміру задачі:

- зменшення на постійну величину (зазвичай, на одиницю);
- зменшення на постійний множник (зазвичай, на два);
- змінне зменшення розміру.

Реалізацію деяких алгоритмів, які базуються на методі зменшення розміру задачі на *постійну величину*, розглянемо у пункті 3.6.2; а тих, що на постійний множник – у пункті 3.6.3. Метод *змінного* зменшення розміру задачі розглянемо у наступному параграфі під час вивчення *алгоритму Евкліда* визначення найбільшого спільного дільника двох цілих чисел.

Метод *послідовного поліпшення* розв’язку – один із загальних підходів до розв’язування задач *оптимізації*, який базується на тому, що спочатку довільно обираємо деякий *допустимий* розв’язок задачі (або опорний план), а далі тим чи іншим способом відшукуємо розв’язки, які є наближеннями до оптимального розв’язку. Метод уперше сформулював Данциг 1947 р. Метод базується на трьох істотних моментах. Необхідно:

- вказувати спосіб обчислення опорного плану;
- встановити ознаку, яка дає змогу перевірити, чи опорний план є оптимальним;
- вказати спосіб переходу від неоптимального опорного плану до іншого опорного плану, ближчого до оптимального.

Прикладами можуть слугувати методи лінійного програмування: симплекс-метод, метод потенціалів, визначення максимального потоку тощо.

Метод *декомпозиції* (*divide-and-conquer*) базується на тому, що початкова задача *рекурсивно* ділиться на декілька підзадач доти, доки вони не стають достатньо простими для застосування деякого алгоритму прямого розв'язування. Опісля розв'язки підзадач об'єднують, щоб отримати розв'язок початкової задачі.

Прикладами можуть слугувати методи сортування злиттям і швидкого сортування, метод Карацуби множення великих чисел, множення матриць методом Штрассена, обхід бінарного дерева, визначення найближчої пари точок і побудови опуклої оболонки, швидке перетворення Фур'є тощо.

Метод *динамічного програмування* (ДП) зародився у рамках дослідження операцій (Річард Беллман, 1950-і роки), де його інтерпретують як варіант методу декомпозиції розв'язування *багатоетапних* задач оптимізації, які мають задовольняти *принцип оптимальності*: оптимальний розв'язок задачі має складатися із оптимальних розв'язків підзадач на цих етапах (тобто метод ДП можна застосувати до тих задач, в яких функція мети є адитивною, а процес розв'язування піддається розбиттю на етапи).

Згодом метод ДП знайшов широке застосування в інформатиці, де його інтерпретують як підхід до розв'язування задач, в яких унаслідок застосування методу декомпозиції виникають підзадачі, що *перекриваються*.

Зазвичай, ці підзадачі виникають з рекурентних співвідношень, які пов'язують розв'язок початкової задачі із розв'язками підзадач меншого розміру. Метод ДП вимагає, щоб розв'язок кожної підзадачі було визначено тільки один раз і збережено у таблиці, з якої потім визначають розв'язок початкової задачі.

Метод *перетворення* полягає у тому, що екземпляр задачі перетворюється до іншого екземпляра, який з певної причини легше розв'язати. Існує три базові варіанти цього методу: *спрощення* екземпляра задачі (перетворення до екземпляра цієї ж задачі з деякими спеціальними властивостями, які спрощують її розв'язок); зміна *відображення* задачі (перетворення одного відображення екземпляра задачі до іншого відображення екземпляра цієї ж задачі); *зведення* задачі (перетворення цієї задачі до іншої задачі, яку можна розв'язати відомим алгоритмом).

3.6.2. Реалізація найпростіших рекурентних співвідношень

Перед створенням рекурсивної функції, зазвичай, необхідно записати *рекурентне співвідношення* (див. пункт 3.4.5), яке визначатиме метод обчислення функції. Довільне рекурентне співвідношення має містити, як мінімум, дві умови:

- умову продовження рекурсії (*крок* рекурсії);
- умову завершення рекурсії (*базу* рекурсії).

У рекурсивній функції необхідно обов'язково задати інструкцію `if` (чи оператор умови) з базою рекурсії, яка неодмінно спричинить повернення з функції без виконання рекурсивного виклику. Якщо цього не зробити, то рекурсивна функція викликатиме сама себе доти, доки не вичерпається її стек. Зазначимо, що у деяких рекурентних співвідношеннях є дві бази.

Очевидно, що будь-який алгоритм, реалізований у рекурсивній формі, може бути реалізований в ітераційній формі і навпаки. Для кращого розуміння запишемо загальні рекурентні співвідношення пункту 3.4.5 у вигляді функції з цілим (чи натуральним) аргументом, а потім реалізуємо їх за допомогою рекурсивного та ітераційного процесів.

Рекурентне співвідношення (1) у функціональному вигляді:

$$u(i)=\begin{cases} c=\text{const}, & i=0; \\ g(u(i-1)), & i=\overline{1;\,n}, \end{cases} \qquad u(i)=\begin{cases} c=\text{const}, & i=1; \\ g(u(i-1)), & i=\overline{2;\,n}, \end{cases} \tag{30}$$

де g – функція чи вираз.

Схема 1 (реалізація *лівої* частини (30)):

Рекурсивний процес	Ітераційний процес

`int u(int n)`
`{return (!n)? c : g(u(n-1));}`
або
`int u(int n) {`
`  if(!n) return c;`
`  return g(u(n-1));`
`}  //c-літерал`
	`int u(int n)`
	`{`
	`  int i, ui;`
	`  for (ui=c, i=0; i<n;`
	`       i++, ui = g(ui));`
	`  return ui;`
	`}  //c-літерал`

Схема 2 (реалізація *правої* частини (30)):

Рекурсивний процес	Ітераційний процес

`int u(int n)`
`{return (n==1)? c : g(u(n-1));}`
або
`int u(int n)`
`{`
`  if(n==1) return c;`
`  return g(u(n-1));`
`} //c-літерал`
	`int u(int n)`
	`{`
	`  int i, ui;`
	`  for (ui=c, i=1; i<n;`
	`       i++, ui = g(ui));`
	`  return ui;`
	`} //c-літерал`

Розглянемо рекурентне співвідношення (5), яке визначає факторіал цілого числа n ($n \geq 0$). Запишемо (5) у функціональному вигляді:

$$u(i) = \begin{cases} 1, & i = \overline{0;1}, \\ i \cdot u(i-1), & i = \overline{2;n}. \end{cases} \qquad (31)$$

Хоча формула (31) містить дві бази, однак для її реалізації можна використати *модифіковану схему 2* (для `n==0` чи `n==1` у рекурсивному процесі виконуватиметься тільки повернення з функції без виконання рекурсивного виклику чи ініціалізація змінних `i=u=1` – в ітераційному процесі):

Рекурсивний процес	Ітераційний процес

`int fact(int n)`
`{`
`  return (n<=1)?`
`          1 : n*fact(n-1);`
`}`
	`int fact(int n)`
	`{`
	`  int i, u;`
	`  for(i=u=1; i<n; u*=++i);`
	`  return u;`
	`}`

Розглянемо детальніше процес виконання рекурсивної функції `fact` на прикладі обчислення `fact(4)`. Для пояснення задіємо механізм позначок `m1` – `m5`. Окрім цього, "`->`" означатиме, що виконання функції ще не завершено, а "`=`" – виконання функції завершено з поверненням результату.


```
m1: fact(4) -> 4*fact(3); m2: fact(3) -> 3*fact(2)
m3: fact(2) -> 2*fact(1);
m4: fact(1) = 1;           m3: fact(2) = 2*1 = 2
m2: fact(3) = 3*2 = 6;     m1: fact(4) = 4*6 = 24
```

Зазначимо, що для обчислення факторіала числа використовують загальний метод *зменшення розміру* задачі на *одиницю*. Зв'язок між розв'язком задачі даного розміру і розв'язком цієї ж задачі меншого розміру визначають за допомогою другого рядка формули (31).

Рекурентне співвідношення (2), яке визначає *поточне* значення елемента послідовності через значення двох *попередніх* елементів, у функціональному вигляді є таким (надалі розглядатимемо тільки праві частини відповідних формул):

$$u(i) = \begin{cases} c1 = \text{const}, & i = 1; \\ c2 = \text{const}, & i = 2; \\ h(u(i-2), u(i-1)), & i = \overline{3;n}, \end{cases} \qquad (32)$$

де h – деяка функція чи вираз.

Схема 3 (реалізація формули (32)):

Рекурсивний процес	Ітераційний процес

`int u(int n)`
`{`
`  return (n==1)? c1 :`
`   ((n==2)? c2 :`
`        h(u(n-2), u(n-1)));`
`}`
`//c1, c2 - літерали`
	`int u(int n)`
	`{`
	`  int a, b, i, ui;`
	`  if(n==1) ui=c1;`
	`  if(n==2) ui=c2;`
	`  for(a=c1, b=c2, i=2; i<n;`
	`      i++, ui=h(a, b),`
	`      a=b, b=ui);`
	`  return ui;`
	`}`
	`//c1, c2 - літерали`

Якщо у формулі (32) `c1=c2=c`, то рекурсивний та ітераційний процес матимуть такий вигляд, як проілюстровано на схемі 4.

Схема 4.

Рекурсивний процес	Ітераційний процес

`int u(int n)`
`{`
` return (n<=2)? c :`
`      h(u(n-2), u(n-1));`
`}`
`//c - літерал`
	`int u(int n)`
	`{`
	` int a, b, i, ui;`
	` for(ui=a=b=c, i=2; i<n;`
	`   i++, ui=h(a, b),`
	`   a=b, b=ui);`
	`  return ui;`
	`}`
	`//c - літерал`

За цією схемою можна реалізувати рекурентне співвідношення (4), яке визначає числа Фібоначчі. Запишемо (4) у вигляді:

$$u(i)=\begin{cases}1, & i=\overline{1;2},\\ u(i-2)+u(i-1), & i=\overline{3;n}.\end{cases} \qquad (33)$$

За *схемою 4* реалізація формули (33) виглядатиме так:

Рекурсивний процес	Ітераційний процес

`int fib(int n)`
`{`
` return (n<=2)? 1 :`
`       fib(n-2)+fib(n-1);`
`}`
	`int fib(int n)`
	`{ int a, b, i, u;`
	`  for(u=a=b=1, i=2; i<n;`
	`     i++, u=a+b,`
	`     a=b, b=u);`
	`  return u;`
	`}`

Розглянемо детальніше процес виконання рекурсивної функції `fib` на прикладі обчислення `fib(4)`:

```
m1: fib(4) -> fib(2)+fib(3) ;
  m11: fib(2) = 1; m12: fib(3) -> fib(1)+fib(2)
        m121: fib(1) =1; m122: fib(2) = 1
        (m121; m122) ⇒ m12: fib(3) = 1+1 = 2
(m11; m12)   ⇒ m1:  fib(4) = 1+2 = 3
```


Визначення чисел Фібоначчі слугує яскравим прикладом застосування методу декомпозиції (початкова задача весь час *рекурсивно* ділиться на дві підзадачі), унаслідок застосування якого виникають підзадачі, що *перекриваються*. Під час визначення `fib(4)` повторюється обчислення `fib(2)`. Якщо визначати `fib(5)`, то повторюються обчислення `fib(3)`, `fib(2)` і `fib(1)`. Аналогічні міркування можна застосувати і до `fib(6)`, `fib(7)` і т. д.

У цьому випадку замість викликів рекурсивної функції рекомендують застосовувати метод динамічного програмування, згідно з яким необхідно, щоб розв'язок кожної підзадачі було визначено тільки один раз і збережено у таблиці, з якої потім визначають розв'язок початкової задачі.

Отже, необхідно лише заповнити елементи одновимірного масиву `(n + 1)`-м послідовним значенням `fib(i)`, використовуючи рекурентне співвідношення (33). Очевидно, що останній елемент цього масиву міститиме значення `fib(n)`.

Якщо нас цікавить тільки значення `fib(n)`, то можна обійтися і без використання масиву, обмежившись запам'ятовуванням тільки двох останніх елементів послідовності (саме це відбувається в *ітераційному процесі* реалізації формули (33)).

У цьому пункті далі розглянемо ще три приклади реалізації рекурентних співвідношень: у прикладах 27 і 29 рекурентне співвідношення є наслідком загального методу *зменшення розміру* задачі на *одиницю*, а у прикладі 28 – методу *декомпозиції*.

Приклад 27. Визначити рекурентне співвідношення для обчислення степеня числа ($a^n, a \in R; a \neq 0; n = 0;1;2;\ldots$). Скласти фрагменти програми, які реалізують рекурсивний та ітераційний процеси обчислення степеня числа.

➤ *Попередні міркування*. Оскільки

$$a^n=\underbrace{a\cdot a\cdot\ldots\cdot a}_{n}=a\cdot(\underbrace{a\cdot\ldots\cdot a}_{n-1});a^0=1,$$

то рекурентне співвідношення обчислення степеня числа є таким:

$$u(i)=\begin{cases}1, & i=0;\\ a\cdot u(i-1), & i=\overline{1;n}.\end{cases} \qquad (34)$$

Отже, для обчислення степеня числа використано загальний метод *зменшення розміру* задачі на *одиницю*. Зв'язок між розв'язком задачі такого розміру і розв'язком цієї ж задачі меншого розміру визначають за допомогою другого рядка формули (34).

Реалізація формули (34) виглядатиме так:

Рекурсивний процес	Ітераційний процес

`double Pow(double a, int n)`
`{`
`  return (!n)? 1 :`
`         a*Pow(n-1);`
`}`
	`double Pow(double a, int n)`
	`{`
	`  int i; double u;`
	`  for(i=0, u=1; i<n; i++, u*=a);`
	`  return u;`
	`}`

Формула (34) не відображає факту, що обчислення залежить від двох параметрів (a та n). Перепишемо її у такому вигляді:

$$u(a,i) = \begin{cases} 1, & i = 0; \\ a \cdot u(a,i-1), & i = 1; n. \end{cases} \quad (34`)$$

Формула (34`) точніше відображає обчислювальний процес порівняно з формулою (34), і цілковито відповідає функції Pow, яка залежить від двох параметрів (a та n).

Приклад 28. Визначити рекурентне співвідношення для обчислення біноміальних коефіцієнтів C_n^i за допомогою загального методу *декомпозиції*. Скласти фрагмент програм реалізації рекурсивного процесу їхнього обчислення.

➢ *Попередні міркування*. Біноміальним коефіцієнтом (позначення C_n^i, $C(n, i)$ або $\binom{n}{i}$) називають кількість комбінацій (підмножин) з i елементів у n-елементній множині, де $0 \le i \le n$. Назва "біноміальні коефіцієнти" походить від участі цих чисел у формулі бінома:

$$(a+b)^n = \sum_{i=0}^{n} C_n^i \cdot a^{n-i} \cdot b^i .$$

У комбінаториці доводять, що

$$C_n^i = \begin{cases} 1, & i = 0, i = n; \\ C_{n-1}^{i-1} + C_{n-1}^{i}, & n > i > 0. \end{cases} \quad (35)$$

Отже, задача обчислення C_n^i, згідно з формулою (35), зводиться до двох підзадач: обчислення C_{n-1}^{i-1} та C_{n-1}^{i} (тобто тут застосовано загальний метод декомпозиції).

Рекурсивний процес реалізація формули (35) виглядатиме так:

```
int c(int n, int i)
{
  return (!i || i==n)? 1 : c(n-1,i-1)+c(n-1,i);
}
```

Якщо розглянути процес виконання рекурсивної функції `c(n, i)` на прикладі обчислення `c(5, 3)`: $C_5^3 \to C_4^2 + C_4^3$; $C_4^2 \to C_3^1 + C_3^2$; $C_4^3 \to C_3^2 + C_3^3$; ..., то побачимо, що обчислення C_3^2 дублюється. Якщо цей процес продовжити, то повторюватимуться й інші біноміальні коефіцієнти, тобто загальний метод декомпозиції і в цьому випадку утворює підзадачі, що *перекриваються*.

Отже, застосовувати рекурсивний процес для реалізації рекурентного співвідношення (35) *не доцільно*. У цих випадках рекомендують застосовувати метод динамічного програмування, який буде реалізовано згодом.

Замінимо рекурентне співвідношення (35) на інше, яке базується на прямій формулі обчислення C_n^i згідно з означенням:

$$C_n^i = \frac{n!}{i!(n-i)!}; \quad 0 \le i \le n.$$

Визначимо рекурентний множник:

$$R = \frac{C_n^i}{C_n^{i-1}} = \frac{n!}{i!(n-i)!} \cdot \frac{(i-1)!(n-i+1)!}{n!} = \frac{n-i+1}{i}.$$

Рекурентне співвідношення обчислення C_n^i згідно з означенням:

$$C_n^i = \begin{cases} 1, & i=0; \\ C_n^{i-1} \cdot \dfrac{n-i+1}{i}, & i=1;n. \end{cases} \quad (36)$$

Запишемо (37) у функціональному вигляді з використанням позначення $C_n^i = comb(n,i)$:

$$comb(n,i) = \begin{cases} 1, & i=0; \\ \dfrac{n-i+1}{i} \cdot comb(n,i-1), & i=1;n. \end{cases} \quad (37)$$

Отже, тепер для обчислення біноміальних коефіцієнтів C_n^i використано загальний метод *зменшення розміру* задачі на *одиницю*. Зв'язок між розв'язком задачі такого розміру i та розв'язком цієї ж задачі меншого розміру $(i-1)$ визначають за допомогою другого рядка формули (37).

Приклад 29. Скласти програму визначення біноміальних коефіцієнтів для бінома $(a+b)^n$. Обчислення біноміальних коефіцієнтів реалізувати за допомогою рекурсивного та ітераційного процесів (для порівняння) згідно з рекурентним співвідношення (37).

➢ *Попередні міркування*. Реалізація рекурентного співвідношення (37) за допомогою рекурсивного процесу *буквально* (один до одного) повторює формулу цього рекурентного співвідношення (у програмі це функція comb_R).

Для розуміння реалізації рекурентного співвідношення (37) за допомогою ітераційного процесу (у програмі це функція comb_I) варто порівняти це співвідношення зі співвідношенням (34`): параметр n у (37) відповідає параметру a у (34`), а параметр i – параметру n. Необхідно звернути увагу на те, що оператори множення та ділення реалізовані в арифметиці раціональних чисел, щоб не втратити значущих розрядів (це неодмінно сталося б, якщо б ми користувалися арифметикою цілих чисел).

На вході (Enter n>=0). Кожен такий рядок є окремим тестом, що задає значення змінної n (степеня бінома). Ознакою завершення введення даних є рядок, в якому введене значення n є *від'ємним*.
На виході. Для кожного n одержуємо відповідний набір біноміальних коефіцієнтів для рекурсивної (comb_R:) та ітераційної (comb_I:) реалізації рекурентного співвідношення (37). За результатами тестування можна впевнитися, що значення цих коефіцієнтів співпадають (тобто є незалежними від способу реалізації рекурентного співвідношення).
Програма:

```
#include <iostream>
using namespace std;
//------------------------------------------------------
int comb_R(int n, int i)
// Рекурсивний процес реалізації (37)
{ return (!i)? 1 : comb_R(n,i-1)*(n-i+1)/i; }
//------------------------------------------------------
int comb_I(int n, int i)
// Ітераційний процес реалізації (37)
{ int k, u;
  for(u=1, k=0; k<i; k++, u=int(double(u)*(n-k+1)/k));
  return u;
}
//------------------------------------------------------
void main() { int i, n;
 while (cout<<"Enter n>=0: ", cin>>n, n>=0) {
  cout<<"comb_R:"<<endl;
  for(i=0; i<=n;
   cout<<"c("<<n<<","<<i<<")="<<comb_R(n,i)<<" ",i++);
   cout<<endl; cout<<"comb_I:"<<endl;
  for(i=0; i<=n;
   cout<<"c("<<n<<","<<i<<")="<<comb_I(n,i)<<" ",i++);
   cout<<endl;
 }
 cin>>i; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_4_29\Debug\Example_4_29.exe
Enter n>=0: 2
comb_R:
c(2,0)=1 c(2,1)=2 c(2,2)=1
comb_I:
c(2,0)=1 c(2,1)=2 c(2,2)=1
Enter n>=0: 3
comb_R:
c(3,0)=1 c(3,1)=3 c(3,2)=3 c(3,3)=1
comb_I:
c(3,0)=1 c(3,1)=3 c(3,2)=3 c(3,3)=1
Enter n>=0: 5
comb_R:
c(5,0)=1 c(5,1)=5 c(5,2)=10 c(5,3)=10 c(5,4)=5 c(5,5)=1
comb_I:
c(5,0)=1 c(5,1)=5 c(5,2)=10 c(5,3)=10 c(5,4)=5 c(5,5)=1
Enter n>=0: 7
comb_R:
c(7,0)=1 c(7,1)=7 c(7,2)=21 c(7,3)=35 c(7,4)=35 c(7,5)=21 c(7,6)=7 c(7,7)=1
comb_I:
c(7,0)=1 c(7,1)=7 c(7,2)=21 c(7,3)=35 c(7,4)=35 c(7,5)=21 c(7,6)=7 c(7,7)=1
Enter n>=0: -3
```

Увага! До цього моменту ми використовували рекурентні співвідношення, в яких визначалося значення *поточного* елемента послідовності через одне чи два значення *попередніх* елементів (тобто таких, що є безпосередніми сусідами поточного елемента). Назвемо такі рекурентні співвідношення *найпростішими*.

Реалізація *найпростішого* рекурентного співвідношення за допомогою рекурсивного процесу *буквально* (один до одного) повторює формулу цього співвідношення, якщо її "розгортати" *від кінця до початку* через повторні виклики функції.

Реалізація *найпростішого* рекурентного співвідношення за допомогою ітераційного процесу *буквально* повторює формулу цього співвідношення, якщо її "розгортати" *від початку до кінця* через збереження обчислених значень у локальній змінній (`ui / u`).

Ітерація і рекурсія базуються на конструкціях керування: ітерація використовує конструкцію повторення (циклу), рекурсія – конструкцію галуження. Як рекурсія, так і ітерація реалізують повторення деяких дій: ітерація через явне використання конструкції циклу, рекурсія – за допомогою повторних викликів функції.

Ітерація та рекурсія містять перевірку на завершення: ітерація завершується, якщо перестає виконуватися умова продовження циклу; рекурсія завершується, якщо розпізнається нерекурсивний випадок (база рекурсії).

Ітерація та рекурсія можуть відбуватися нескінченно: ітерація потрапляє у нескінченний цикл, якщо умова продовження циклу ніколи не стає хибною; рекурсія триває нескінченно, якщо крок рекурсії не редукує підзадачі так, щоб вони збігалися до бази.

Рекурсивні функції необхідно використовувати обережно:

- рекурсивні процеси рекурентних співвідношень виконуються значно повільніше, ніж їхні ітеративні еквіваленти, з-за витрат часу на організацію повторних викликів функції;
- за кожного нового виклику функції створюється нова копія параметрів функції та локальних змінних у стекові, а це означає, що стек може швидко вичерпатися.

Зазначимо, що наведені у цьому пункті приклади можна відшукати майже у кожній книжці з програмування, де вони є своєрідною базою пояснення рекурсії. Це пов'язано з їхньою простотою та зрозумілістю. Однак усі ці приклади застосування рекурсії, фактично, є прикладами того, як не слід застосовувати рекурсію (тобто *програмувати* їх треба все ж таки через ітераційні процеси).

Вищесказане не означає, що у програмуванні можна обійтися без рекурсії. Це зовсім не так! Яскравим прикладом застосування рекурсії є "*олімпіадні*" задачі, на розв'язування яких встановлюють обмежений час. Рекурсивний підхід дає змогу швидше зрозуміти суть задачі, записати і налагодити алгоритм її розв'язування тощо.

Під час програмування рекурсії необхідно впевнитися у тому, що задачу не можна *легше* розв'язати за допомогою ітераційного процесу. Рекурсію варто використовувати тоді, коли нерекурсивний підхід ускладнює програмування або не зумовлює до сильного зменшення часу роботи алгоритму, об'єму пам'яті тощо.

Рекурсивний підхід переважає ітераційний і у випадках, коли рекурсія природніше відображає *математичний* бік задачі і зумовлює до програми, яка є простішою для розуміння. Іншою причиною вибору рекурсивного способу розв'язування може слугувати те, що ітераційний спосіб розв'язування може не бути очевидним. Однак, якщо вдалося розв'язати задачу ітераційно, то, зазвичай, її рекурсивного аналогу не шукають.

У наступному пункті розглянемо задачі, які варто розв'язувати саме за допомогою рекурсивного підходу.

3.6.3. Зменшення розміру задачі на постійний множник

У попередньому пункті порівнювали рекурсивні та ітераційні процеси реалізації деяких базових алгоритмів, які відображалися найпростішими рекурентними співвідношеннями. Нагадаємо, що ці співвідношення отримано унаслідок застосування таких загальних методів конструювання алгоритмів як метод декомпозиції та метод зменшення розміру задачі на одиницю.

Якщо унаслідок застосування методу декомпозиції отримують підзадачі, що *перекриваються*, то застосовувати рекурсивну процедуру для реалізації відповідного алгоритму не рекомендують через її *експоненту* складність. У цих випадках застосовують метод динамічного програмування (чи звичайний ітераційний процес, як частковий випадок цього методу).

Якщо використовують метод зменшення розміру задачі на одиницю, то рекурсивний та ітераційний процеси реалізації методу є простими і зрозумілими. Однак через витрати часу на організацію повторних викликів функції рекурсивні процеси виконуються дещо повільніше, ніж їхні ітераційні еквіваленти. Отже і в цьому випадку рекомендують (хоча і не так категорично) використовувати ітераційні процеси.

Перейдемо тепер до застосування загального методу зменшення розміру задачі на постійний *множник*. Для пояснення цього методу наведемо декілька класичних прикладів. Спочатку удосконалимо алгоритм піднесення до степеня (приклад 27).

Часова складність виконання функції Pow у прикладі 27 є *лінійною* (O(*n*)). Однак існує алгоритм обчислення степеня числа за логарифмічний час (O($\log_2 n$)), який базується на відомій формулі $a^{n\cdot m} = (a^m)^n$. Наприклад: $a^{10} = a^{2\cdot 5} = \left(a^5\right)^2 = \left(a\cdot\left(a^2\right)^2\right)^2$

Рекурентне співвідношення для обчислення степеня числа за логарифмічний час має такий вигляд (ділення – *цілочислове*):

$$a^n = \begin{cases} 1, & n = 0; \\ a\cdot\left(a^{n/2}\right)^2, & n - \text{непарне}; \\ \left(a^{n/2}\right)^2, & n - \text{парне}. \end{cases} \qquad (38)$$

Нехай $u(a,n)$ – функція, що обчислює a^n (тобто $a^n = u(a,n)$). Тоді $(a^{n/2})^2 = (a^2)^{n/2} = u(a^2, n/2)$. З огляду на ці зауваження, запишемо співвідношення (38) у вигляді, який є зручним для програмування:

$$u(a,n) = \begin{cases} 1, & n = 0; \\ a\cdot u(a^2, n/2), & n - \text{непарне}; \\ u(a^2, n/2), & n - \text{парне}. \end{cases} \qquad (39)$$

Алгоритм, що базується на рекурентних співвідношеннях (38) / (39) є результатом застосування методу зменшення розміру задачі *удвічі*.

Необхідно чітко розуміти різницю цього алгоритму від алгоритму, який базується на методі *декомпозиції*, що розв'язує дві задачі розміру *n*/2, тобто:

$$a^n = \begin{cases} 1, & n = 0; \\ a^{\lfloor n/2 \rfloor}\cdot a^{\lceil n/2 \rceil}, & n > 0. \end{cases} \qquad (40)$$

Алгоритм, що базується на формулі (40) є неефективним, оскільки знову ж таки виникають підзадачі, що *перекриваються* (тобто одержуємо *експоненту* складність алгоритму).

Рекурентні співвідношення, які не є найпростішими, назвемо *складними* рекурентними співвідношеннями. Рекурентні співвідношення (38) – (40) є складними.

Що стосується *складних* рекурентних співвідношень, то усі *переваги* рекурсивного процесу реалізації (простота, зрозумілість, легкість кодування) цих співвідношень *зберігаються*. Наприклад, складне рекурентне співвідношення (39) можна відобразити такою рекурсивною функцією С++ (*буквальне* відтворення формули):

```
double Pow_R(double a, int n)
{
   if(!n) return 1;
   if(n%2) return a*Pow_R(a*a, n/2);
   return Pow_R(a*a, n/2);
}
```

Можна запропонувати ще простіший варіант:

```
double Pow_R(double a, int n)
{
  return  (!n)?  1 :
          (n%2)?  a*Pow_R(a*a, n/2) :
                  Pow_R(a*a, n/2);
}
```

Однак з реалізацією *складних* рекурентних співвідношень за допомогою ітераційного процесу можуть виникнути проблеми. Доволі часто *буквально* відтворити співвідношення не вдається, отож його необхідно модифікувати. Якщо навіть співвідношення і не змінюється, то його кодування вимагає значних зусиль та часу.

Окрім цього, не завжди вдається "розгорнути" формулу *складного* рекурентного співвідношення *від початку до кінця* (втрачається *універсальність*). Інколи необхідно "розгортати" її так, як у рекурсивному процесі: *від кінця до початку*. Наприклад, складне рекурентне співвідношення (39) можна відобразити такою нерекурсивною функцією C++ (впізнати формулу (39) можна, однак треба докласти зусиль):

```
double Pow_I(double a, int n)
{
  double u=1;
  while (n>0)
  {
     if(n%2) u*=a;
     n/=2, a*=a;
  }
  return u;
}
```

У зв'язку з вищесказаним з приводу реалізації *складних* рекурентних співвідношень за допомогою ітераційного процесу, у цьому пункті надалі наводитимемо тільки рекурсивний процес реалізації алгоритму.

Популярним прикладом застосування методу зменшення розміру задачі удвічі є *задача Йосипа*, яку названо так за іменем Флавія Йосипа, відомого єврейського історика, учасника і літописця

повстання євреїв проти римлян 66-70 рр. Йосип протягом 47-ми днів очолював оборону фортеці Йотапата. Після того, як вона впала, він та 40 фанатиків сховалися у печері неподалік. Однак повстанці вирішили вчинити самогубство, щоб не здатися римлянам. Йосип запропонував, щоб усі встали в коло і почергово кожен убивав свого сусіда (тобто кожного другого) доти, доки не залишиться одна людина, яка має вчинити самогубство. Йосип, який вирішив здатися, став на таке місце у колі, щоб він залишився останнім.

Приклад 30 (аналог задачі Йосипа). Скласти програму, яка визначатиме номер людини у колі, що залишиться останньою внаслідок групового самогубства за схемою Флавія Йосипа. Вважати, що спочатку маємо n людей (у задачі Йосипа $n = 41$).

➤ *Попередні міркування.* Розставимо навколо n людей, пронумерувавши їх від 1 до n. Починаючи відлік з першої людини і рухаючись по колу, видалятимемо кожну другу людину доти, доки не залишиться одна людина з номером J (очевидно, що J є функцією від n, тобто $J = J(n)$, $1 \le J \le n$). Якщо $n = 6$, то за чергою будуть видалені люди з номерами 2, 4, 6; 3, 1. Отже, $J(6) = 5$. Якщо $n = 9$, то за чергою будуть видалені люди з номерами 2, 4, 6, 8; 1, 5, 9, 7. Отже, $J(7) = 3$ і т. д.

Складемо рекурентне співвідношення для визначення $J(n)$. Очевидно, що $J(1) = 1$.

Якщо n – парне ($n = 2k$), то після першого проходження кола будуть видалені люди з парними номерами: 2, 4, ..., $2k$. Залишаться люди з непарними номерами, а відлік продовжиться з номера 1. Це еквівалентно тому, що розмір задачі зменшився *удвічі* (стало k людей). Єдина відмінність полягає у нумерації людей: номер 3 під час другого проходження кола матиме *новий* номер 2; номер 5 – *новий* номер 3 і т. д. Очевидно, щоб отримати початкові (стартові) номери людей, необхідно новий номер подвоїти і зменшити на 1, тобто

$$J(2k) = 2 \cdot J(k) - 1. \qquad (41)$$

Якщо n – непарне ($n = 2k + 1$), то після першого проходження кола будуть видалені люди з парними номерами: 2, 4, ..., $2k$, а особу з

номером 1 буде видалено одразу ж за особою з номером $2k$. Залишиться k людей з номерами 3, 5, 7, …, $2k+1$. Очевидно, що у цьому випадку для того, щоб отримати початкові (стартові) номери людей, необхідно новий номер подвоїти і збільшити на 1, тобто

$$J(2k+1) = 2 \cdot J(k) + 1. \qquad (42)$$

Об'єднуючи формули (41) і (42), одержимо таке *складне* рекурентне співвідношення розв'язування задачі Йосипа:

$$J(n) = \begin{cases} 1, & n = 1; \\ 2 \cdot J(n/2) + 1, & n - \text{непарне}; \\ 2 \cdot J(n/2) - 1, & n - \text{парне}. \end{cases} \qquad (43)$$

На вході (`Enter n>0`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (кількості жертв). Ознакою завершення введення даних є рядок, в якому введене значення `n` *не є додатним*. *На виході*. Для кожного `n` одержуємо номер (`Jn=`) людини, яка залишиться останньою.

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------------------
int J(int n)
{
  return  (n==1)?  1 :
          (n%2)?  2*J(n/2)+1 :
                  2*J(n/2)-1;
}
//--------------------------------------------------------
void main()
{ int  n;
  while (cout<<"Enter n>0: ", cin>>n, n>0)
    cout<<"Jn="<<J(n)<<endl;
  cin>>n; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_4_...
Enter n>0: 6
Jn=5
Enter n>0: 9
Jn=3
Enter n>0: 10
Jn=5
Enter n>0: 41
Jn=19
Enter n>0: 40
Jn=17
Enter n>0: 0
```

Додатковий коментар. Рекурентне співвідношення (43) можна розв'язати в явному вигляді (тобто отримати аналітичний вигляд $J(n)$). Якщо $m-$ таке найбільше ціле число, що $n = 2^m + l$ $(m \geq 0; 0 \leq l < 2^m)$, то

$$J(n) = 2 \cdot l + 1. \qquad (44)$$

Для $n = 6 = 2^2 + 2$: $J(6) = 2 \cdot 2 + 1 = 5$; $n = 9 = 2^3 + 1$: $J(9) = 3$; $n = 41 = 2^5 + 9$: $J(41) = 2 \cdot 9 + 1 = 19$; $n = 40 = 2^5 + 8$: $J(40) = 17$.

І ще один цікавий факт: значення $J(n)$ можна одержати шляхом циклічного зсування вліво на один біт двійкового зображення n. Наприклад, $J(6) = J(110_2) = 101_2 = 5$; $J(9) = J(1001_2) = 0011_2 = 3$; $J(41) = J(101001_2) = 010011_2 = 19$.

◄

У літературі (чи Інтернеті) можна відшукати безліч задач, які певним чином повторюють постановку задачі Йосипа (насамперед це стосується "*олімпіадних*" задач). Наступний приклад присвячений одній з таких задач.

Приклад 31 (відбір у розвідку). З n солдатів, вишикуваних у шеренгу, потрібно відібрати декількох у розвідку. Для здійснення цього виконують таку операцію: якщо солдатів у шеренгі більше трьох, то вилучаються усіх солдатів, що стоять на парних позиціях, або усіх солдатів, що стоять на непарних позиціях. Цю процедуру

повторюють доти, доки у шерензі залишиться 3 або менше солдатів. Вони і підуть у розвідку. Скласти програму обчислення кількості способів формування груп розвідників, які налічуватимуть трьох солдатів.

➤ *Попередні міркування.* Нехай $R(n)$ – це кількість способів формування груп розвідників з n солдатів. Оскільки група має налічувати трьох розвідників, то $R(1) = 0$, $R(2) = 0$, $R(3) = 1$.

Якщо n парне, то, застосовуючи операцію вилучення солдатів у шерензі, одержимо або $n / 2$ солдатів, що залишилися на парних позиціях, або $n / 2$ солдатів, що залишилися на непарних позиціях, тобто $R(n) = 2R(n / 2)$.

Якщо n непарне, то застосовуючи операцію вилучення солдатів у шерензі, одержимо або $n / 2$ солдатів, що залишилися на парних позиціях, або $(n / 2 + 1)$ солдатів, що залишилися на непарних позиціях, тобто $R(n) = R(n / 2) + R(n / 2 + 1)$.

Одержали таке *складне* рекурентне співвідношення:

$$R(n) = \begin{cases} 0, & n = 1, n = 2; \\ 1, & n = 3; \\ R(n/2) + R(n/2+1), & n - \text{непарне}; \\ 2R(n/2), & n - \text{парне}. \end{cases} \quad (45)$$

На вході (`Enter n>0`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (кількості солдатів). Ознакою завершення введення даних є рядок, в якому введене значення `n` *не є додатним*.

На виході. Для кожного `n` одержуємо кількість способів формування груп розвідників з трьох солдатів (`Rn=`).

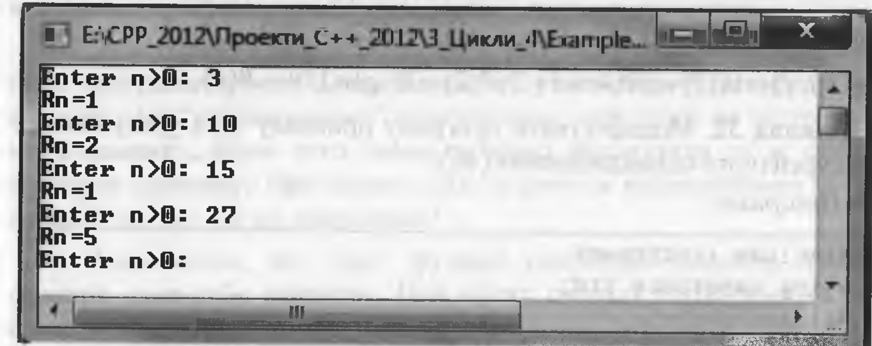
Програма:

```
#include <iostream>
using namespace std;
//-----------------------------------------------------
int R(int n) {
  return (n<=2)? 0 : (n==3)? 1 :
            (n%2)? R(n/2)+R(n/2+1) : 2*R(n/2);
}
```



```
//-----------------------------------------------------
void main() { int n;
  while (cout<<"Enter n>0: ", cin>>n, n>0)
     cout<<"Rn="<<R(n)<<endl;
  cin>>n; // Пауза
}
```

Результати тестування програми:



Рекурентне співвідношення (45) є результатом застосування методу зменшення розміру задачі вдвічі. Однак третій рядок формули (45) спричинюватиме появу підзадач, які перекриватимуться.

Водночас специфіка підзадач (45) дає змогу перебороти перекривання задач за допомогою введення нового рекурентного співвідношення, яке залежатиме від трьох параметрів:

$$g(n,i,j) = \begin{cases} 0, & n = 1; \\ j, & n = 2; \\ i, & n = 3; \\ g(n/2, i, i+2j), & n - \text{непарне}; \\ g(n/2, 2i+j, j), & n - \text{парне}. \end{cases} \quad (46)$$

У рекурентному співвідношенні (46) немає рядків, які могли б спричинити перекривання задач. Однак, виникають логічні питання: звідки це все взялося і як воно працюватиме? Очевидно, що "сила" формули (46) походить від допоміжних параметрів i та j.

Доведемо правильність рекурентного співвідношення (46). Нехай $g(k,i,j) = iR(k) + jR(k+1)$. Тоді одержуємо таке:

$$g(2k,i,j) = iR(2k) + jR(2k+1) = 2iR(k) + jR(k) + jR(k+1) =$$
$$= (2i+j)R(k) + jR(k+1) = g(k,2i+j,j);$$
$$g(2k+1,i,j) = iR(2k+1) + jR(2k+2) = iR(k) + iR(k+1) + 2jR(k+1) =$$
$$= iR(k) + (i+2j)R(k+1) = g(k,i,i+2j);$$
$$g(1,i,j) = iR(1) + jR(2) = 0; \quad g(2,i,j) = iR(2) + jR(3) = j;$$
$$g(3,i,j) = iR(3) + jR(4) = i + 2jR(2) = i; \quad g(n,1,0) = R(n).$$

Приклад 32. Модифікувати програму прикладу 31 з урахуванням рекурентного співвідношення (46).

➢ *Програма*:

```
#include <iostream>
using namespace std;
//--------------------------------------------------------
int g(int n, int i, int j) {
  return (n==1)? 0 : (n==2)? j : (n==3)? i :
         (n%2)? g(n/2,i,i+2*j) : g(n/2,2*i+j,j);
}
//--------------------------------------------------------
int R(int n) { return g(n,1,0); }
//--------------------------------------------------------
void main() { int n;
  while (cout<<"Enter n>0: ", cin>>n, n>0)
      cout<<"Rn="<<R(n)<<endl;
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example...
Enter n>0: 10
Rn=2
Enter n>0: 27
Rn=5
Enter n>0: -2
```


3.6.4. Рекурсивна функція та стек

Рекурсивні функції лише на перший погляд виглядають як звичайні фрагменти програм. Щоб відчути їхню специфіку, Доволі подумки простежити процес їхнього виконання за текстом програми. У звичайній програмі ми слідуватимемо ланцюжком викликів функцій, однак жодного разу повторно не ввійдемо в один і той же фрагмент, доки з нього не вийдемо. Можна сказати, що процес виконання програми однозначно "*лягає*" на текст програми.

Інша справа – рекурсія. Якщо спробувати відстежити за текстом програми процес її виконання, то увійшовши у рекурсивну функцію, ми "*рухатимемося*" за її текстом доти, доки не зустрінемо її виклику, після чого знову почнемо виконувати ту ж саму функцію спочатку. При цьому слід зазначити найважливіше – її перший виклик ще не завершився!

Виявляється, що текст функції копіюється щоразу, коли функція сама себе викликає. Цей ефект справді відтворюється в комп'ютері. Однак копіюється *не весь* текст функції, а тільки її частини, пов'язані з *даними* (аргументами, локальними змінними і точками повернення).

Алгоритмічна частина (інструкції та вирази) рекурсивної функції та глобальні змінні не змінюються, отож вони присутні у пам'яті комп'ютера в єдиному екземплярі.

Щоб зрозуміти, як саме копіюються дані під час викликів рекурсивної функції, необхідно познайомитися з поняттям *стека.*

Стеком називають динамічну структура даних, в якій у кожен момент часу доступний тільки верхній (останній) елемент. Послідовність обробки елементів стека добре відображають абревіатури LIFO (Last In First Out – "останнім увійшов, першим вийшов") і FILO (First In Last Out - "першим увійшов, останнім вийшов").

Реалізувати стек можна будь-яким зручним для програміста способом, наприклад, масивом. Тоді початком стека (його "*верхнім*" елементом) буде останній компонент масиву, а звільнення стека відбуватиметься у напрямі від кінця масиву до його початку. За такої реалізації немає необхідності у постійному переміщенні

компонент масиву. Однак найефективнішою є реалізація стека за допомогою однозв'язного лінійного списку.

Під час виконання рекурсивної функції компілятор створює для неї спеціальний *стек* (надалі називатимемо його *апаратним стеком*) і в ньому розміщує необхідні дані. Кожен рекурсивний виклик функції породжує новий "*екземпляр*" аргументів і локальних змінних, причому старий "*екземпляр*" не знищується, а зберігається у стеку за принципом вкладеності. Відбувається це у такій послідовності:

- у стеку резервується місце для формальних параметрів, в які записуються значення фактичних параметрів (аргументів);
- під час виклику функції у стек записується точка повернення – адреса тієї частини програми, де перебуває виклик функції;
- на початку тіла функції у стеку резервується місце для локальних (автоматичних) змінних.

Пераховані змінні утворюють групу (фрейм стека). Стек "*пам'ятає*" історію рекурсивних викликів у вигляді послідовності (ланцюжків) таких фреймів. Програма у кожен конкретний момент працює з останнім викликом і з останнім фреймом. Під час завершення кроку рекурсії програма повертається до попередньої версії рекурсивної функції і до попереднього фрейму в стекові.

Незважаючи на те, що код рекурсивної функції буває дуже коротким, виконання функції може виявитися неефективним, оскільки на організацію фреймів витрачають і час, і пам'ять – у програму вставляють відповідні команди, які виконуються за кожного виклику функції.

Однак ця неефективність не дуже істотна, порівняно з іншою – витратою пам'яті під стек. Розмір стека збільшується за кожного виклику доти, доки не буде досягнуто точки повернення (бази рекурсії). Розмір стека пропорційний глибині рекурсії.

Глибина рекурсії – це максимальна кількість вкладеності рекурсивних викликів. У загальному випадку глибина залежатиме від вхідних даних. Наприклад, у рекурсивній функції `fact(n)` глибина рекурсії дорівнюватиме `n`. У принципі, це може зумовити до переповнення стека та до аварійного завершення програми.

Отже, під час виконання рекурсивної функції компілятор створює *апаратний* стек і в ньому розміщує необхідні дані. Однак те, що робиться автоматично, можна реалізувати в явному вигляді з використанням спеціально організованого *програмного* стека. Тоді рекурсивний алгоритм стає циклічним алгоритмом, однак простота та зрозумілість алгоритму зберігається.

Програмний стек використовують у випадках, в яких глибина рекурсії за певних умов може стати настільки великою, що спричинить до переповнення апаратного стека. Під час використання програмного стека програміст може передбачити ці випадки і відповідно на них відреагувати (закодувати необхідні дії).

Усі наведені вище приклади рекурсії базувалися на відомих *рекурентних* співвідношеннях. Однак у багатьох випадках рекурсивна природа задачі не очевидна. Наприклад, відомо чимало задач, які ефективно розв'язують за допомогою стека. Очевидно, що у цих задачах замість створення програмного стека можна використати апаратний стек (тобто реалізувати алгоритм рекурсивно).

Щоб з'ясувати специфіку реалізації алгоритму розв'язку задачі за допомогою рекурсії, необхідно ознайомитися зі *схемами організації* (чи *структурою*) і *типами* рекурсивних функцій.

Довільна рекурсивна функція (позначимо `Rec`) містить деяку множину інструкцій `S` і один або декілька рекурсивних викликів `Rec`. Рекурсивний виклик обов'язково супроводжується *умовою* (завершення чи продовження викликів `Rec`), оскільки безумовні виклики `Rec` зумовлять до *нескінченного* процесу. Отже, рекурсивний виклик є інструкцією галуження чи інструкцією виразу з умовою.

Структура рекурсивної функції є *найпростішою*, якщо функція містить тільки *один* рекурсивний виклик, у протилежному випадку – *складною*.

Аргументами `Rec` під час рекурсивного виклику слугують вирази, операндами яких є параметри та локальні змінні `Rec`. Виклик `Rec` з конкретним набором аргументів називатимемо *копією* `Rec`.

Рекурсивні функції поділяють на два *типи*: лінійні та нелінійні. Якщо кожна гілка рекурсивного виклику містить тільки одну копію `Rec`, то функція – *лінійна*, а протилежному випадку – *нелінійна*.

Розрізняють три форми найпростішої структури рекурсивної функції (домовимося, що умова – це умова *продовження* рекурсивного виклику):

- форма з виконанням дій `S` після рекурсивного виклику (на рекурсивному *поверненні* (*підйомі*)):

```
void Rec() {if(умова) Rec(); S;}
```

- форма з виконанням дій `S` до рекурсивного виклику (на рекурсивному *спуску*):

```
void Rec() {S; if(умова) Rec();}
```

- форма з виконанням дій `S1` на рекурсивному спуску і дій `S2` на рекурсивному підйомі:

```
void Rec() {S1; if(умова) Rec(); S2;}
```

Увага! У прототипах указано заголовок `void Rec()`, оскільки тут важливо продемонструвати *форми*, не відволікаючись на деталі.

Отже, дії `S` (чи `S1` і `S2`) – це дії, які не зв'язані з рекурентним викликом функції `Rec`. Назви "рекурсивний спуск" і "рекурсивний підйом" пов'язані з поняттям глибини рекурсії – функція опускається на глибину рекурсії і піднімається "звідти".

В усіх попередніх прикладах з рекурсією ми використовували *першу форму*, в якій `S` – це *база* рекурсії. Доведемо це для загальних схем.

Схема 1 (реалізація *лівої* частини (30)):

```
int u(int n) {
  if(n) return g(u(n-1));
  return c; //c-літерал
}
```

Схема 2 (реалізація *правої* частини (30)):

```
int u(int n) {
  if(n>1) return g(u(n-1));
  return c; //c - літерал
}
```

Функція `u` (за схемами 1 і 2) є лінійною рекурсивною функцією з найпростішою структурою першої форми.

Схема 3 (реалізація формули (32)):

```
int u(int n) {
  if(n>2) return h(u(n-2), u(n-1));
  return (n==1)? c1 : c2; //c1, c2 - літерали
}
```

Схема 4 (реалізація формули (32), якщо `c1=c2=c`):

```
int u(int n) {
  if(n>2) return h(u(n-2), u(n-1));
  return c; //c - літерал
}
```

Функція `u` (за схемами 3 і 4) є нелінійною рекурсивною функцією з найпростішою структурою першої форми.

Формула (31) відображає рекурентне співвідношення обчислення факторіала, яке реалізуємо за допомогою рекурсивної функції `fact`, зведеної до першої форми:

```
int fact(int n) {
  if(n>1) return n*fact(n-1);
  return 1;
}
```

Розгортаючи процес рекурсивних викликів і обчислень (на рекурсивному підйомі) функції `fact`, одержимо таке:

Рівень рекурсії	Рекурсивний спуск ↓	Рекурсивний підйом ↑
0	`fact(4)`	Повернення`fact(4)=24`
1	`fact(4) -> 4*fact(3)`	`fact(4) = 4*6 = 24`
2	`fact(3) -> 3*fact(2)`	`fact(3) = 3*2 = 6`
3	`fact(2) -> 2*fact(1)`	`fact(2) = 2*1 = 2`
4		`fact(1) = 1`

Перша форма найпростішої структури рекурсивної функції (з обчисленнями на рекурсивному підйомі) є "природною" формою реалізації рекурентних співвідношень, які розглядали до цього

часу. Розгортання рекурсивного процесу тут відбувається від кінцевих значень деякого параметра функції до початкових.

Інші форми найпростішої структури рекурсивної функції очевидно розгортатимуться від початкових значень параметра функції до кінцевих, що характерніше для ітераційних процесів.

Для реалізації рекурентних співвідношень за допомогою другої і третьої форм необхідно застосовувати додаткові "штучні" прийоми, наприклад: використання глобальних змінних, введення додаткового параметра рекурсивної функції тощо. Проілюструємо це на такому прикладі.

Приклад 33. Скласти програму обчислення скінченного добутку

$$P = \prod_{i=l}^{k} f(i). \qquad (47)$$

Для тестування програми вважатимемо, що $f(i) = \dfrac{3i^3 - 4i}{i^2 + 2i + 5}$.

➢ *Попередні міркування.* Для коректного обчислення скінченного добутку необхідно вимагати виконання умови $k \ge l$. Добуток накопичуватимемо у глобальній змінній m. Процес накопичення:

$$m = 1;\ m = m \cdot f(i),\ \ i = \overline{l;\, k};\ \ P = m. \qquad (48)$$

Формула (48), фактично, є відображенням *ітераційного* процесу обчислення скінченного добутку, однак вона допомагає визначити дію `S1` на початку відповідної рекурсивної функції, вигляд рекурсивного виклику цієї функції та дію `S2` наприкінці.

Якщо для обчислення добутку використовуватимемо функцію `prod(l, k)`, то дія `S1` полягатиме у виконанні інструкції `m*=f(l)`, де початкове значення глобальної змінної (`m=1`) має задаватися у головній функції перед викликом `prod(l, k)`.

Після цього стає очевидним, що рекурсивний виклик функції матиме вигляд `prod(l+1, k)`, який продовжуватиметься доти, доки справджуватиметься умова `l < k`. Дія `S2` полягатиме у поверненні результату функції (значення скінченного добутку).

На вході (`Enter l:` та `Enter k (k>=l):`). Ці рядки визначають окремий тест, оскільки задають значення нижньої та верхньої межі

зміни індексу скінченного добутку. Ознакою завершення введення даних є рядки, після яких одержимо такі значення `l` та `k`, що `l > k`. *На виході.* Для кожних `l` та `k` одержуємо значення скінченного добутку (`product=`).

Програма:

```
include <iostream>
using namespace std;
double m; // Глобальна змінна
//-------------------------------------------------------
double f(int i) {
   double x = static_cast<double>(i*i);
   return (3*x *i-4*i) /(x+2*i+5);
}
//-------------------------------------------------------
double prod(int l, int k) {  m*=f(l);
   if(l<k) return prod(l+1, k); return m;
}
//-------------------------------------------------------
void main() { int l, k;
  while (cout<<"Enter l: ", cin>>l,
         cout<<"Enter k (k>=l): ", cin>>k, k>=l)
        {m=1; cout<<"product="<<prod(l,k)<<endl; }
  cin>>k; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_3_33\Debug\...
Enter l: 1
Enter k (k>=l): 10
product=-4.29379e+007
Enter l: 1
Enter k (k>=l): 5
product=-28.5883
Enter l: -6
Enter k (k>=l): -2
product=-142713
Enter l: -2
Enter k (k>=l): 5
product=0
Enter l: 5
Enter k (k>=l): 1
```

Якщо `f(i) = i` та `l = 1`, то функція `prod(1, k)`, фактично, обчислюватиме значення факторіала `(k!)`. Для кращого розуміння процесу обчислення вставимо у функцію `prod(1, k)` інструкцію виведення проміжних результатів.

Модифікація програми для обчислення факторіала:

```
#include <iostream>
using namespace std;
double m; // Глобальна змінна
//------------------------------------------------------
double f(int i) { return i; }
//------------------------------------------------------
double prod(int l, int k){
   m*=f(l);
   if(l<k)
    { cout<<"m="<<m<<endl; return prod(l+1, k); }
   cout<<endl;
   return m;
}
//------------------------------------------------------
void main() { int l, k;
  while (cout<<"Enter l: ", cin>>l,
         cout<<"Enter k (k>=l): ", cin>>k, k>=l)
        {m=1; cout<<"product="<<prod(l,k)<<endl; }
  cin>>k; // Пауза
}
```

Результати тестування модифікованої програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цикли_4\Example_3_33\De...
Enter l: 1
Enter k (k>=l): 4
m=1
m=2
m=6

product=24
Enter l: 2
Enter k (k>=l): 1
```


Розгортаючи процес рекурсивних викликів і обчислень на рекурсивному спуску функції prod, одержимо таке:

Рівень рекурсії	Рекурсивний спуск	Рекурсивний підйом
0	`prod(1, 4)`	`prod(1, 4) = 24`
1	`m=1*1=1; prod(2, 4) ->`	`prod(2, 4) = 24`
2	`m=1*2=2; prod(3, 4) ->`	`prod(3, 4) = 24`
3	`m=2*3=6; prod(4, 4) ->`	`prod(4, 4) = 24`
4	`m=6*4=24`	

Приклад 34. Скласти програму обчислення добутку (47) за схемою (48) без використання глобальної змінної.

➢ *Попередні міркування.* Все те саме, що у прикладі 33, однак замість глобальної змінної `m` у функцію `prod` вводимо параметр `m`.

Програма:

```
#include <iostream>
using namespace std;
//------------------------------------------------------
 double f(int i) {
    double x = static_cast<double>(i*i);
    return (3*x *i-4*i) /(x+2*i+5);
}
//------------------------------------------------------
double prod(double m, int l, int k) { m*=f(l);
   if(l<k) return prod(m, l+1, k);
   return m;
}
//------------------------------------------------------
void main() { int l, k;
  while (cout<<"Enter l: ", cin>>l,
         cout<<"Enter k (k>=l): ", cin>>k, k>=l)
         cout<<"product="<<prod(1.0, l, k)<<endl;
  cin>>k; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цикли_4\Example_3_34\Debug\...
Enter l: 1
Enter k (k>=l): 5
product=-28.5883
Enter l: -6
Enter k (k>=l): -2
product=-142713
Enter l: 1
Enter k (k>=l): 10
product=-4.29379e+007
Enter l: 4
Enter k (k>=l): 2
```

∢

Рекурсивний процес у прикладах 33 і 34 "*притягнутий за вуха*" з метою відображення третьої форми рекурсивної функції. Однак можна записати рекурентне співвідношення формули (47):

$$P(i)=\begin{cases} f(i), & i=l; \\ P(i-1)\cdot f(i), & i=\overline{l+1;k}. \end{cases} \qquad (49)$$

Приклад 35. Скласти рекурсивну функцію з найпростішою структурою першої форми для обчислення співвідношення (49).

➢

```
double prod_1(int l, int k) {
   if(k>l) return prod_1(l, k-1)*f(k);
   return f(l);
}
```

∢

Рекурентне співвідношення для $S=\sum_{i=l}^{k} f(i)$ має вигляд:

$$S(i)=\begin{cases} f(i), & i=l; \\ S(i-1)+f(i), & i=\overline{l+1;k}. \end{cases} \qquad (50)$$

Увага! Автор *усвідомлює*, що розв'язки прикладів 33 – 35 є дещо "екзотичними" (тут раціональніше використовувати ітераційні процеси), однак наводить їх винятково з навчально-методичних позицій для глибшого розкриття суті рекурсивних функцій і процесів.

3.7. Програмування задач цілочислової арифметики

3.7.1. Теоретичні положення

Множину $Z=\{...;-3;-2;-1;0;1;2;3;...\}$ називають множиною *цілих* чисел, а $N=\{1;2;3;...\}$ – множиною *натуральних* чисел. Відносно операцій додавання і множення множина Z є моногенним асоціативно-комутативним кільцем з одиницею.

Нехай $a,b\in Z$ $(b\neq 0)$. Число a *ділиться* на число b, якщо $\exists\, q\in Z$, що $a=qb$. Синоніми: b ділить a, a кратне b, b – дільник a. Позначення: $b|a$ або $a \vdots b$. Число 0 не може бути дільником жодного числа, однак воно ділиться на будь-яке ненульове число.

Наприклад, дільниками числа 6 є числа: –6; –3; –2; –1; 1; 2; 3; 6. Зазвичай, перелічують тільки натуральні дільники: 1; 2; 3; 6. Якщо $a>0$ і $b|a$, то $|b|\leq a$.

Натуральне число $a\neq 1$, яке не має дільників, окрім 1 і a, називають *простим* числом. Прості числа: 2; 3; 5; 7; 11; 13; 17; … . Простих чисел є нескінченно багато. Натуральне число $a\neq 1$, яке не є простим, називають *складеним* числом. Числа 0 і 1, а також цілі від'ємні числа не є ні простими, ні складеними.

Нехай $a,b\neq 0, c\in Z$. Якщо $b|a$ і $b|c$, то b називають спільним дільником цілих чисел a і c. Серед спільних дільників a і c, зазвичай, виокремлюють *найбільший спільний дільник* (*greatest common divisor*), який позначають НСД(a,c) або $\gcd(a,c)$.

Для однозначності визначення НСД(a,c) необхідно, щоб хоча б одне з чисел a і c не дорівнювало нулю (для зручності, домовимося, що НСД(0,0) = 0). Очевидно, що

$$\text{НСД}(a,c)\geq 0;\ \ \text{НСД}(a,0)=|a|. \qquad (51)$$

Приклад 36.

➢ НСД(9,6) = НСД(6,9) = НСД(−9,6) = НСД(−9,−6) = 3;
НСД(0,7) = 7; НСД(0,−4) = 4; НСД(0,0) = 0. ∢

Теорема 7. Нехай $a,b\neq 0, c,k,m\in Z$. Якщо $b|a$ і $b|c$, то

$$b|(ak+cm). \qquad (52)$$

➢ Оскільки $b|a$ і $b|c$, то $a=qb$, $c=wb$, де $q,w\in Z$.
Тоді $ak+cm=qbk+wbm=b(qk+wm)$. ∢

Наслідок 1 (з теореми 7). Нехай $a, b \neq 0, c \in Z$. Якщо $b|a$ і $b|c$, то

$$b|(a+c);\ \ b|(a-c). \tag{53}$$

Формули (52) і (53) справедливі й для $b = \text{НСД}(a,c)$, адже b тут хоча й найбільший спільний дільник, однак все ж таки *дільник*.

Поняття найбільшого спільного дільника поширюють на довільну кількість цілих чисел за допомогою рекурентної формули:

$$\begin{aligned}&\text{НСД}(a_1, a_2, \dots, a_{n-1}, a_n) = \text{НСД}(a_1, \text{НСД}(a_2, \dots, a_{n-1}, a_n)) = \\ &= \text{НСД}(a_1, \text{НСД}(a_2, \text{НСД}(\dots, \text{НСД}(a_{n-1}, a_n)))),\end{aligned} \tag{54}$$

де $n \in N, n > 2$.

Нехай $a, c, d \in Z$ $(a \neq 0; c \neq 0)$. Якщо $a|d$ і $c|d$, то d називають спільним кратним цілих чисел a і c. Серед спільних кратних a і c, зазвичай, виокремлюють *найменше додатне спільне кратне* (*least common multiple*), яке позначають $\text{НСК}(a,c)$ або $\text{lcm}(a,c)$. Для коректності визначення $\text{НСК}(a,c)$ необхідно, щоб кожне з чисел a і c не дорівнювало нулю.

Приклад 37.

➢ $\text{НСК}(9,6) = \text{НСК}(6,9) = \text{НСК}(-9,6) = \text{НСК}(-9,-6) = 18.$ ⊲

Поняття найменшого спільного кратного поширюють на довільну кількість цілих чисел за допомогою рекурентної формули:

$$\begin{aligned}&\text{НСК}(a_1, a_2, \dots, a_{n-1}, a_n) = \text{НСК}(a_1, \text{НСК}(a_2, \dots, a_{n-1}, a_n)) = \\ &= \text{НСК}(a_1, \text{НСК}(a_2, \text{НСК}(\dots, \text{НСК}(a_{n-1}, a_n)))),\end{aligned} \tag{55}$$

де $n \in N, n > 2$.

Наведемо без доведення таку теорему.

Теорема 8. Нехай $a, c \in Z$. Якщо $ac > 0$,

$$\text{то } ac = \text{НСД}(a,c) \cdot \text{НСК}(a,c). \tag{56}$$

Формула (56) дає змогу визначити $\text{НСК}(a,c)$, якщо відомий $\text{НСД}(a,c)$. Для визначення $\text{НСД}(a,c)$ існують різні алгоритми. Наприклад, зі шкільних років можна згадати алгоритм, який базується на розкладанні цілих чисел на прості множники.

Все було б чудово, однак розкладання цілого числа на прості множники є значно важчою задачею, ніж власне задача визначення НСД двох цілих чисел. Найчастіше для визначення $\text{НСД}(a,c)$ використовують славнозвісний *алгоритм Евкліда*. Перш ніж розпочати його вивчення, нам необхідно пригадати ще одну операцію над цілими числами – операцію *ділення з остачею*.

Нехай $a, b \in Z$ $(b \neq 0)$. Операція *ділення з остачею* числа a на число b полягає у визначенні чисел $q, r \in Z$, що

$$a = bq + r, \tag{57}$$

де $0 \leq r < |b|$. Число q називають *неповною часткою*, а r – *остачею*. Якщо $r = 0$, то кажуть, що a ділиться на b *без остачі*.

Відзначимо, що остача r – це *невід'ємне* ціле число, а неповна частка q – *довільне* ціле число. У теорії чисел доводять, що для будь-яких $a, b \in Z$ $(b \neq 0)$ існує єдиний розклад числа a згідно з (57). Очевидно, що $0 = b \cdot 0 + 0$.

Приклад 38. Поділити з остачею: а) 41 на 7; б) 41 на –7;

в) –41 на 7; г) –41 на –7.

➢ а) $41 = 5 \cdot 7 + 6$; б) $41 = -5 \cdot (-7) + 6$;
в) $-41 = -6 \cdot 7 + 1$; г) $-41 = 6 \cdot (-7) + 1$. ⊲

Приклад 39. Поділити з остачею: а) 7 на 41; б) 7 на –41;

в) –7 на 41; г) –7 на –41.

➢ а) $7 = 0 \cdot 41 + 7$; б) $7 = 0 \cdot (-41) + 7$;
в) $-7 = -1 \cdot 41 + 34$; г) $-7 = 1 \cdot (-41) + 34$. ⊲

У Microsoft C++ результат операції цілочислового ділення a/b та операції отримання залишку від цілочислового ділення $a\%b$ відповідають, відповідно, визначенню чисел $q, r \in Z$ у розкладі (57) ***тільки*** для $a \geq 0$.

Це відбувається тому, що результат операції цілочислового ділення a/b отримують цілочисловим діленням їхніх абсолютних значень з подальшим присвоєнням частці необхідного знака. Результат операції отримання залишку від цілочислового ділення:

$$a\%b = a - (a/b) \cdot b. \tag{58}$$

Приклад 40. За умовами прикладів 38 і 39 скласти програму, в якій відобразити результати операції цілочислового ділення a/b та операції отримання залишку від цілочислового ділення $a\%b$.

➢ *На вході* (`Enter a b!=0`). Кожен рядок є окремим тестом, що містить цілі числа a і $b \neq 0$. Ознакою завершення введення вхідних даних є рядок, у якому перше число дорівнює нулю.

На виході (`Result`). Для кожної пари цілих чисел, окрім останньої пари, одержуємо значення $q = a/b$ та $r = a\%b$.

Програма:

```
#include <iostream>
using namespace std;
//-------------------------------------------------------
void main()
{
  int a, b;
  while (cout<<"Enter a  b!=0: ", cin>>a>>b, a)
   {
     cout<<"  Result: q="<<a/b<<"; "<<"r="<<a%b<<endl;
   }
  cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цілі_числа\Ділення_Залишок\Debug\Ділення_Залишок...
Enter a b!=0: 41 7
  Result: q=5; r=6
Enter a b!=0: 41 -7
  Result: q=-5; r=6
Enter a b!=0: -41 7
  Result: q=-5; r=-6
Enter a b!=0: -41 -7
  Result: q=5; r=-6
Enter a b!=0: 7 41
  Result: q=0; r=7
Enter a b!=0: -7 41
  Result: q=0; r=-7
Enter a b!=0: 7 -41
  Result: q=0; r=7
Enter a b!=0: -7 -41
  Result: q=0; r=-7
Enter a b!=0: 0 5
```

◄

Приклад 41. За умовами прикладів 38 і 39 скласти програму, в якій відобразити результати (неповну частку q і залишок r) операції ділення з остачею чисел $a, b \in Z$ ($b \neq 0$) згідно з (57).

➢ *На вході* (`Enter a b!=0`). Кожен рядок є окремим тестом і містить цілі числа a і $b \neq 0$. Ознакою кінця вхідних даних є рядок, в якому перше число дорівнює нулю.

На виході (`Result`). Для кожної пари цілих чисел, окрім останньої пари, одержуємо значення неповної частки q та остачі r.

Програма:

```
#include <iostream>
using namespace std;
//-------------------------------------------------------
void main()
{ int a, b,q,r;
  while (cout<<"Enter a b!=0: ", cin>>a>>b, a)
   {
     q=(b>0)? floor(float(a)/b) : ceil(float(a)/b);
      r=a-q*b;
     cout<<"  Result: q="<<q<<"; "<<"r="<<r<< endl;
   }
  cin>>a; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цілі_числа\Ділення_з_Правильним_залишком\Debu...
Enter a b!=0: 41 7
  Result: q=5; r=6
Enter a b!=0: 41 -7
  Result: q=-5; r=6
Enter a b!=0: -41 7
  Result: q=-6; r=1
Enter a b!=0: -41 -7
  Result: q=6; r=1
Enter a b!=0: 7 41
  Result: q=0; r=7
Enter a b!=0: -7 41
  Result: q=-1; r=34
Enter a b!=0: 7 -41
  Result: q=0; r=7
Enter a b!=0: -7 -41
  Result: q=1; r=34
Enter a b!=0: 0 7
```

◄

3.7.2. *Визначення найбільшого спільного дільника*

Як уже зазначено вище, для визначення найбільшого спільного дільника використовують алгоритм Евкліда. Цей алгоритм базується на такій теоремі.

Теорема 9. Нехай $a, c \in Z\ (a \geq 0; c > 0)$. Тоді

$$\text{НСД}(a,c) = \text{НСД}(c, a\%c). \quad (59)$$

➤ Згідно з (57) існують такі числа $q, r \in Z$, що

$$a = cq + r, \quad (60)$$

де $0 \leq r < c$.

Оскільки $a \geq 0$, то $r = a\%c$ (див. формулу (58) і приклад 40).

Нехай $d = \text{НСД}(a,c)$. У цьому випадку $d|a$ і $d|c$. Згідно з теоремою 7 та співвідношенням (60): $d|(a - cq)$; $d|r$. Оскільки $r < c$, то $d = \text{НСД}(c,r) = \text{НСД}(c, a\%c)$.

Аналогічно доводимо справедливість (59) і у зворотному порядку. Нехай $d = \text{НСД}(c,r)$. У цьому випадку $d|c$ і $d|r$. Згідно з теоремою 7 та співвідношенням (60): $d|(cq + r)$; $d|a$. Оскільки $r \leq a$, то $d = \text{НСД}(a,c)$. ≺

Оскільки для <u>додатних</u> чисел a і c остача $r = a\%c$ завжди *менша* за дільник c, то послідовно застосовуючи формулу (59), визначатимемо найбільший спільний дільник d для все менших і менших чисел. Зрештою одержимо ситуацію $\text{НСД}(t, 0)$. Це, згідно з (51), означатиме, що $\text{НСД}(a,c) = t$.

Приклад 42. Визначити найбільший спільний дільник чисел 16 і 28.

➤ а) НСД(16,28) = НСД(28,16) = НСД(16,12)= НСД(12,4) =

= НСД(4,0) = 4;

б) НСД(28,16) = НСД(16,12)= НСД(12,4) = НСД(4,0) = 4. ≺

Аналізуючи приклад (42), зазначимо таке: якщо $a < c$, то

$$\text{НСД}(a,c) = \text{НСД}(c, a\%c) = \text{НСД}(c, a),$$

тобто аргументи функції НСД у цьому випадку *переставляються*. Під час подальших викликів функції НСД перший аргумент завжди більший за другий. Нулем може стати тільки другий аргумент. Це саме відбувається і для $a = 0$: $\text{НСД}(0, c) = \text{НСД}(c, 0\%c) = \text{НСД}(c, 0)$.

Отже, опираючись на теорему 9 та формулу (51), для чисел $a, c \in Z\ (a \geq 0; c > 0)$ можна записати таку рекурентну формулу визначення їхнього найбільшого спільного дільника:

$$\text{НСД}(a,c) = \begin{cases} a, c = 0; \\ \text{НСД}(c, a\%c), c \neq 0. \end{cases} \quad (61)$$

Якщо ж a і c – довільні цілі числа, то, опираючись на те, що $\text{НСД}(a,c) = \text{НСД}(|a|, |c|)$, формулу (61) можна записати так:

$$\text{НСД}(a,c) = \begin{cases} |a|, c = 0; \\ \text{НСД}(c, |a|\%c), c \neq 0. \end{cases} \quad (62)$$

Відповідна функція на C++ матиме такий вигляд:

```
int gcd(int a, int c)
{
    if (!c) return abs(a);
    return gcd(c, abs(a)%c);
}
```

Використовуючи тернарний умовний оператор, цю функцію можна записати простіше:

```
int gcd(int a, int c)
{   return (!c)? abs(a) : gcd(c, abs(a)%c); }
```

Опираючись на теорему (8) і на той факт, що $\text{НСК}(a,c) = \text{НСК}(|a|, |c|)$, запишемо функцію визначення найменшого спільного кратного двох *ненульових* цілих чисел:

```
int lcm(int a, int c)
{
  if (!(a*c)) return -1; // Помилка: a=0 і/або c=0
  return abs(a / gcd(a,c) * c);
}
```

Приклад 43. Виконати тестування функцій `gcd` і `lcm`.

➤ *На вході* (`Enter a c`). Кожен рядок є окремим тестом і містить цілі числа a і c, які *водночас* не дорівнюють нулю. Ознакою завершення вхідних даних є рядок, в якому *обидва* числа дорівнюють 0.

На виході (Result). Для кожної пари цілих чисел, окрім останньої пари, одержуємо значення НСD = gcd(a, c) та НСК=lcm(a, c). Якщо НСК=−1, то це означає, що $a = 0$ і/або $c = 0$.

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//-------------------------------------------------------
int gcd(int a, int c)
//-------------------------------------------------------
// Визначає найбільший спільний дільник (НСД) a і c
// Передумова: аргументи a і c - довільні цілі числа
// Постумова: повертає НСД(a, c); НСД(0, 0)=0
//-------------------------------------------------------
{    return (!c)? abs(a) : gcd(c, abs(a)%c);  }
//-------------------------------------------------------
int lcm(int a, int c)
//-------------------------------------------------------
// Визначає найменше спільне кратне (НСК) чисел a і c
// Передумова: аргументи a і c - ненульові цілі числа
// Постумова: повертає НСК(a, c)
// НСК(a, c)=-1 - ознака помилки (a=0 і/або c=0)
//-------------------------------------------------------
{ if (!(a*c)) return -1; return abs(a/gcd(a,c) * c); }
//-------------------------------------------------------
void  main() { int a, c;
  while (cout<<"Enter a c: ", cin>>a>>c, a||c)
   {
     cout<<"  Result: HCD="<<gcd(a,c);
     cout<<"; HCK="<<lcm(a, c)<<endl;
   }
  cin>>a; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цілі_числа\НСД_НСК\Debug\НСД_...
Enter a c: 3 2
  Result: HCD=1; HCK=6
Enter a c: -6 4
  Result: HCD=2; HCK=12
Enter a c: 16 28
  Result: HCD=4; HCK=112
Enter a c: -16 28
  Result: HCD=4; HCK=112
Enter a c: 0 5
  Result: HCD=5; HCK=-1
Enter a c: -6 0
  Result: HCD=6; HCK=-1
Enter a c: 0 0
```

Наведемо ітераційний варіант визначення НСД(a, c):

```
int gcd_iter(int a, int c)
//-------------------------------------------------------
// Визначає найбільший спільний дільник (НСД) a і c
// Передумова: аргументи a і c - довільні цілі числа
// Постумова: повертає НСД(a, c); НСД(0, 0)=0
//-------------------------------------------------------
{
  int r;
  a=abs(a); c=abs(c);
  do
    { r=a%c, a=c, c=r; } while(r);
  return a;
}
```

Як ми уже зазначали раніше, *алгоритм Евкліда* визначення найбільшого спільного дільника двох цілих чисел базується на такому <u>загальному</u> методі конструювання алгоритмів, як *метод зменшення розміру* задачі на *змінну* величину. Варто нагадати, що у попередньому параграфі розглянуто зменшення розміру задачі на *постійну величину* та зменшення розміру задачі на *постійний множник*.

3.7.3. Прості числа

Просте число – це натуральне число, яке має тільки два натуральні дільники (1 і саме число). Нагадаємо, що одиниця, нуль та усі від'ємні цілі числа не можуть бути простими числами за означенням. Здавна для визначення усіх простих чисел, які є меншими за деяке натуральне число *n*, використовують простий алгоритм під назвою *решето Ератосфена*[1].

Розглянемо цей алгоритм. Спочатку складемо список чисел від 2 до *n*. На першому кроці алгоритму з цього списку вилучаємо (або перекреслюємо) усі числа, які діляться на 2 (кратні 2). Потім обираємо з початку списку наступне просте число (на другому кроці це буде число 3; на третьому – 5 і т. д.) і вилучаємо зі списку числа, які діляться на нього. Робота алгоритму триватиме доти, доки у списку будуть числа, які можна вилучити. Числа, які залишилися після роботи цього алгоритму – прості.

Продемонструємо роботу алгоритму для $n = 30$ (стільки чисел можна розмістити в одному рядку):

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
2 3   5   7   9    11    13    15    17    19    21    23    25    27    29
2 3   5   7        11    13          17    19          23    25          29
2 3   5   7        11    13          17    19          23                29
```

Виникає питання: для яких простих чисел *p* з початку списку ще залишаються кратні їм числа у решті списку, які треба вилучити (тобто, за яких *p* алгоритм має продовжувати роботу)?

Відповідь така: необхідно вилучати кратні числа для всіх простих чисел *p* з початку списку, для яких $p^2 \leq n$. Справді, якщо *p* – чергове просте число, кратні якого вилучатимуться зі списку, то вилучення необхідно починати з числа p^2, оскільки попередні числа, кратні *p* (тобто числа: $2p, 3p, \ldots, (p-1)p$) вилучено на попередніх кроках алгоритму.

У нашому прикладі $5^2 < 30$, однак $7^2 > 30$ (під час роботи алгоритму вилучено числа, кратні 2, 3 і 5).

[1] Ератосфен – давньогрецький учений

Приклад 44. Скласти програму визначення усіх простих чисел, які є меншими за деяке натуральне число *n*.

➢ *Попередні міркування*. Для зберігання чисел використовуватимемо динамічний масив `Prime`.

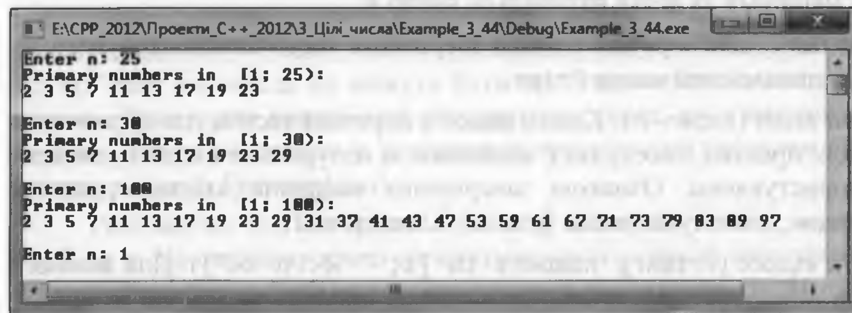
На вході (`Enter n`). Кожен рядок є окремим тестом для визначення усіх простих чисел, які є меншими за натуральне число *n*, введене користувачем. Ознакою завершення введення вхідних даних є рядок, в якому це число буде не більшим за 1.

На виході (`Primary numbers in [1; " << n <<"`)). Для кожного проміжку `[1; n)` виводиться список простих чисел, які входять у нього.

Програма:

```
#include <iostream>
using namespace std;
//-------------------------------------------------------
void  main()
{
 int *Prime;  int i,j,n;
 while (cout<<"Enter n: ", cin>>n, n>1)
  {
   Prime=new int[n];
   for (i=0; i<n; i++) Prime[i]=i; // Початковий список
   // На місці складених чисел ставимо нулі
   for (j=2; j*j <= n; j++)
     for (i=2; i<n; i++)
        if (((i % j) == 0) && (i != j)) Prime[i]=0;
   cout<<"Primary numbers in [1; " << n <<"): "<<endl;
   for (i=2; i < n; i++)
      if (Prime[i] > 0) cout << Prime[i] << " " ;
   cout <<endl<<endl;
   delete [] Prime;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:



Приклад 45. Скласти програму, яка визначатиме, чи деяке натуральне число *n* буде простим?

➤ *Попередні міркування*. Найпростіший спосіб перевірки *n* на простоту – перевірка подільності *n* на числа 2, 3, … $\lfloor\sqrt{n}\rfloor$ (якщо *n* розкладається на добуток двох чи більше співмножників, то один зі співмножників не перевищуватиме $\sqrt{n}$). Тут немає потреби заводити масив для зберігання простих чисел. Як тільки задане число *n* поділиться на одне з менших чисел з проміжку $[2; p]$, де $p^2 \le n$ – це означатиме, що воно є складеним. Якщо число *n* не поділиться на жодне з менших чисел з проміжку $[2; p]$, то *n* – просте число.

На вході (`Enter n`). Кожен рядок є окремим тестом для визначення того, чи число *n*, введене користувачем, є простим числом. Ознакою завершення введення вхідних даних є рядок, в якому це число буде не більшим за 1.

На виході. Якщо число *n*, введене користувачем, є простим числом, то виводитиметься повідомлення (`Is primary number`), у протилежному випадку – (`Is Not primary number`).

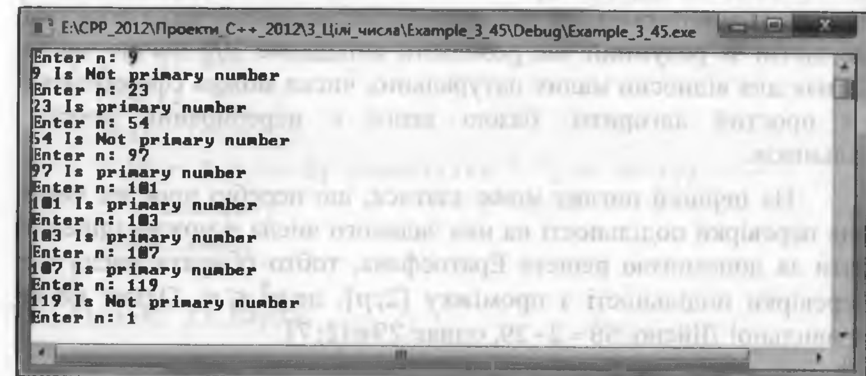
Програма:

```
#include <iostream>
using namespace std;
//---------------------------------------------------------
bool Is_Prime(const int number)
{ // Визначає, чи аргумент number є простим числом
```



```
  // Передумова: number - довільне натуральне число > 1
  // Постумова: повертає true, якщо number є простим
  // числом; інакше - false
 short int p = 2;
 while (p*p <= number)
  { if(number % p == 0) return false; else p++; }
 return true;
}
//---------------------------------------------------------
void main()
{
  int i,j,n;
  while (cout<<"Enter n: ", cin>>n, n>1)
  {
   if(Is_Prime(n))
   cout<<n<< " Is primary number "<<endl;
   else cout<<n<< " Is Not primary number "<<endl;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:



Час роботи T алгоритму перевірки на простоту числа n, наведеного у прикладі 46, *експоненентно* залежить від довжини запису n у пам'яті комп'ютера (тобто $T = \Theta(2^{\beta/2})$, де $\beta = \log(n+1)$). Отже, застосовувати цей алгоритм можна до відносно малих натуральних чисел, чи до тих чисел, що мають малі дільники.

Для практичних застосувань винайдені алгоритми, які з імовірністю α близькою до 1 дають правильні відповіді щодо простоти великих чисел за *поліноміальний* час. Однак платою за це є можливість отримання неправильної відповіді з ймовірністю $1 - \alpha$. Хоча строго доведено, що ймовірність $1 - \alpha \to 0$, однак похибка все ж таки можлива. Найвідомішим з цих алгоритмів є імовірнісний алгоритм (тест) Міллера-Рабіна.

Наступним важливим питанням, яке ми розглянемо, буде питання розкладу довільного натурального числа n на прості множники (або питання *факторизації* натурального числа n).

З теорії чисел відомо, що будь-яке складене натуральне число можна єдиним чином відобразити у вигляді добутку *простих* співмножників, наприклад:

$$48 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 3;\ 225 = 3 \cdot 3 \cdot 5 \cdot 5;\ 1050 = 2 \cdot 3 \cdot 5 \cdot 5 \cdot 7.$$

Розкладання дуже великого натурального числа на прості множники є *складнішою* задачею, порівняно із задачею перевірки цього числа на простоту. Навіть найпотужніші комп'ютери, які використовують сьогодні найдосконаліші на даний час алгоритми, не здатні за розумний час розкласти випадкове 200-значне число. Однак для відносно малих натуральних чисел можна сформулювати простий алгоритм, базою якого є перебирання простих дільників.

На перший погляд може здатися, що перебір простих чисел для перевірки подільності на них заданого числа n можна здійснювати за допомогою решета Ератосфена, тобто обирати числа для перевірки подільності з проміжку $[2; p]$, де $p^2 \leq n$. Однак це не правильно! Дійсно, $58 = 2 \cdot 29$, однак $29 \notin [2; 7]$.

Насправді ж перебір простих чисел i для перевірки подільності на них числа n, необхідно здійснювати на проміжку $[2; n/2]$, де $n/2$ – можливий найбільший дільник n.

Приклад 46. Скласти програму, яка розкладатиме довільне натуральне число n на прості множники.

➢ *Попередні міркування*. Тут немає потреби заводити масив для зберігання співмножників. Як тільки задане число n поділиться на одне з менших чисел i з проміжку $[2; n/2]$, то число i одразу ж виведемо на консоль, а число n, зменшимо в i разів. Конкретне просте число i застосовуватимемо для перевірки доти, доки n ділитиметься на нього.

На вході (`Enter n`). Кожен рядок є окремим тестом для визначення простих множників числа n, введеного користувачем. Ознакою завершення введення вхідних даних є рядок, в якому це число буде не більшим за 1.

На виході (`n:`). Для кожного числа n через пропуск виводиться список простих чисел, які є співмножниками цього числа. Якщо список порожній, то це означатиме, що n – просте число.

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------------------
void main()
{
  int i,n,m; // m - верхня межа перебору дільників
  while (cout<<"Enter n: ", cin>>n, n>1)
  { i = 2; m=n;
    cout<< "n: ";
    while (i<=m/2)
      if(n % i == 0) {cout<<i<< " "; n /= i;}
       else i++;
    cout<<endl;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цілі_числа...
Enter n: 48
n: 2 2 2 2 3
Enter n: 45
n: 3 3 5
Enter n: 13
n:
Enter n: 1050
n: 2 3 5 5 7
Enter n: 1
```

Якщо виникає потреба визначити усі дільники деякого натурального числа *n* (як прості, так і складені), то головну функцію необхідно записати у такому вигляді:

```
void main()
{ int i,n;
  while (cout<<"Enter n: ", cin>>n, n>1)
   {i = 2; cout<< "n: ";
    for (i = 2; i<=n/2; i++) if(n%i==0) cout<<i<< " ";
  cout<<endl;
}
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\3_Цілі_числа\Example_3_4...
Enter n: 96
n: 2 3 4 6 8 12 16 24 32 48
Enter n: 13
n:
Enter n: 45
n: 3 5 9 15
Enter n: 1
```


3.7.4. Виокремлення та опрацювання цифр числа

Для отримання цифр числа використовуємо операцію отримання залишку (`%`) від цілочислового ділення числа на `10`. Під час першого ділення отримаємо цифру з розряду одиниць (наприклад, `167 % 10 = 7`). Операція цілочислового ділення вилучає цю цифру з розряду одиниць (`167 / 10 = 16`), а наступна операція отримання залишку поверне число десятків (`16 % 10 = 6`) і т. д.

Зауважимо, що при цьому отримуємо цифри числа у зворотному порядку. Цей факт можна використати при розв’язанні відомої задачі про знаходження *паліндрома* – числа, величина якого не змінюється, якщо порядок цифр у його записі змінити на протилежний.

Приклад 47. Дано натуральне число *n*. Знайти кількість і суму десяткових цифр у записі цього числа.

➢ *На вході* (`Enter n`). Кожен рядок є окремим тестом для визначення кількості та суми десяткових цифр у записі числа *n*, введеного користувачем. Ознакою завершення введення вхідних даних є рядок, в якому це число буде не більшим за 1.

На виході. Для кожного числа *n* виводиться кількість (`Kilkist cyfr`) та сума (`Suma cyfr`) десяткових цифр у його записі.

Програма:

```
#include <iostream>
using namespace std;
//--------------------------------------------------------
void main()
{
  unsigned int n; unsigned short s=0, k=0;
  while (cout<<"Enter n: ", cin>>n, n>1)
  { while (n>0)
     {s += n % 10; k++; n /= 10;}
    cout<< "Suma cyfr = " << s;
    cout<< " Kilkist cyfr = " << k << endl;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
Enter n: 123
Suma cyfr = 6  Kilkist cyfr = 3
Enter n: 34567
Suma cyfr = 31  Kilkist cyfr = 8
Enter n: 1
```

Приклад 48. Визначити, чи натуральне число *n* є паліндромом?

➢ *На вході* (Enter n). Кожен рядок є окремим тестом для визначення того, чи число *n*, введене користувачем, є паліндромом. Ознакою завершення введення вхідних даних є рядок, в якому це число буде не більшим за 1.

На виході. Якщо число *n* – паліндром, то виводиться повідомлення (Palindrom), а інакше – (Ne palindrom).

Програма:

```
#include <iostream>
using namespace std;
//----------------------------------------------------
void main()
{
  unsigned int n, m,s;
  while (cout<<"Enter n: ", cin>>n, n>1)
  {
        m=n; // копія числа
        s=0;
        while (n>0)
        {s=s*10+n%10; n /= 10;}
        if( m==s) cout<< "Palindrom " << endl;
         else cout<< "Ne palindrom " << endl;
  }
  cin>>n; // Пауза
}
```


Результати тестування програми:

```
Palindrom
Enter n: 3333
s= 3333
Palindrom
Enter n: 6776
s= 6776
Palindrom
Enter n: 123321
s= 123321
Palindrom
Enter n:
12345
s= 54321
Ne palindrom
Enter n: 1
```

Подібний алгоритм використовують і під час представлення цілого десяткового числа в іншій системі числення. Пригадайте *правило переведення*: обчислюємо остачу від ділення певного числа на нову основу, отриману частку ділимо знову на нову основу – отримуємо наступну остачу і так далі доти, доки чергова частка дорівнюватиме нулю. Отримані остачі у зворотному порядку формують число за новою основою.

Приклад 49. Дано десяткове натуральне число *n*. Отримати його зображення у вісімковій системі числення.

➢ *На вході* (Enter n). Кожен рядок є окремим тестом для переведення числа *n*, введеного користувачем, з десяткової системи числення у вісімкову. Ознакою завершення введення вхідних даних є рядок, в якому це число буде не більшим за 1.

На виході (8-е chyslo=) отримуємо число *n*, зображене у вісімковій системі числення.

Програма:

```
#include <iostream>
using namespace std;
//-------------------------------------------------------------
void main()
{
  unsigned int n, m,s;
  while (cout<<"Enter n: ", cin>>n, n>1)
  {
      m=1; // - степінь 10; спочатку 10^0=1
      s=0;
      while (n>0)
       { s=s+(n%8)*m; n /= 8; m*=10;}
      cout<< "8-e chyslo= "<<s<<endl;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\3_Цілі_числа\Exa...
Enter n: 7
8-e chyslo= 7
Enter n: 8
8-e chyslo= 10
Enter n: 16
8-e chyslo= 20
Enter n: 234
8-e chyslo= 352
Enter n: 1
```


? Запитання для самоперевірки

1. Що таке інструкція виразу?
2. Що таке порожня інструкція?
3. Які форми має умовна інструкція?
4. Порівняйте операцію умови та умовну інструкцію.
5. Для яких задач зручно використовувати інструкцію вибору?
6. Запишіть і охарактеризуйте інструкцію циклу `while`.
7. Запишіть і охарактеризуйте інструкцію циклу `do … while`.
8. Запишіть і охарактеризуйте інструкцію циклу `for`.
9. У яких випадках застосовують інструкцію `break`?
10. У яких випадках застосовують інструкцію `continue`?
11. У яких випадках застосовують інструкцію `goto`?
12. З якою метою використовують змінну стану програми?
13. Як організувати тестування простої програми?
14. Що таке псевдовипадкове число?
15. З якою метою використовують функцію `rand()`?
16. З якою метою використовують функцію `srand()`?
17. Як зробити випадковим `seed` (параметр функції `srand()`)?
18. Запишіть фрагмент програми одержання псевдовипадкових цілих чисел на довільному проміжку $[a; b]$.
19. Що називають рекурентним співвідношенням?
20. Запишіть загальну формулу для поточного елемента рекурентного співвідношення, якщо відоме одне значення попереднього елемента.
21. Запишіть загальну формулу для поточного елемента рекурентного співвідношення, якщо відомі два значення попередніх елементів.
22. Як визначити рекурентний множник?
23. Що називають числовим рядом?
24. Що називають частковою сумою ряду?
25. Сформулюйте необхідну умову збіжності числового ряду.
26. Сформулюйте достатню ознаку Д'Аламбера збіжності числового ряду.
27. Сформулюйте достатню ознаку Коші збіжності числового ряду.
28. Що називають функціональним рядом?
29. Що називають областю збіжності функціонального ряду?
30. Що називають залишком функціонального ряду?

31. Що називають степеневим рядом?
32. Що називають узагальненим степеневим рядом?
33. Сформулюйте ознаку Абеля збіжності степеневого ряду.
34. Як визначити радіус збіжності степеневого ряду за ознакою Д'Аламбера?
35. Як визначити радіус збіжності степеневого ряду за ознакою Коші?
36. Що називають рядом Маклорена?
37. Що називають рядом Тейлора?
38. Запишіть формулу числового інтегрування методом прямокутників.
39. Запишіть формулу числового інтегрування методом трапецій.
40. Запишіть формулу числового інтегрування методом Сімпсона.
41. Що означає принцип Рунге для числового інтегрування?
42. Що називають проміжком ізоляції розв'язків?
43. Сформулюйте теорему, на якій базується метод визначення проміжків ізоляції.
44. Для чого використовують метод ділення проміжку ізоляції навпіл?
45. Що називають рекурсивною функцією?
46. Перелічіть загальні методи конструювання алгоритмів.
47. Порівняйте ітераційні та рекурсивні процеси.
48. У чому полягає загальний метод зменшення розміру задачі на постійну величину?
49. У чому полягає загальний метод зменшення розміру задачі на постійний множник?
50. Що таке стек?
51. Опишіть реалізацію рекурсивної функції за допомогою стека.
52. Як реалізуються рекурсивні обчислення на рекурсивному поверненні?
53. Як реалізуються рекурсивні обчислення на рекурсивному виклику?
54. Дайте означення найбільшого спільного дільника двох чисел (НСД).
55. Дайте означення найменшого спільного кратного двох чисел (НСК).
56. Опишіть алгоритм Евкліда визначення НСД двох чисел.
57. Як визначити НСК двох чисел, якщо відомий їхній НСД.
58. Опишіть алгоритм Ератосфена визначення простих чисел.
59. Опишіть спосіб отримання цифри із зображення числа?
60. Опишіть спосіб переведення числа з однієї системи числення в іншу?

Завдання для програмування

Завдання 1. Створити консольну програму розв'язання таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи). Кількість чисел, які вводитимуть з клавіатури консолі, отримати від користувача у діалоговому режимі.

1. Ввести з клавіатури групу чисел і визначити серед них найменше число та номер його першого входження у групу.
2. Ввести з клавіатури групу чисел і визначити серед них найбільше число та номер його останнього входження у групу.
3. Ввести з клавіатури групу чисел і обчислити серед них суму всіх невід'ємних значень.
4. Ввести з клавіатури групу чисел і визначити серед них кількість додатних, від'ємних і нульових чисел.
5. Ввести з клавіатури групу чисел і обчислити серед них суму чисел, що належать проміжку [–3; 5].
6. Ввести з клавіатури групу чисел і визначити серед них відсоток додатних чисел і відсоток нульових чисел.
7. Ввести з клавіатури групу чисел. Визначити серед них середнє арифметичне чисел, розташованих за порядком за першим ненульовим. Вважати, що обов'язково є хоча б одне ненульове значення, причому не на останньому місці.
8. Ввести з клавіатури групу чисел. Визначити серед них середнє арифметичне чисел, розташованих за порядком за першим ненульовим. Вважати, що ненульових значень може не бути взагалі.
9. Ввести з клавіатури групу чисел. Визначити серед них різницю між найбільшим і найменшим числом.
10. Ввести з клавіатури групу чисел і визначити серед них середнє арифметичне чисел, що належать проміжку [–5; 5].
11. Ввести з клавіатури групу чисел. Визначити серед них перше за порядком число, що не дорівнює нулю, та його номер у групі.
12. Ввести з клавіатури групу чисел. Визначити порядковий номер першого нульового числа та відсоток чисел у групі, які стоять перед ним.
13. Ввести з клавіатури групу чисел. Визначити серед них останнє за порядком число, що не дорівнює нулю, та його номер у групі.
14. Ввести з клавіатури групу чисел. Визначити порядковий номер останнього нульового числа та відсоток чисел у групі, які стоять після нього.
15. Ввести з клавіатури групу чисел. Визначити суму модулів тих чисел, які передують першому нульовому, або усіх чисел групи, якщо нульових чисел немає.

Завдання 2. Створити консольну програму розв'язання задач із **завдання 1** (задачу обрати згідно з номером студента у списку студентів підгрупи). Тестування програми реалізувати за допомогою *псевдовипадкових* чисел (кількість цих чисел отримати від користувача у діалоговому режимі).

Завдання 3. Створити консольну програму табулювання значення функції із **завдання 2.1**[1] (функцію обрати згідно з номером студента у списку студентів підгрупи) на проміжку $[a; b]$ з кроком h (значення параметрів a, b і h отримати від користувача у діалоговому режимі; при цьому контролювати виконання умов $a < b$, $h > 0$). Під час табулювання функції перевіряти приналежність аргументу до області визначення функції.

Завдання 4. Створити консольну програму табулювання значення функції із **завдання 2.2** (функцію обрати згідно з номером студента у списку студентів підгрупи) на проміжку $[a; b]$ з кроком h (значення параметрів a, b і h отримати від користувача у діалоговому режимі; при цьому контролювати виконання умов $a < b$, $h > 0$). Під час табулювання функції перевіряти умови, які забезпечуватимуть можливість застосування математичних функцій..

Завдання 5. Створити консольну програму обчислення скінченної суми (задачу обрати згідно з номером студента у списку студентів підгрупи). Значення параметра n отримати від користувача у діалоговому режимі (з відповідним контролем).

1. $$s = \sum_{i=2}^{n} \frac{i^3 + 2}{(i-3)(i-6)}.$$

2. $$s = \sum_{i=1}^{n} \frac{3i^2 + 2}{(i-1)(i-5)}.$$

3. $$s = \sum_{i=2}^{n} \frac{2i^3 + 5}{(i-4)(i-7)}.$$

4. $$s = \sum_{i=0}^{n} \frac{5i^2 + 2i - 3}{(i-1)(i-4)}.$$

5. $$s = \sum_{i=1}^{n} \frac{4i^2 + 3i - 1}{(i-3)(i-5)}.$$

6. $$s = \sum_{i=2}^{n} \frac{5i^3 - 2i + 4}{(i-2)(i-6)}.$$

7. $$s = \sum_{i=1}^{n} \frac{5i^2 + 2i^2 + 3}{(i-1)(i-4)}.$$

8. $$s = \sum_{i=0}^{n} \frac{i^2 + 2i + 7}{(i-3)(i-6)}.$$

9. $$s = \sum_{i=-2}^{n} \frac{i^2 + 2 \cdot i + 2}{(i+1)(i-3)}.$$

10. $$s = \sum_{i=-3}^{n} \frac{i^2 + 2 \cdot i + 3}{(i+1)(i+3)}.$$

11. $$s = \sum_{i=-2}^{n} \frac{i^2 - 2 \cdot i + 4}{(i+3)(i+1)}..$$

12. $$s = \sum_{i=-4}^{n} \frac{i^2 + 5 \cdot i + 3}{(i+4)(i-3)}.$$

13. $$s = \sum_{i=-2}^{n} \frac{3i^2 + 7 \cdot i + 4}{(i+1)(i-2)}.$$

14. $$s = \sum_{i=-3}^{n} \frac{i^2 + 4 \cdot i + 1}{(i+2)(i-1)}.$$

15. $$s = \sum_{i=-2}^{n} \frac{i^2 + 8 \cdot i + 3}{(i+1)(i-4)}.$$

16. $$s = \sum_{i=-4}^{n} \frac{i^2 + 3 \cdot i + 9}{(i+5)(i-7)}.$$

[1] Завдання 1 з розділу 2.

Завдання 6. Створити консольну програму обчислення скінченного добутку (задачу обрати згідно з номером студента у списку студентів підгрупи). Значення параметра n отримати від користувача у діалоговому режимі (з відповідним контролем).

1. $$p = \prod_{i=2}^{n} \frac{i^3 - 2}{(i-3)(i-6)}.$$

2. $$p = \prod_{i=1}^{n} \frac{3i^2 - 2}{(i-1)(i-5)}.$$

3. $$p = \prod_{i=2}^{n} \frac{2i^3 - 5}{(i-4)(i-7)}.$$

9. $$p = \prod_{i=-2}^{n} \frac{i^2 - 2 \cdot i - 5}{(i+1)(i-3)}.$$

10. $$p = \prod_{i=-3}^{n} \frac{i^2 + 2 \cdot i - 3}{(i+1)(i+3)}.$$

11. $$p = \prod_{i=-2}^{n} \frac{i^2 + 2 \cdot i - 5}{(i+3)(i+1)}.$$

4. $s=\sum_{i=0}^{n}\frac{5i^2-2i+3}{(i-1)(i-4)}$.

12. $p=\prod_{i=-4}^{n}\frac{i^2+5\cdot i-3}{(i+4)(i-3)}$.

5. $p=\prod_{i=1}^{n}\frac{4i^2-3i+1}{(i-3)(i-5)}$.

13. $p=\prod_{i=-2}^{n}\frac{3i^2+7\cdot i+4}{(i+1)(i-2)}$.

6. $p=\prod_{i=2}^{n}\frac{5i^3+2i-4}{(i-2)(i-6)}$.

14. $p=\prod_{i=-3}^{n}\frac{i^2-4\cdot i+1}{(i+2)(i-1)}$.

7. $p=\prod_{i=1}^{n}\frac{5i^2-2i^2-3}{(i-1)(i-4)}$.

15. $p=\prod_{i=-2}^{n}\frac{i^2+8\cdot i-3}{(i+1)(i-4)}$.

8. $p=\prod_{i=0}^{n}\frac{i^2-2i+7}{(i-3)(i-6)}$.

16. $p=\prod_{i=-4}^{n}\frac{i^2-3\cdot i+9}{(i+5)(i-7)}$.

Завдання 7. Створити консольну програму обчислення скінченної суми (задачу обрати згідно з номером студента у списку студентів підгрупи). Значення параметра x отримати від користувача у діалоговому режимі. Для *частини* загального члена необхідно обов'язково визначати рекурентну формулу (очевидно, що це стосуватиметься піднесення до степеня k чи обчислення факторіала.

Увага! Орієнтиром під час виконання цього завдання може слугувати виконання **прикладу 25**.

1. $\sum_{k=1}^{9}\frac{\sin(2kx)}{(2k)!}$.

9. $\sum_{k=1}^{12}x^k\cos^k\left(\frac{k}{4}\right)$.

2. $\sum_{k=3}^{14}\frac{k^2\cos^k x}{\left(2k^2-1\right)}$.

10. $\sum_{k=2}^{11}\frac{k^2\sin^k x}{\left(2k^2-1\right)}$.

3. $\sum_{k=1}^{12}\frac{\cos^k(2x)}{k}$.

11. $\sum_{k=1}^{8}\frac{k^2x^k}{(2k)!}$.

4. $\sum_{k=1}^{6}\frac{(-1)^k}{k^2}x^k$.

12. $\sum_{k=1}^{11}\frac{kx^k}{4k^2-1}$.

5. $\sum_{k=1}^{10}k(2k+1)x^k$.

13. $\sum_{k=1}^{10}\frac{x^{2k}}{2k(2k-1)}$.

6. $\sum_{k=1}^{11}\frac{x^{2k+1}}{4k^2-1}$.

14. $\sum_{k=1}^{8}\frac{x^{2k-1}}{2^k(2k-1)}$.

7. $\sum_{k=2}^{11}\frac{(k-1)\cdot x^k}{(k+1)\cdot 3^k}$.

15. $\sum_{k=2}^{9}\frac{x^{2k+1}}{(2k+1)\cdot 2^k}$.

8. $\sum_{k=1}^{10}k(2k-1)x^{k+1}$.

16. $\sum_{k=1}^{11}\frac{kx^{k-1}}{4k^2+1}$.

Завдання 8. Створити консольну програму обчислення скінченної суми, вважаючи x параметром, значення якого одержують від користувача. Для *усього* загального члена необхідно обов'язково визначити рекурентну формулу. Постійні величини необхідно виносити за знак суми.

1. $\sum_{k=1}^{7}\frac{(-1)^k x^{2k}}{k!}$.

9. $\sum_{k=1\,1}^{9}\frac{x^k}{k!}$.

2. $\sum_{k=1}^{9}\frac{x^{k+1}}{(k+1)!}$.

10. $\sum_{k=1}^{8}\frac{(-1)^k x^{2k}}{(2k)!}$.

3. $\sum_{k=3}^{11}\frac{x^{2k}}{(2k-1)!}$.

11. $\sum_{k=1}^{12}\frac{\sin^k(x)}{k!}$.

4. $\sum_{k=1}^{7}\frac{(x\ln 3)^k}{2^k}$.

12. $\sum_{k=1}^{11}\frac{(x+1)^{2k}}{(2k)!}$.

5. $\sum_{k=1}^{7}\frac{(-1)^k x^k}{2^k k!}$.

13. $\sum_{k=2}^{10}\frac{x^{2k-1}}{2(k-1)!}$.

6. $\sum\limits_{k=1}^{17} \frac{(-1)^k x^k}{k!}$.

7. $\sum\limits_{k=2}^{9} (-1)^k x^{2k}$.

8. $\sum\limits_{k=1}^{12} \frac{(x-1)^{2k}}{(x+1)^k}$.

14. $\sum\limits_{k=0}^{9} \frac{x^{k-2}}{(k+2)!}$.

15. $\sum\limits_{k=0}^{16} \frac{(x-1)^k}{2k!}$.

16. $\sum\limits_{k=2}^{9} \frac{(-1)^k x^{k-1}}{(k-1)!}$.

Завдання 9. Обчислити значення ряду 2очністю ε $(\varepsilon > 0)$, вважаючи x параметром, значення якого одержують від користувача. Для *усього* загального члена необхідно обов'язково визначити рекурентну формулу. Постійні величини необхідно виносити за знак суми. З метою уникнення можливого зациклення під час обчислення ряду необхідно ввести величину максимальної кількості ітерацій, яка слугуватиме індикатором для аварійного виходу з циклу.

1. $\sum\limits_{k=1}^{\infty} \frac{(-1)^k x^{2k}}{k!}$.

2. $\sum\limits_{k=1}^{\infty} \frac{x^{k+1}}{(k+1)!}$.

3. $\sum\limits_{k=1}^{\infty} \frac{x^{2k}}{(2k-1)!}$.

4. $\sum\limits_{k=1}^{\infty} \frac{(x\ln 3)^k}{2^k}$.

5. $\sum\limits_{k=1}^{\infty} \frac{(-1)^k x^k}{2^k k!}$.

9. $\sum\limits_{k=1}^{\infty} \frac{x^k}{k!}$.

10. $\sum\limits_{k=1}^{\infty} \frac{(-1)^k x^{2k}}{(2k)!}$.

11. $\sum\limits_{k=1}^{\infty} \frac{\sin^k(x)}{k!}$.

12. $\sum\limits_{k=1}^{\infty} \frac{(x+1)^{2k}}{(2k)!}$.

13. $\sum\limits_{k=1}^{\infty} \frac{x^{2k-1}}{2(k-1)!}$.

6. $\sum\limits_{k=0}^{\infty} \frac{(-1)^k x^k}{k!}$.

7. $\sum\limits_{k=1}^{\infty} (-1)^k x^{2k}$.

8. $\sum\limits_{k=0}^{\infty} \frac{(x-1)^{2k}}{(x+1)^k}$.

14. $\sum\limits_{k=0}^{\infty} \frac{x^{k-2}}{(k+2)!}$.

15. $\sum\limits_{k=0}^{\infty} \frac{(x-1)^k}{2k!}$.

16. $\sum\limits_{k=1}^{\infty} \frac{(-1)^k x^{k-1}}{(k-1)!}$.

Завдання 10. Створити консольну програму табулювання значення функції (функцію обрати згідно з номером студента у списку студентів підгрупи) на проміжку $[a;\ b]$ з кроком h (значення параметрів a, b і h отримати від користувача у діалоговому режимі; при цьому необхідно контролювати виконання умов $a < b$ $h > 0$). Під час табулювання функції перевіряти приналежність аргументу до області визначення функції. Обчислення функції *реалізувати* за допомогою її розкладу у степеневий ряд. Обчислені значення *порівнювати* зі значеннями відповідних бібліотечних функцій. З метою уникнення можливого зациклення під час обчислення ряду необхідно ввести величину максимальної кількості ітерацій, яка слугуватиме індикатором для аварійного виходу з циклу.

1. $y = \ln\frac{x+1}{x-1} = 2\sum\limits_{n=0}^{\infty}\frac{1}{(2n+1)x^{2n+1}};\ |x| > 1.$

2. $y = e^{-x} = \sum\limits_{n=0}^{\infty}\frac{(-1)^n \cdot x^n}{n!};\ |x| < \infty.$

3. $y = \ln(1-x) = -\sum\limits_{n=0}^{\infty}\frac{x^n}{n};\ \ -1 \le x < 1.$

4. $y = \ln\frac{x+1}{1-x} = 2\sum\limits_{n=0}^{\infty}\frac{x^{2n+1}}{2n+1};\ \ |x| < 1.$

5. $y = \operatorname{arctg} x = \frac{\pi}{2} + \sum\limits_{n=0}^{\infty}\frac{(-1)^{n+1}}{(2n+1)x^{2n+1}};\ |x| > 1.$

6. $y = \operatorname{arcctg} x = \dfrac{\pi}{2} + \sum\limits_{n=0}^{\infty} \dfrac{(-1)^{n+1} \cdot x^{2n+1}}{2n+1}$; $|x| \le 1$.

7. $y = \operatorname{arctg} x = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^{n} \cdot x^{2n+1}}{2n+1}$; $|x| \le 1$.

8. $y = \cos x = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^{n} \cdot x^{2n}}{(2n)!}$; $|x| < \infty$.

9. $y = e^x = \sum\limits_{n=0}^{\infty} \dfrac{x^n}{n!}$; $|x| < \infty$.

10. $y = \dfrac{\sin x}{x} = \sum\limits_{n=0}^{\infty} \dfrac{(-1)^{n} \cdot x^{2n}}{(2n+1)!}$; $|x| < \infty$.

11. $\dfrac{1}{1-x} = 1 + x + x^2 + ... + x^n ...$; $x \in (-1;\ 1)$.

12. $\ln(1+x) = x - \dfrac{x^2}{2} + \dfrac{x^3}{3} - \dfrac{x^4}{4} + ... + (-1)^{(n-1)} \dfrac{x^n}{n!} + ...$; $x \in (-1;\ 1]$.

13. $\operatorname{arctg} x = x - \dfrac{x^3}{3} + \dfrac{x^5}{5} - \dfrac{x^7}{7} + ... + (-1)^n \dfrac{x^{2n+1}}{2n+1} + ..$; $x \in [-1;\ 1]$.

14. $\sqrt[4]{x+1} = 1 + \dfrac{x}{4} - \dfrac{1 \cdot 3 \cdot x^2}{4 \cdot 8} + \dfrac{1 \cdot 3 \cdot 7 \cdot x^3}{4 \cdot 8 \cdot 12} - \dfrac{1 \cdot 3 \cdot 7 \cdot 11 \cdot x^4}{4 \cdot 8 \cdot 12 \cdot 16} + ..$; $x \in [-1;\ 1]$.

15. $\sqrt{1-x} = 1 - \dfrac{x}{2} - \dfrac{1 \cdot 1 \cdot x^2}{2 \cdot 4} + \dfrac{1 \cdot 1 \cdot 3 \cdot x^3}{2 \cdot 4 \cdot 6} - \dfrac{1 \cdot 1 \cdot 3 \cdot 5 \cdot x^4}{2 \cdot 4 \cdot 6 \cdot 8} + ..$; $x \in [-1;\ 1]$.

Завдання 11. Створити консольну програму обчислення виаченого інтеграла $\int\limits_a^b f(x)dx$ за допомогою числового інтегрування методом *прямокутників* з автоматичним вибором кроку. Функцію необхідно обирати із **завдання 2.1** (функцію обрати згідно з номером студента у списку студентів підгрупи). Значення меж інтегрування a і b отримати від користувача у діалоговому режимі; при цьому контролювати виконання умови $a < b$, а також приналежність a і b до області визначення функції).

Завдання 12. Виконати завдання 11, замінивши фразу "методом *прямокутників*" на фразу "методом *трапецій*".

Завдання 13. Виконати завдання 11, замінивши фразу "методом *прямокутників*" на фразу "методом *Сімпсона*".

Завдання 14. Визначити проміжок ізоляції хоча б одного розв'язку рівняння (це рівняння обрати згідно з номером студента у списку студентів підгрупи) та уточнити його за допомогою методу ділення проміжку ізоляції навпіл з точністю *eps* (задається користувачем).

1. $\cos^3(2x) - \log_2(x^2 - 2) = 0$.
2. $\sin^4(3x) - 2e^{x-2} + \sqrt[4]{x-3} = 0$.
3. $\cos^2(2x) - \sin x + \ln(4 - x^2) = 0$.
4. $\sin^3(3x) - \cos x + \sqrt{9 - x^2} = 0$.
5. $\sqrt{x^2 - 4} - 2\log_5(x^2 + 1) = 0$.
6. $\sqrt{x^3 - 3x} + 2\sin x = 0$.
7. $(x^2 - 2x)^3 - 4\log_3(x^3 - 4) = 0$.
8. $\sqrt[3]{x^4 - 3x^3} - 2\sin x = 0$.
9. $2^{\ln(x^3 + 2x)} - 4\cos x = 0$.
10. $3^{x^3 - 2x} + 4\log_4(x^2 - 9) = 0$.
11. $\sin^{\frac{2}{5}}(3x) + \cos x + \sqrt{9 - x^2} = 0$.
12. $\sin^{\frac{4}{7}}(3x) + 2\cos x + \ln(4 - x^2) = 0$.
13. $\sqrt[4]{x^5 - 3x^3} + 2\sin x = 0$.
14. $\sqrt[4]{x^5 - 5x^3} - 4x^{\frac{2}{3}} = 0$.
15. $\sqrt{x^3 - 5x} + 10\sin x = 0$.

Завдання 15. Записати рекурсивну та ітераційну функцію обчислення заданої величини y. Використовуючи ці функції, обчислити декілька значень y для різних $n > 5$ (та різних m за потреби) і порівняти результати.

1. $y = \cfrac{1}{2n + \cfrac{1}{2n - 2 + \cfrac{1}{2n - 4 + \ldots + \cfrac{1}{2}}}}$.

2. $y = \sqrt{1 + \sqrt{3 + \sqrt{5 + \ldots \sqrt{2n+1}}}}$.

3. $y = \cfrac{1}{2 + \cfrac{1}{4 + \cfrac{1}{6 + \ldots + \cfrac{1}{2n}}}}$.

4. $y = \sqrt{2 + \sqrt{4 + \sqrt{6 + \ldots \sqrt{2n}}}}$.

5. $y = x_n = 3x_{n-1} + 5;\ x_0 = -3$.

6. $y = \ln(2 + \ln(4 + \ldots + \ln(2n)))$.

7. $y = \ln(3 + \ln(5 + \ldots + \ln(2n+1)))$.

8. $y = \begin{cases} 1, & n = 1; \\ \sum_{i=2}^{n} f\left(\frac{n}{i}\right), & n > 2. \end{cases}$

9. $y = \begin{cases} 1, & n = 1; \\ \prod_{i=2}^{n} f\left(\frac{n}{i}\right), & n > 2. \end{cases}$

10. $y = \cfrac{1}{1 + \cfrac{1}{2 + \cfrac{1}{3 + \ldots + \cfrac{1}{n}}}}$.

11. $y = \begin{cases} 1, & n = 1; \\ \sum_{i=2}^{n} f\left(\frac{n^2}{i}\right), & n > 2. \end{cases}$

12. $y = \begin{cases} 1, & n = 1; \\ \prod_{i=2}^{n} f\left(\frac{3}{n \cdot i}\right), & n > 2. \end{cases}$

13. $y = x_n = 5x_{n-1};\ x_0 = -4$.

14. $y = x_n = 2x_{n-1};\ x_0 = 1$.

15. $y = A(m, n) = \begin{cases} n + 1, & m = 0; \\ A(m-1, 1), & m > 0, n = 0; \\ A(m-1, A(m, n-1)), & m > 0, n > 0. \end{cases}$

Завдання 16. Створити консольну програму розв'язання таких задач:

1. На проміжку $[a; b]$, де a і b – натуральні числа, вивести всі трійки Піфагорових чисел (тобто, $k^2 + l^2 = m^2$).
2. Обчислити суму часток та суму остач таких виразів: $(5p^2+3)/(3p)$, $p=1,\ldots,10$.
3. Обчислити середнє арифметичне остач таких виразів: $(4p^2+2)/(2p)$, $p=1,\ldots,12$.
4. Нехай у деякій країні є купюри номіналом по 3 і 5 одиниць грошей. Довести, що довільну грошову суму $n > 7$ можна виплатити цими купюрами. Визначити цілі a і b, що $n = 3a+5b$.
5. Вивести на консоль усі числа, які менші за n і діляться на m, потім –ті, які діляться на $m - 2$, $m - 4$,..., діляться на 2.
6. Дано ціле $m > 1$. Отримати усі прості k, за яких $m < 4k < 2m$.
7. Визначити всі натуральні числа, які не перевищують заданого n і діляться на кожну зі своїх цифр.
8. Задано натуральне число n. З'ясувати, чи є серед чисел n, $n+1$, ..., $2n$ числа-близнюки, тобто прості числа, різниця між якими дорівнює двом.
9. Визначити натуральне число від 1 до 10 000 з максимальною сумою дільників.
10. Натуральне число називається досконалим, якщо воно дорівнює сумі всіх своїх дільників, за винятком самого себе. Наприклад, число 6 – досконале, оскільки 6 = 1+2+3. Для заданого натурального числа n одержати всі досконалі числа, які менше за n.
11. Одержати у порядку зростання n перших натуральних чисел, які не діляться на жодне з чисел 2, 3 і 5.
12. Хлопчик Женя народився 2011 року. Відомо, що з дня його народження станом на *сьогодні* минуло n днів ($n > 500$). Визначити, скільки минуло років, місяців і днів хлопчику від дати його народження?
13. Дівчинка Наталка народилася 2009 року. Відомо, що 2.09.12 р. їй було 820 днів. Визначити дату її народження.
14. Годинникова стрілка утворює кут α ($0 < \alpha < 2\pi$) з променем, який проходить через центр циферблата, і точку, що відповідає 12 годині.

Визначити значення кута для хвилинної стрілки, а також кількість годин і повних хвилин.

15. Дано цілі числа *m* і *n* ($0 < m \le 12$; $0 \le n < 60$), які задають момент часу "*m* годин, *n* хвилин". Визначити кількість повних хвилин, які мають пройти до того моменту, коли стрілки співпадуть.

Завдання 17. Створити консольну програму розв'язання таких задач:

1. Дано десяткове натуральне число. Отримати його зображення у сімковій системі числення.
2. Дано вісімкове натуральне число. Отримати його зображення у десятковій системі числення.
3. Дано десяткове натуральне число. Отримати його зображення у трійковій системі числення.
4. Дано четвіркове натуральне число. Отримати його зображення у десятковій системі числення.
5. Дано натуральне число *n*. Чи містить воно три однакові цифри?
6. Дано натуральне число *n*. Чи є в ньому повторення цифр?
7. Дані натуральні числа *n* і *m*. Отримати суму *m* останніх цифр числа *n*.
8. Дано натуральне число *n*. Чи входить цифра 3 у зображення числа *n*. Переставити першу і останню цифри числа *n*.
9. Дано натуральне число. Знайти кількість і добуток десяткових цифр у записі цього числа.
10. Дано натуральне число. Знайти середнє арифметичне цифр у зображенні цього числа.
11. Дано натуральне число. Знайти першу за порядком позицію (зліва направо), у якій знаходиться найменша цифра у зображенні цього числа. Вивести також і цю найменшу цифру.
12. Дано натуральне число. Знайти останню за порядком позицію (зліва направо), у якій знаходиться найменша цифра у зображенні цього числа. Вивести також і цю найменшу цифру.
13. Дано натуральне число. Знайти першу за порядком позицію (зліва направо), у якій знаходиться найбільша цифра у зображенні цього числа. Вивести також і цю найбільшу цифру.
14. Дано натуральне число. Знайти останню за порядком позицію (зліва направо), у якій знаходиться найбільша цифра у зображенні цього числа. Вивести також і цю найбільшу цифру.
15. Ввести ціле додатне число. Визначити кількість одиниць у двійковому зображенні цього числа.

4. МАСИВИ ТА ВКАЗІВНИКИ

План викладу матеріалу:

1. Масиви.
2. Вказівники.
3. Взаємодія функцій та вказівників.
4. Багатомодульні проекти.
5. Стандартні задачі обробки одновимірних масивів.
6. Рядки символів у стилі С.

Ключові терміни розділу

- *Визначення масиву*
- *Вимірність масиву*
- *Ініціалізація масивів*
- *Особливості ініціалізації рядків*
- *Одновимірні масиви*
- *Багатовимірні масиви*
- *Вказівник. Види вказівників*
- *Оператор адресації*
- *Оператор рознайменування*
- *Оператор* `new`
- *Оператор* `delete`
- *Динамічні масиви*
- *Зв'язок масивів і вказівників*
- *Взаємодія функцій та вказівників*
- *Вказівник на функції*
- *Аргументи-масиви*
- *Багатомодульні проекти*
- *Роздільне компілювання модулів*
- *Шаблон багатомодульного консольного проекту*
- *Узагальнений алгоритм обчислення визначених інтегралів*
- *Сервісний модуль обробки масивів*
- *Стандартні задачі обробки одновимірних масивів*
- *Схема Горнера обчислення многочлена*
- *Прості методи сортування одновимірних масивів*
- *Сусіднє розміщення елементів*
- *Пошук в одновимірному масиві*
- *Злиття двох упорядкованих масивів*
- *Стандартні функції обробки рядків у стилі С*
- *Кодування символів*
- *Функції Windows API для рядків*
- *Відображення літер кирилиці*
- *Поняття локалі*
- *Таблиця кодувань CP866*
- *Таблиця і система кодувань CP1251*
- *Стандарт Unicode*
- *Функція* `CharToOem()`
- *Засоби локалізації у стилі С*
- *Функція* `SetConsoleCP()`
- *Функції класифікації символів*
- *Сканування формату рядка*
- *Логіка змінних стану*
- *Структурна логіка*

4.1. Масиви

Масив (*array*) – це колекція (чи набір) змінних однакового типу, значення яких зберігаються у послідовних осередках пам'яті. Звертання до змінних колекції (або *елементів масиву*) відбувається за допомогою назви, спільної для усіх них, і фіксованого набору спеціальних параметрів доступу – *індексів* масиву.

Залежно від кількості параметрів доступу (*індексів* масиву) розрізняють *одновимірні* та *багатовимірні* масиви (двовимірні, тривимірні ...). Одновимірний масив відображає такий математичний об'єкт, як *вектор*, а двовимірний масив – *матрицю*. У програмах найчастіше використовують одновимірні масиви.

4.1.1. Одновимірні масиви

Оскільки у цьому пункті розглядатимемо *тільки* одновимірні масиви, то слова *одновимірний* / *одновимірні* інколи опускатимемо.

У пам'яті комп'ютера (*фізичний* рівень) зберігаються *значення* елементів масиву (дані), причому важливим є те, що:

- значення елементів масиву розміщені *послідовно* один за одним;
- кожне значення елемента масиву займає у пам'яті комп'ютера однакову кількість байтів, оскільки *усі* елементи масиву мають *однаковий* тип;
- елементи масиву пронумеровано послідовними цілими числами, починаючи з *нуля*.

Ці три обставини дають змогу компіляторові дуже просто визначати адресу пам'яті довільного елемента одновимірного масиву. Якщо, наприклад, масив містить 10 елементів типу `int` (4 байти), то компілятор виокремлює 10 послідовних осередків пам'яті, які нумерує числами `0`; `1`; `2`; ...; `9`. Адресу i-го елемента $(i = \overline{0; 9})$ можна визначити за формулою $A_0 + 4 \cdot i$, де A_0 – адреса *нульового* елемента.

Логічний рівень масиву (чи *логічна* структура) пов'язаний з описом та використанням масиву у програмі. На логічному рівні *масив* – це послідовність даних, які мають спільну *назву* та спільний *тип*.

Визначення одновимірного масиву:

```
тип ідентифікатор[константний_вираз];
```

де `ідентифікатор` задає назву масиву, `константний_вираз` – кількість елементів (чи *розмір*) масиву, `тип` – тип елементів масиву.

Наприклад, інструкція опису

```
double W[20];
```

визначає одновимірний масив `W` типу `double` розміром 20 елементів (тобто масив `W` може зберігати 20 чисел раціонального типу подвійної точності). У пам'яті комп'ютера масив `W` займає 20×8 = 160 (байтів); елементи `W` матимуть номери `0`; `1`; ...; `19`.

Доступ до елементів масиву на логічному рівні задає вираз

```
назва_масиву[індекс],
```

де `індекс` – це ціле число, яке набуває значення від `0` до `(розмір_масиву - 1)`.

Найчастіше доступ до конкретних елементів масиву `W` здійснюють за допомогою виразу `W[i]`, де `i` – *індекс*, значеннями якого є цілі числа з проміжку [0; 19]. Наприклад, `W[0]` – значення першого елемента масиву , `W[1]` – другого, `W[19]` – останнього (20-го). На місці індексу `i` у `W[i]` можна записувати й складніші вирази, якот `W[2*i+1]` тощо.

Іноді *спрощують*, кажучи, що `W[0]` – значення нульового елемента масиву, `W[1]` – першого, `W[19]` – останнього (19-го). Тобто елементів 20, а останній – 19-й?! У попередньому абзаці теж не все добре: останній (20-й) елемент має 19-й номер?!

Який варіант правильніший? Відповідь – жоден! Вірніше, обидва варіанти – погані. Однак, якщо пам'ятати, що елементи масиву пронумеровано з 0, то все стає на свої місця, отож говорити можна і так, як у першому варіанті, і так, як у другому.

Компілятор C++ не виконує жодної перевірки на дотримання задекларованих меж значень індексу масиву. Наприклад, використання `W[29]` не викличе жодної реакції з боку компілятора. Отже, усю відповідальність за дотримання меж зміни індексу покладено на програміста ("порятунок потопельника – справа рук самого потопельника").

Під час визначення масиву його елементам можна задавати початкові значення, які перелічують у *списку ініціалізації* (список обмежують фігурними дужками). Наприклад:

```
int  A[10] = {1,2,3,4,5,6,7,8,9,10};
char S1[5] = {'a','b','c','d','e'};
char S2[5] = {'a','b','c','d','\0'};
```

Тут `S1` – *масив*, що містить 5 символів, а `S2` – *рядок символів* (закінчується нульовим символом `'\0'` - ознакою закінчення рядка).

Якщо початкових значень є менше, ніж елементів масиву, то елементи, які залишаються не ініціалізованими, автоматично одержують нульові початкові значення. Наприклад, інструкція

```
int B[10] = {10,20,35};
```

задає значення першим трьом елементам, а інші дорівнюватимуть `0`.

Якщо масив під час визначення не ініціалізований, то його елементи мають випадкові значення, отож їх не можна використовувати у виразах до присвоєння значень.

У масивах символів кінцеві елементи, не зазначені у списку ініціалізації, за домовленістю отримують *нульове значення* (позначку закінчення рядка). Отож наведене вище визначення змінної `S2` з її ініціалізацією може бути і таким:

```
char S2[5] = {'a','b','c','d'};
```

У визначенні зі списком ініціалізації розмір масиву можна не вказувати (кількість елементів масиву дорівнюватиме кількості елементів у списку початкових значень). Наприклад, визначення

```
int C[] = {1,2,3,4,5};
```

створює масив `C` з п'яти елементів.

Під час такої ініціалізації символьних масивів жодного додаткового байта для нульового символу *не виділяють*, наприклад:

```
char S3[]={'a','b','c'}; // Масив 3-х символів
```

Ініціалізація ж масиву-рядка виглядає так:

```
char S4[]="abc";
```


Над одновимірним масивом виконують дві базові операції: *отримують* (витягають) значення деякого елемента масиву для задіяння його у певному виразі і *розміщують* деяке значення у конкретному елементі масиву.

Отримують значення елемента масиву за допомогою доступу до цього елемента завдяки *індексу* (наприклад, `3*W[2]+5`). Розміщують деяке значення в елементі масиву через завдання початкових значень під час визначення масиву, через введення значення з клавіатури (наприклад, `cin>>W[2]`) чи за допомогою інструкції присвоювання значень (наприклад, `W[0]=23, W[1]=25`).

С++ не має засобів уведення / виведення усіх елементів масиву одразу ж, отож уведення / виведення значень здійснюють поелементно (у циклі). Обробку значень масиву, зазвичай, здійснюють не для окремих, а для усіх елементів масиву, отже, і тут треба використовувати цикли.

Наведемо декілька прикладів роботи з масивами. У перших двох прикладах заповнення масиву значеннями здійснюватимемо шляхом задавання початкових значень під час визначення масиву.

Приклад 1. Сформувати масив шляхом задавання початкових значень під час визначення масиву. Вивести на екран консолі значення квадратів елементів цього масиву.

➢ *На вході* – нічого.

На виході. Одержуємо значення елементів масиву (`ar:`) та їхні квадрати (`ar2:`).

Програма:

```
#include <iostream>
using namespace std;
//-----------------------------------------------------
void main() {int i, ar[]={3,5,7,11,13,17}; // Масив
 // Виведення елементів масиву
 cout << "ar:  "; // Позначка елементів масиву
 for(i=0; i<6; cout<<ar[i]<<"  ", i++); cout<<endl;
 // Виведення квадратів елементів масиву
 cout << "ar2: "; // Позначка квадратів
 for(i=0; i<6; cout<<ar[i]*ar[i]<<" ", i++);  cin>>i;
}
```

Результати тестування програми:

```
ar:   3  5   7  11  13  17
ar2: 9 25 49 121 169 289
```

Приклад 2. Сформувати масив шляхом задавання початкових значень під час визначення масиву. Вивести індекс першого входження найбільшого елемента одновимірного масиву.

➢ *На вході* – нічого.

На виході. Одержуємо значення елементів масиву (`ar:`) та індекс першого входження найбільшого елемента масиву (`IMax=`).

Програма:

```
#include <iostream>
using namespace std;
//----------------------------------------------------------
void main()
{
  int ar[]={6, 3, 45, 134, -23, 56, 134, 24, 2, 6};
  int size=sizeof(ar)/sizeof(ar[0]), i;
  // Виведення елементів масиву
  cout << "ar:  "; // Позначка елементів масиву
  for(i=0; i<size; cout<<ar[i]<<"  ", i++); cout<<endl;
  int IMax = 0;
  for (int i = 1; i < size; i++ )
     if (ar[i] > ar[IMax]) IMax = i ;
  cout << "IMax=  "<<IMax;
  cin>>i;
}
```

Додатковий коментар. Зверніть увагу, що розмір масиву (змінна `size`) компілятор визначатиме самостійно за допомогою того, що кількість байтів усього масиву (`sizeof(ar)`) ділиться на кількість байтів окремого елемента масиву (`sizeof(ar[0])`).

Результати тестування програми:

```
ar:  6  3  45  134  -23  56  134  24  2  6
IMax=  3
```

Зручним засобом тестування програм є використання *псевдовипадкових* чисел (див. 3.4.2). Застосуємо цей засіб і для заповнення значеннями елементів масиву. Покажемо це на прикладі 3.

Приклад 3. Сформувати масив і заповнити його *псевдовипадковими* числами. Визначити перше від початку масиву найменше число і останнє за порядком найбільше число та поміняти їх місцями.

➢ *Попередні міркування.* Оскільки розмір масиву явно не завданий, то його необхідно *отримати* від користувача у режимі діалогу. Однак проблема полягає у тому, що компілятор розміщує масив у пам'яті комп'ютера (у сегменті даних або у стекові, залежно від місця визначення) до початку виконання програми (такі масиви називають *статичними*).

Внаслідок цього розмір статичного масиву можна задати тільки константою чи константним виразом. Є ще *динамічні* масиви, в яких виокремлення пам'яті відбувається під час виконання програми і розмір масиву можна оперативно одержувати від користувача (про них говоритимемо згодом).

Отже, одержати розмір статичного масиву від користувача можна, однак використати його для визначення масиву не вдасться. Заводити ж динамічний масив ми ще не вміємо! Що робити? За цих умов вчиняють так: визначають масив деякого фіксованого розміру (у цьому прикладі – 20), а фактичну кількість елементів масиву задають через деякий параметр (у цьому прикладі – параметр `n`). Елементи масиву – псевдовипадкові цілі числа з проміжку [–20; 20].

Алгоритм розв'язання задачі: визначити *індекси* першого від початку найменшого числа в масиві та останнього за порядком найбільшого числа в масиві, а потім відповідні елементи масиву поміняти місцями.

На вході (`Enter n (0<n<=20):`). Кожен такий рядок є окремим тестом, що задає значення змінної n (кількості фактичних елементів масиву). Ознакою закінчення тестування є рядок, в якому введене значення є *недодатним*.

На виході. Одержуємо початковий (`ar:`) і модифікований масиви.

Програма:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
//-----------------------------------------------
void main()
{ int i, n, kmin=0, kmax=0;
  double scale; int ar[20];
  while (cout<<"Enter n (0<n<=20): ", cin>>n, n>0)
  { (n>20)? n=20 : n; // Перевірка розміру масиву
    srand(time(NULL));
    cout << "ar: ";
    // Заповнення і виведення початкового масиву
    for(i=0; i<n; i++)
    {
       scale = rand()/double(RAND_MAX);
       ar[i] = int(scale*40 - 19.5);
       cout<<ar[i]<<" ";
    }
    cout<<endl;
    // Виконання алгоритму
    for(i=1; i<n; i++)
    {
      if (ar[kmin]>ar[i])    kmin=i;
      if (ar[kmax] <= ar[i]) kmax=i;
    }
    i= ar[kmin]; ar[kmin]= ar[kmax]; ar[kmax]=i;
    // Виведення модифікованого масиву
    for (i=0; i<n; i++) cout<<ar[i]<<" "; cout<<endl;
  }
  cin>> n; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_3\Debug\Example_4...
Enter n (n>0): 12
ar:  -12  9  -8  -8  13  -10  5  10  -9  -16  -3  2
-12 9 -8 -8 -16 -10 5 10 -9 13 -3 2
Enter n (n>0): 6
ar:  -12  -9  13  -12  -17  20
-12 -9 13 -12 20 -17
Enter n (n>0): 8
ar:  -12  11  5  -1  -11  0  3  -14
-12 -14 5 -1 -11 0 3 11
Enter n (n>0): 0
```

Приклад 4. Сформувати масив через введення значень з клавіатури. Вивести на екран консолі значення кубів елементів масиву.

➢ *На вході* – послідовне введення елементів масиву (`ar[i]=`).

На виході. Одержуємо елементи масиву (`ar:`) та їхні куби (`ar3:`).

Програма:

```
#include <iostream>
using namespace std;
void main() {int i, ar[5]; // Масив
 // Введення елементів масиву
 for(i=0; i<5; cout<<" ar["<<i<<"]=", cin>>ar[i], i++);
 /* Виведення елементів масиву */   cout << "ar  ";
 for(i=0; i<5; cout<<"  "<<ar[i], i++); cout<<endl;
 // Виведення кубів елементів масиву
 cout << "ar3: "; // Позначка кубів
 for(i=0; i<5; cout<<ar[i]*ar[i]*ar[i]<<" ", i++);
 cin>>i; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_4\Debug\Example_4_4.exe
 ar[0]=2
 ar[1]=3
 ar[2]=4
 ar[3]=5
 ar[4]=7
ar    2  3  4  5  7
ar3: 8 27 64 125 343
```

4.1.2. Багатовимірні масиви

Багатовимірний масив – це масив, елементами якого є масиви меншого виміру. Приклад визначення двовимірного масиву:

```
int A2[10][5];
```

Ця інструкція визначає двовимірний масив цілих чисел A2, який можна інтерпретувати так: кожен з 10-ти елементів масиву A2 є одновимірним масивом розміру 5 (всього маємо 10 × 5 = 50 елементів масиву).

Зауважимо, що двовимірні масиви C++ моделюють матриці (масив A2 моделює матрицю, що складається з 10-ти рядків і 5-ти стовпців). Іноді вживатимемо короткий термін *матриця* замість терміна *двовимірний масив*.

Доступ до значень елементів багатовимірного масиву забезпечується через індекси, кожен з яких беруть у квадратні дужки. Наприклад, A2[3][2] – значення елемента у четвертому рядку та третьому стовпці цього масиву (нумерація індексів з 0).

Розміщують деяке значення в елементі багатовимірного масиву шляхом задавання початкових значень під час визначення масиву, через введення значення з клавіатури (наприклад, cin>>A2[i][j]) чи за допомогою інструкції присвоювання значень (наприклад, A2[0][j] = 23, A2[2*i][j+1] = i+2*j).

Якщо багатовимірний масив ініціалізують під час визначення, то список значень за кожною розмірністю беруть у фігурні дужки. Наведена нижче інструкція визначає тривимірний масив A3 з 4 × 3 × 2 елементами:

```
int A3[4][3][2] = {{{0,1},{2,3},{4,5}},
                   {{6,7},{8,9},{10,11}},
                   {{12,13},{14,15},{16,17}},
                   {{18,19},{20,21},{22,23}}};
```

Отримуємо масив A3, чотири елементи якого є матрицями вигляду:

$$\begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}; \quad \begin{pmatrix} 6 & 7 \\ 8 & 9 \\ 10 & 11 \end{pmatrix}; \quad \begin{pmatrix} 12 & 13 \\ 14 & 15 \\ 16 & 17 \end{pmatrix}; \quad \begin{pmatrix} 18 & 19 \\ 20 & 21 \\ 22 & 23 \end{pmatrix}.$$

Елемент A3[0][1][0] дорівнює 2, елемент A3[3][0][1] дорівнює 19 і т. д. Якщо у списку ініціалізації у будь-якій розмірності не вистачає даних, то усі наступні не перелічені елементи вважають нулями. Під час опрацювання багатовимірного масиву використовують *вкладені* цикли.

Приклад 5. Сформувати матрицю 2 × 4 шляхом ініціалізації елементів під час її визначення. Вивести суму від'ємних елементів цієї матриці.

➤ *На вході* – нічого.

На виході. Одержуємо значення елементів масиву (ar_2:) та суму від'ємних елементів масиву (s=).

Програма:

```
#include <iostream>
using namespace std;
//------------------------------------------------------
void main()
{ int ar_2[2][4]={{-6, 3, 4, 4}, {-3, 5, -1, 2}},
      s=0, i, j;
  // Формування суми і виведення елементів масиву
  cout << "ar_2:  "<< endl; // Позначка масиву
  for(i=0; i<2; i++)
   {for(j=0; j<4; j++)
     {if(ar_2[i][j]<0) s += ar_2[i][j];
       cout << ar_2[i][j]<<" "; } cout << endl;
   }
  cout << endl<<"s=  "<<s<< endl;
  cin>>i; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4-5\...
ar_2:
-6 3 4 4
-3 5 -1 2

s=  -10
```

4.2. Вказівники

4.2.1. Загальні положення

Вказівник – це змінна, значення якої дорівнює *адресі* розміщення у пам'яті деякої іншої змінної. Назва цієї іншої змінної відсилає до її значення *прямо*, а вказівник – *побічно* (*непрямо*). Посилання на значення за допомогою вказівника називають *непрямою адресацією*.

Зауважимо, що у цьому розділі термін "*змінна*" у багатьох випадках використано для позначення як власне змінних, так і структурованих даних (масивів, структур, об'єднань тощо).

У С++ розрізняють три типи вказівників: вказівники на *об'єкти*; вказівники на *функції*; вказівники типу `void`. Вказівники, подібно до інших змінних, перед використанням у програмі повинні визначатися відповідною інструкцією.

Вказівник на *об'єкт* містить адресу області місця у пам'яті, де зберігається значення певного типу (простого чи структурованого). Найпростіше визначення вказівника на *об'єкт* (у подальшому просто вказівника) має вигляд:

```
тип *ідентифікатор;
```

де `тип` – один з базових чи визначених користувачем типів, окрім посилання й бітового поля; `ідентифікатор` задає назву вказівника.

Наприклад:

```
int *countPtr, count;
```

визначає змінну `countPtr` типу `int*` (тобто, вказівник на ціле число) і змінну `count` цілого типу. Символ * в оголошенні відноситься тільки до `countPtr`. Кожну змінну, яку оголошують вказівником, повинна мати перед собою символ *. Якщо необхідно, щоб і змінна `count` була вказівником, треба записати:

```
int *countPtr, *count;
```

Символ * у цих записах позначає операцію *непрямої адресації*.

Визначення вказівника типу `void`, який є *універсальним* вказівником на *будь-який тип даних*, має вигляд:

```
void *Pv;
```


Перед використанням вказівника `Pv`, йому в процесі роботи необхідно присвоїти значення іншого вказівника на деякий конкретний тип даних, наприклад:

```
Pv = countPtr;
```

Вказівники повинні ініціалізовуватися під час визначення, або за допомогою інструкції присвоювання. Вказівник може містити початкове значення `NULL` (символічна константа) чи адресу. Вказівник зі значенням `NULL` *ні на що не вказує*, наприклад:

```
int *countPtr = NULL;
```

Для присвоювання вказівнику адреси деякої змінної використовують *оператор адресації* &, що повертає адресу свого операнда. Наприклад, якщо маємо визначення

```
int y = 5; int *yP, x;
```

то оператор `yP = &y;` присвоює вказівнику `yP` адресу змінної `y`.

Для одержання *значення*, на яке вказує вказівник, використовують операцію *, яку, зазвичай, називають оператором *непрямої адресації* (або оператором *рознайменування*). Він повертає значення об'єкта, на який вказує його операнд (тобто вказівник). Наприклад, оператор присвоювання

```
x = *yP;
```

присвоїть змінній `x` значення `5`.

Операцію рознайменування не можна застосовувати до вказівника типу `void`, оскільки для нього невідомо, який розмір пам'яті треба рознайменовувати.

Вказівник можна присвоювати іншому вказівнику, якщо обидва вказівники мають однаковий тип. У протилежному випадку необхідно використати операцію надання типу, щоб перетворити значення вказівника у правій частині присвоювання до типу вказівника у лівій частині присвоювання.

Винятком з цього правила є вказівник типу `void`, якому можна присвоювати усі типи вказівників без надання типу. Однак вказівник типу `void` неможливо присвоїти безпосередньо вказівнику іншого типу – вказівник типу `void` спочатку необхідно довести до типу відповідного вказівника.

4.2.2. Зв'язок масивів і вказівників

Масиви і вказівники у C++ тісно зв'язані, їх можна використовувати майже еквівалентно. Назва масиву – це *сталий вказівник* на перший елемент масиву. Його відмінність від звичайного вказівника тільки у тому, що його не можна модифікувати. Вказівники можна використовувати для виконання будь-якої операції, включаючи індексування масиву. Нехай маємо таке визначення:

```
int b[5] = {1,2,3,4,5}, *Pt;
```

У такий спосіб визначено масив цілих чисел `b[5]` і вказівник на ціле `Pt`. Оскільки назва масиву є вказівником на перший елемент масиву, то інструкція `Pt=b;` еквівалентна інструкція `Pt=&b[0];`

Вказівники можна індексувати так само, як і масиви. Наприклад, вираз `Pt[3]` посилається на елемент масиву `b[3]`.

Над вказівниками можна виконувати такі арифметичні операції: вказівник можна збільшувати (++), зменшувати (--), додавати до вказівника цілі числа (+ чи +=), віднімати від нього цілі числа (- чи -=) або ж віднімати один вказівник від іншого.

Додавання/віднімання вказівників з цілими числами відрізняється від звичайної арифметики. Додати до вказівника `1` означає збільшити його на кількість байтів, яку відводять для збереження значень змінної, на яку він вказує.

Якщо продовжити наведений вище приклад, у якому вказівнику `Pt` присвоєно значення `b` – вказівника на перший елемент масиву, то після виконання інструкції

```
Pt += 2;
```

`Pt` вказуватиме на третій елемент масиву `b` (тобто `b[2]`). Значення вказівника `Pt` зміниться на кількість байтів, які займає один елемент масиву, помножену на `2`. Оскільки кожен елемент типу `int` займає `4` байти, то значення `Pt` збільшиться на `8`. Аналогічні правила діють і під час віднімання від вказівника цілого значення.

Вирази з вказівниками можна використовувати і під час рознайменувань. Для одержання значення четвертого елемента масиву `b` (тобто `b[3]`) можна, наприклад, використати вираз `*(Pt + 3)`.

Змінні-вказівники можна віднімати один від іншого. Наприклад, якщо `Pt` указує на перший елемент `b`, а вказівник `Pt1` – на третій, то результат виразу `Pt1 - Pt` дорівнюватиме `2`.

Арифметика вказівників утрачає будь-який зміст, якщо її виконують не над вказівниками на масив. Порівняння вказівників операторами `>`, `<`, `>=`, `<=` також мають сенс тільки для вказівників на той самий масив. Однак операції порівняння `==` і `!=` мають сенс для будь-яких вказівників (вказівники *рівні*, якщо вони вказують на одну і ту ж адресу пам'яті).

4.2.3. Динамічний розподіл пам'яті

Досі у всіх прикладах пам'ять для змінних чи інших об'єктів виділяли автоматично у *програмному стекові* – області оперативної пам'яті, зарезервованої програмою під час запуску (тобто відбувся *локальний розподіл* (*local allocation*) пам'яті).

Усю пам'ять, необхідну для створення локальних змінних, викликів функцій та інших дій, взято зі стека. Ця пам'ять розподіляється за необхідністю і звільняється, коли потреба у ній зникає.

Локальний розподіл має свої плюси і мінуси. З одного боку, пам'ять у стекові виділяється дуже швидко. З іншого боку, стек має фіксований розмір, отож може переповнитися. Для змінних і невеликих масивів достатньо використовувати локальний розподіл.

Однак під час роботи з великими масивами, структурами чи класами доцільно вдатися до *динамічного розподілу* (*dynamic allocation*) пам'яті з "*купи*". Під *купою* (*heap*) у Windows-програмах розуміють усю віртуальну пам'ять комп'ютера (вільну оперативну пам'ять і вільний простір на жорсткому диску).

Об'єкти, які розміщуватимуться у купі, називатимемо *динамічними*. У C++ для роботи з динамічними об'єктами використовують спеціальні оператори `new` і `delete`.

За допомогою оператора `new` виокремлюється осередок пам'яті для динамічного розміщення об'єкта і повертається адреса цього осередку, яка присвоюється певному вказівнику. Сам вказівник розташовується у стекові. Оператор `delete` видаляє динамічний об'єкт з пам'яті.

Основна форма цих операторів:

```
p = new type; … delete p;
```

Тут type – це специфікатор типу об'єкта, для якого виділяється пам'ять у купі, а p – вказівник на цей тип; new – оператор, який повертає вказівнику p адресу осередка пам'яті, динамічно виокремленого у купі.

Оператор delete звільняє цю пам'ять у купі, коли у ній немає необхідності. Виклик оператора delete з неправильним вказівником може спричинити руйнування системи динамічного виділення пам'яті і можливого краху програми.

Змінним, які динамічно розміщуються у пам'яті, можна присвоїти початкові значення, використовуючи таку форму new:

```
p = new type (початкове_значення);
```

Для одновимірного масиву, що динамічно розміщується у пам'яті, використовують таку форму оператора **new**:

```
p = new type [size];
```

Після виконання цієї інструкції вказівник p вказуватиме на початковий (з нульовим індексом) елемент масиву із size елементів заданого типу. Через різноманітні технічні причини неможливо під час виконання оператора new одночасно ініціалізувати масив.

Для видалення динамічно розміщеного одновимірного масиву необхідно використати таку форму оператора delete:

```
delete [] p;
```

Надалі вживатимемо термін "*динамічний масив*", розуміючи під ним масив, який динамічно розміщується у пам'яті. Створимо, для прикладу, динамічний вектор, кількість елементів якого задаватиме користувач через введення з клавіатури (для кращого розуміння наведено два варіанти реалізації):

```
int n, *vect; // Варіант 1
// Введення n - кількості елементів масиву
vect = new int[n]; // Виокремлення пам'яті
// Опрацювання масиву vect
delete [] vect; // Видалення масиву
```



```
int n; // Варіант 2
// Введення n - кількості елементів масиву
int *vect = new int[n];
// Опрацювання масиву vect
delete [] vect; // Видалення масиву
```

У другому варіанті інструкція:

```
int *vect = new int[n];
```

водночас створює вказівник і виокремлює пам'ять у купі.

Для створення динамічного двовимірного масиву (матриці) розміру $n \times k$ треба спочатку за допомогою оператора **new** виокремити пам'ять у купі під n вказівників на рядки, а згодом для кожного вказівника на рядок виокремити пам'ять у купі для k елементів цього рядка:

```
int n,k,i, **m;
// Введення n i k - кількості рядків і стовпців
m = new int * [n];
for (i=0; i<n; i++) m[i] = new int[k];
// Опрацювання масиву m
// Видалення масиву m
for (i=0; i<n; i++) delete [] m[i];
delete [] m;
```

Доступ до елементів динамічних масивів аналогічний доступу до елементів локально розміщених масивів, наприклад vect[3], m[2][i] тощо. Наведемо приклади створення та опрацювання динамічних масивів.

Приклад 6. Сформувати одновимірний масив і заповнити його випадковими числами. Визначити перше від початку масиву найменше число. Вивести індекс цього числа у масиві та саме число.

➢ *Попередні міркування*. Оскільки розмір масиву явно не задано, то його необхідно отримати від користувача у режимі діалогу. Після цього створимо одновимірний *динамічний* масив заданого розміру та заповнимо його псевдовипадковими цілими числами з проміжку [–20; 20].

На вході (`Enter n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінної n (кількості елементів масиву). Ознакою завершення тестування є рядок, в якому введене значення не є додатним.

На виході. Одержуємо індекс першого від початку масиву найменшого числа та саме число (`ar[індекс]=число`).

Програма:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
//-----------------------------------------------
void main()
{ int i, n, kmin;
  double scale;
  while (cout<<"Enter n (n>0): ", cin>>n, n>0)
  { int *ar = new int[n];
    srand(time(NULL));
    cout << "ar:  ";
    // Заповнення і виведення початкового масиву
    for(i=0; i<n; i++)
    {
      scale = rand()/double(RAND_MAX);
      ar[i] = int(scale*40 - 19.5);
      cout<<ar[i]<<"  ";
    }
    cout<<endl;
    // Виконання алгоритму
    kmin=0;
    for(i=1; i<n; i++) if (ar[kmin]>ar[i])  kmin=i;
    // Виведення результату
    cout<<"ar["<<kmin<<"]="<<ar[kmin]<<endl;
    delete [] ar;
  }
  cin>> n; // Пауза
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_5\Debug\Example_4_5.exe
Enter n (n>0): 6
ar:  -13  11  0  -4  14  -9
ar[0]=-13
Enter n (n>0): 9
ar:  -13  -16  -7  1  -17  -16  -4  17  10
ar[4]=-17
Enter n (n>0): 13
ar:  -13  -5  11  14  -11  -8  -6  -18  12  -15  -14  8  -9
ar[7]=-18
Enter n (n>0): 3
ar:  -13  -6  7
ar[0]=-13
Enter n (n>0): 0
```

⮜

Приклад 7. Дано натуральне число n. Отримати дійсну матрицю $a[i,j]_{i,j=1,\dots,n}$, для якої виконуються умови:

$$a[i,j]=\begin{cases} i+j, & \text{якщо } i<j; \\ 2, & \text{якщо } i=j; \\ i\cdot j, & \text{якщо } i>j. \end{cases}$$

➢ *Попередні міркування*. Оскільки розмір матриці n явно не задано, то його необхідно отримати від користувача у режимі діалогу. Після цього створимо відповідний двовимірний *динамічний* масив та заповнимо його згідно з заданою формулою. Індекс i позначає рядок квадратної матриці, а j – стовпець. Рівні індекси ($i=j$) мають елементи головної діагоналі. Умова $i<j$ вказує на те, що елементи розміщені над головною діагоналлю, а $i>j$ – елементи знаходяться під головною діагоналлю.

На вході (`Enter n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінної n (розміру матриці). Ознакою завершення тестування є рядок, в якому введене значення не є додатним.

На виході. Одержуємо сформовану матрицю розміру n.

Програма:

```
#include <iostream>
using namespace std;
//-----------------------------------------------
```

```
void main()
{ int n, i ,j; int **a;
  while (cout<<"Enter n (n>0): ", cin>>n, n>0)
  {
   a = new int*[n];
   for(i=0; i<n; i++) a[i]= new int[n];
   for(i=0; i<n;i++) a[i][i] = 2;// Головна діагональ
   for(i=0; i<n-1; i++)  // Над головною діагоналлю
   for(j=i+1; j<n;j++) a[i][j] = i+j+2;
   for(i=1; i<n; i++) // Під головною діагоналлю
   for(j=0; j<=i-1; j++) a[i][j]=(i+1)*(j+1);
   for(i=0; i<n; i++) // Виведення матриці
     { for(j=0; j<n; j++) cout<<a[i][j]<<"  ";
       cout<<endl;
     }
   for(i=0; i<n; i++) delete [] a[i];
   delete [] a;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_7\Debug\Exa...
Enter n (n>0): 3
2  3  4
2  2  5
3  6  2
Enter n (n>0): 4
2  3  4  5
2  2  5  6
3  6  2  7
4  8  12  2
Enter n (n>0): 6
2  3  4  5  6  7
2  2  5  6  7  8
3  6  2  7  8  9
4  8  12  2  9  10
5  10  15  20  2  11
6  12  18  24  30  2
Enter n (n>0):
```

Увага! Якщо в попередньому прикладі йдеться тільки про сукупність чисел на екрані, то масив можна й не заводити.

4.3. Взаємодія функцій та вказівників

Нагадаємо, що початкове знайомство з функціями відбулося у пункті 2.7. Там функції використовували для опрацювання скалярних величин. У цьому параграфі ми суттєво поглибимо наші знання про функції. Розглянемо питання: використання функцій для опрацювання масивів; роль вказівників на функції та відповідних типів у формуванні інваріантів функцій щодо дій; створення багатомодульних проектів тощо.

4.3.1. Передача функції аргументів-масивів

Нагадаємо, що назва масиву є сталим *вказівником* на його нульовий елемент. Отже, під час використання аргументу, який відображає назву деякого масиву (тобто *аргументу-масиву*), у функцію передається копія вказівника з адресою нульового елемента масиву.

При цьому інформація про кількість елементів масиву втрачається, і її необхідно передавати за допомогою окремого параметра. Під час передачі *багатовимірних* масивів *усі* розмірності, якщо вони невідомі на етапі компіляції, необхідно передавати за допомогою відповідної кількості параметрів.

Щоб функція могла прийняти масив, переданий під час виклику функції, відповідний параметр функції повинен бути *вказівником* (або описуватися *парою символів* "`[]`" у випадку одновимірних масивів).

Під час передачі функції аргументу-масиву він у пам'яті не дублюється, хоча синтаксис відповідного параметра відповідає синтаксису параметра за значенням. Назва аргументу-масиву перетворюється у вказівник на початок цього масиву, а *копія вказівника* передається у функцію за значенням.

Це дуже вигідно, оскільки копіювання великих масивів вимагає значних затрат часу та пам'яті. Однак чималим мінусом цього підходу є те, що внаслідок опрацювання масиву функцією окремі елементи масиву можуть змінити значення. Якщо зміна значень елементів масиву є не бажаною, то для попередження можливих змін використовують модифікатор `const`.

Для глибшого розуміння вищесказаного наведемо декілька прикладів дуже простих функцій, які працюватимуть з масивами.

Приклад 8. Скласти та протестувати функцію, яка визначає кількість додатних елементів одновимірного масиву цілих чисел.

➢ *Попередні міркування*. Оскільки розмір масиву явно не задано, то його необхідно отримати від користувача у режимі діалогу. Після цього створимо одновимірний *динамічний* масив заданого розміру та заповнимо його псевдовипадковими цілими числами з проміжку [–20; 20]. Для визначення кількості додатних елементів довільного одновимірного масиву створимо функцію `kilDod`.

На вході (`Enter n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (розміру масиву). Ознакою завершення тестування є рядок, в якому введене значення не є додатним.

На виході. Одержуємо сформований масив (`ar:`) розміру `n` та кількість додатних елементів цього масиву (`m=`).

Програма:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
//-------------------------------------------------------------
int kilDod(const int *a, const int n)
// Заголовок функції може бути й таким:
//  int kilDod(const int a[], const int n)
// Визначення кількості додатних значень у
// масиві a; n - кількість елементів масиву.
// Передумова: n - додатне число.
// Постумова. Кількість додатних значень k
// є результатом функції kilDod.
{ int k=0;
  for(int i=0; i<n; i++) if (a[i]>0) k++;
  return k;
}
//-------------------------------------------------------------
```



```
void main()
{ int i,n;
  double scale;
  while (cout<<"Enter n (n>0): ", cin>>n, n>0)
  { int *ar = new int[n];
    srand(time(NULL));
    cout << "ar:  ";
    // Заповнення і виведення початкового масиву
    for(i=0; i<n; i++)
    {
       scale = rand()/double(RAND_MAX);
       ar[i] = int(scale*40 - 19.5);
       cout<<ar[i]<<"  ";
    }
    cout<<endl;
    int m=kilDod(ar,n); // m - кількість додатних
    cout<<"m="<<m<<endl;
    delete [] ar;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Example_4_8\Debug\Examp...
Enter n (n>0): 9
ar:  -4  6  -19  6  -1  2  -17  14  -9
m=4
Enter n (n>0): 9
ar:  -4  19  -11  4  9  -18  -2  0  -8
m=3
Enter n (n>0): 10
ar:  -3  4  2  0  -9  20  -11  11  -7  19
m=5
Enter n (n>0): 10
ar:  -3  -10  -8  -13  12  4  7  -6  13  -18
m=4
Enter n (n>0): 0
```

Приклад 9. Дано натуральне число *n*. Скласти програму визначення суми тих елементів матриці $a[i,\ j]_{i,\ j=1,\dots,n}$, які розміщені на головній діагоналі або над нею, що перевищують усі елементи, які розміщені під головною діагоналлю.

➢ *Попередні міркування*. Оскільки розмір матриці *n* явно не задано, то його необхідно отримати від користувача у режимі діалогу. Після цього створимо двовимірний *динамічний* масив заданого розміру та заповнимо його псевдовипадковими раціональними числами з проміжку [–20; 20].

Серед елементів матриці, які розміщені під головною діагоналлю, знайдемо найбільший елемент. Тоді елементи, які розміщені на головній діагоналі або над нею і більші за знайдений найбільший елемент, задовольнятимуть умові задачі.

Для визначення суми відповідних елементів матриці створимо функцію `Sum_Nad`.

На вході (`Enter n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (розміру матриці). Ознакою завершення тестування є рядок, в якому введене значення є *недодатним*.

На виході. Одержуємо сформовану матрицю (`ar:`) розміру `n` та суму відповідних елементів цього масиву (`s=`).

Програма:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
#include<iomanip>
using namespace std;
//-----------------------------------------------------------
double Sum_Nad(double **a, const int n)
// Визначення суми тих елементів матриці a,
// які розміщені на головній діагоналі або над нею,
// що перевищують усі елементи, які розміщені
// під головною діагоналлю.
// Параметр n - кількість елементів масиву.
// Передумова: n - додатне число.
// Постумова. Сума відповідних елементів s
// є результатом функції Sum_Nad.
```



```
//-----------------------------------------------------------
{
  int i,j; double max, s;
  max = a[1][0];
  for(i=2; i<n; i++)
   for(j=0; j<=i-1; j++)
    if(a[i][j]>max) max=a[i][j];
  s = 0;
  for(i=0; i<n; i++)
   for(j= i; j<n; j++)
    if (a[i][j]>max) s += a[i][j];
  return s;
}
//-----------------------------------------------------------
void main()
{ int i,j,n;
  double scale,s, **ar;
  while (cout<<"Enter n (n>0): ", cin>>n, n>0)
  { ar = new double* [n];
    for (i=0; i<n; i++) ar[i] = new double[n];
    srand(time(NULL));
    cout << "ar:  "<<endl;
    // Заповнення і виведення початкового масиву
    for(i=0; i<n; i++)
     {
      for(j=0; j<n; j++)
       {
         scale = rand()/double(RAND_MAX);
         ar[i][j] = scale*40 - 19.5;
         cout<<setprecision(2)<<ar[i][j]<<"  ";
       }
      cout<<endl;
     }
    cout<<endl;
    s = Sum_Nad(ar,n);
    cout<<"s="<<s<<endl;
    for(i=0; i<n; i++) delete [] ar[i];
    delete [] ar;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_9\Debug\Example_4_9...
Enter n (n>0): 4
ar:
15  -0.61  19  -6.2
-11  -0.59  2.8  1.8
-13  -6.4  17  8
-14  17  2.5  -12

s=19
Enter n (n>0): 3
ar:
16  11  11
-0.53  -2.5  -7
-11  17  8.4

s=0
Enter n (n>0):
```

Приклад 10. Дано натуральні числа *n* і *k*. Скласти програму визначення суми від'ємних елементів прямокутної матриці *n* × *k*.

➢ *Попередні міркування*. Оскільки розміри матриці *n* і *k* явно не задано, то їх необхідно отримати від користувача у режимі діалогу. Після цього створимо двовимірний *динамічний* масив заданих розмірів і заповнимо його псевдовипадковими раціональними числами з проміжку [–20; 20]. Для визначення суми від'ємних елементів матриці створимо функцію `Sum_Neg`.

На вході (`Enter n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінних n і k. Ознакою завершення тестування є рядок, в якому хоча б одне значення n або k не є додатним.

На виході. Одержуємо сформовану матрицю (`ar:`) розміру n та суму від'ємних елементів цього масиву (`s=`).

Програма:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
#include<iomanip>
using namespace std;
//----------------------------------------------------------
double Sum_Neg(double **a, const int n, const int k)
// Визначення суми від'ємних елементів матриці a
```



```
// Параметр n / k - кількість рядків / стовпців матриці
// Передумова: n,k - додатні числа.
// Постумова. Сума відповідних елементів  s
// є результатом функції Sum_Neg.
//----------------------------------------------------------
{ int i,j; double s=0;
  for(i=0; i<n; i++)
   for(j=0; j<k; j++) if(a[i][j]<0) s += a[i][j];
  return s;
}
//----------------------------------------------------------
void main() { int i,j,n,k; double scale,s, **ar;
 while(cout<<"Enter n k > 0: ", cin>>n>>k, n>0 && k>0)
 {ar = new double* [n];
  for (i=0; i<n; i++) ar[i] = new double[k];
  srand(time(NULL));  cout << "ar: "<<endl;
  // Заповнення і виведення початкового масиву
  for(i=0; i<n; i++) {
    for(j=0; j<k; j++) {scale=rand()/double(RAND_MAX);
         ar[i][j] = scale*40 - 19.5;
         cout<<setprecision(3)<<ar[i][j]<<"  ";}
    cout<<endl;                                    }
    cout<<endl; s=Sum_Neg(ar,n,k);
    cout<<"s="<< setprecision(3)<<s<<endl;
    for(i=0; i<n; i++) delete [] ar[i]; delete [] ar;
 }  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_10\Debug...
Enter n k (n/k>0): 3 4
ar:
11.3  -16.1  10.9  16.8
-4.82  1.91  -15.4  1.3
-3.1  -7.09  -3.76  12.7

s=-50.29
Enter n k (n/k>0): 4 0
```

Параметри-масиви можна використовувати під час організації викликів *рекурсивної* функції. Наведемо такий приклад.

Приклад 11. Скласти та протестувати рекурсивну функцію визначення суми елементів одновимірного масиву цілих чисел.

➢ *Попередні міркування*. Оскільки розмір масиву явно не задано, то його необхідно отримати від користувача у режимі діалогу. Після цього створимо одновимірний *динамічний* масив заданого розміру та заповнимо його псевдовипадковими цілими числами з проміжку [–20; 20]. Для визначення суми елементів цього масиву створимо рекурсивну функцію `Sum_Rec`.

На вході (`Enter n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінної `n` (розміру масиву). Ознакою завершення тестування є рядок, в якому введене значення є *недодатним*.

На виході. Одержуємо сформований масив (`ar:`) розміру `n` та суму елементів цього масиву (`s=`).

Програма:

```
#include <iostream>
#include<cstdlib>
#include<ctime>
using namespace std;
//---------------------------------------------------------
int Sum_Rec(const int *a, const int n)
// Заголовок функції може бути й таким:
//  int Sum_Rec(const int a[], const int n)
// Визначення суми елементів масиву a;
// n - кількість елементів масиву.
// Передумова: n - додатне число.
// Постумова. Сума елементів масиву
// є результатом функції Sum_Rec.
{
  return n==1?  a[0] :
                          Sum_Rec(a, n-1)+a[n-1];
}
//---------------------------------------------------------
```



```
void main()
{ int i,n;
  double scale;
  while (cout<<"Enter n (n>0): ", cin>>n, n>0)
  { int *ar = new int[n];
    srand(time(NULL));
    cout << "ar:  ";
    // Заповнення і виведення початкового масиву
    for(i=0; i<n; i++)
    {
       scale = rand()/double(RAND_MAX);
       ar[i] = int(scale*40 - 19.5);
       cout<<ar[i]<<"  ";
    }
    cout<<endl;
    int s=Sum_Rec(ar,n); // m - кількість додатних
    cout<<"s="<<s<<endl;
    delete [] ar;
  }
  cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Exampl...
Enter n (n>0): 10
ar:  -2  -16  -8  11  -12  -2  -3  15  18  5
s=6
Enter n (n>0): 8
ar:  -2  -3  -1  9  -2  16  11  1
s=29
Enter n (n>0): 6
ar:  -2  -4  -16  17  -4  -8
s=-17
Enter n (n>0): 5
ar:  -2  -4  9  -14  -7
s=-18
Enter n (n>0):
```

4.3.2. Вказівник на функції

З попереднього викладу зрозуміло, що вказівник зберігає *адресу* розміщення у пам'яті деякого об'єкта (змінної, масиву, екземпляра структури чи класу тощо). Це дає змогу звертатися у різний час до різних об'єктів *спільного типу* через один вказівник.

Однак вказівник також може зберігати *адресу* розміщення у пам'яті довільної функції (у цьому випадку говорять про *вказівник на функції*). Це дає змогу викликати функцію, адреса якої зберігається у цьому вказівнику. Змінюючи значення вказівника на функції, можна почергово викликати різні функції.

За аналогією з масивами, можна розтлумачити по-іншому. Назва функції – це *сталий вказівник* на функцію, яка тотожна адресі точки входу в конкретну функцію (адреса першої машинної команди). *Вказівник-змінна* на функції – це змінна, значеннями якої є *сталі вказівники* на функцію.

Для забезпечення коректного виклику функції через вказівник-змінну на функції необхідно, щоб цей вказівник підтримував інформацію про типи параметрів функції та тип результату функції. Це необхідно для того, щоб виокремити *сталі вказівники*, які можуть бути значеннями вказівника-змінної на функції. Вказівник-змінну на функції визначають так:

```
return_type (*name) (list_type);
```

де `return_type` – тип результату функції; `name` – ідентифікатор вказівника-змінної на функції; `list_type` – список типів параметрів функції. Надалі вживатимемо термін *вказівник на функції*, розуміючи під цим *вказівник-змінну на функції*. Наприклад, інструкція:

```
int (*fun)(int, int*);
```

визначає ідентифікатор `fun` як вказівник на функцію з двома параметрами (типу `int` і вказівника на `int`). Сама ж функція `fun` має повертати значення типу `int`. Округлі дужки, які обмежують вказівник на функцію, обов'язкові. Без них конструкцію

```
int *fun(int, int *);
```

інтерпретуватимуть як прототип функції `fun`, яка повертає вказівник на `int`.

Вказівник на функцію, зазвичай, використовують для *непрямого виклику* функцій та для *передачі назви* функції як аргументу в іншу функцію. Розглянемо застосування непрямого виклику функцій на базі прикладу 12.

Приклад 12. Скласти програму, в якій використано непрямі виклики функцій для організації найпростіших обчислень.

➤ *На вході* – нічого.
На виході. Одержуємо значення заданих числових виразів.
Програма:

```
#include <iostream>
using namespace std;
//---------------------------------------------------------------
int Sum(const int a, const int b)
// Додавання цілих чисел
// Передумова: a, b - довільні цілі числа.
// Постумова. Сума a+b - результат функції Sum.
{  return a+b; }
//---------------------------------------------------------------
int Prod(const int a, const int b)
// Множення цілих чисел
// Передумова: a, b - довільні цілі числа.
// Постумова. Добуток a*b - результат функції Prod.
{  return a*b; }   чсим
//------------------г8н2------------------------------------------
----
void main()
{ int (*calc)(int, int); // Вказівник на функції
  calc=Sum;
  cout<<endl << "4+7= " <<calc(4,7);
  calc=Prod;
  cout<<endl << "4*7= " <<calc(4,7);
  cout<<endl << "(2*3+5)*4= " <<
                       calc(Sum(calc(2,3),5),4);
  cout<<endl;
  int n; cin>>n; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_12\...
4+7= 11
4*7= 28
(2*3+5)*4= 44
```

Увага! У попередньому прикладі обчислення виразів можна реалізувати тільки за допомогою функцій `Sum` і `Prod` без використання механізму непрямих викликів функцій. Цей механізм вжито лише з навчально-методичною метою.

Якщо вказівник на функції використовують для *передачі назви* функції як аргументу в іншу функцію, то це, зазвичай, зв'язано з реалізацією *узагальненого* алгоритму (тобто алгоритму, інваріантного як до різних даних, так і до певних дій).

У C++ узагальнений алгоритм визначають за допомогою функції, в якій хоча б один із параметрів є вказівником на функції. Під час звертання до цієї функції параметр-вказівник замінюється назвою функції, яка має таку ж саму *сигнатуру* (кількість і типи параметрів, тип функції), як параметр-вказівник.

Розглянемо застосування вказівника на функції для реалізації *узагальненого* алгоритму на базі прикладу 13.

Приклад 13. Скласти програму, в якій використано вказівник на функції для реалізації узагальненого алгоритму *обчислення суми* натуральних логарифмів та *суми* кубів елементів одновимірного масиву.

➢ *Попередні міркування*. Сформуємо тестовий одновимірний масив `ar` у головній функції шляхом задавання початкових значень під час його визначення. Узагальнений алгоритм обчислення *суми значень деякої функції* `f` від елементів довільного одновимірного масиву а реалізуємо за допомогою функції `SumAr`. Третій параметр `f` – вказівник на функції із сигнатурою

```
double назва_функції(const double)
```


Цій сигнатурі у програмі відповідають сигнатури функції `Ln` (одержання натурального логарифма числа) та `Cub` (одержання куба числа). Якщо під час звертання до функції `SumAr` параметр-вказівник `f` замінити назвами функції `Ln` / `Cub`, то одержимо, відповідно, суму натуральних логарифмів / кубів елементів масиву `ar`.

На вході – нічого.

На виході. Одержуємо значення елементів масиву (`ar:`), суму натуральних логарифмів (`Sum_Ln=`) та суму кубів (`Sum_Cub=`) елементів масиву `ar`.

Програма:

```
#include <iostream>
#include <cmath>
using namespace std;
//--------------------------------------------------------
double Ln(const double a)
// Повертає натуральний логарифм числа a.
// Передумова: a - додатне раціональне число.
// Постумова: ln(a) - результат функції Ln.
{  return log(a); }
//--------------------------------------------------------
double Cub(const double a)
// Повертає куб числа a.
// Передумова: a  - довільне раціональне число.
// Постумова: a*a*a - результат функції Cub.
{  return a*a*a; }
//--------------------------------------------------------
double SumAr(const double *a,const int n,
                           double (*f)(const double) )
// Повертає суму значень функції f від елементів a.
// Передумова: елементи a - додатні раціональні числа.
// Постумова: сума значень - результат функції SumAr.
{
  double s=.0;
  for (int i=0; i< n; i++) s+=f(a[i]);
  return s;
}
```

```
//-----------------------------------------------------------
void main()
{ double ar[]={6, 3, 4.5, 1.4, 2.3, 5.6, 2, 1,6};
  int size=sizeof(ar)/sizeof(ar[0]), i;
  // Виведення елементів масиву
  cout << "ar:  ";
  for(i=0; i<size; cout<<ar[i]<<"  ", i++); cout<<endl;
  // Виведення суми логарифмів елементів масиву
  cout << "Sum_Ln=  "<<SumAr(ar,size,Ln)<<endl;
  // Виведення суми кубів елементів масиву
  cout << "Sum_Cub=  "<<SumAr(ar,size,Cub)<<endl;
  cin>>size;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_13...
ar:  6  3  4.5  1.4  2.3  5.6  2  1  6
Sum_Ln=  9.7715
Sum_Cub=  749.652
```

Для поліпшення читабельності та зрозумілості програми за допомогою оператора typedef визначимо новий *тип* pf – вказівник на функції із сигнатурою double назва_функції(const double).

Необхідні зміни у програмі відображає такий фрагмент програми:

```
…
typedef double (*pf)(const double);
double SumAr(const double *a,const int n, pf f)
// Повертає суму значень функції f від елементів масиву
a.
…
{  double s=.0;
   for (int i=0; i< n; i++) s+=f(a[i]);
   return s;
}
…
```


4.4. Багатомодульні проекти

4.4.1. Відокремлене компілювання модулів

Нагадаємо, що програма на C++, як фізична структура, складається з множини незалежних модулів. Розрізняють *вихідні* (початкові), *об'єктні* та *виконавчі* модулі.

До цього моменту довільна навчальна програма розміщувалася в *одному* модулі, що складався з файлу реалізації. Однак реальні програми є доволі складними сутностями, які підтримують розгалужену логіку функціонування. Отож здебільшого програми мовою C++ містять декілька модулів. Метою цього пункту є поглиблення знань про модульну структуру програми, базові аспекти якої розглянуто у параграфі 1.2.

Отже, початковий текст програми мовою C++ міститься в одному чи декількох вихідних модулях. Окремої конструкції модуля у C++ немає; модуль ототожнюють з парою <файл_специфікацій; файл_реалізації>, які мають спільне призначення.

З метою зручності тестування та налагодження програми файл_специфікацій та файл_реалізації конкретного модуля, зазвичай, мають спільну назву (хоча це й не обов'язково). У найпростіших програмах файл специфікацій можуть не використовувати (як це було у наших навчальних програмах до цього часу).

Файл *специфікацій* (файл заголовків, заголовковий файл, header file, include file) має розширенням .h, а файл *реалізації* (вихідний файл, source file) – розширення .cpp.

Зазвичай, файли специфікацій містять директиви передпроцесора та *інтерфейсні частини* модулів (символьні константи; прототипи функцій; оголошення структур, класів і шаблонів; визначення глобальних змінних). Файли реалізації містять *визначення* змінних, методів класів, функцій тощо. Для забезпечення максимальної автономності файлу *реалізації* модуля необхідно у його файлі *специфікації* підключати усі потрібні *стандартні* файли специфікацій.

У файлах специфікацій не слід розміщувати визначення функцій, оскільки це може спричинити порушення правила *одного* визначення. Не слід також використовувати директиву #include для вставлення файлів реалізації в будь-які інші файли.

Процес підготовки програми Visual C++ до виконання (тобто процес створення *виконавчого* модуля) складається з трьох етапів: опрацювання директив передпроцесора, компілювання та компонування програми (див. рис. 1 на ст. 22).

На першому етапі працює *передпроцесор* (*preprocessor*) – програма, яка перед компілюванням програми виконує опрацювання директив передпроцесора, розміщених у вихідних модулях. Передпроцесор цілковито обробляє файл *специфікацій*, а файл *реалізації* готує для подальшої обробки компілятором.

Результатом роботи передпроцесора є *розширений* вихідний модуль на базі *файлу реалізації*. Цей модуль є внутрішнім об'єктом проекту (не доступним програмістові), з яким надалі працюватиме компілятор.

На другому етапі працює *компілятор* (*compiler*) – програма, яка переводить (компілює) кожний розширений вихідний модуль у машинний код. Результатом цього компілювання є набір *об'єктних модулів* (файлів з розширенням `.obj`) - для кожного розширеного вихідного модуля створюється власний *об'єктний* модуль.

Відокремлене компілювання розширених вихідних модулів є значною перевагою системи програмування на C++. Якщо зміни вносилися тільки в один модуль, то можна перекомпілювати тільки його та зв'язати з раніше відкомпільованими версіями інших модулів. Цей механізм полегшує роботу з великими програмами. У середовищі розробки Visual Studio є засоби, які автоматично відстежують внутрішні залежності між модулями програми, а також те, коли ці модулі змінювалися останній раз.

На третьому етапі працює *компонувальник* (*linker*) – програма, яка з'єднує коди об'єктних модулів з кодами стандартних функцій із бібліотечних файлів, а також з'єднує всі об'єктні модулі в єдине ціле – *виконавчий модуль* (файл з розширенням `.exe`).

Водночас з відокремленою компіляцією зв'язані певні проблеми. Якщо у визначеннях двох різних функцій з двох різних файлів реалізації використовують одні і ті самі оголошення певних сутностей, то ці оголошення необхідно розмістити в обох файлах. У протилежному випадку компілятор генеруватиме повідомлення про невідомий тип, оскільки ці файли компілюються окремо.

Однак просте копіювання цих оголошень в усіх файлах, де їх застосовують, може зумовити до нових проблем, оскільки необхідно постійно пам'ятати, що зміни треба вносити в обидва файли, а це може легко зумовити до помилок.

Цю проблему успішно розв'язують за допомогою *додаткового* файлу специфікацій. Достатньо ввести у нього усі необхідні оголошення, а потім за допомогою директиви `#include` ввести його у всі файли реалізації, де ці оголошення використовують. За такого підходу для зміни оголошення сутності достатньо буде зробити це один раз у відповідному файлі специфікацій.

У найпростішому випадку проект програми може містити тільки два модулі:

– *модуль функцій* проекту, який складатиметься з файлу специфікацій (містить оголошення струкгур даних і прототипи функцій, які використовують ці структури) та файлу реалізації (містить визначення функцій, прототипи яких розміщені у файлі специфікацій);
– *модуль тестування* функцій, що складатиметься тільки з одного файлу реалізації з програмним кодом функції `main()`, в якій ці функції тестуватимуть.

Така стратегія дає змогу *неодноразово* використовувати раніше реалізовані функції та інші сутності (структури, класи тощо) з інших проектів. Для цього достатньо скопіювати у папку проекту вихідний модуль (відповідні файли специфікацій та реалізації) і підключити файли модуля до проекту за допомогою вікна `Solution Explorer` (див. рис. 4 на ст. 39).

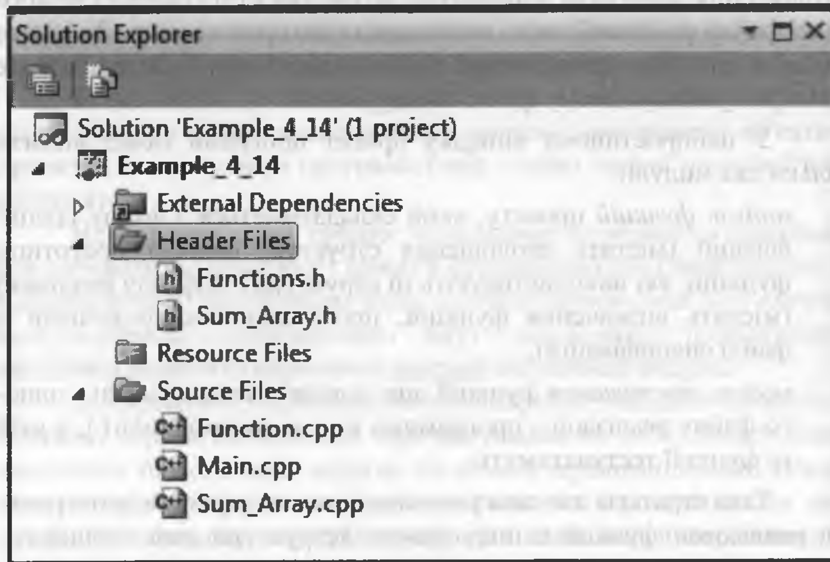
Приклад 14. За умовами прикладу 13 скласти програму з декількома модулями (*багатомодульний проект*).

➢ *Попередні міркування.* Багатомодульний проект створимо за допомогою шаблону порожнього проекту (`Empty project`). Усі необхідні файли додаватимемо до проекту вручну за допомогою вікна `Solution Explorer`. У програмі доцільно створити три модулі:

– *модуль* `SumArray` для функції `SumAr` обчислення суми значень деякої функції `f` від елементів довільного одновимірного масиву (містить файли `SumArray.h` і `SumArray.cpp`);

– *модуль* `Functions` для функцій `Ln` (одержання натурального логарифма числа) та `Cub` (одержання куба числа), містить файли `Functions.b` і `Functions.cpp`;
– *модуль* `Main` *тестування* функцій, що складатиметься тільки з одного файлу реалізації `Main.cpp`.

Багатомодульний проект прикладу 14 (`Example_4_14`):



Файл специфікацій модуля `SumArray`:

```
// Файл Sum_Array.h
#ifndef _Sum_Array_h
#define _Sum_Array_h
//--------------------------------------------------------
typedef double (*pf)(const double);
double SumAr(const double *ar, const int n, pf f);
// Повертає суму значень f від елементів масиву ar.
// Передумова: елементи ar - додатні раціональні числа.
// Постумова: сума значень - результат функції SumAr.
//--------------------------------------------------------
#endif // _Sum_Array_h
```


Файл реалізації модуля `SumArray`:

```
// Файл Sum_Array.cpp
#include "Sum_Array.h"
double SumAr(const double *ar, const int n, pf f)
{
   double s=.0;
   for (int i=0; i< n; i++) s+=f(ar[i]);
   return s;
}
```

Файл специфікацій модуля `Functions`:

```
// Файл Functions.h
#ifndef _Functions_h
#define _Functions_h
#include <cmath>
//----------------------------------------------------------
double Ln(const double a);
// Повертає натуральний логарифм числа a.
// Передумова: a - додатне раціональне число.
// Постумова: ln(a) - результат функції Ln.
//----------------------------------------------------------
double Cub(const double a);
// Повертає куб числа a.
// Передумова: a - довільне раціональне число.
// Постумова: a*a*a - результат функції Cub.
//----------------------------------------------------------
#endif // _Functions_h
```

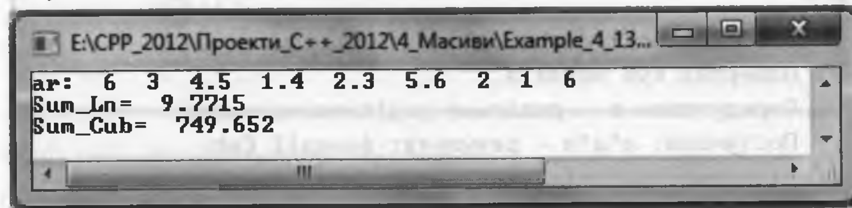
Файл реалізації модуля `Functions`:

```
// Файл Function.cpp
#include "Functions.h"
//----------------------------------------------------------
double Ln(const double a)
{  return log(a); }
//----------------------------------------------------------
double Cub(const double a)
{  return a*a*a; }
```

Файл реалізації модуля Main:

```
// Файл Main.cpp
#include <iostream>
#include "Functions.h"
#include "Sum_Array.h"
using namespace std;
void main()
{ double ar[]={6, 3, 4.5, 1.4, 2.3, 5.6, 2, 1,6};
  int size = sizeof(ar)/sizeof(ar[0]), i;
  // Виведення елементів масиву
  cout << "ar:  ";
  for(i=0; i<size; cout<<ar[i]<<"  ", i++); cout<<endl;
  // Виведення суми логарифмів елементів масиву
  cout << "Sum_Ln=  "<<SumAr(ar,size,Ln)<<endl;
  // Виведення суми кубів елементів масиву
  cout << "Sum_Cub=  "<<SumAr(ar,size,Cub)<<endl;
  cin>>size;
}
```

Результати тестування програми:



⮜

Як бачимо, результати виконання програм прикладів 13 і 14 цілковито співпадають, чого ми й очікували. Модулі SumArray і Functions відтестовані, їх можна неодноразово використовувати.

Однак організація багатомодульного проекту в прикладі 14 має очевидний *недолік*: файл Sum_Array.h у проекті підключаємо у двох файлах реалізації (Sum_Array.cpp і Main.cpp). З метою усунення цього недоліку під час створення багатомодульного консольного проекту скористаємося шаблоном Win32 Console Application.

4.4.2. Шаблон багатомодульного консольного проекту

Під час створення довільного проекту Visual Studio генерує *декілька файлів*, які складають *каркас* застосування (програми). Щоб не затіняти початкове знайомство з мовою С++ технічними деталями, ми досі усі навчальні програми реалізовували за допомогою *одного* файлу реалізації, використовуючи під час створення проекту шаблон порожнього проекту (Empty project). Усі необхідні файли у цих випадках додавалися до проекту вручну за допомогою вікна Solution Explorer.

Якщо певні функції готують для багаторазового використання у різних проектах, то цей підхід не годиться. Отже, навіть у простих проектах доцільно використовувати багатомодульну організацію. Для створення багатомодульного консольного проекту у середовищі Visual Studio використовують шаблон Win32 Console Application. Використання цього шаблону пояснюватимемо на такому прикладі.

Приклад 15. За умовами прикладу 13 скласти програму з декількома модулями (*багатомодульний проект*), використовуючи шаблон Win32 Console Application.

Оскільки весь наступний матеріал до кінця цього пункту присвячено виконанню прикладу 15, то спеціальних позначок початку (⮞) та завершення прикладу (⮜) не використовуватимемо.

Виконаємо такі команди: ➤ File ➤ New ➤ Project … Після цього відкривається вікно New Project (див. рис. 3 на ст. 38). Далі виконаємо таке:

⇩ Installed Template: ⬀ Win32 ⬀ Win32 Console Application

Name:= Example_4_15   // Назва / папка проекту

Location:= E:\CPP_2012\Проекти_С++_2012\

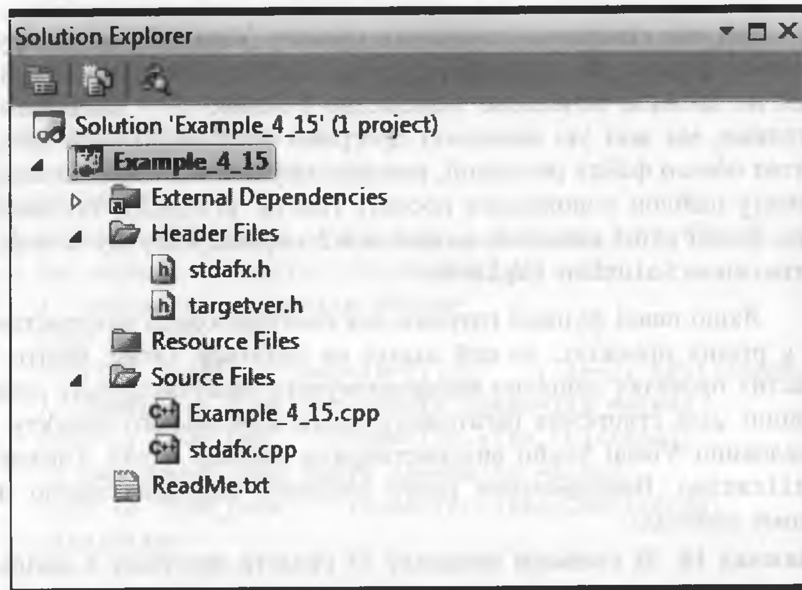
// Каталог обираємо через навігацію за допомогою ❑Browse

☑ (зняти позначку) Create directory for solution ❑ Ok.

➤ Next // Перехід на нове діалогове вікно

☑ Precompiled header ⦿ Console Application ➤ Finish

Після цього Visual Studio генерує такий *каркас* проекту:



Розглянемо вмістиме файлів каркасу проекту.

Файл targetver.h:

```
#pragma once
// Including SDKDDKVer.h defines the highest available
// Windows platform.
// If you wish to build your application for a previous
// Windows platform, include WinSDKVer.h and
// set the _WIN32_WINNT macro to the platform you wish to
// support before including SDKDDKVer.h.
#include <SDKDDKVer.h>
```

Отже, якщо перекласти коментарі цього файлу, то стане зрозуміло, що включення SDKDDKVer.h забезпечує визначення найпізнішої доступної платформи Windows для побудови проекту. Якщо потрібно виконати побудову проекту для попередньої версії Windows, то необхідно додатково включати WinSDKVer.h і задавати у макросі _WIN32_WINNT значення цієї платформи перед включенням SDKDDKVer.h. Надалі на цей файл уваги не звертатимемо.

Файл stdafx.h (коментарі перекладено):

```
// stdafx.h: файл специфікацій для підключення системних
// стандартних файлів специфікацій та додаткових файлів
// специфікацій проекту, які часто використовуються,
// однак рідко змінюють.
#pragma once
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
// TODO: тут підключають додаткові файли специфікацій
```

Файл stdafx.cpp (коментарі перекладено):

```
// stdafx.cpp : файл реалізації, що використовують
// для побудови передкомпільованого файлу заголовків
// Example_4_15.pch і передкомпільованого файлу типів
// stdafx.obj
#include "stdafx.h"
// TODO: додаткові файли специфікацій необхідно
// підключати у STDAFX.H, а не у цьому файлі
```

Файли stdafx.h і stdafx.cpp використовують для побудови передкомпільованого файлу заголовків Example_4_15.pch і передкомпільованого файлу типів stdafx.obj. Створення цих файлів дає змогу значно прискорити роботу компілятора.

Як відомо у С++ модульне компонування – кожен розширений модуль компілюється окремо. Оскільки обсяг деяких стандартних файлів специфікацій перевищує певні розумні межі (наприклад, windows.h підключає ще до десяти win*.h), то обробляти усі ці файли для кожного файлу реалізації доволі затратно.

У РСН-файлі створюється загальна база даних включень файлів специфікацій для всього проекту. Використовуючи цю базу даних, компілятор обробляє розширені модулі значно швидше.

За домовленістю для довільного файлу реалізації у проекті за шаблоном Win32 Console Application *автоматично* включається опція "Use Precompiled Header" з файлом stdafx.h.

Оскільки для автономності модуля необхідно, щоб його файл реалізації зв'язувався тільки з відповідним файлом специфікації, то треба вміти відключати опцію "Use Precompiled Header" у файлі реалізації. Для цього у вікні Solution Explorer виконують такі дії:

⇗Назва_файлу.cpp ❑ ПКМ // натискаємо праву кнопку миші

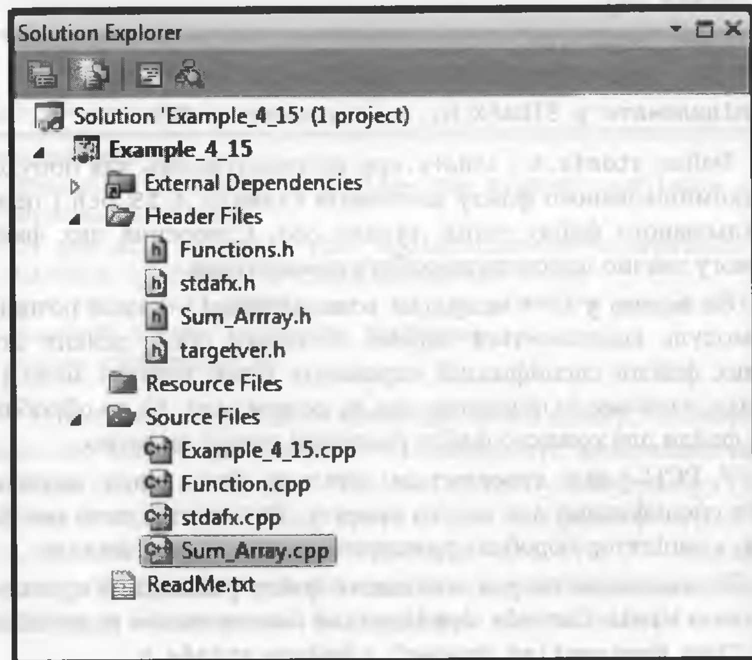
⇗ Properties ❑ ЛКМ // відкривається діалогове вікно

⇓ C/C++ ⇗ Precompiled Headers

⇓ Precompiled Header ⇗ Not Using Precompiled Headers ❑ Ok

За допомогою вікна Solution Explorer підключимо до нашого проекту Example_4_15 файли SumArray.h, SumArray.cpp, Functions.h і Functions.cpp з проекту Example_4_14 (ці файли необхідно попередньо скопіювати у папку проекту Example_4_15). Для файлів SumArray.cpp і Functions.cpp перевіримо факт *відключення* опції "Use Precompiled Header".

Після цього проект набуде такого вигляду:



Модифікуємо файл stdafx.h (коментарі опущено):

```
// stdafx.h : файл специфікацій …
#pragma once
#include "targetver.h"
#include <stdio.h>
#include <tchar.h>
// TODO: тут підключають додаткові файли специфікацій
#include <iostream>
#include "Functions.h"
#include "Sum_Arrray.h"
using namespace std;
```

Файл специфікацій модуля SumArray (без змін, див. на ст. 288).
Файл реалізації модуля SumArray (без змін, див. на ст. 289).
Файл специфікацій модуля Functions (без змін, див. на ст. 289).
Файл реалізації модуля Functions (без змін, див. на ст. 289).

Файл Example_4_15.cpp:

```
// Example_4_15.cpp: Тестування модуля SumArray
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{ double ar[]={6, 3, 4.5, 1.4, 2.3, 5.6, 2, 1,6};
  int size=sizeof(ar)/sizeof(ar[0]), i;
  // Виведення елементів масиву
  cout << "ar:  ";
  for(i=0; i<size; cout<<ar[i]<<"  ", i++); cout<<endl;
  // Виведення суми логарифмів елементів масиву
  cout << "Sum_Ln=  "<<SumAr(ar,size,Ln)<<endl;
  // Виведення суми кубів елементів масиву
  cout << "Sum_Cub=  "<<SumAr(ar,size,Cub)<<endl;
  cin>>size; return 0;
}
```

У файлах специфікацій *модулів* (SumArray.h і Functions.h) для уникнення зациклювання і повторного компілювання директив #include використані *охоронці включення* (include guards; див. 1.3). У файлах stdafx.h і targetver.h для цього використано

директиву #pragma once. Це зроблено свідомо, як з навчально-методичною метою, так і для того, щоб ідентифікувати різні типи файлів специфікацій.

Оскільки головною метою створення модуля є забезпечення можливості його повторного виконання, то модуль має бути максимально *автономним*. Пояснимо це для модуля Functions. У файлі Function.cpp використана стандартна функція log(·). Щоб компілятор правильно опрацював назву log, необхідно підключити бібліотеку математичних функцій (#include <cmath>).

Це підключення здійснено у файлі Function.h, адже саме файли Function.cpp і Function.h копіюють і підключають до інших проектів. Якщо ж #include <cmath> розмістити у stdafx.h, то в нових проектах про це треба *пам'ятати* і розміщувати #include <cmath> у stdafx.h по-новому!

Винятком є файл тестування з головною функцією _tmain (у нашому випадку Example_4_15.cpp), директиви для роботи якого, зазвичай, розміщують саме у файлі stdafx.h.

4.4.3. Узагальнений алгоритм обчислення визначених інтегралів

Тепер настала черга закрити питання, зв'язані з обчисленням визначених інтегралів (див. 3.5.3) та розв'язуванням рівнянь з однією змінною (див. 3.5.4). В обох випадках реалізація *методів* (обчислення інтегралу чи розв'язування рівняння) є *недосконалою*, оскільки існує жорсткий зв'язок між функцією методу (integral_prm2 чи binary_division) і функцією f. Щоб функції методів стали універсальними, їм необхідно передавати назву конкретної функції f. Покажемо це для функції integral_prm2.

Раніше було введено поняття *узагальненого* алгоритму як алгоритму, інваріантного і до різних даних, і до певних *дій*. У С++ узагальнений алгоритм визначають за допомогою функції, в якій хоча б один із параметрів є вказівником на функції.

Під час звертання до цієї функції параметр-вказівник заміняється назвою конкретної функції, яка має таку ж саму *сигнатуру*, що і цей параметр. Конкретні функції мають різний вигляд, отож реалізують *різні дії*, які полягають в обчисленні значень функцій.

Якщо у множину параметрів функції методу обчислення визначеного інтегралу integral_prm2 ввести параметр, що є вказівником на підінтегральну функцію f, то функція integral_prm2 стане універсальною (може повертати значення визначеного інтегралу для *довільної* підінтегральної неперервної функції однієї змінної).

Відповідний алгоритм обчислення визначених інтегралів методом лівих прямокутників буде узагальненим, оскільки він стає інваріантним до дії, яка зв'язана з обчисленням значень підінтегральної функції.

Приклад 16. Створити і протестувати модуль для обчислення визначених інтегралів за узагальненим алгоритмом лівих прямокутників.

➢ *Попередні міркування*. Теоретичні положення обчислення визначених інтегралів за алгоритмом лівих прямокутників розглянуто у 3.5.3.

Багатомодульний проект створимо за допомогою шаблону Win32 Console Application.

У програмі доцільно створити три модулі:

– модуль My_Integral для функції integral_prm2 обчислення визначених інтегралів для деякої функції f за узагальненим алгоритмом лівих прямокутників (містить файли My_Integral.h і My_Integral.cpp);
– модуль Functions для функцій Ln (одержання натурального логарифма числа) та Cub (одержання куба числа), містить файли Functions.h і Functions.cpp;
– модуль тестування функцій, що складатиметься тільки з одного файлу реалізації Example_4_16.cpp.

Файл stdafx.h (опущено стандартні підключення):

```
…
#include <iostream>
#include "Functions.h"
#include "My_Integral.h"
using namespace std;
```

Файли модуля `Functions` див. на ст. 289.

Файл специфікацій модуля `My_Integral`:

```
// Файл My_Integral.h
#ifndef _My_Integral_h
#define _My_Integral_h
//---------------------------------------------------------
typedef double (*pf)(const double);
double integral_prm2(double a, double b, double eps, pf f);
// Обчислення визначених інтегралів методом лівих
// прямокутників від підінтегральної функції f
// на проміжку [a; b].
// Передумова: a<b (перевірка у функції відсутня).
// Постумова: результат функції - значення інтеграла.
//---------------------------------------------------------
#endif // _My_Integral_h
```

Файл реалізації модуля `My_Integral`:

```
// Файл My_Integral.cpp
#include "My_Integral.h"
double integral_prm2(double a, double b, double eps, pf f)
{double h, v, w, s=0.;
 // v - наближення інтеграла після k-го поділу
 // w - наближення інтеграла після (k+1)-го поділу
 // s - накопичення суми значень функції
 int i, k=0, n=10;  h=(b-a)/n;
 // k - номер поділу; для уникнення зациклення: k < 10
 for(i=0; i<=n-1; s+=f(a+i*h), i++);
 v=0; w=h*s; // Тут w - початкове наближення інтеграла
 while (fabs(v - w)/3>=eps && k<10)
  { k++; v=w; for(i=0; i<=n-1; s+=f(a+h/2+i*h), i++);
    h=h/2; w=h*s; // w - наступне наближення інтеграла
    n*=2;
  }
 return w;
}
```

Можна легко визначити, що $\int x^3 dx = \frac{1}{4} \cdot x^4 + c, c = \text{const};$ $\int \ln x dx = x(\ln x - 1) + c, c = \text{const}, x > 0.$

На базі цих первісних можна обчислити такі інтеграли: $\int_1^3 x^3 dx = 20;$ $\int_2^7 x^3 dx = 596{,}25;$ $\int_1^3 \ln x dx \approx 1{,}29583;$ $\int_2^7 \ln x dx \approx 7{,}23507.$

Модуль *тестування*:

```
// Example_4_16.cpp
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{double a, b, eps=1e-12;
 while (cout<<"Enter a b (a<b): ", cin>>a>>b, a!=-99)
  {if(a>b) {double r=a; a=b; b=r;} // Забезпечення a>b
   cout<<"Int_Ln="<<integral_prm2(a, b, eps, Ln)<<endl;
   cout<<"Int_Cub="<<integral_prm2(a,b,eps,Cub)<<endl;
  }  cin>>a; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_...
Enter a b (a<b): 1 3
Int_Ln=  1.29573
Int_Cub=  19.9975
Enter a b (a<b): 2 7
Int_Ln=  7.23477
Int_Cub=  596.168
Enter a b (a<b): -99 5
```

Читачам пропоную самостійно узагальнити алгоритм бінарного поділу відрізка ізоляції розв'язку рівняння (див. 3.5.4).

4.4.4. Сервісний модуль обробки масивів

Під час тестування алгоритмів роботи з масивами необхідно спочатку *заповнити* масив початковими даними і *вивести* його на консоль. Після виконання алгоритму на консоль виводять результати: це можуть бути як деякі скалярні величини, так і весь масив.

Створимо *сервісний модуль*, який міститиме дві функції; одна заповнюватиме масив випадковими числами з деякого проміжку, а інша – виводитиме масив на екран консолі. Необхідно мати два модулі: для одновимірних і багатовимірних масивів.

Спочатку створимо сервісний модуль `Array_1` з функціями:

- `Fill_Array_1_Int` – заповнюватиме одновимірний масив `ar` розміру `n` випадковими цілими числами з проміжку `[a; b]`;
- `Print_Array_1` – виводитиме одновимірний масив `ar` на екран.

Файл специфікацій модуля `Array_1`:

```
// Файл Array_1.h: заповнення і виведення масиву
#ifndef _Array_1_h
#define _Array_1_h
#include<cstdlib>
#include<ctime>
#include <iostream>
using namespace std;
//--------------------------------------------------
void Fill_Array_1_Int(int *ar, int n, int a, int b);
// Заповнює одновимірний масив ar розміру n випадковими
// цілими числами з проміжку [a; b].
// Передумова: a<b (за потреби функція корегує a та b).
// Постумова: повертає заповнений масив ar.
//--------------------------------------------------
void Print_Array_1(int *ar, int n);
// Виводить на екран консолі елементи одновимірного масиву.
#endif // _Array_1_h
```

Файл реалізації модуля `Array_1`:

```
#include "Array_1.h"
void Fill_Array_1_Int(int *ar, int n, int a, int b)
{ double scale; if(a>b) {int r=a; a=b; b=r;}
  srand(time(NULL));
  for(int i=0; i<n; i++) { scale=rand()/double(RAND_MAX);
                           ar[i]=int(a+scale*(b-a)+0.5);}
}
void Print_Array_1(int *ar, int n)
{ for(int i=0; i<n; i++) cout<<ar[i]<<" "; cout<<endl; }
```


Приклад 17. Відтестувати сервісний модуль обробки одновимірних масивів `Array_1`.

➢ Тестування модуля полягатиме у створенні динамічного масиву розміру *n*, заповнення його випадковими цілими числами з проміжку [*a*; *b*] і виведення на екран консолі. Величини *n*, *a* і *b* задаються користувачем у режимі діалогу. Багатомодульний проект створимо за допомогою шаблону `Win32 Console Application`.

На вході (`Enter a b n (a<b):`). Кожен такий рядок є окремим тестом, що задає значення змінних *a* і *b* (меж проміжку випадкових чисел) і змінної *n* (розміру масиву). Ознакою завершення тестування є рядок, в якому введене значення *a* дорівнюватиме –99.

На виході. Одержуємо сформований масив.

Програма тестування:

```
// Example_4_17.cpp: тестування модуля Array_1
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{int a, b, n;
 while(cout<<"Enter a b n (a<b):", cin>>a>>b>>n, a!=-99)
  { int *ar = new int[n];
    Fill_Array_1_Int(ar, n, a, b);
    Print_Array_1(ar, n);  delete [] ar;
  }
  cin>>a; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_17\Debug\...
Enter a b n (a<b): -50 50 10
-1  48  -35  30  -10  -20  -47  35  38  45
Enter a b n (a<b): 0 45 10
22  40  43  1  3  20  33  11  43  26
Enter a b n (a<b): 10 99 15
53  20  38  97  13  74  58  58  57  10  51  40  76  66  44
Enter a b n (a<b): -40 40 12
0  32  6  9  -14  -8  -20  -34  -25  15  -16  -8
Enter a b n (a<b): -99 3 5
```

➢

Після цього створимо сервісний модуль Array_2 (для обробки матриць).

Файл специфікацій модуля Array_2:

```
// Файл Array_2.h: заповнення і виведення матриці
#ifndef _Array_2_h
#define _Array_2_h
#include<cstdlib>
#include<ctime>
#include <iostream>
using namespace std;
//-------------------------------------------------------
void Fill_Array_2_Int(int **ar, int m, int n, int a, int b);
// Заповнює матрицю ar з m рядків і n стовпців
// випадковими цілими числами з проміжку [a; b].
// Передумова: a<b (за потреби функція корегує a та b).
// Постумова: повертає заповнений масив ar.
//-------------------------------------------------------
void Print_Array_2(int **ar, int m, int n);
// Виводить на екран консолі елементи матриці.
//-------------------------------------------------------
#endif // _Array_2_h
```

Файл реалізації модуля Array_2:

```
#include "Array_2.h"
void Fill_Array_2_Int(int **ar, int m, int n, int a, int b)
{ double scale; int i, j; if(a>b) {int r=a; a=b; b=r;}
  srand(time(NULL));
  for(int i=0; i<m; i++)
   for(int j=0; j<n; j++)
    { scale=rand()/double(RAND_MAX);
      ar[i][j]=int(a+scale*(b-a)+0.5); }
}
void Print_Array_2(int **ar, int m, int n)
{ int i, j;
  for(i=0; i<m; i++)
   {for(j=0; j<n; j++) cout<<ar[i][j]<<" "; cout<<endl;}
  cout<<endl;
}
```


4.5. Стандартні задачі обробки одновимірних масивів

4.5.1. Загальні положення

Математичне формулювання задачі обробки одновимірного масиву, зазвичай, починається так: "Дано натуральне число n і послідовність (сукупність, множина тощо) дійсних чисел $a_1, a_2, \ldots, a_n$. Серед послідовності цих чисел *визначити* ...".

Кількість цих чисел (величина n) наперед невідома, її вводить з клавіатури користувач. Отож для моделювання послідовності чисел $a_1, a_2, \ldots, a_n$ найкраще застосовувати одновимірний динамічний масив.

Окрім цього, для спрощення тестування замість дійсних чисел використовуватимемо цілі числа. Це даватиме змогу використовувати сервісний модуль Array_1 (див. 4.4.4).

Існує чимало різноманітних задач на одновимірні масиви. Як і всі задачі загалом, умовно їх можна розділити на три типи:

1) *прості* задачі, які розв'язують методом грубої сили (в "одне розуміння");
2) *стандартні* задачі, для розв'язування яких необхідно володіти певними загальними прийомами чи методами;
3) *нестандартні* задачі, рішення яких потребує знання термінології та методології певної предметної області, допоміжних алгоритмів, спеціальних прийомів чи методів тощо.

Очевидно, що без уміння вирішувати задачі перших двох типів неможливо вирішувати нестандартні задачі. До простих задач можна зачислити усі задачі обробки одновимірних масивів (див. 4.1 – 4.4). Це задачі:

1) визначення найбільшого/найменшого елемента масиву;
2) визначення суми/добутку елементів (безумовне й умовне);
3) підрахунок елементів, що задовольняють певній умові тощо.

Під час розв'язування *простих* задач іноді замість методу грубої сили вдається застосувати деякий "*хитрий*" прийом, який значно спрощує розв'язок задачі. Покажемо це у прикладах 18 і 19.

Приклад 18. В одновимірному масиві a кожен елемент дорівнює 0 чи 1. Замінити всі нулі одиницями, і навпаки.

➢ *Попередні міркування.* Досить однієї інструкції присвоювання:

```
for (i=0; i<n; i++) a[i] = 1-a[i];
```

⋞

Приклад 19. У масиві кожен елемент дорівнює 0, 1 чи 2. Переставити елементи масиву так, щоб спочатку розміщувалися всі 0, потім усі 1 і, нарешті, 2. Додаткового масиву не заводити.

➢ *Попередні міркування.* Можна не переставляти елементи масиву, а підрахувати кількості 0, 1 і заповнити масив у потрібний спосіб.

Фрагмент програми:

```
…
k0=0; k1=0;
for (i=0; i<n; i++)
  if (a[i] == 0) k0++; else if(a[i] == 1) k1++;
for (i=0;  i<n;  i++)   a[i] = 2;
for (i=0;  i<k0;  i++)   a[i] = 0;
for (i=k0; i<k0+k1; i++)   a[i] = 1;
…
```

⋞

Перелічимо деякі *стандартні* задачі обробки одновимірних масивів:

1) сортування масиву (прості методи);
2) визначення сусіднього розміщення елементів;
3) видалення елемента;
4) включення елемента в задану позицію;
5) перерозміщення (інвертування, циклічне зсування) елементів;
6) пошук заданого елемента:
   а) у невпорядкованому масиві; б) в упорядкованому масиві;
7) злиття двох упорядкованих масивів в один упорядкований масив тощо.

До *нестандартних* задач можна зачислити поліпшені методи сортування масивів, пошук моди і медіани масиву; алгоритми генерування комбінаторних об'єктів, алгоритми на графах тощо.

4.5.2. Схема Горнера обчислення многочлена

До стандартних задач обробки одновимірних масивів належить і задача обчислення значення многочлена степеня n у довільній точці x, оскільки коефіцієнти многочлена, зазвичай, зберігають в одновимірному масиві. Запишемо многочлен степеня n у вигляді:

$$P_n(x) = a_0 \cdot x^n + a_1 \cdot x^{n-1} + \ldots + a_{n-1} \cdot x + a_n .$$

Для організації циклічного процесу обчислення значення многочлена запишемо $P_n(x)$ у вигляді схеми Горнера:

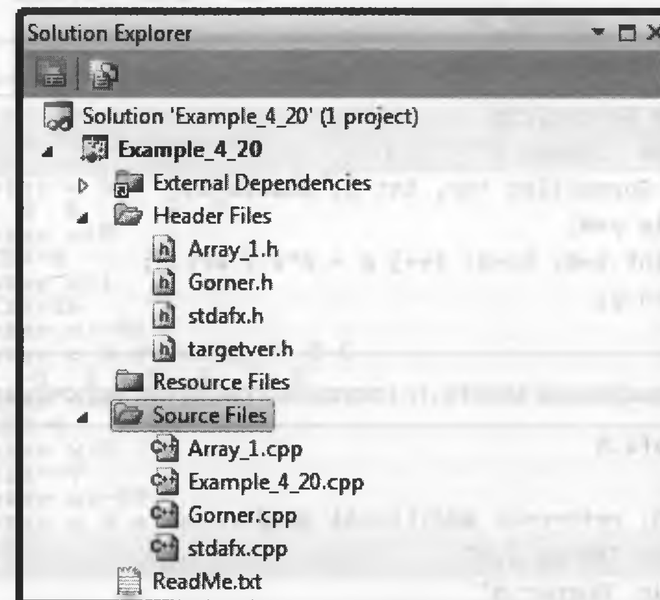
$$P_n(x) = (\ldots((a_0 \cdot x + a_1) \cdot x + a_2) \cdot x + \ldots + a_{n-1}) \cdot x + a_n .$$

Циклічний процес обчислення значення многочлена за схемою Горнера:

```
p=0; for (i=0; i<=n; i++) p = p*x + a[i];
```

Приклад 20. Скласти і протестувати модуль, який міститиме функцію обчислення значення многочлена з цілими коефіцієнтами для довільного дійсного числа за схемою Горнера.

➢ *Багатомодульний проект* прикладу 20 (`Example_4_20`):



Сервісний модуль Array_1 для одновимірних масивів (див. 4.4.4) містить файл специфікацій Array_1.h (див. ст. 300) та файл реалізації Array_1.cpp (див. ст. 300).

Файл специфікацій модуля Gorner (містить функцію Gorner обчислення значення многочлена з цілими коефіцієнтами для довільного дійсного числа за схемою Горнера):

```
// Файл Gorner.h
#ifndef _Gorner_h
#define _Gorner_h
//----------------------------------------------------
double Gorner(int *ar, int n, double x);
// Обчислює значення многочлена з цілими коефіцієнтами
// для довільного дійсного числа x за схемою Горнера.
// Передумова: n - степінь многочлена; масив ar
// розміру  n+1  містить коефіцієнти многочлена.
// Постумова: значення многочлена - результат функції.
//----------------------------------------------------
#endif // _Gorner_h
```

Файл реалізації модуля Gorner:

```
// Файл Gorner.cpp
#include "Gorner.h"
double Gorner(int *ar, int n, double x)
{ double p=0;
  for(int i=0; i<=n; i++) p = p*x + ar[i];
  return p;
}
```

Файл специфікацій stdafx.h (опущено стандартні включення):

```
// stdafx.h
…
// TODO: reference additional headers
#include "Array_1.h"
#include "Gorner.h"
```


На вході (Enter a b n (n>0):). Кожен такий рядок є окремим тестом, що задає значення змінних *a* і *b* (меж проміжку випадкових чисел) і змінної *n* (степеня многочлена). Коефіцієнти многочлена – випадкові числа з проміжку [*a*; *b*]. Ознакою завершення тестування є рядок, в якому введене значення *n* не є додатним. Після цього відбувається введення значень *x* доти, доки істинна умова (x!=-99).

На виході. Одержуємо значення многочлена у точці *x*.

Програма тестування (Example_4_20.cpp):

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{ int a, b, n; double x;
  while(cout<<"Enter a b n (n>0):", cin>>a>>b>>n, n>0)
  { int *ar = new int[n+1];
    Fill_Array_1_Int(ar, n+1, a, b);
    Print_Array_1(ar, n+1);
    while(cout<<"Enter x:", cin>>x, x!=-99)
      cout<<"p("<<x<<")="<< Gorner(ar, n, x)<<endl;
    delete [] ar;
  }  cin>>a; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Ex...
Enter a b n (n>0):1 6 5
4  4  5  6  5  2
Enter x:0
p(0)=2
Enter x:1
p(1)=26
Enter x:-99
Enter a b n (n>0):-3 3 6
0  -1  1  -1  3  1  1
Enter x:0
p(0)=1
Enter x:2
p(2)=-9
Enter x:-99
Enter a b n (n>0):2 3 -4
```

4.5.3. Прості методи сортування масиву

Сортування масиву – це переросміщення елементів масиву у заданому порядку (домовимося розглядати випадок впорядкування масиву за *неспаданням*: $a_1 \le a_2 \le ... \le a_n$). Під *внутрішнім* сортуванням розуміють сортування, яке виконується в оперативній пам'яті комп'ютера без використання допоміжних масивів. *Головна мета* сортування – полегшити подальший пошук потрібних елементів.

Сортування методом вибору. Ідея методу полягає в тому, що визначають максимальний елемент масиву і міняють його місцями з останнім елементом (з індексом $n-1$). Потім максимум визначають серед елементів від першого до передостаннього і міняють його з елементом, що має індекс $n-2$ і т. д. Можна визначати не максимум, а *мінімум* і ставити його на перше місце (з індексом 0), потім друге і т. д. Обчислювальна складність сортування методом вибору – величина порядку $n \cdot n$, яку, зазвичай, записують як $O(n^2)$. Це пояснюють тим, що кількість порівнянь під час визначення першого максимуму дорівнює $n-1$; потім $n-2, n-3, ..., 1$; разом: $n \cdot (n-1)/2$.

Наведемо фрагмент програми, що реалізує сортування *вибором*:

```
for(k = n-1; k > 0; k--)
{  m = 0; // m - початкове місце max
   for(i = 1; i <= k; i++) if (a[i] > a[m]) m = i;
   x = a[m]; a[m] =a[k]; a[k] = x;
}
```

Сортування методом обміну (метод "*бульбашки*"). Ідея методу полягає у тому, що послідовно порівнюють пари сусідніх елементів масиву. Якщо вони розміщені не в тому порядку, то здійснюємо перестановку, міняючи місцями пару сусідніх елементів. Після першого такого проходження масиву на останньому місці виявиться максимальний елемент ("спливає" перша "бульбашка"). Наступне проходження має розглядати елементи до передостаннього і т. д. Усього потрібно $(n-1)$-не проходження по масиву. Обчислювальна складність такого сортування $O(n^2)$.

Наведемо фрагмент програми, що реалізує сортування *обміном*:

```
for(k = n-2;k >= 0;k--) // Кількість пар порівняння
 for (i = 0; i <= k; i++)
  if (a[i] > a[i+1]) // Міняємо сусідні елементи
   { x = a[i]; a[i] = a[i+1]; a[i+1] = x; }
```

Сортування методом включення. Ідея методу полягає у тому, що кожного разу, маючи уже впорядкований масив з k елементів, ми додаємо ще один елемент, включаючи його в масив так, щоб впорядкованість не порушилася. Сортування може відбуватися водночас з введенням масиву. На початку сортування впорядкована частина масиву містить тільки один елемент. Різні методи пошуку місця для елемента, що включається, зумовлюють до різних модифікацій сортування включенням. Найпростішим методом пошуку у невпорядкованому масиві є лінійний пошук, який полягає у послідовному перегляді всього масиву у циклі `while` з подвійною умовою. Перша умова контролює індекс на приналежність масиву (`i>=0`), а друга – це умова продовження пошуку. У тілі циклу є інструкція, яка змінює індекс масиву.

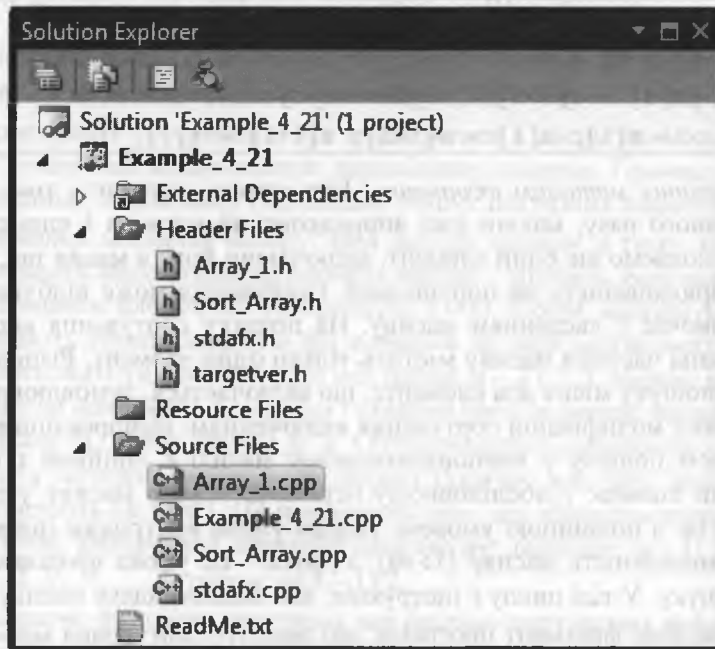
Наведемо фрагмент програми, що реалізує сортування *методом включення* за *лінійного пошуку* місця включення:

```
// k+1 - кількість елементів
// упорядкованої частини масиву
for(k = 0; k < n-1; k++)
 {  x = a[k+1]; i = k;
    while ((i>=0) && (a[i]>x)) { a[i+1] = a[i]; i--;}
    a[i+1] = x;
 }
```

Аналізуючи цей фрагмент, можна стверджувати, що під час використання лінійного пошуку обчислювальна складність методу сортування включенням становить $O(n^2)$.

Приклад 21. Скласти і протестувати модуль, який міститиме функції простих методів сортування одновимірного масиву.

➢ *Багатомодульний проект* прикладу 21 (Example_4_21):



Сервісний модуль Array_1 для одновимірних масивів (див. 4.4.4) містить файл специфікацій Array_1.h (див. ст. 300) та файлу реалізації Array_1.cpp (див. ст. 300).

Файл специфікацій модуля Sort_Array (містить функції простих методів сортування за неспаданням одновимірного масиву):

```
// Файл _Sort_Array.h: сортування масиву
#ifndef _Sort_Array_h
#define _Sort_Array_h
//  Модуль реалізації простих методів сортування
//  за неспаданням одновимірного масиву a розміру n
//  Передумова: - немає.
//  Постумова: функції повертають посортований масив a.
//------------------------------------------------------------
void Sort_Array_Select(int *a,  int n);
// Метод вибору
//------------------------------------------------------------
void Sort_Array_Change(int *a, int n);
// Метод обміну
//------------------------------------------------------------
void Sort_Array_Insert_L(int *a, int n);
// Метод включення на базі лінійного пошуку
//------------------------------------------------------------
#endif // _Sort_Array_h
```


Файл реалізації модуля Sort_Array:

```
#include "Sort_Array.h"
void Sort_Array_Select(int *a, int n)
{ int k, m, i, x;
  for(k = n-1; k > 0; k--)
   { m = 0; // m - початкове місце max
     for(i = 1; i <= k; i++) if (a[i] > a[m]) m = i;
     x = a[m]; a[m] =a[k]; a[k] = x;
   }
}

void Sort_Array_Change(int *a,  int n)
{ int k, i, x;
  for(k = n-2; k >= 0; k--) // Кількість пар порівняння
   for (i = 0; i <=k; i++)
    if (a[i] > a[i+1]) // Міняємо сусідні елементи
     { x = a[i]; a[i] = a[i+1]; a[i+1] = x; }
}

void Sort_Array_Insert_L(int *a,  int n)
{
  int k, i, x;
  // k+1 - кількість елементів
  // упорядкованої частини масиву
  for(k = 0; k < n-1; k++)
   {  x = a[k+1]; i = k;
      while ((i>=0) && (a[i]>x)) { a[i+1] = a[i]; i--;}
     a[i+1] = x;
   }
}
```

Файл специфікацій `stdafx.h` (опущено стандартні включення):

```
// stdafx.h
…
// TODO: reference additional headers
#include "Array_1.h"
#include "Sort_Array.h"
```

На вході (`Enter a b n (n>0):`). Кожен такий рядок є окремим тестом, що задає значення змінних *a* і *b* (меж проміжку випадкових чисел) і змінної *n* (розміру масиву). Елементи масиву *ar* – випадкові числа з проміжку [*a*; *b*]. Ознакою завершення тестування є рядок, в якому введене значення *n* не є додатним.

На виході. Одержуємо упорядкований різними методами масив *ar*.

Програма тестування (`Example_4_21.cpp`):

```
// Example_4_21.cpp
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{ int a, b, n; double x;
  while(cout<<"Enter a b n (n>0):", cin>>a>>b>>n, n>0)
  { int *ar = new int[n];
    Fill_Array_1_Int(ar, n, a, b);
    cout<<"ar1: "; Print_Array_1(ar, n);
    Sort_Array_Select(ar, n); // Метод вибору
    cout<<"Sort_Array_Select: "<<endl;
    Print_Array_1(ar, n);
    Fill_Array_1_Int(ar, n, 2*a, 2*b);
    cout<<"ar2: "; Print_Array_1(ar, n);
    Sort_Array_Change(ar, n); // Метод обміну
    cout<<"Sort_Array_Change: "<<endl;
    Print_Array_1(ar, n);
    Fill_Array_1_Int(ar, n, 3*a, 3*b);
    cout<<"ar3: "; Print_Array_1(ar, n);
    Sort_Array_Insert_L(ar, n); // Метод включення
    cout<<"Sort_Array_Insert_L: "<<endl;
    Print_Array_1(ar, n);
    delete [] ar;
  }  cin>>a; return 0;
}
```


Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Ex...
Enter a b n (n>0):-10 10 10
ar1: 10  6  4  -3  1  6  0  3  2  4
Sort_Array_Select:
-3  0  1  2  3  4  4  6  6  10
ar2: 20  13  7  -7  3  11  -1  5  3  9
Sort_Array_Change:
-7  -1  3  3  5  7  9  11  13  20
ar3: 29  19  11  -11  4  17  -2  8  5  13
Sort_Array_Insert_L:
-11  -2  4  5  8  11  13  17  19  29
Enter a b n (n>0):
```

4.5.4. Визначення сусіднього розміщення елементів

До задач визначення сусіднього розміщення елементів в одновимірному масиві зачислимо задачі, в яких необхідно виокремити одну чи декілька *компактних* підмножин елементів, що задовольняють певній ознаці, та обчислити для кожної з підмножин деяку *агреговану* величину (кількість елементів, суму, середнє арифметичне значення тощо). Далі постає питання визначення найменшого / найбільшого значення цієї *агрегованої* величини тощо.

Тут під терміном "компактна підмножина елементів" розумітимемо підмножину, в якій елементи, що задовольняють ознаці, *розміщені поруч*.

Прикладами таких задач можуть слугувати задачі:

- визначення найбільшої кількості максимальних (чи мінімальних) елементів масиву, які розміщені поруч;
- визначення найбільшої кількості від'ємних (чи додатних) елементів масиву, які розміщені поруч;
- визначення найбільшої кількості однакових чисел, які розміщені поруч;

– визначення середнього арифметичного значення тих елементів масиву, які розміщені поруч та належать проміжку [*a*; *b*] тощо.

Для перевірки виконання ознаки необхідно переглянути весь масив (зазвичай, задіюватимемо цикл `for`). Під час встановлення істинності ознаки використовуватимемо *прийом скидання* накопиченого значення агрегованої величини. Пояснимо детальніше.

Під час проходження масиву можна отримати від однієї до декількох компактних підмножин, отож доречно ввести поняття *поточної компактної підмножини* (ПКП). Нехай `k` – агрегована величина ПКП.

Оскільки у *виродженому* випадку кожна компактна підмножина може містити тільки один елемент (тобто у всьому масиві немає сусідніх елементів, що задовольняють певній ознаці), то можна встановити початкове значення: `k = 1` (під час визначення кількості елементів) чи `k = 0` (під час визначення суми елементів, середнього арифметичного значення елементів тощо).

Надалі для зручності вважатимемо, що агрегована величина `k` відображає значення кількості елементів ПКП.

Першою ПКП буде підмножина, що містить елемент `a[0]`. Далі у циклі порівнюватимемо `a[0]` з `a[1]`. Нехай ознака справджуватиметься, тоді `k = 2` (відбудеться накопичення значення агрегованої величини). Нехай ознака справджуватиметься і під час наступного порівняння `a[1]` з `a[2]`, тоді `k = 3`.

Якщо ж під час чергового порівняння `a[2]` з `a[3]` ознака не справдиться, то це означатиме завершення цієї ПКП (міститиме елементи `a[0]`, `a[1]` і `a[2]`). Значення `k = 3` збережемо у деякій змінній `q` (або модифікуємо `q`). Наступна ПКП початково міститиме елемент `a[3]`, а `k = 1` (*скидання* накопиченого значення агрегованої величини).

Приклад 22. Дано натуральне число *n* і послідовність цілих чисел a_1, a_2, ..., a_n. Скласти фрагмент програми визначення найбільшої кількості однакових чисел, які розміщені поруч.

➢ *Попередні міркування*. Компактні підмножини у цій задачі – це послідовності *однакових* сусідніх елементів масиву. Нехай `k` – агрегована величина, в якій зберігатимемо значення кількості елементів ПКП, а `q` – шукана величина максимуму *однакових* сусідніх елементів (очевидно, що початкове значення `q = 1`).

Фрагмент програми:

```
k = 1; // Кількість елементів поточної підмножини
q = 1; // Шукана величина максимуму
for (i=1; i<n; i++)
  if (a[i]==a[i-1]) k++;  // Продовження підмножини
   else                   // Кінець підмножини
    { if (q < k) q = k;   // Уточнення максимуму
      k =1;               // Скидання значення k
    }
if (q < k) q = k; // Врахування останньої підмножини
```

Прийом скидання накопиченого значення агрегованої величини іноді можна застосовувати до задач, які розв'язують методом грубої сили. Покажемо це на такому прикладі.

Приклад 23. Дано натуральне число *n* і послідовність цілих чисел a_1, a_2, ..., a_n. Визначити, скільки разів трапляється у цій послідовності найменше число.

➢ *Попередні міркування*. За методом грубої сили треба організувати два цикли: у першому визначити найменше число, у другому – кількість його повторень. Однак, можна обійтися одним циклом.

Якщо умова `a[i]<min` справджується, то знайдено *нове* найменше число, і відлік кількості повторень `nmin` найменшого числа треба починати з 1 (*скидання* накопиченого значення). Якщо ж ця умова не справджується, то з двох можливих ситуацій (`a[i] == min` чи `a[i] > min`) нас цікавитиме `a[i]==min` – це означатиме, що знайдено елемент, який збігається з поточним найменшим числом (збільшуємо лічильник кількості повторень `nmin`).

```
min = a[0]; nmin = 1;
for(i=1; i<n; i++)
 if (a[i]<min) {min = a[i]; nmin = 1;}
  else if (a[i] == min ) nmin++;
```

4.5.5. Переміщення елементів масиву

Розглянемо підходи до розв'язування задач:

- долучення деякого числа до заданої позиції масиву;
- вилучення елемента масиву із заданої позиції;
- перерозміщення (циклічне зсування, інвертування) елементів масиву.

Під час розв'язування цих задач необхідно організовувати *переміщення* елементів усього масиву чи деякої підмножини елементів масиву (*підмасиву*).

Під час долучення деякого числа x у задану позицію k (цій позиції відповідає індекс масиву $k-1$) необхідно попередньо розсунути масив, тобто перемістити підмасив $a_k,..., a_n$ вправо на одну позицію. Переміщення підмасиву треба виконувати з кінця масиву.

Оскільки для моделювання послідовності чисел $a_1, a_2,..., a_n$ найчастіше використовують одновимірний динамічний масив, то при переміщенні підмасиву можливі дві ситуації:

- масив перед переміщенням заповнений не цілковито (після долучення нового числа кількість ініційованих (непорожніх) елементів масиву збільшиться на одиницю);
- масив перед переміщенням цілковито заповнений (після долучення нового числа попереднє останнє число вилучається з масиву).

Приклад долучення числа 1000 у п'яту позицію за умови часткового заповнення масиву ("?" позначає порожню позицію):

6	9	5	1	12	18	24	?	?
6	9	5	1	1000	12	18	24	?

Приклад долучення числа 1000 у п'яту позицію за умови цілковитого заповнення масиву:

6	9	5	1	12	18	24	36	48
6	9	5	1	1000	12	18	24	36

Фрагмент програми долучення нового елемента x у позицію k:

```
for (i=n-2; i >= k-1; i--) a[i+1] = a[i]; a[k-1] = x;
```


Вилучаючи k-й елемент з масиву, необхідно зсунути підмасив $a_{k+1},..., a_n$ вліво на одну позицію. Відповідний фрагмент програми виглядає так:

```
for (i = k-1; i <= n-2; i++) a[i] = a[i+1];
```

Приклад вилучення числа 1000 з п'ятої позиції за умови часткового заповнення масиву:

6	9	5	1	1000	12	18	24	?
6	9	5	1	12	18	24	?	?

Приклад вилучення числа 1000 з п'ятої позиції за умови цілковитого заповнення масиву:

6	9	5	1	12	18	24	36	48
6	9	5	1	18	24	36	48	48

Під час *циклічного зсування* масиву *усі* елементи масиву зберігаються, однак змінюють свої позиції. Приклад циклічного зсування масиву *вправо* на одну позицію:

а) ситуація часткового заповнення масиву:

6	9	5	1	7	12	18	24	?
?	6	9	5	1	7	12	18	24

б) ситуація цілковитого заповнення масиву:

6	9	5	1	12	18	24	36	48
48	6	9	5	1	12	18	24	36

Під час циклічного зсування масиву вправо на одну позицію напрям перегляду масиву та інструкція присвоювання у циклі такі ж самі, як і при долученні елемента у масив. Змінюються умова продовження циклу та інструкції присвоювання поза циклом. *Фрагмент* програми циклічного зсування масиву *вправо* на одну позицію:

```
x = a[n-1];  // збереження останнього елемента
// Усі елементи масиву зсуваємо вправо на одну позицію
for (i = n-2; i >= 0; i--) a[i+1] = a[i];
a[0] = x;    // останній елемент займає першу позицію
```

Фрагмент програми циклічного зсування масиву *вправо* на *k* позицій:

```
for (j = 1; j <= k; j++)
{
 x = a[n-1];  // збереження останнього елемента
 // Усі елементи масиву зсуваємо вправо на одну позицію
 for (i = n-2; i >= 0; i--) a[i+1] = a[i];
 a[0] = x;    // останній елемент займає першу позицію
}
```

Приклад циклічного зсування масиву *вліво* на одну позицію:
а) ситуація часткового заповнення масиву:

6	9	5	1	7	12	18	24	?
9	5	1	7	12	18	24	?	6

б) ситуація цілковитого заповнення масиву:

6	9	5	1	12	18	24	36	48
9	5	1	12	18	24	36	48	6

Під час циклічного зсування масиву *вліво* на одну позицію напрям перегляду масиву та інструкція присвоювання у циклі такі ж самі, як і при вилученні елемента з масиву. Змінюються умова продовження циклу та інструкції присвоювання поза циклом.

Фрагмент програми циклічного зсування масиву *вліво* на одну позицію:

```
x = a[0];  // збереження першого числа послідовності
// Усі елементи масиву зсуваємо вліво на одну позицію
for (i =0; i <= n-2; i++) a[i] = a[i+1];
a[n-1] = x;    // перше число займає останню позицію
```

Фрагмент програми циклічного зсування масиву *вліво* на *k* позицій:

```
for (j = 1; j <= k; j++) {
  x = a[0];
  for (i = 0; i <= n-2; i++) a[i] = a[i+1];  a[n-1] = x;
}
```


Наведені вище фрагменти програм мають важливе значення, оскільки є складовою багатьох алгоритмів обробки одновимірних масивів. На підтвердження цього факту наведемо такий приклад.

Приклад 24. Дано натуральне число n і послідовність дійсних чисел $a_1, a_2,..., a_n$. Переставити числа так, щоб спочатку розмістилися усі невід'ємні, а потім – усі від'ємні. Порядок розміщення як серед невід'ємних чисел, так і серед від'ємних має зберегтися. Скласти фрагмент програми розв'язування цієї задачі.

➢ *Попередні міркування.* Організуємо цикл послідовного перегляду усіх елементів одновимірного масиву. Введемо змінну `i` – індекс елемента перевірки (спочатку `i = 0`). Якщо `a[i] < 0`, то перемістимо `a[i]` в останню позицію масиву, попередньо зсунувши підмасив від `a[i+1]` до `a[n-1]` на одну позицію вліво; у протилежному випадку `i++`.

Перший від'ємний член спочатку займе останнє місце в масиві, а далі під час виявлення нового від'ємного елемента він зсунеться на одну позицію вліво, а на його місці (останньому місці у масиві) розміститься новий від'ємний елемент.

Під час виявлення наступних від'ємних елементів вони почергово розміщуватимуться на останньому місці у масиві, посуваючи вліво попередні від'ємні елементи. За такої організації переміщень початковий порядок розміщення від'ємних елементів збережеться. Початковий порядок невід'ємних чисел також збережеться, оскільки вони тільки зсуваються вліво, а взаємно між собою місцями не міняються.

Фрагмент програми:

```
i = 0; // початковий індекс елемента перевірки
for (k = 0; k < n; k++)
 if (a[i] < 0)
  { x = a[i]; // збереження від'ємного елемента
    // Зсування масиву вліво на 1 позицію
    for(j = i; j <= n-2; j++) a[j] = a[j+1];
    a[n-1] = x; // від'ємний елемент у кінець масиву
  }
 else i++; // наступний індекс елемента перевірки
```

◄

4.5.6. Злиття двох упорядкованих масивів

Під час реалізації деяких алгоритмів обробки масивів (зокрема, у деяких алгоритмах швидкого сортування) виникає задача утворити з двох масивів, *упорядкованих* за неспаданням значень, новий масив, який буде *упорядкованим* за неспаданням значень.

Приклад 25. Дано натуральні числа n, m і два масиви дійсних чисел $a_1, a_2,..., a_n$ і $b_1, b_2,..., b_m$, впорядкованих за неспаданням значень. Скласти фрагмент програми формування з елементів цих масивів нового масиву c, впорядкованого за неспаданням значень.

➢ *Попередні міркування*. Методом грубої сили цю задачу можна розв'язати так: спочатку переписати елементи масивів a і b у масив c, а потім масив c посортувати. Однак такий підхід – нераціональний, оскільки не враховує впорядкованості масивів a і b.

Для розв'язування задачі використаємо три цикли. У першому циклі, порівнюючи значення чергових елементів a і b, записуємо менше з них (правильніше, не більше) у масив c. Індекс масиву, елемент якого записали у масив c, збільшуємо на одиницю.

Збільшення кожного індексу реалізується на *окремій* гілці умовної інструкції. Якщо збільшується індекс i, то інший індекс j при цьому – не збільшується, і навпаки. Це означає, що один з елементів a чи b братиме участь у порівнянні на наступній ітерації.

Отже, перший цикл працюватиме доти, доки не вичерпається один із масивів a чи b. Наступні цикли забезпечують запис решти елементів одного з масивів у масив c (у конкретному випадку працюватиме тільки один з цих двох циклів).

Фрагмент програми:

```
// Уведення n елементів масиву a
// Уведення m елементів масиву b
 i = 0; j = 0;
 while ((i<n) && (j<m))
   if (a[i]<=b[j]) { c[i+j] = a[i]; i++; }
    else { c[i+j] = b[j]; j++; }
 while (i<n) { c[m-1+i] = a[i]; i++; }
 while (j<m) { c[n-1+j] = a[j]; j++; }
```

⮘

4.5.7. Алгоритми пошуку

Алгоритми пошуку застосовують для визначення у масиві *індексу* елемента, що має задане значення x. Розрізняють постановки задачі пошуку для першого і останнього входження x. Вважатимемо, що у всіх наведених алгоритмах, здійснюють пошук у масиві a розмірності n *індексу* першого від початку масиву елемента (перше входження), який дорівнює x.

Розрізняють пошук у впорядкованому та невпорядкованому масивах. У невпорядкованому масиві пошук виконують *лінійно* – послідовно переглядають увесь масив від початку до кінця.

Для організації лінійного пошуку використовують цикл `while` (чи `do … while`) з подвійною умовою. Перша умова контролює індекс на приналежність масиву (`i<n`), а друга – це умова продовження пошуку (`a[i]<>x`).

У тілі циклу є *тільки* один оператор, який змінює індекс масиву. Після виходу з циклу необхідно перевірити, за якою з умов відбувся вихід з циклу. В операторі `if`, зазвичай, повторюють першу умову циклу. Подаємо фрагмент програми лінійного пошуку:

```
 i = 0;
 while ((i<n) && (a[i] != x)) i++;
 if (i<n)… // Виведення/запам'ятовування індексу i
           // першого входження x
  else …   // Інформація про відсутність входження x
```

Розглянемо пошук у *впорядкованому* масиві. У цьому випадку його можна значно пришвидшити, застосовуючи метод половинного ділення масиву (*двійковий* або *бінарний пошук*).

Нехай маємо масив a з n елементів, який впорядковано за *неспаданням* ($a_i \le a_{i+1}$). Ідея алгоритму полягає у тому, що масив кожен раз ділять навпіл і обирають ту частину, де може розміщуватися потрібний елемент. Ділення продовжують доти, доки частина масиву для пошуку не складатиметься з одного елемента. Після цього залишається тільки перевірити елемент, що залишився, на виконання умови пошуку.

Фрагмент програми методу бінарного пошуку в масиві:

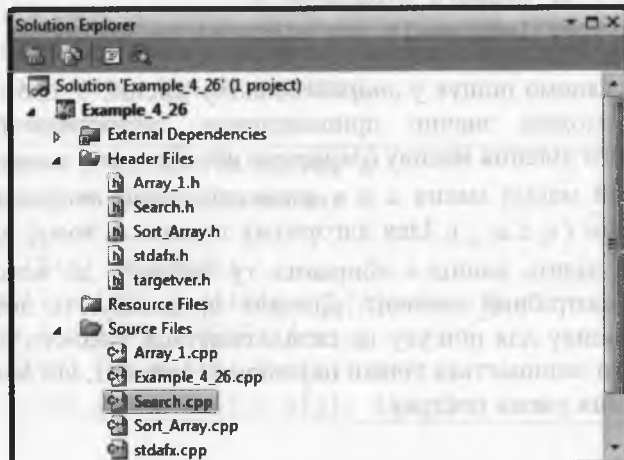
```
l = 0; r = n-1; // ліва і права межі пошуку
while (l < r)
 {
  m = (l + r) / 2; // середина проміжку
  if (x > a[m]) l = m+1; else r = m;
 } // while
if (a[l]==x))… // Запам'ятовування індексу l
                  // першого входження x
 else …  // Інформація про відсутність входження x
```

В обох фрагментах (як для лінійного, так і бінарного пошуку) встановлюють позицію l першого входження *x* у масив *a*. Наступні входження *x* визначають за допомогою лінійного пошуку в *a*, починаючи з позиції l+1.

У процесі роботи алгоритму двійкового пошуку розмір фрагмента, де цей пошук продовжуватиметься, кожного разу зменшуватиметься приблизно вдвічі. Це забезпечує обчислювальну складність алгоритму порядку $\log_2 n$, де n – кількість елементів масиву.

Приклад 26. Скласти і протестувати модуль, який міститиме функції лінійного та бінарного пошуку в одновимірному масиві.

➢ *Багатомодульний проект* прикладу 26 (Example_4_26):



Сервісний модуль Array_1 для одновимірних масивів (див. 4.4.4) містить файл специфікацій Array_1.h (див. ст. 300) та файл реалізації Array_1.cpp (див. ст. 300).

Модуль Sort_Array сортування одновимірних масивів (див. 4.5.3) містить файл специфікацій Sort_Array.h (див. ст. 312) та файл реалізації Sort_Array.cpp (див. ст. 313).

Файл специфікацій модуля Search (містить функції реалізації лінійного та бінарного (двійкового) пошуку заданого числа в одновимірному масиві):

```
// Файл Search.h: пошук елемента у масиві
#ifndef _Search_h
#define _Search_h
//-------------------------------------------------
// Функція визначає в одновимірному масиві ar розміру n
// індекс (позначення l) першого від початку масиву
// елемента, який дорівнює x.
// Постумова: функція повертає 0<=l<n, якщо такий
// елемент знайдено, чи l==-1 - інакше.
//-------------------------------------------------
int Search_Linear(int *ar,  int n,  int x);
// Лінійний пошук
//-------------------------------------------------
int Search_Binary(int *ar,  int n,  int x);
// Бінарний (двійковий) пошук
// Передумова: масив ar - впорядкований за неспаданням.
#endif // _Search_h
```

Файл реалізації модуля Search:

```
// Файл Search.cpp: пошук елемента у масиві
#include "Search.h"
int Search_Linear(int *ar, int n, int x){int i=0, l=-1;
 while((i<n)&&(ar[i]!=x)) i++; if(i<n) l=i; return l;}
int Search_Binary(int *ar,  int n,  int x)
{ int m, l=0, r=n-1; // l-ліва, r - права межі пошуку
  while (l < r) { m = (l + r) / 2; // середина проміжку
    if (x > ar[m]) l = m+1; else r = m; } // while
  if (ar[l]!=x) l=-1; return l;
}
```

Програма тестування (`Example_4_26.cpp`):

```
#include "stdafx.h"
int _tmain(int argc, _TCHAR* argv[])
{int a, b, n, x, l;
 while(cout<<"Enter a b n (n>0):", cin>>a>>b>>n, n>0)
  {int *ar=new int[n]; Fill_Array_1_Int(ar, n, a, b);
   cout<<"ar1: "; Print_Array_1(ar, n);
   cout<<"x= "; cin>>x; l=Search_Linear(ar,n, x);
   if(l==-1) cout<<"Not found " <<endl;  else
   {cout<<"l= "<<l<<endl;cout<<"The following indices l:";
    for(int i=l+1; i<n; i++) if(ar[i]==x)cout<<i<<" ";}
   cout<<endl;
   Fill_Array_1_Int(ar, n, 2*a, 2*b); cout<<"ar2: ";
   Sort_Array_Select(ar, n); Print_Array_1(ar, n);
   cout<<"x= "; cin>>x; l=Search_Binary(ar,n, x);
   if(l==-1) cout<<"Not found " <<endl;  else
   {cout<<"l= "<<l<<endl;cout<<"The following indices l:";
    for(int i=l+1; i<n; i++) if(ar[i]==x)cout<<i<<" ";}
   cout<<endl; delete [] ar;
  }  cin>>a; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Example_4_26\Debug\Example_4_26.e...
Enter a b n (n>0):-10 10 15
ar1: 2  -1  5  -7  -2  5  -2  2  -6  -7  2  10  5  -4  5
x= -2
l= 4
The following indices l: 6
ar2: -18  -11  -10  -10  -7  -7  -1  1  1  1  2  4  7  8  18
x= -10
l= 2
The following indices l: 3
Enter a b n (n>0):-20 20 15
ar1: 4  15  14  12  -4  18  7  0  11  2  -1  -17  -2  17  -1
x= 22
Not found

ar2: -36  -34  -22  -13  -7  -6  -1  0  3  4  9  14  20  27  28
x= 30
Not found

Enter a b n (n>0):
```


4.6. Рядки символів у стилі С

4.6.1. Загальні положення

Будь-яку зв'язану послідовність символів називають *рядком символів* (чи просто *рядком*). У мові C++ розрізняють два підходи до відображення рядків символів. Згідно з першим – рядком вважають одновимірний масив елементів типу `char` з кінцевим нульовим символом `NULL` (`'\0'`). Цей підхід успадкований від мови С, отож відповідні рядки символів називають рядками символів у стилі С (чи просто рядками С).

У C++ також підтримують широкосимвольні рядки символів з кінцевим нульовим символом, які відображаються одновимірними масивами елементів типу `wchar_t`.

Згідно з іншим підходом, рядок – об'єкт типу `basic_string`, який є шаблонним класом, визначеним у C++. Існують дві спеціалізації `basic_string`: `string` і `wstring`. Клас `string` оперує символами типу `char`, а `wstring` – символами типу `wchar_t`. Рядки типу `basic_string` розглядатимемо після знайомства з *класами*.

Визначення рядка символів у стилі С виглядає так:

```
char назва_рядка [довжина_рядка];
```

Наприклад:

```
char r[10]; const int len=80; char str[len];
```

Оскільки рядки у стилі С є масивами символів, то назва рядка є *вказівником* на його перший елемент. Символ `NULL` належить до символів рядка, отож розмір масиву, який відображає рядок, має на одиницю перевищувати кількість реальних символів рядка.

Під час визначення рядка йому можна присвоїти початкове значення (*ініціалізувати* рядок). Розрізняють два способи ініціалізації рядків: 1) за списком ініціалізації; 2) за рядковими літералами.

Згідно з першим способом елементам масиву символів при своюють початкові значення, які перелічують у *списку ініціалізації* (список обмежують фігурними дужками). Наприклад:

```
char s1[5]={'a','b','c','d','e'}; // масив з 5-ти символів
char s2[5]={'a','b','c','d','\0'}; // рядок з 5-ти символів
```

Як уже зазначалося раніше, якщо початкових значень у списку ініціалізації є менше, ніж елементів масиву, то елементи, які залишаються не ініціалізованими, автоматично одержують нульові початкові значення (у масивах символів нульове значення є ознакою кінця рядка). Отож наведене вище визначення змінної S2 з її ініціалізацією може бути й таким:

```
char s2[5] = {'a','b','c','d'};
```

Згідно з другим способом, під час ініціалізації рядка символів йому присвоюють *рядковий літерал* (послідовність символів, обмежену лапками). Оскільки компілятор С++ автоматично додає символ NULL до рядкового літерала, то така ініціалізація значно простіша та зрозуміліша.

Наприклад, визначимо рядок S розміром 10 символів, який ініціалізується рядком-літералом (тобто рядковим літералом):

```
char S[10] = "Це рядок";
```

За визначенням компілятор у пам'яті сформує масив S з 10-ти елементів типу char: 'Ц', 'е', ' ', 'р', 'я', 'д', 'о', 'к', '\0', '\0'.

Доступ до окремих символів рядка здійснюють за допомогою індексів, які відраховують від нуля. Тобто: S[0]– перший символ ('Ц'), S[1]– другий символ ('е'), ..., S[9] – останній десятий символ ('\0'). Доступ до конкретного символу використовують для його заміни іншим символом. Наприклад, оператор S[8] = '!' замінює передостанній символ '\0' символом '!'.

У визначенні рядка символів з ініціалізацією рядком-літералом довжину рядка можна не вказувати (довжина рядка дорівнюватиме кількості значущих символів рядка-літерала плюс одиниця), наприклад:

```
char text[]="Ми вивчаємо С++"; // 16 символів

char s[]="Univer"; // 7 символів
```

Під час такої ініціалізації символьних масивів жодного додаткового байта для нульового символу *не виділяється*, наприклад:

```
char s3[]={'a','b','c'}; // Масив з 3-х символів
```

Якщо ж ініціалізувати треба все-таки рядок, то записують так:

```
char s3[]="abc";
```


Отже, під час ініціалізації рядка або під час введення рядка з клавіатури компілятор автоматично записує один чи декілька нульових байтів наприкінці рядка (кількість реальних символів і кількість нульових байтів сумарно дорівнюють розміру рядка). Якщо ж рядок формують програмно, то нульові байти необхідно задавати явно.

Задати значення рядку символів My_String з клавіатури консолі можна за допомогою інструкції:

```
cin>> My_String;
```

Якщо значення рядка My_String містить символ пропуску, то ця інструкція зчитає значення рядка тільки до першого пропуску. Щоб зчитати значення усього рядка до символу завершення рядка (натиснення клавіші Enter), необхідно застосувати інструкцію:

```
cin.get(My_String, m);
```

де m – максимальна кількість символів під час введення. Якщо використати цю інструкцію двічі підряд, то другий рядок не зчитається. Для уникнення цієї помилки необхідно використати наступну серію інструкцій:

```
cin.get(My_String, m1);
cin.get();
cin.get(My_String_2, m2);
```

Замість методу cin.get(My_String, m) можна використовувати метод cin.getline(My_String, m). Виведення значення рядка My_String на екран консолі реалізують інструкцією:

```
cout<<My_String;
```

Приклад 27. Скласти програму, яка демонструватиме способи формування значень рядків:

– за допомогою їхньої ініціалізації під час визначення;
– шляхом введення символів з клавіатури консолі;
– програмним способом.

➤ *Попередні міркування*. Для розуміння програми достатньо прочитати коментарі. Масив символів s4 в cout сприймається як рядок (виводитимуться випадкові символи доти, доки не буде '\0').

Програма демонстрації:

```
#include <iostream>
using namespace std;
void main(void)
{// Ініціалізація значень рядків
 char s1[] = "Hello, world!";
 char s2[5] = {'a','b','c','d','\0'};
 char s3[5] = {'w','z','t'};
 // Виведення рядків
 cout << "s1: " << s1<<endl;
 cout << "s2: " << s2<<endl;
 cout << "s3: " << s3<<endl;
 char s4[]={'x','y'}; // Ініціалізація масиву символів
 // Виведення  масиву символів, як рядка
 cout << "s4: " << s4<<endl;
 // Програмне формування значення рядка
 char abetka [27]; // 26 букв плюс NULL
 int i=0;
 for (char c = 'a'; c <= 'z'; c++, i++) abetka[i] = c;
 abetka[i] = NULL;
 // Виведення рядка, сформованого у програмі
 cout << "Abetka: " << abetka <<" OK!"<<endl;
 // Введення значень рядка з консолі
 char s5[25]; cout<<"Enter s5: "<<endl; cin.get(s5,80);
 // Виведення рядка, отриманого з консолі
 cout << "s5: " << s5<<endl;  cin>>i; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Приклади_На_Ряд...
s1: Hello, world!
s2: abcd
s3: wzt
s4: xy|||||||||||||||wzt
Abetka: abcdefghijklmnopqrstuvwxyz  OK!
Enter s5:
It is a string!
s5: It is a string!
```


Відзначимо, що інструкція

```
char *s_Ptr = "Hello, my friends!";
```

визначає змінну s_Ptr, яка є вказівником на рядок-літерал (константу), значення якої змінювати не можна!

Для того, щоб вказівник на char міг оперувати з рядками, значення яких можуть змінюватися під час виконання програми, йому необхідно присвоїти назву конкретної змінної-рядка, або виділити для рядка осередок динамічної пам'яті (прикл. 28).

Оскільки змінні-рядки зображають *масиви* символів, то рядки не можна *порівнювати* між собою чи *копіювати* оператором присвоювання. У цих випадках необхідно організовувати цикли для посимвольної обробки рядків (прикл. 28).

Приклад 28. Скласти програму копіювання рядка символів.

Перший варіант *програми*:

```
#include <iostream>
using namespace std;
void main(void) {char s1[15], s2[] = "Hello, world!";
 int i=0; while(s2[i]){s1[i]=s2[i]; i++;} s1[i]=NULL;
 cout << "s2: " << s2<<endl; cin>>i; // Пауза
}
```

Другий варіант *програми*:

```
#include <iostream>
using namespace std;
void main(void) { char s2[] = "Hello, world!";
 char *s1 = new char[15], *src=s2; while(*s1++=*src++);
 cout << "s2: " << s2<<endl; cin>>s2; // Пауза
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Пр...
s2: Hello, world!
```

Для організації порівнянь між рядками та копіювання рядків, зазвичай, використовують *стандартні* функції.

4.6.2. Стандартні функції обробки рядків

Прототипи значної кількості стандартних функцій обробки рядків у стилі C оголошені у системному файлі специфікацій `cstring`. Розглянемо прототипи найуживаніших функцій обробки рядків C (при цьому домовимося замість довгої фрази "*рядок, на який вказує вказівник* `s`" просто вживати термін "*рядок* `s`" тощо):

1) `char *strcat(char *d, const char *s)` – приєднує рядок `s` до кінця рядка `d`. Повертає `d`. Якщо рядки перекриваються, то результат не визначений.
2) `int strcmp(const char *s1, const char *s2)` – здійснює *лексикографічне* порівняння рядків, враховуючи регістр літер. Якщо рядок `s1` менший за рядок `s2`, то функція повертає від'ємне ціле. Якщо обидва рядки рівні, то функція повертає `0`. Якщо рядок `s1` більший за рядок `s2`, то функція повертає додатне ціле.
3) `char *strcpy(char *d, const char *s)` – копіює рядок `s` у рядок `d`. Повертає `d`. Якщо рядки перекриваються, то результат не визначений.
4) `size_t strcspn(const char *s1, const char *s2)` – повертає індекс першого символу в рядку `s1`, який співпадає з будь-яким символом рядка `s2`. Якщо співпадіння не виявлено, то повертається довжина рядка `s1`.
5) `int stricmp(const char *s1, const char *s2)` – здійснює *лексикографічне* порівняння рядків аналогічно до функції `strcmp`, однак регістр літер ігнорується.
6) `size_t strlen(const char *s)` – повертає кількість значущих символів рядка `s` (`NULL` не враховується).
7) `char *strncat(char *d, const char *s, size_t n)` – приєднує не більше `n` символів рядка `s` до кінця рядка `d`. Повертає `d`. Якщо рядки перекриваються, то результат не визначений.
8) `int strncmp(const char *s1, const char *s2, size_t n)` – здійснює *лексикографічне* порівняння не більше `n` перших символів рядків, враховуючи регістр літер. Якщо рядок `s1` менший за рядок `s2`, то функція повертає від'ємне ціле. Якщо обидва рядки рівні, то функція повертає `0`. Якщо рядок `s1` більший за рядок `s2`, то функція повертає додатне ціле.

9) `char *strncpy(char *d, const char *s, size_t n)` – копіює `n` перших символів рядка `s` у рядок `d`. Повертає `d`. Якщо рядки перекриваються, то результат не визначений. Якщо `s` містить більше, ніж `n` символів, то рядок `d` не буде обмежений символом `NULL`.
10) `char *strpbrk(const char *s1, const char *s2)` – повертає вказівник на перший символ у рядку `s1`, який співпадає з будь-яким символом рядка `s2`. Якщо співпадіння не виявлено, то повертається нульовий вказівник.
11) `char *strchr(const char *s1, int ch)` – повертає вказівник на останнє входження молодшого байта `ch` у рядку `s1`. Якщо входження не виявлено, то повертається нульовий вказівник.
12) `size_t strspn(const char *s1, const char *s2)` – повертає індекс першого символу в рядку `s1`, який не співпадає з жодним символом рядка `s2`. Якщо неспівпадіння не виявлено, то повертається довжина рядка `s1`.
13) `char *strstr(const char *s1, const char *s2)` – повертає вказівник на перше входження рядка `s2` у рядок `s1`. Якщо немає співпадіння, то повертається нульовий вказівник.
14) `char *strtok(char *s, const char *d)` – повертає вказівник на наступну лексему рядка `s`. Символи у рядку `d` специфікують роздільники, які визначають межі лексем. Якщо не залишається лексем для повернення, то повертається нульовий вказівник. Щоб розбити рядок на лексеми, перший виклик `strtok` має отримати в `s` вказівник на цей рядок. Наступні виклики мають передавати `s` нульовий вказівник.

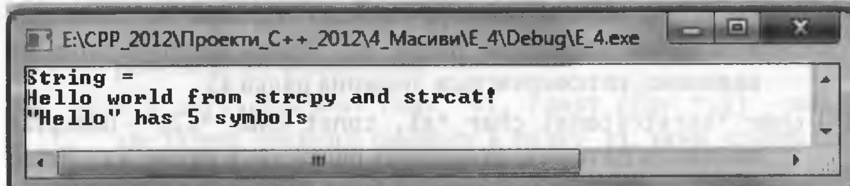
Деякі функції використовують тип `size_t`. Це деяка форма *беззнакового цілого*, визначена у `<cstring>`. У `<cstring>` також визначено декілька функцій з префіксом "mem". Це функції: `memchr()`, `memcmp()`, `memcpy()`, `memmove()` і `memset()`. Перелічені функції оперують із символами, однак не використовують домовленість про завершальний нульовий символ.

Приклад 29. Скласти програму, яка демонструє використання функцій strcpy, strcat і strlen.

Програма демонстрації функцій:

```
#include <iostream>
using namespace std;
int main() { char string[80];
 strcpy(string, "Hello world from " );
 strcat(string, "strcpy "); strcat(string, "and ");
 strcat(string,  "strcat!");
 cout<<"String = \n"<< string<<endl;
 strcpy( string, "Hello" );
 cout<<"\""<<string<<"\""<<" has ";
 cout<<strlen(string)<<" symbols "<<endl;
 int c; cin>>c;  return 0;
}
```

Результати тестування програми:



Приклад 30. Скласти програму, яка демонструє використання функцій пошуку strstr, strchr і strpbrk.

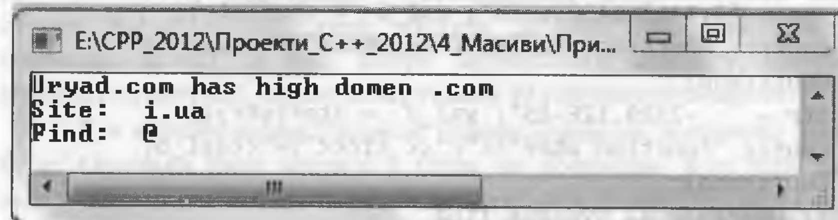
Програма демонстрації функцій:

```
using namespace std;
int main() {const char *s1="Uryad.com"; const char *ptr;
 const char *e_mail="igordudz@i.ua";
 const char *nabir[]={".com",".org",".ua",".ru"};
 // Приклад на пошук підрядків
 for(int i=0; i<4; i++)
 {ptr=strstr(s1,nabir[i]);
  if(ptr) cout<<s1<<" has high domen "<<nabir[i]<<endl; }
 // Приклад на пошук окремого символу
 ptr=strchr(e_mail,'@');
 if(ptr) cout<<"Site:  "<<ptr+1<<endl;
```



```
 // Приклад на пошук символу з набору
 ptr=strpbrk(e_mail,"@.");
 if(ptr) cout<<"Find:  "<<*ptr<<endl;
 int i; cin>>i; return 0;
}
```

Результати тестування програми:



Кожна цифру - 0, 1, 2, 3, 4, 5, 6, 7, 8 і 9 - можна зобразити у вигляді символу, отож будь-яке число, яке зберігається у комп'ютері, можна замінити набором цифр (символів), і навпаки. Для цього необхідно скористатися функціями, заголовки яких описані у файлі специфікацій cstdlib. Розглянемо найуживаніші функції перетворення рядків:

1) `int atoi(const char *s)` – перетворення алфавітно-цифрового (символьного) масиву в ціле число;
2) `float atof(const char *s)` – перетворення алфавітно-цифрового (символьного) масиву в раціональне число;
3) `itoa(int v, char *s, int r)` – перетворення цілого виразу v у символьний масив s у системі числення r.

Приклад 31. Скласти програму, яка демонструє використання функцій atoi, atof та itoa.

Програма демонстрації функцій:

```
#include <cstdlib>
#include <iostream>
using namespace std;
int main(void) { char *str; int val_i; double val_d;
 // Приклад на функцію atoi
 str = "  -2309 "; val_i = atoi(str);
 cout<< "Function atoi"<<"("<< str<<")="<<val_i;
```

```
cout<<endl;
// Приклади на функцію atof
str = "  5556402735183407160354 ";
val_d = atof(str);
cout<< "Function atof"<<"("<< str<<")="<<val_d;
cout<<endl;
str = "3.1412764583d210"; val_d  = atof(str);
cout<< "Function atof"<<"("<< str<<")="<<val_d;
cout<<endl;
str = "  -2309.12E-15"; val_d  = atof(str);
cout<< "Function atof"<<"("<< str<<")="<<val_d;
cout<<endl;
// Приклади на функцію itoa
char buffer[65];
for(int r=10; r>=2; --r ) {
 itoa( 16, buffer, r ); // C4996
 cout<< r<<" "<<buffer<<endl;
}
cin>>val_d; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Example_4_28...
Function atoi(   -2309 )=-2309
Function atof(  5556402735183407160354 )=5.5564e+021
Function atof(3.1412764583d210)=3.14128e+210
Function atof(   -2309.12E-15)=-2.30912e-012
10 16
9 17
8 20
7 22
6 24
5 31
4 100
3 121
2 10000
```


4.6.3. Кодування символів

Щоб мати змогу виводити у повідомленнях консольних застосувань кириличні літери, необхідно дещо краще орієнтуватися у питаннях кодування символів. Ядром переважної більшості сучасних систем кодування є код ASCII (American Standard Code for Information Interchange – стандартний американський код обміну інформацією), який містить *базовий* набір із 128-ти символів: великі та малі літери латинського алфавіту, цифри, символи пунктуації, символи керування та спеціальні символи.

Для відображення *базового* набору символів достатньо 7-ми бітів. Оскільки байт містить 8 бітів, то додавання 8-го біта дає змогу збільшити кількість кодів таблиці ASCII до 255. Коди від 128-ми до 255-ти – це *розширення* базової таблиці ASCII. Перша *розширена таблиця кодувань* ASCII отримала назву ISO 646 (назва міжнародного стандарту), а відповідний восьмибітний код – Latin-1 (додані латинські літери зі штрихами та діакритичні символи, деякі літери грецького алфавіту та символи *псевдографіки*).

Незабаром з'явився новий стандарт ISO 8859, в якому вводилося поняття *кодової сторінки* (*Code page*), тобто набору з 256-ти символів для визначеної мови чи групи мов. Наприклад, кодова сторінка ISO 8859-1 (синонім – кодова сторінка CP437) це, фактично, код Latin-1, ISO 8859-2 – слов'янські мови з латинським алфавітом, ISO 8859-5 – слов'янські мови з кириличним алфавітом (не набула широкого розповсюдження, її переважно використовують у системах керування базами даних).

Однобайтові таблиці кодувань з підтримкою кириличних літер будують за однаковим принципом: перші 128 символів (коди від 0 до 127) відповідають базовому набору символів ASCII, а наступні 128 символів (коди від 128 до 255) заповнюють певним способом. Якщо розглядати тільки ті таблиці кодувань (чи кодові сторінки), які "*дожили*" до цього часу, то матимемо таке:

- Таблиця кодувань **CP866** (кодування MS DOS, таблиця наведено на ст. 146), яка збереглася завдяки віконцю `cmd.exe` в ОС Windows. У цій таблиці між літерами "*п*" і "*р*" залишена "дірка" з 48-ми кодів, які в CP437 відповідають символам псевдографіки. Це дає змогу запускати програми, написані для

CP437, на комп'ютерах з CP866. Зазначимо відсутність кириличної літери "ї" (її необхідно вводити за допомогою латинської розкладки клавіатури).

- Таблиці кодувань **KOI** (код обміну інформацією) заповнені кириличними літерами так, що якщо відкинути старший біт коду, то отримаємо більш-менш відповідну латинську літеру. Це зумовлює до того, що кириличні літери у таблиці розміщені не в алфавітному порядку. З історичних причин модифікація KOI8-R поширена в UNIX-подібних операційних системах.
- Таблицю і систему кодувань **CP1251** (або кодування **win-1251**, таблицю наведено на ст. 147) використовують в однойменній операційній системі. Символи псевдографіки відсутні, оскільки Windows формує графічне середовище. Російські літери у таблиці розміщені за алфавітом і без "дірок". Літери української абетки, які відсутні в російській абетці, розміщені довільно. За певних обставин можуть виникати проблеми з набором кириличної літери "ї". На жаль, літері "*я*" належить останній код 255, а це число у C++, будучи зведеним до типу `char`, даватиме `-1` (код константи `EOF`). Отже, якщо програміст під Windows неакуратно працює з типами даних, то введена літера "*я*" іноді може зумовити до завершення читання даних.
- Таблицю кодувань **MAC-CYR** використовують на комп'ютерах Macintosh / Apple в операційній системі MacOS, а також в інших операційних системах для забезпечення сумісності з MacOS.

Доволі часто вживають поняття *системи кодувань ANSI* (*American National Standards Institute*, Американського національного інституту стандартів), як системи, яку використовує Windows (точніше, це *псевдонім* поточного кодування у локалі Windows).

Кодові сторінки є важливим компонентом локалізації (їх задають у ключах реєстру `HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \Control \ Nls \ CodePage`). Для користувачів комп'ютерів в Англії та США абревіатура *ANSI* – це псевдонім win-1252 (Latin-1), для користувачів українізованих (чи русифікованих) комп'ютерів – win-1251 (cyrillic).

У Windows для позначення кодових сторінок, які містять символи *псевдографіки*, використовують абревіатуру *OEM* (буквально: *Original Equipment Manufacturer*, оригінальне постачання виробника; в реєстрі `OEMCP`). Фактично `OEMCP` – це кодова сторінка за домовленістю, яку використовують для відображення текстових файлів, створених в операційній системі MS DOS. Для користувачів українізованих (чи русифікованих) комп'ютерів `OEMCP` – це, зазвичай, кодова сторінка CP866.

Загалом, ANSI не тотожне CP1251, а OEM – CP866. Відповідності між ними (залежать від регіональних налаштувань) прописані у реєстрі (див. `HKEY_LOCAL_MACHINE \ SYSTEM\ CurrentControlSet \Control \ Nls \ CodePage`).

Використання різних кодових сторінок створює чимало незручностей як для користувачів, так і для програмістів. Під час спроби прочитати текстовий файл за допомогою кодової сторінки, несумісної з тією, в якій його створено, виникають незрозумілі символи (т. зв. "*кракозябри*"). Окрім цього, можна зазначити ще й такі недоліки: програмне забезпечення має стежити за кодовими сторінками; не можна змішувати різні мови; відсутні кодові сторінки багатьох поширених мов тощо.

Для подолання цих недоліків у 1991 р. було створено стандарт міжнародної системи кодування для обміну інформацією Unicode (у стандарті символи називають *code points – точками коду*). Перша версія Unicode 1.1 (стандарт ISO / IEC 10646-1 : 1993) була просто *таблицею* точок коду з фіксованим розміром 16 біт на точку, тобто загальна кількість точок коду становила 2^{16} = 65 536.

На початку в Unicode планували кодувати тільки символи, які найчастіше використовують. Символи, які зрідка вживають, передбачали розмістити в *області символів користувача* (коди `U + D800 ... U + F8FF`, де `U` – символ-ознака Unicode).

Надалі, однак, було ухвалено рішення кодувати усі можливі символи і у зв'язку з цим значно розширити кодову область. Водночас коди символів стали розглядати не як 16-бітові значення, а як *абстрактні числа*, які в комп'ютері можна відображати різними *способами*.

Оскільки у багатьох операційних системах (наприклад, у MS Windows NT) фіксовані 16-бітові значення уже використовували для кодування символів, було вирішено усі найуживаніші символи кодувати тільки у *межах* 65 536 позицій, які утворили *базовий багатомовний рівень* (англ. *Basic Multilingual Plane*, *BMP*).

Для сумісності зі старими 16-бітовими системами було винайдено систему кодування UTF-16, де усі позиції, за винятком позицій з інтервалу `U + D800 ... U + DFFF`, відображають символи безпосередньо як 16-бітові числа.

Позиції з інтервалу `U + D800 ... U + DFFF` (2 048 позицій) використовують для відображення символів у вигляді "*сурогатних пар*" (перший елемент пари з області `U + D800 ... U + DBFF`, а другий елемент – з області `U + DC00 ... U + DFFF`). Отже, в UTF-16 можна відобразити $2^{20} + 2^{16} - 2\,048 = 1\,112\,064$ символів.

Окрім 16-бітової системи кодування (UTF-16), стандарт Unicode підтримує ще 32-бітову (UTF-32) та 8-бітову (UTF-8) системи кодування. Найпростіше влаштована система UTF-32, в якій існує однозначна відповідність між точкою коду та 32-бітовим двійковим значенням.

Система кодування UTF-8 забезпечує найкращу сумісність зі старими системами, які використовували 8-бітові символи. Текст, що містить тільки символи з кодами, меншими за 128, під час запису в UTF-8 перетворюється на звичайний текст ASCII. У довільному тексті UTF-8 будь-який байт зі значенням, меншим за 128, відображає символ ASCII з цим же кодом. Решта символів Unicode відображаються послідовностями від 2-х до 4-х байтів.

Хоча системи кодування UTF-8 і UTF-32 дають змогу закодувати до 2 147 483 648 символів, було ухвалено рішення використовувати тільки 1 112 064 символів (для сумісності з UTF-16). Число 1 112 064 називають *верхньою межею* кодового простору Unicode.

Сьогодні стандарт Unicode 6.0 містить близько 110 000 символів, яких достатньо для спілкування усіма відомими мовами світу та для відображення історичних документів мертвими мовами. Окрім цього, стандарт містить наукові та технічні знаки, геометричні фігури та графічні позначки, фонетичні символи тощо.

4.6.4. Функції **Windows API** *відображення літер кирилиці*

Консольне застосування створюється у середовищі розробки Visual Studio операційної системи Windows, отож усі *літерали*, які відображають символи чи рядки символів, кодуються у системі ANSI (кодова сторінка win-1251). Однак виконавчий модуль консольного застосування виконується у вікні консолі, яке імітує операційну систему DOS, де використовують систему кодування OEM (кодова сторінка CP866).

Коди латинських літер є однаковими в обох системах кодування, а коди *кириличних* літер – різними. Повідомлення програми (*літерал*), записане кириличними літерами, яке без проблем відображається у текстовому редакторі коду C++, під час виведення на консоль матиме вигляд послідовності "*кракозябрів*". Отож повідомлення консольних застосувань часто записують латинськими літерами (як це зроблено в усіх наших попередніх програмах).

Для правильного введення / виведення у вікні консолі кириличних літер можна використовувати функції Windows API: `CharToOem()` для перетворення літер із системи кодування ANSI у систему кодування OEM і функцію `OemToChar()` для зворотного перетворення. Для використання цих функцій необхідно підключати файл специфікацій `windows.h`.

У програмах на C++ найчастіше використовують функцію `CharToOem()`, яка має такий прототип:

```
BOOL CharToOem (LPCTSTR lpszSrc, LPSTR lpszDst);
```

де `lpszSrc` – вказівник на рядок, що перетворюється, `lpszDst` – вказівник на буфер для перетвореного рядка.

Функція `CharToOem()` існує у двох варіантах: для однобайтових символів (ANSI функція) і для широких символів. Якщо функцію використовують як ANSI-функцію, то рядок можна перетворити одразу ж, установкою для параметрів `lpszDst` і `lpszSrc` однакового значення. Цього не можна робити, якщо `CharToOem()` використовують як функцію для обробки широких символів.

Аргументами функції `CharToOem()` можуть бути рядки символів у стилі C. Значення, яке повертає функція `CharToOem()`, завжди не нульове, за винятком випадку, коли передаються однакові ад-

реси для lpszSrc і lpszDst під час використання версії функції для широких символів (у цьому випадку функція повертає нуль). Використання функції CharToOem() проілюструємо на такому прикладі.

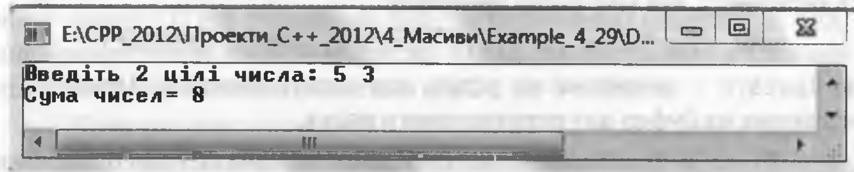
Приклад 32. Скласти програму, яка демонструє використання функції CharToOem().

Програма демонстрації функцій:

```
#include <windows.h>
#include <iostream>
using namespace std;
void main() {
  int a, b; char st[30];   // Рядок
  CharToOem("Введіть 2 цілі числа: ",st);
  cout<<st; cin>>a>>b; CharToOem("Сума чисел= ",st);
  cout<<st<<a+b<<endl; cin>>a;   // Пауза
}
```

Додатковий коментар. У програмі маємо два літерали – рядки символів: "Введіть 2 цілі числа: " та "Сума чисел= ", кириличні літери яких закодовані в ANSI. Для збереження значень цих рядків у системі кодування OEM зарезервовано *масив* символів (char st[30]). Цей масив заповниться відповідними значеннями рядків у кодах OEM після виконання функції CharToOem.

Результати тестування програми:



Для введення рядків з символами пропуску і/або табуляції можна також використовувати функцію gets(array), прототип якої описано у файлі специфікації cstdio. Функція gets зчитує символи, введені з клавіатури, у рядок-масив array доти, доки не зчитає символ нового рядка (натискання користувачем клавіші Enter).

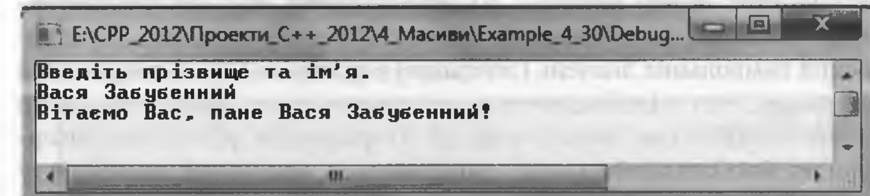
Приклад 33. Розробити програму виведення на екран повідомлення-запрошення на введення користувачем його прізвища та імені; отримати відповідь від користувача. Для коректного відображення кириличних літер використати функцію CharToOem(). Скористатися отриманими даними для формування привітання користувачеві.

Попередні міркування. Компілятор отримані з консолі літери зберігає у пам'яті комп'ютера у кодовій таблиці CP866; під час виведення на екран також інтерпретує ці літери як коди CP866. Отже, використовувати функції OemToChar() / CharToOem() для перетворення даних, отриманих з консолі, не потрібно.

Програма:

```
#include <windows.h>
#include <iostream>
#include <cstdio>
using namespace std;
void main() {char c[20], st[20]; // Визначення масивів
  CharToOem("Введіть прізвище та ім'я. ",st);
  cout<< st<<endl;  gets(c);
  CharToOem("Вітаємо Вас, пане ",st);
  cout<< st<<c<<"!"<<endl;
  cin>>c;   // Організація паузи
}
```

Результати *тестування* програми:



Окрім функцій перетворення кириличних літер CharToOem() і OemToChar() файл специфікацій windows.h містить прототипи функцій Windows API *налаштування консолі* (тобто консольного вікна DOS): GetConsoleCP, GetConsoleOutputCP, SetConsoleCP і SetConsoleOutputCP.

Функція

```
UINT GetConsoleCP(void);
```

повертає номер поточної кодової сторінки, яку використовує консоль, асоційована із запущеним процесом (програмою), для перетворення кодів клавіш клавіатури у відповідні символи. Тип функції `UINT` визначено у файлі `window.h` інструкцією:

```
typedef unsigned int UINT;
```

Функція

```
BOOL SetConsoleCP(uiNT wCodePageiD);
```

встановлює кодову сторінку, яку використовуватиме консоль, асоційована із запущеним процесом, для перетворення кодів клавіш клавіатури у відповідні символи. Параметр `wCodePageiD` задає номер кодової сторінки. За успішного завершення `SetConsoleCP` повертається ненульове значення; у протилежному випадку повертається нуль.

Функція

```
UINT GetConsoleOutputCP(void);
```

повертає номер поточної кодової сторінки, яку використовує консоль, асоційована із запущеним процесом, для перетворення символьних значень (літералів) у програмі С++ у зображення символів, що відображаються у консольному вікні.

Функція

```
BOOL SetConsoleOutputCP (UINT WCodePageiD);
```

встановлює номер поточної кодової сторінки, яку використовуватиме консоль, асоційована із запущеним процесом, для перетворення символьних значень (літералів) у програмі С++ у зображення символів, що відображаються у консольному вікні. Параметр `wCodePageiD` задає номер кодової сторінки. За успішного завершення `SetConsoleOutputCP` повертається ненульове значення; у протилежному випадку повертається нуль.

Під час використання функцій налаштування консолі кириличні літери відображатимуться правильно, якщо для консольного вікна вибрано шрифт Lucida Console або Consolas. Для вибору шрифту необхідно натиснути праву кнопку миші на заголовку

вікна, виконати команду `Властивості`, а потім на вкладці `Шрифт` обрати потрібний шрифт.

Приклад 34. Розробити програму виведення запрошення на введення користувачем прізвища та імені; отримати відповідь від користувача та сформувати привітання. Для коректного відображення кириличних літер використати функції налаштування консолі.

Програма:

```
#include <iostream>
#include "windows.h"
using namespace std;
int main() {
 // Поточна кодова сторінка для введення з консолі
 cout<<"Current Console_CP=\t" << GetConsoleCP()<<endl;
 // Нова кодова сторінка 1251 для введення з консолі
 SetConsoleCP(1251);
 cout<< "New Console_CP=\t" << GetConsoleCP()<<endl;
 // Поточна кодова сторінка для виведення на консоль
 cout<<"Current Console_Output_CP=\t"
             <<GetConsoleOutputCP()<<endl;
 // Нова кодова сторінка 1251 для виведення на консоль
 SetConsoleOutputCP(1251);
 cout<< "New Console_Output_CP=\t" <<
               GetConsoleOutputCP()<<endl;
 char Name[20];  cout<<"Введіть прізвище та і'мя: ";
 cin.get(Name,80);
 cout<<"Вітаємо Вас, "<<Name<<"!"<<endl; cin>>Name;
}
```

Результати *тестування* програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Приклади...
Current Console_CP=      866
New Console_CP= 1251
Current Console_Output_CP=      866
New Console_Output_CP=  1251
Введіть прізвище та і'мя: Пупкін Іван
Вітаємо Вас, Пупкін Іван!
```

4.6.5. Засоби локалізації у стилі C

Для врахування особливостей, пов'язаних з країною та мовою, використовують спеціальні середовища – *локальні контексти* чи *локалі* (locale – місце дії), які налічують набір параметрів і функцій, що забезпечують підтримку національних та культурних стандартів.

Локальний контекст визначається рядком формату:

`мова[_зона[.код]]` або `[мова_зона].код`

де `мова` – позначка мови, а `зона` – країна, в якій використовують мову. Кваліфікатор `зона` дає змогу підтримувати національні стандарти для різних країн, що використовують одну мову. Кваліфікатор `код` визначає кодову сторінку.

У таблиці наведено приклади рядків, які визначають локальні контексти (ці рядки не стандартизовані, отож підтримка тих чи інших локальних контекстів залежить від реалізації мови):

Рядок	*Локальний контекст*
`C`	Використовують за домовленістю (стандарт ANSI-C, англійська мова)
`en_US`	Англійська мова (США)
`de_DE`	Німецька мова (Німеччина)
`de_AT`	Німецька мова (Австрія)
`Ukrainian_Ukraine` `ukr`	Українська мова (Україна), кодова сторінка 1251 (за домовленістю)
`Ukrainian_Ukraine.866`	Українська мова (Україна), кодова сторінка 866
`Russian_Russia`	Російська мова (Росія), кодова сторінка 1251 (за домовленістю)
`Russian_Russia.866`	Російська мова (Росія), кодова сторінка 866

Засоби мови C++ для роботи з локальними контекстами у стилі С оголошені у стандартному заголовку `locale`. Налаштування програми на конкретний локальний контекст виконує функція:

```
char *setlocale(int category, const char *locale);
```

Параметр `category` визначає категорію функцій, на які впливає `setlocale`:

- `LC_ALL` – усі категорії;
- `LC_COLLATE` – порівняння та перетворення рядків;
- `LC.CTYPE` – обробка символів;
- `LC.MONETARY` – форматування сум грошей;
- `LC_NUMERIC` – форматування чисел;
- `LC_TIME` – форматування часу.

Параметр `locale` є вказівником на рядок, який визначає локальний контекст. Якщо `locale` вказує на порожній рядок, то використовують локальний контекст з кодовою сторінкою, одержаною з операційної системи. Якщо `locale` дорівнює нулю (`NULL`), то поточний локальний контекст не змінюється.

За успішного завершення роботи функція `setlocale` повертає вказівник на рядок з описом локального контексту або нульовий вказівник – при невдачі.

Приклад 35. Розробити програму, яка виводитиме на екран консолі два українські слова: 1) задане безпосередньо в програмі (рядок-літерал); 2) введене з клавіатури консолі. Необхідно використати *локальні контексти*: поточний контекст за домовленістю, за налаштуваннями операційної системи, за налаштуваннями DOS та за різними способами явного задавання контекстів.

Програма:

```
#include <iostream>
#include <locale>
using namespace std;
```

```
int main()
{
 char *program = "Програма", console[15];
 // Локальний контекст за домовленістю
 cout << "Default locale is\t"<< setlocale(LC_ALL,
                                          NULL)<<endl;
 cout << "Enter the Ukrainian word: ";
 cin >> console;
 cout << program << "\t" << console << endl;
 // Локальний контекст з налаштуваннями ОС
 cout<<"OS locale is\t"<<setlocale(LC_ALL, "") << endl;
 cout << program << "\t" << console << endl;
 // Локальний контекст з налаштуваннями DOS
 cout<<"DOS locale is\t"<<setlocale(LC_CTYPE, ".866");
 cout << endl;
 cout <<program << "\t" << console << endl;
 // Локальний контекст з кодовою сторінкою 1251
 cout << "Locale: "<< setlocale(LC_CTYPE, ".1251");
 cout << endl;
 cout <<program << "\t" << console << endl;
 // Локальний контекст з кодовою сторінкою .866
 cout<< "Locale: "<< setlocale(LC_CTYPE, ".866");
 cout << endl;
 cout <<program << "\t" << console << endl;
 // Локальний контекст з українською мовою
 cout<< "Locale: "<< setlocale(LC_CTYPE, "Ukrainian");
 cout << endl;
 cout <<program << "\t" << console << endl;
 int i; cin>>i; // Пауза
 return 0;
}
```


Результати *тестування* програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Приклади_На_Рядки...
Default locale is       C
Enter the Ukrainian word: Клавіатура
╧ЕюуЕрьр        Клавіатура
OS locale is    Ukrainian_Ukraine.1251
Програма        ?< ўі вга
DOS locale is   Ukrainian_Ukraine.866
╧ЕюуЕрьр        Клавіатура
Locale: Ukrainian_Ukraine.1251
Програма        ?< ўі вга
Locale: Ukrainian_Ukraine.866
╧ЕюуЕрьр        Клавіатура
Locale: Ukrainian_Ukraine.1251
Програма        ?< ўі вга
```

З результатів тестування бачимо, що за домовленістю встановлюється стандартний локальний контекст C, за яким українське слово з програми виводиться правильно, а введене з консолі – неправильно.

Виклик `setlocale(LC_ALL, "")` встановлює локальний контекст з налаштуваннями операційної системи, де за домовленістю передбачено використання української мови з кодовою сторінкою 1251, за якою українське слово з програми виводиться правильно, а введене з консолі – неправильно.

Оскільки в консольному вікні використовують кодову сторінку 866, то для правильного виведення українських слів, попередньо введених з консолі, необхідно змінювати локальний контекст через виклик `setlocale(LC_CTYPE, ".866")`, однак при цьому українські слова, задані у програмі, виводяться неправильно.

Якщо ж змінити локальний контекст через виклик функції `setlocale(LC_CTYPE, ".1251")`, то все буде навпаки: українські слова з програми виводитимуться правильно, а введені з консолі – неправильно.

Отже, під час виведення українських слів з програми (рядків-літералів) необхідно встановлювати кодову таблицю 1251, а перед виведенням українських слів, попередньо введених з консолі, необхідно встановлювати кодову таблицю 866.

Приклад 36. Розробити програму, яка правильно виводитиме на екран консолі два українські слова: 1) задане безпосередньо в програмі (рядок-літерал); 2) введене з клавіатури консолі.

Програма:

```
#include <iostream>
#include <locale>
using namespace std;
int main() {
 char *program = "Програма", console[15];
 // Локальний контекст для виведення літералів
 setlocale(LC_CTYPE, "Ukrainian");
 cout << "Українське слово з програми: ";
 cout << "\t" << program << endl;
 cout << "Введіть українське слово: ";
 cin >> console;
 cout << "Українське слово з консолі: ";
 // Локальний контекст для виведення українських
 // слів, попередньо введених з клавіатури
 setlocale(LC_CTYPE, "Ukrainian_Ukraine.866");
 cout <<"\t" << console << endl;
 int i; cin>>i;
 return 0;
}
```

Результати *тестування* програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Приклади_На_Р...
Українське слово з програми:   Програма
Введіть українське слово: Клавіатура
Українське слово з консолі:    Клавіатура
```

Якщо порівняти різні способи відображення кириличних літер на екрані консолі, то, на думку автора, найпростішим способом є спосіб налаштування консолі за допомогою функцій `SetConsoleCP` і `SetConsoleOutputCP` (див. 4.6.4).

Приклад 37 (модифікація прикладу 29). Скласти програму, яка демонструє використання функцій `strcpy`, `strcat` і `strlen` для літер кирилиці.

Програма демонстрації функцій:

```
#include <iostream>
#include "windows.h"
using namespace std;
int main()
{
  SetConsoleCP(1251);
  SetConsoleOutputCP(1251);
  char str[80];
  strcpy(str, "Привіт з функції " );
  strcat(str, "strcpy "); strcat(str, "та функції ");
  strcat(str,  "strcat!");
  cout<<"str = \n"<< str<<endl;
  strcpy( str, "Привіт" );
  cout<<"\""<<str<<"\""<<" має  ";
  cout<<strlen(str)<<" символів "<<endl;
  int c; cin>>c;
  return 0;
}
```

Результати *тестування* програми:

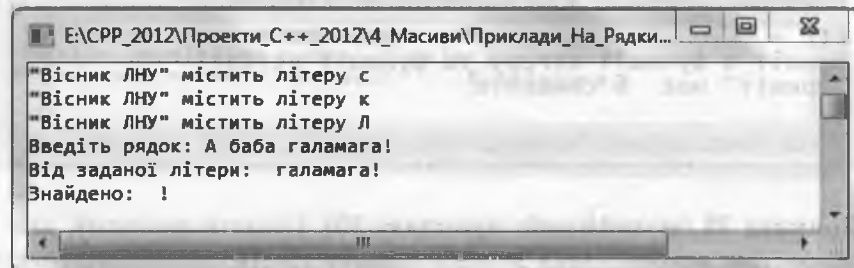
```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Приклади_На_Рядки_Остат...
str =
Привіт з функції strcpy та функції strcat!
"Привіт" має  6 символів
```

Приклад 38 (модифікація прикладу 30). Скласти програму, яка демонструє використання функцій пошуку `strstr`, `strchr` і `strpbrk` для літер кирилиці.

Програма демонстрації функцій:

```
#include <iostream>
#include "windows.h"
using namespace std;
int main() {
 SetConsoleCP(1251);
 SetConsoleOutputCP(1251);
 const char *s1="Вісник ЛНУ", *p;
 char console[80], *ptr;
 const char *nabir[]={"с","к","Л","Д"};
 // Приклад на пошук підрядків
 for(int i=0; i<4; i++) { p=strstr(s1,nabir[i]);
  if(p) {cout<<"\""<<s1<<"\""<<" містить літеру ";
        cout <<nabir[i]<<endl; }  } // for
 // Приклад на пошук окремого символу
 cout<<"Введіть рядок: "; cin.getline(console,80);
 ptr=strchr(console,'г');
 if(ptr) cout<<"Від заданої літери:  "<<ptr<<endl;
 // Приклад на пошук символу з набору
 ptr=strpbrk(console,".!");
 if(ptr) cout<<"Знайдено:  "<<*ptr<<endl;
 int i; cin>>i; return 0;
}
```

Результати *тестування* програми:



4.6.6. Стандартні функції класифікації символів

Під час реалізації деяких алгоритмів обробки рядків необхідно вміти класифікувати символи рядка. Наприклад, необхідно обчислити кількість літер верхнього регістру чи кількість цифр у рядку тощо. Мова C++ дає змогу легко розв'язати ці та інші подібні задачі за допомогою використання однієї чи декількох спеціальних стандартних функцій класифікації символів, прототипи яких оголошені у заголовку `<cctype>`. Короткий опис цих функцій:

`int isalnum(unsigned char c)`	– повертає не нуль, якщо `c` – літера або цифра, і нуль – в інших випадках;
`int isalpha(unsigned char c)`	– повертає не нуль, якщо `c` – літера, і нуль – в інших випадках;
`int iscntrl(unsigned char c)`	– повертає не нуль, якщо `c` – символ керування, і нуль – в інших випадках;
`int isdigit(unsigned char c)`	– повертає не нуль, якщо `c` – цифра, і нуль – в інших випадках;
`int isgraph (unsigned char c)`	– повертає не нуль, якщо `c` – символ виведення на друк, відмінний від пропуску, і нуль – в інших випадках;
`int islower(unsigned char c)`	– повертає не нуль, якщо `c` – літера нижнього регістру, і нуль – в інших випадках;
`int isprint(unsigned char c)`	– повертає не нуль, якщо `c` – символ виведення на друк, включаючи пропуск, і нуль – в інших випадках;
`int ispunct(unsigned char c)`	– повертає не нуль, якщо `c` – символ пунктуації (будь-який символ, окрім літер, цифр і пропуску), і нуль – інакше;
`int isspace(unsigned char c)`	– повертає не нуль, якщо `c` – символ пропуску, і нуль – в інших випадках;
`int isupper(unsigned char c)`	– повертає не нуль, якщо `c` – велика літера верхнього регістру, і нуль – в інших випадках;
`int isxdigit(unsigned char c)`	– повертає не нуль, якщо `c` – 16-а цифра, і нуль – в інших випадках;

Усі ці функції, окрім `isdigit` та `isxdigit`, є залежними від локалі; їхня поведінка може змінитись зі зміною локалі.

У заголовку `<cctype>` також оголошені прототипи функцій:

`int tolower(unsigned char c)`	– перетворює літеру `c` до нижнього регістру (залежна від локалі);
`int toupper(unsigned char c)`	– перетворює літеру `c` до верхнього регістру (залежна від локалі).

Приклад 39. Скласти програму, яка обчислюватиме кількість літер, цифр, пропусків і символів пунктуації у реченні, введеному користувачем з клавіатури консолі. Усі літери цього речення перетворити на літери верхнього регістру.

Програма:

```
#include <iostream>
#include <locale>
using namespace std;
int main() { int letters=0, spaces=0, digits=0,
 puncts=0; char cons [80];
 // Локальний контекст для виведення літералів
 setlocale(LC_CTYPE, "Ukrainian");
 cout << "Введіть речення: ";
 // Локальний контекст для правильної обробки
 // українських літер, введених з клавіатури
 setlocale(LC_CTYPE, "Ukrainian_Ukraine.866");
 cin.getline(cons, 80);  char *p = cons;
 while(*p)
 {if(isalpha(unsigned char(*p))) letters++;
  if(isspace(unsigned char(*p))) spaces++;
  if(ispunct(unsigned char(*p))) puncts++;
  if(isdigit(unsigned char(*p))) digits++; p++; }
 p=cons; while(*p){*p=toupper(unsigned char(*p)); p++;}
 cout << cons << endl;
 setlocale(LC_CTYPE, "Ukrainian_Ukraine.1251");
 cout <<"Літер: "<<letters;
 cout <<"\tЦифр: "<<digits<< endl;
 cout <<"Пропусків: "<<spaces;
 cout <<"\tСимволів пунктуації: "<<puncts<< endl;
 int i; cin>>i; return 0;
}
```


Результати *тестування* програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Приклади_На_Рядк...
Введіть речення: 23 гарбузи, 12 apples! Стоп!!
23 ГАРБУЗИ, 12 APPLES! СТОП!!
Літер: 17      Цифр: 4
Пропусків: 4   Символів пунктуації: 4
```

Функції, оголошені у `<cctype>`, налаштовані на CP866. У цьому легко переконатися, якщо програму записати так:

```
#include <iostream>
#include "windows.h"
using namespace std;
int main() {int letters=0,spaces=0, digits=0, puncts=0;
 SetConsoleCP(1251);  SetConsoleOutputCP(1251);
 char cons[80]; cout << "Введіть речення: ";
 cin.getline(cons,80); char *p=cons;
 while(*pstr)
 { if(isalpha(unsigned char(*p))) letters++;
   if(isspace(unsigned char(*p))) spaces++;
   if(ispunct(unsigned char(*p))) puncts++;
   if(isdigit(unsigned char(*p))) digits++;  p++; }
 p=cons; while(*p){*p=toupper(unsigned char(*p)); p++;}
 cout<< cons<< endl; cout <<"Літер: "<<letters;
 cout <<"\tЦифр: "<<digits<< endl;
 cout <<"Пропусків: "<<spaces;
 cout <<"\tСимволів пунктуації: "<<puncts<< endl;
 int i; cin>>i; return 0;
}
```

Результати *тестування* програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Приклади_На...
Введіть речення: 13 boys, 15 дівчат!
13 BOYS, 15 дівчат!
Літер: 4      Цифр: 4
Пропусків: 3   Символів пунктуації: 2
```

У багатьох програмах на С++ використовують властивості впорядкованості значень кодів латинських літер і цифр. Опираючись на визначення змінних

```
int n; char c;
```

покажемо такі стандартні прийоми обробки символів:

- отримати символ десяткової цифри зі значення цілої змінної, що лежить у діапазоні 0 ... 9:

```
c = n + '0';
```

- отримати символ 16-кової цифри зі значення цілої змінної, що лежить у діапазоні 0 … 15:

```
c = (n <= 9)? n + '0': n - 10 + 'A';
```

- отримати значення цілої змінної зі символу десяткової цифри:

```
if (c >='0' && c <='9') n = c - '0';
```

- отримати значення цілої змінної зі символу 16-кової цифри:

```
if (c >='0' && c <='9') n = c - '0';
else if (c >='A' && c <='F') n = c - 'A' + 10;
```

- перетворити малу (рядкову) латинську літеру у велику:

```
if(c >='a' && c <='z') c = c - 'a' + 'A';
```

- перетворити рядок, що містить зображення десяткового раціонального числа, у раціональне число (`StringToFloat`):

```
double StringToFloat(char c[])
{ int i; double n, v;
  for (i=0; !(c[i]>='0' && c[i]<='9'); i++)
   if (c[i]=='\0') return 0;  // Пошук першої цифри
  // Нагромадження цілої частини числа
  for (n=0; c[i]>='0' && c[i]<='9'; i++)
   n = n*10 + c[i] - '0';      // Цифра за цифрою
  if (c[i]!= '.') return n;
  // Нагромадження дробової частини числа
  for (i++, v=0.1; c[i]>='0' && c[i]<='9';i++)
   { n += v*(c[i] - '0'); v = v/10;}
  return n;
}
```


4.6.7. Сканування формату рядка

Послідовність символів у рядку підпорядковується деякому *закону слідування* (або *формату*). Наприклад, рядок зі *слів* можна визначити як набір послідовностей *значущих* символів (відмінних від пропуску), розділених послідовностями пропусків. Сканувати формат рядків символів можна двома способами:

1) за допомогою логіки *змінних стану*;
2) за допомогою *структурної* логіки.

У першому випадку програма містить базовий цикл *перегляду символів* рядка. У залежності від поточного символу змінюється значення однієї чи декількох змінних, які відображають значення певних *параметрів рядка* (змінних стану).

Наприклад, для виокремлення слів достатньо однієї змінної стану, яка зберігатиме значення *кількості* значущих символів усередині слова.

Якщо, наприклад, визначатимемо у рядку слово максимальної (чи мінімальної) довжини, то цю змінну стану необхідно *обнулювати* на кожному пропускові і збільшуватиметься на 1 на кожному символі слова. Очевидно, що у момент обнуління ненульового значення цієї змінної програма знаходитиметься в кінці поточного слова.

Необхідно пам'ятати, що після завершення базового циклу перегляду символів рядка *останнє* слово, якщо після нього немає символу пропуску, залишиться необробленим!

Під час використання *структурної логіки* встановлюється однозначна відповідність між елементами формату (наприклад, словами) і керуючими конструкціями алгоритму (зазвичай, циклами). Опис формату нібито "зашивається" у алгоритм.

Наприклад, якщо певний алгоритм обробляє у рядку окремі слова, то у програмі має бути базовий цикл, на кожному кроці якого переглядається одне слово. Усередині базового циклу мають бути два цикли – по послідовності пропусків і по послідовності символів слова.

Проілюструємо ці два способи сканування формату рядків символів на **прикладі** визначення у рядку символів слова *мінімальної* довжини (необхідні пояснення відображені у коментарях):

```
// Функція find_min повертає індекс початку слова
// мінімальної довжини, першого серед таких слів
// за розташуванням у рядку,
// або -1 за відсутності слів (порожній рядок)
//---------------------------------------------------------
// 1-й спосіб - за допомогою логіки змінних стану
// Змінна стану n - зберігає кількість символів слова
int find_min(char s[])
{ int i, n = 0, lmin =100, b=-1;
  for (i=0; s[i]!=0; i++) // Базовий цикл
  {
    if (s[i]!=' ') n++;
    else
      { if (n < lmin) { lmin = n; b = i-n; } n=0; }
  }
  // Перевірка для останнього слова
  if (n < lmin) b = i-n;
  return b;
}
//---------------------------------------------------------
// 2-й спосіб - за допомогою структурної логіки
// 3 цикли: перегляд слів, пропусків і символів
int find_min(char s[])
{ int i=0, n, lmin =100, b=-1;
  while (s[i]!=0) // Цикл перегляду слів рядка
  { // Ігнорування пропусків перед словами
    while (s[i]==' ') i++;
    // Визначення кількості символів поточного слова
    for (n = 0; s[i]!=' ' && s[i]!=0; i++, n++);
    // Контекст вибору мінімуму
    if (n < lmin) { lmin = n; b = i - n; }
  }
  return b;
}
```


Приклад 40. Скласти програму, яка обчислюватиме кількість слів у рядку, введеного користувачем з консолі.

Програма:

```
#include <iostream>
#include <locale>
using namespace std;
int words(char c[])
// Функція визначення кількості слів
{
  int nw=0; // Лічильник кількості слів
  for(int i=0; c[i]!=0; i++)  // Базовий цикл
   if (c[i]!=' ' && (i==0 || c[i-1]==' ')) nw++;
  // Черговий символ - не пропуск -> початок нового
  // слова, якщо цей символ перший у рядку чи перед
  // ним стоїть пропуск
  return nw;
}
//---------------------------------------------------------
int main()
{
 int w; char console[80];
 // Локальний контекст для виведення літералів
 setlocale(LC_CTYPE, "Ukrainian");
 cout << "Введіть речення: "<< endl;
 cin.getline(console,80);
 w = words(console);
 cout <<"У реченні " << w<<" слів."<< endl;
 cin>>w;  return 0;
}
```

Результати *тестування* програми:

```
E:\CPP_2012\Проекти_C++_2012\4_Масиви\Приклади_На_...
Введіть речення:
У попа була собака. Він її любив!
У реченні 7 слів.
```

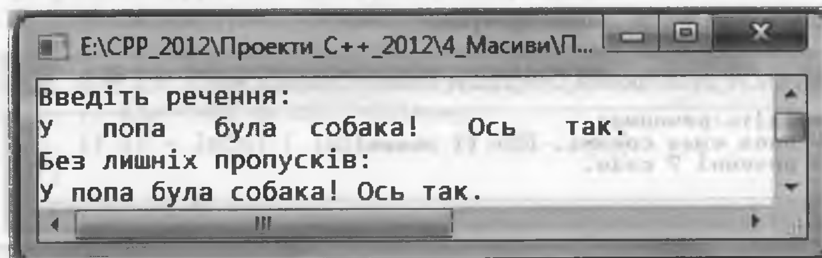
Приклад 41. Скласти програму, яка вилучатиме зайві пропуски у рядку символів.

Попередні міркування. Для початкового значення у рядку с індекс і змінюватиметься *рівномірно*, а для модифікованого значення цього рядка індекс j змінюватиметься тільки у моменти додавання наступного символу.

Програма:

```
#include <iostream>
#include "windows.h"
using namespace std;
void no_spaces(char *c)
// Функція вилучення зайвих пропусків
{ int j=0;
   for(int i=0; c[i]!=0; i++)  // Базовий цикл
    if (c[i]!=' ')
     { if (i!=0 && c[i-1]==' ') c[j++]=' ';
       c[j++]=c[i]; }
   c[j++]=0;
}
//-------------------------------------------------------
int main() { char console[80];
  SetConsoleCP(1251);  SetConsoleOutputCP(1251);
  cout << "Введіть речення: "<< endl;
  cin.getline(console,80);  no_spaces(console);
  cout <<"Без зайвих пропусків: " << endl;
  cout <<console<< endl; cin>>console; return 0;
}
```

Результати *тестування* програми:



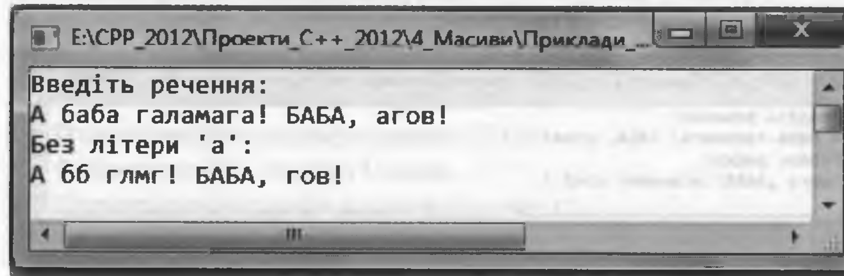
Приклад 42. Скласти програму, яка вилучатиме з рядка символів фіксовану літеру.

Попередні міркування. Для початкового значення у рядку s індекс і змінюватиметься *рівномірно*, а для модифікованого значення цього рядка індекс j змінюватиметься тільки у моменти додавання символу, відмінного від символу, що зберігається у змінній с.

Програма:

```
#include <iostream>
#include "windows.h"
using namespace std;
void delete_letter(char *s, char c)
// Функція вилучення у рядку s літери c
{ int i=0; // Індекс початкового рядка s
  int j=i; // Індекс модифікованого рядка s
  while(s[i]) {if(s[i]!=c) s[j++]=s[i]; i++;}
  while(s[j]) {s[j]='\0'; j++;}
}
//-------------------------------------------------------
int main() { char  console[80];
 SetConsoleCP(1251);  SetConsoleOutputCP(1251);
 cout << "Введіть речення: "<< endl;
 cin.getline(console,80);
 delete_letter(console, 'a');
 cout <<"Без літери 'a': " << endl;
 cout <<console<< endl;
 cin>>console;  return 0;
}
```

Результати *тестування* програми:



Приклад 43. Скласти програму, яка здійснюватиме реверс рядка (розміщення символів у зворотному порядку).

Програма:

```
#include <iostream>
#include <cstring>
#include "windows.h"
using namespace std;
void reverse_string(char *s)
//  Реверс символів у рядку s
{
  char h;
  int i=0;   // Індекс від початку рядка
  int j=strlen(s)-1; // Індекс від кінця рядка
  do
       { h=s[i]; s[i++]=s[j]; s[j--]=h; }
  while (i<j);
}
//--------------------------------------------------
int main() {
 char  console[80];
 SetConsoleCP(1251);  SetConsoleOutputCP(1251);
 cout << "Введіть речення: "<< endl;
 cin.getline(console,80);
 reverse_string(console);
 cout <<"Реверс рядка: " << endl;
 cout <<console<< endl;
 cin>>console;  return 0;
}
```

Результати *тестування* програми:

```
E:\CPP_2012\Проекти_С++_2012\4_Масиви\Приклади_На_Рядки_Остаточ...
Введіть речення:
А баба галамага! БАБА, агов!
Реверс рядка:
!вога ,АБАБ !агамалаг абаб А
```


? Запитання для самоперевірки

1. Що таке масив? Як описують одновимірний масив?
2. Як описують багатовимірний масив? Що таке елемент масиву?
3. Охарактеризуйте спосіб ініціалізації одновимірного масиву.
4. Охарактеризуйте спосіб ініціалізації багатовимірного масиву.
5. Охарактеризуйте спосіб ініціалізації рядків.
6. Що розуміють під динамічним розміщенням даних?
7. Що таке вказівник? Як оголошують вказівник?
8. Вкажіть на особливості присвоєння значень вказівникам.
9. Для чого використовують операцію рознайменування?
10. Як виокремлюють і звільняють пам’ять для динамічного масиву?
11. Охарактеризуйте зв’язок масивів і вказівників.
12. Охарактеризуйте взаємодію функцій та вказівників.
13. Охарактеризуйте спосіб використання вказівника на функції.
14. Що розуміють під відокремленим компілюванням модулів?
15. Охарактеризуйте шаблон багатомодульного консольного проекту.
16. Напишіть фрагмент програми організації схеми Горнера для обчислення значення многочлена у заданій точці.
17. З якою метою використовують лінійний пошук?
18. З якою метою використовують бінарний пошук?
19. Напишіть фрагмент програми організації лінійного пошуку.
20. Напишіть фрагмент програми організації бінарного пошуку.
21. Охарактеризуйте спосіб сортування вектора вибором.
22. Охарактеризуйте спосіб сортування вектора обміном.
23. Охарактеризуйте спосіб сортування вектора включенням.
24. Напишіть фрагмент програми злиття двох упорядкованих масивів.
25. Охарактеризуйте стандартні функції обробки рядків у стилі C.
26. Охарактеризуйте способи кодування символів.
27. Охарактеризуйте функції Windows API для рядків.
28. Охарактеризуйте таблицю кодувань CP866.
29. Охарактеризуйте систему кодувань CP1251.
30. Охарактеризуйте стандарт Unicode.
31. Охарактеризуйте засоби локалізації у стилі C.
32. Охарактеризуйте функції класифікації символів.

Завдання для програмування

Завдання 4.1. Дано натуральне число *n* і дійсні числа $a_1, a_2, ..., a_n$. Скласти програми розв'язання таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи; у задачах 1 – 7 передбачити опрацювання ситуації відсутності від'ємних елементів):

1. Знайти добуток елементів масиву, які розміщені після останнього від'ємного числа.
2. Знайти середнє арифметичне елементів масиву, які розміщені після останнього від'ємного числа.
3. Замінити всі елементи, які розміщені перед першим від'ємним числом, на число 2.
4. Замінити всі елементи, які розміщені після останнього від'ємного числа, на число 10.
5. Знайти середнє арифметичне елементів масиву, які розміщені між першим та останнім від'ємним числом.
6. Замінити всі від'ємні числа на їхні квадрати і впорядкувати після цього масив за незростанням.
7. Замінити всі від'ємні числа на їхні модулі і впорядкувати після цього масив за незростанням.
8. Знайти добуток елементів масиву, які розміщені після останнього входження найменшого числа.
9. Знайти суму елементів масиву, які розміщені перед останнім входженням найменшого числа.
10. Знайти середнє арифметичне елементів масиву, які розміщені після останнього входження найбільшого числа.
11. Знайти добуток елементів масиву, які розміщені перед останнім входженням найбільшого числа.
12. Знайти середнє арифметичне елементів масиву, які розміщені між останніми входженнями найбільшого та найменшого чисел.
13. Знайти суму елементів масиву, які розміщені між першими входженнями найбільшого та найменшого числа.
14. Замінити всі додатні числа на їхні квадрати і впорядкувати після цього масив за незростанням.
15. Замінити всі нульові числа на число 3 і впорядкувати після цього масив за незростанням.

Завдання 4.2. Дано натуральне число *n* і цілі числа $a_1, a_2, ..., a_n$. Скласти програми розв'язання таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи):

1. Впорядкувати масив за незростанням методом обміну.
2. Впорядкувати масив за незростанням методом вибору.
3. Впорядкувати масив за незростанням методом включення.
4. Стиснути масив, видаливши з нього числа, модуль яких менший за 3.
5. Стиснути масив, видаливши з нього всі нулі.
6. Стиснути масив, видаливши з нього всі від'ємні числа.
7. Після кожного нуля вставити у масив число 100.
8. Після кожного від'ємного числа вставити у масив два нулі.
9. Кожне від'ємне число у масиві оточити з обох боків нулями.
10. Впорядкувати масив за неспаданням модулів чисел методом обміну.
11. Впорядкувати масив за неспаданням модулів чисел методом вибору.
12. Впорядкувати масив за неспаданням модулів чисел методом включення.
13. Впорядкувати масив за незростанням модулів чисел методом обміну.
14. Впорядкувати масив за незростанням модулів чисел методом вибору.
15. Впорядкувати масив за незростанням модулів чисел методом включення.

Завдання 4.3. Дано цілочисельну прямокутну матрицю. Скласти програми розв'язання таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи):

1. Визначити кількість рядків, які не містять нулів.
2. Визначити кількість стовпців, які не містять нулів.
3. Визначити кількість стовпців, які містять хоча б один нуль.
4. Визначити кількість рядків, які містять хоча б один нуль.
5. Визначити добуток елементів у кожному рядку, що не містить нуля.
6. Визначити добуток елементів у кожному стовпці, що не містить нуля.
7. Визначити суму елементів у кожному рядку, що не містить від'ємних чисел.
8. Визначити суму елементів у кожному стовпці, що не містить від'ємних чисел.

9. Визначити середнє арифметичне елементів у кожному рядку, який містить хоча б один нуль.
10. Визначити середнє арифметичне елементів у кожному стовпці, який містить хоча б один нуль.
11. Визначити суму елементів у кожному рядку, який містить хоча б одне від'ємне число.
12. Визначити суму елементів у кожному стовпці, який містить хоча б одне від'ємне число.
13. Знайти номер першого з рядків, що не містить від'ємних чисел.
14. Знайти номер останнього з рядків, що не містить від'ємних чисел.
15. Знайти номер останнього зі стовпців, що не містить нуля.

Завдання 4.4. Скласти програми для розв'язання таких задач опрацювання рядків (задачу обрати згідно з номером студента у списку студентів підгрупи):

1. Відомо, що у рядку є хоча б один символ `"*"` (зірочка). Знайти номер першого входження `"*"` у рядок і перелічити кількість усіх входжень `"*"` у рядок (величина *n*). Долучити символ `"*"` у кінець рядка, повторити його *n* разів.
2. Відомо, що у рядку є хоча б один символ `"*"` (зірочка). Знайти номер останнього входження `"*"`у рядок і перелічити кількість усіх входжень `"*"` у рядок (величина *n*). Долучити символ `"*"` на початок рядка, повторити його *n* разів.
3. Відомо, що у рядку є літери і цифри. Перетворити рядок так, щоб спочатку розміщувалися літери, а потім цифри. Порядок символів необхідно зберегти.
4. Відомо, що у рядку є літери і цифри. Перетворити рядок так, щоб спочатку розміщувалися літери у прямому порядку, а потім цифри у зворотному порядку.
5. Відомо, що у рядку є літери і цифри. Перетворити рядок так, щоб спочатку розміщувалися цифри у прямому порядку, а потім літери у зворотному порядку.
6. Прочитати з клавіатури два рядки символів. Якщо вони однакові (за символами, інтервали не рахуються), то надрукувати слово `"Так"`, у протилежному випадку – `"Hi"`.

7. Прочитати з клавіатури рядок символів. Ввести шаблон і вибрати з рядка усі слова, що відповідають шаблону. *Слова* у тексті рядка розділяють пропусками.
8. Нехай цифрам від 1 до 9 відповідають латинські літери від A(a) до I(i). Прочитати з клавіатури рядок символів. Скласти новий рядок з цифр, які відповідають тільки зазначеним літерам. Рядок-результат впорядкувати за неспаданням.
9. Прочитати з клавіатури рядок символів. Вибрати з нього латинські літери від A(a) до I(i), перетворити малі (рядкові) літери у великі, і впорядкувати їх за алфавітним порядком.
10. Кожен символ `"+"` у рядку замінити на символ `"–"`, якщо перед `"+"` стоїть непарна цифра.
11. Прочитати з клавіатури рядок символів. Скопіювати його, замінивши водночас літери `"R"`, `"S"`, `"T"` на `"K"`, `"L"`, `"M"`, відповідно.
12. Прочитати з клавіатури рядок символів. Перелічити в ньому окремо символи `"("` та `")"`. За неспівпадання кількості повторень додати необхідний символ наприкінці рядка, повторивши його необхідну кількість разів.
13. Перелічити у рядку усі послідовності символів `"cd"` і видалити ті з них, перед якими стоїть літера `"b"`.
14. Перелічити у рядку усі послідовності символів `"cd"` і видалити ті з них, після яких стоїть літера `"b"`.
15. Прочитати з клавіатури слово і побудувати всі його *анаграми* (усі можливі буквосполучення, які складаються з літер заданого слова).

Завдання 4.5.

Термінологія. *Текст* – довільна послідовність символів в одному рядку. *Слова* у тексті розділені пропусками. *Речення* – послідовність слів, що завершується крапкою (знаком оклику чи запитання). Скласти програми (і протестувати) розв'язання таких задач опрацювання рядків (задачу обрати згідно з номером студента у списку студентів підгрупи):

1. Визначити у тексті максимальну довжину послідовності символів, що не є літерами.
2. Знайти максимальну за довжиною монотонну (неспадну або незростаючу) підпослідовність натуральних чисел, заданих у тексті так, що числа відокремлені пропусками.

3. Визначити всі слова тексту, що складаються з тих самих літер, що й перше слово цього тексту.
4. У тексті знайти пари слів, що є дзеркальними відображеннями одне одного (наприклад, "ole" і "elo").
5. У тексті знайти всі симетричні слова (наприклад "oko", "ABBA").
6. Знайти всі слова, що трапляються у кожному з двох речень тексту.
7. Відредагувати деякий текст, видаливши з нього всі слова з непарними номерами і перевертаючи слова з парними номерами (наприклад, "How do you do? " ⇒ "od od? ").
8. Для кожного символу тексту зазначити, скільки разів він трапляється у тексті. Повідомлення про символ виводити тільки один раз.
9. Деякий текст має два речення. Знайти найкоротше слово першого рядка, якого нема у другому реченні.
10. Відредагувати деякий текст, видаливши у ньому всі слова, які налічують входження тільки двох однакових літер (наприклад, "Akka ababa cece knopka." ⇒ "knopka.").
11. Замінити в англомовному тексті закінчення слів "ing" на "ed", стиснувши водночас текст.
12. В англомовному тексті знайти слово, у якому кількість голосних літер (*a*, *e*, *i*, *o*, *u*) є найбільшою.
13. Характеристика слова – кількість *різних* символів, що у нього входять. Впорядкувати слова у тексті за спаданням їхніх характеристик.
14. Відстань між двома словами *однакової* довжини – це кількість позицій, у яких стоять різні символи. В деякому тексті знайти пару найвіддаленіших слів заданої довжини.
15. Визначити всі слова тексту, що складаються з тих самих літер, що й останнє слово цього тексту.

5. ЗАДАЧІ КОМБІНАТОРНОЇ ОПТИМІЗАЦІЇ

План викладу матеріалу:

1. Вступ у теорію оптимізації.
2. Метод динамічного програмування.
3. Метод ДП розв'язування задач про наплічник.
4. Метод ДП на графах.
5. Застосування жадібного методу.

Ключові терміни розділу

- *Функція мети*
- *Точка глобального мінімуму*
- *Точка локального мінімуму*
- *Точки екстремуму*
- *Безумовний екстремум*
- *Градієнт, матриця Гессе*
- *Умовний екстремум*
- *Функція Лагранжа*
- *Класична задача умовної оптимізації*
- *Задача математичного програмування*
- *Лінійне та нелінійне програмування*
- *Дискретна оптимізація*
- *Цілочислова оптимізація*
- *Комбінаторна оптимізація*
- *Задачі, що перекриваються*
- *Принцип оптимальності*
- *Метод динамічного програмування (ДП)*
- *Низхідний та висхідний метод ДП*
- *Задача про обмежений наплічник*
- *Задача про 0-1 наплічник*
- *Граф: вершини, ребра*
- *Неорієнтовані графи*
- *Орієнтований граф (орграф)*
- *Вузли і дуги орграфа*
- *Зображення графа*
- *Шлях і цикл у графі*
- *Шлях і цикл в орграфі*
- *Ациклічний орграф*
- *Лема про вузли ациклічного орграфа*
- *Теорема про нумерацію вершин ациклічного орграфа*
- *Зв'язаний граф; дерево, ліс*
- *Каркас (покривне дерево)*
- *Навантажений (зважений) граф*
- *Зберігання зваженого графа*
- *Найкоротші шляхи*
- *Принцип оптимальності*
- *Релаксація вершин графа*
- *Дерево попередників*
- *Алгоритм Флойда-Уоршолла*
- *Алгоритм Дейкстри*
- *Алгоритм Форда-Беллмана*
- *Алгоритм Пріма*
- *Алгоритм Краскала відшукання мінімального каркасу*
- *Існування циклів зі сумарною від'ємною вагою*

5.1. Вступ у теорію оптимізації

5.1.1. Постановка задачі оптимізації

Задача оптимізації – це задача відшукання *екстремальних* (*найменших* або *найбільших*) значень дійсної функції у деякій області.

Перші задачі оптимізації геометричного змісту виникли ще у давні часи. Розвиток промисловості та науки у XVII–XVIII ст. зумовив до потреби у дослідженні складніших задач на екстремум, а також сприяв виникненню варіаційного числення. Лише у XX ст. стали актуальними *задачі оптимального керування* (у тому або іншому сенсі) різними процесами фізики, техніки, економіки тощо.

Розглянемо спочатку задачу оптимізації, в якій необхідно визначити найменше значення (*мінімум*) дійсної функції $f(x)$ на деякій множині *X*:

$$f(x) \to \min , \ x \in X . \qquad (1)$$

В (1) $f(x)$ називають функцією *мети*, а X – *допустимою* множиною. Довільний елемент $x \in X$ є допустимою точкою для задачі (1). Надалі розглядатимемо *скінченновимірні* задачі оптимізації, тобто задачі, в яких $X \subseteq R^n$.

Точку $x_* \in X$ називають:

1) точкою *глобального мінімуму* функції $f(x)$ на множині X, якщо

$$\forall x \in X: \ f(x_*) \le f(x); \qquad (2)$$

2) точкою *локального мінімуму* функції $f(x)$ на X, якщо

$$\exists \ \varepsilon > 0 \ \ \forall x \in X \bigcap X_\varepsilon(x_*): \ f(x_*) \le f(x), \qquad (3)$$

де $X_\varepsilon(x_*) = \{x : x \in R^n, \ \|x - x_*\| \le \varepsilon\}$ – окіл точки x_* радіуса $\varepsilon > 0$ з центром у точці x_*.

Якщо нерівність (2) або (3) для $x \ne x_*$ є строгою, то x_* – точка *строгого мінімуму* в глобальному або локальному значенні. Глобальний розв'язок є водночас і локальним. Зворотне твердження – неправильне.

Якщо $x_* \in X$ – точка глобального мінімуму, то цей факт ще фіксують так: $f(x_*) = \min\limits_{x \in X} f(x)$ (або $x_* = \arg \min\limits_{x \in X} f(x)$). Надалі домовимося, що $f_* = \min\limits_{x \in X} f(x)$. Множину *всіх* точок глобального мінімуму $f(x)$ на X позначають так:

$$\mathrm{Arg}\min_{x \in X} f(x) = \{x : x \in X, f(x) = f_*\}.$$

Отже, $\arg\min\limits_{x \in X} f(x)$ – деяка точка множини $\mathrm{Arg}\min\limits_{x \in X} f(x)$.

За аналогією з (1) можна розглядати задачу визначення найбільшого значення (*максимуму*) функції $f(x)$ на множині X:

$$f(x) \to \max_{x \in X} . \qquad (4)$$

Легко бачити, що задача (4) еквівалентна задачі

$$-f(x) \to \min_{x \in X}$$

у тому сенсі, що множини локальних, глобальних і строгих розв'язків першої та другої задач збігаються. Задачі (1) і (4) ще називають задачами на *екстремум*. Надалі розглядатимемо здебільшого задачі мінімізації (1).

У разі розв'язування задач на *екстремум* виникає питання про існування розв'язків. З огляду на це, нагадаємо результати математичного аналізу.

Теорема 1 (теорема Вейєрштрасса). Нехай X – компакт у R^n, $f(x)$ – неперервна функція на X. Тоді точка глобального мінімуму функції $f(x)$ на X існує.

Надалі буде корисним дещо інший варіант цієї теореми.

Теорема 2. Нехай X – замкнута множина в R^n, $f(x)$ – неперервна функція на X, причому для деякого $x^0 \in X$ множина

$$N(x^0) = \{x \in X : f(x) \le f(x^0)\}$$

обмежена, тоді точка глобального мінімуму $f(x)$ на X існує.

Задача (1) – це загальне формулювання задач оптимізації. Однак ці задачі ще класифікують за різними ознаками залежно від вигляду функції $f(x)$ та області X.

5.1.2. Задачі безумовної оптимізації

Задача (1) є задачею безумовної оптимізації, якщо $X \equiv R^n$:

$$f(x) \to \min, \ x \in R^n. \tag{5}$$

Під час дослідження будь-якого типу оптимізаційних задач важливе значення мають умови оптимальності (екстремуму). Розрізняють необхідні і достатні умови екстремуму. Коротко нагадаємо їх.

Уведемо позначення:

- $f'(x) = \left(\frac{\partial f(x)}{\partial x_1}, ..., \frac{\partial f(x)}{\partial x_n}\right)^T$ – вектор перших частинних похідних, градієнт функції $f(x)$ у точці $x \in R^n$;
- $f''(x) = \left\{\frac{\partial^2 f(x)}{\partial x_i \partial x_j}\right\}_{i,j=\overline{1,n}}$ – матриця других частинних похідних (гессіан, матриця Гессе) функції $f(x)$ у точці $x \in R^n$.

Теорема 3. Нехай функція $f(x)$ диференційовна в точці $x_* \in R^n$. Якщо x_* – локальний розв'язок задачі (5), то

$$f'(x_*) = 0. \tag{6}$$

Теорема 4. Нехай функція $f(x)$ двічі диференційовна у точці $x_* \in R^n$. Якщо x_* – локальний розв'язок задачі (5), то матриця $f''(x_*)$ невід'ємно визначена, тобто

$$\forall h \in R^n : \left(f''(x_*)h, h\right) \ge 0. \tag{7}$$

Теорема 5. Нехай функція $f(x)$ двічі диференційовна у точці $x_* \in R^n$, $f'(x_*) = 0$, а матриця Гессе додатно визначена, тобто

$$\forall h \in R^n \ (h \ne 0) : (f''(x_*)h, h) > 0. \tag{8}$$

Тоді x_* – строгий локальний розв'язок задачі (5).

У випадку $X = R$ умови (6) – (8) можна записати у вигляді:

$$f'(x_*) = 0; \ f''(x_*) \ge 0 \ \ f''(x_*) > 0.$$

У простих випадках теореми 3 – 5 дають змогу явно розв'язати задачу (5). Теорема 3 дає *необхідні* умови мінімуму першого порядку (використано лише першу похідну $f(x)$). Теореми 4, 5 – *необхідні* та *достатні* умови мінімуму другого порядку.

5.1.3. Задачі умовної оптимізації

Задачу (1) називають задачею умовної оптимізації, якщо $X \ne R^n$. У цьому випадку умови теорем 3 – 5 справджуються, якщо $x_* \in \operatorname{int} X$, тобто x_* – внутрішня точка множини X.

Для багатьох задач умовної оптимізації екстремум досягається на межі X, і в цьому разі класичні результати аналізу не можна застосовувати. Отже, задачі умовної оптимізації складніші, ніж задачі безумовної оптимізації.

Розглянемо *класичну* задачу умовної оптимізації, в якій область X задано системою скінченної кількості рівнянь, тобто

$$f(x) \to \min_{x \in X}, \ X = \{x \in R^n : f_i(x) = 0, i = \overline{1,m}\}. \tag{9}$$

Під час дослідження задачі (9) важливу роль відіграє функція Лагранжа:

$$L(x,\lambda)=\sum_{i=0}^{m}\lambda_i f_i(x), \qquad (10)$$

де $x\in R^n$, $f_0(x)=f(x)$; $\lambda=(\lambda_0,\lambda_1,\ldots,\lambda_m)\in R^{m+1}$ – множники Лагранжа. У цьому випадку виконується така теорема.

Теорема 6 (правило множників Лагранжа). Нехай функції $f_i(x)$, $i=\overline{0,m}$ неперервно-диференційовані в деякому околі точки x_*. Якщо x_* – локальний розв'язок задачі (9), то існує вектор $\lambda^*=(\lambda_0^*,\lambda_1^*,\ldots,\lambda_m^*)\in R^{m+1}$, що не дорівнює нулю і такий, що

$$L'_x(x_*,\lambda^*)=\sum_{i=0}^{m}\lambda^*_i f'_i(x_*)=0. \qquad (11)$$

Якщо вектори (градієнти) $f'_1(x_*),\ldots,f'_m(x_*)$ лінійно незалежні, то виконується умова *регулярності* (тобто $\lambda_0^*\neq 0$).

Будь-яку точку $x\in X$, що задовольняє за деякого вектора $\lambda^*=(\lambda_0^*,\lambda_1^*,\ldots,\lambda_m^*)\neq 0$ умову (11) та умови допустимості

$$f_i(x_*)=0,\; i=\overline{1,m}, \qquad (12)$$

називають *стаціонарною* точкою задачі (9).

Стаціонарну точку визначають системою $n+m$ рівнянь (11), (12) з $n+m+1$ невідомим. Оскільки у випадку регулярності $\lambda_0^*\neq 0$, то можна прийняти $\lambda_0^*=1$.

Отже, розв'язуючи систему рівнянь (11), (12), достатньо розглянути лише два випадки: а) $\lambda_0^*=0$, б) $\lambda_0^*=1$ і тоді у системі буде $n+m$ невідомих.

Якщо виконується умова регулярності, то замість функції Лагранжа розглядатимемо *регулярну* функцію Лагранжа:

$$L(x,\lambda)=f(x)+\sum_{i=1}^{m}\lambda_i f_i(x). \qquad (13)$$

Теорема 7. Нехай функції $f_i(x)$, $i=\overline{0,m}$ двічі диференційовані у точці $x_*\in R^n$, яка задовольняє (12), і неперервно-диференційовані у деякому околі цієї точки, причому $f'_i(x_*)$, $i=\overline{1,m}$ лінійно незалежні. Якщо x_* – локальний розв'язок (9), то $(L''_{xx}(x_*,\lambda^*)h,h)\geq 0$ за довільного λ^*, що задовольняє (11) і ненульових $h\in R^n$ таких, що

$$(f'_i(x_*),h)=0,\; i=\overline{1,m}. \qquad (14)$$

Теорема 8. Нехай функції $f_i(x)$, $i=\overline{0,m}$ двічі диференційовані у точці $x_*\in R^n$, яка задовольняє (12), і для певного $\lambda^*\in R^n$ виконується умова (11), окрім того, $(L''_{xx}(x_*,\lambda^*)h,h)>0$ при всіх ненульових $h\in R^n$, що задовольняють (14). Тоді x_* строгий локальний розв'язок задачі (9).

5.1.4. Задачі математичного програмування

Задачею *математичного програмування* називають задачу умовної оптимізації (1), в якій допустима множина має вигляд:

$$X=\{x\in R^m : f_i(x)=0,\; i=\overline{1,m};\; f_i(x)\leq 0,\; i=\overline{m+1,s};\; x\in X_0\}. \qquad (15)$$

Задачу математичного програмування можна записати так:

$$f(x)\to\min; \qquad (16)$$

$$f_i(x)=0,\; i=\overline{1,m}; \qquad (17)$$

$$f_i(x)\leq 0,\; i=\overline{m+1,s};\; x\in X_0. \qquad (18)$$

Умови типу (17) називають обмеженнями-рівностями, типу (18) – обмеженнями-нерівностями. Ці типи умов називають *функціональними* обмеженнями, а умову $x\in X_0\subseteq R^n$ – прямим обмеженням. У виразі (15) окремих обмежень може не бути, тобто $m=0; s=0; X_0\equiv R^n,\ldots$. Отже, класична задача на умовний екстремум є частинним випадком задачі математичного програмування.

Зазвичай, множина X_0 має просту структуру, наприклад, $X_0 = \{x \in R^n : a_j \le x_j \le b_j, j = \overline{1,n}\}$ (причому може бути $a_j = -\infty$ і/або $b_j = +\infty$), $X_0 = R_+^n$ та ін.

Задачі математичного програмування класифікуються так:

- задачі *лінійного* програмування, якщо *всі* функціональні обмеження та функція мети є лінійними функціями;
- задачі *нелінійного* програмування, якщо функціональні обмеження і/або функція мети є нелінійними функціями;
- задачі *дискретного* програмування, якщо X – скінченна або злічена;
- задачі *стохастичного* програмування, якщо параметри функціональних обмежень і/або функції мети є випадковими величинами;
- задачі *параметричного* програмування, якщо параметри функціональних обмежень і/або функції мети набувають значення з деяких проміжків.

Вище перелічено *базові* класи задач оптимізації. За різними ознаками можна виокремити підкласи цих задач. Наприклад, як окремий підклас розглядають *дробово-лінійне програмування*, коли функціональні обмеження є лінійними, а функція мети – дробово-лінійна.

Серед задач нелінійного програмування виокремлюють задачі опуклого та квадратичного програмування. Задача (16) – (18) належить до *опуклого програмування*, якщо функція мети – опукла (увігнута вниз), усі функціональні обмеження (16) є лінійними функціями, а (17) – опуклими.

Задача (16) – (18) належить до *квадратичного програмування*, якщо функція мети – квадратична, а функціональні обмеження – лінійні.

Серед задач *дискретного* програмування виокремлюють задачі *комбінаторної оптимізації* (X – скінченна множина) та задачі *цілочислової оптимізації*, де окремі чи усі змінні набувають цілочислових значень.

5.1.5. Методи оптимізації

Співвідношення (16) – (18) визначають відповідну класифікацію задач оптимізації (чи задач математичного програмування). Методи, які розв'язують задачі оптимізації, називають *методами оптимізації* (автор просить вибачення за тавтологію). Очевидно, що *метод оптимізації* залежить від класу (чи навіть підкласу) відповідної задачі.

Сьогодні майже завершено теорію відшукання розв'язків задач лінійного програмування. Методи лінійного програмування (симплекс-метод, метод потенціалів, визначення максимального потоку тощо) базуються на *універсальному* методі *послідовного поліпшення* розв'язку.

Цей метод розв'язування задач оптимізації полягає в тому, що спочатку довільно обираємо деякий *допустимий* розв'язок задачі (або опорний план), а далі тим чи іншим способом відшукуємо розв'язки, які є наближеннями до оптимального розв'язку. Метод уперше сформулював Данциг 1947 р. Він базується на трьох істотних моментах. Необхідно:

- вказати спосіб обчислення опорного плану;
- встановити ознаку, яка дає змогу перевірити, чи опорний план є оптимальним;
- вказати спосіб переходу від неоптимального опорного плану до іншого опорного плану, ближчого до оптимального.

Отже, для задач лінійного програмування завжди можна відшукати оптимальний розв'язок за допомогою відповідної модифікації універсального методу послідовного поліпшення розв'язку. У результаті застосування алгоритму універсального методу завжди отримують один з таких варіантів відповіді:

- отримали оптимальний розв'язок;
- умови задачі суперечливі, тобто розв'язку не існує;
- функція мети – необмежена, тобто розв'язку також не існує.

Для задач нелінійного програмування не існує універсального методу розв'язування, що зумовило розроблення значної кіль-

кості *специфічних* методів розв'язування окремих підкласів задач нелінійного програмування.

Для кожного специфічного методу необхідно доводити існування розв'язку задачі та його єдиність, що є доволі складною математичною задачею.

Відомі точні методи розв'язування нелінійних задач, однак у цьому випадку існують труднощі обчислювального характеру, тобто навіть для сучасних комп'ютерів відповідні алгоритми точних методів є доволі трудомісткими, отож здебільшого для розв'язування нелінійних задач виправданим є застосування наближених методів.

Для задач лінійного програмування доведено наявність *єдиного* екстремуму, що досягається в одній (або декількох одночасно) з вершин багатогранника допустимих розв'язків задачі.

Однак у задачах нелінійного програмування може існувати декілька локальних екстремумів, що потребує пошуку серед них глобального екстремуму. Окрім цього, точки локальних чи глобального екстремумів можуть бути як межовими, так і внутрішніми точками області допустимих розв'язків.

Доведено, що множина допустимих розв'язок задачі лінійного програмування завжди є опуклою. У разі, якщо система обмежень задачі є нелінійною, вона може визначати неопуклу множину допустимих розв'язків, або навіть складатися з довільних, не зв'язаних між собою частин.

Для відшукання оптимальних розв'язків задач цілочислової оптимізації застосовують точні методи (відтинання чи комбінаторні) та наближені методи.

Базою *методів відтинання* є ідея поступового "звуження" області допустимих розв'язків задачі. Пошук цілочислового з розв'язку починають з розв'язування задачі з послабленими обмеженнями (тобто без урахування вимог цілочисловості змінних). Далі введенням у модель спеціальних додаткових обмежень, що враховують цілочисловість змінних, багатогранник допустимих розв'язків послабленої задачі поступово зменшують доти, доки змінні оптимального розв'язку не набудуть цілочислових значень.

Комбінаторні методи цілочислової оптимізації базуються на ідеї *цілеспрямованого* перебору всіх допустимих цілочислових розв'язків. Найпоширенішим у цій групі методів є метод *гілок і меж*.

Специфічні методи застосовують також для розв'язування задач стохастичного та параметричного програмування.

Оскільки метою цього розділу є вивчення методів розв'язування деяких задач комбінаторної оптимізації, то власне на них у подальшому зосередимо нашу увагу.

Оскільки у задачах комбінаторної оптимізації множина X – *скінченна*, то оптимальний розв'язок завжди існує і його, у принципі, можна отримати за допомогою вичерпного перебирання усіх елементів X з обчисленням значень функції мети.

Однак для значної кількості задач комбінаторної оптимізації елементами множини X слугують комбінаторні об'єкти (перестановки, комбінації, розміщення тощо). А це означає, що навіть за малої розмірності n простору R^n кількість елементів множини X є настільки великою, що виконати вичерпне перебирання усіх елементів X практично неможливо.

Здебільшого методи розв'язування задач комбінаторної оптимізації базуються на ідеї *цілеспрямованого* перебирання елементів множини X. Насамперед тут виникає проблема створення процедур, які давали б змогу безпосередньо розглядати лише відносно незначну частину елементів X, а решту елементів враховувати деяким побічним способом.

Далі у цьому розділі розглянемо алгоритми розв'язування задач комбінаторної оптимізації, які базуються на методі *динамічного програмування* та на *жадібному* методі. Ці два методи власне і реалізують ідею цілеспрямованого перебирання елементів множини X.

5.2. Метод динамічного програмування

5.2.1. Загальні положення

Метод *динамічного програмування* (ДП) зародився у рамках дослідження операцій (Річард Беллман, 1950-ті роки), де його інтерпретують як математичний *метод* відшукання оптимальних розв'язків з керування *багатокроковими* процесами, в яких стани досліджуваних систем змінюються у *часі* чи *поетапно*.

Звідси і термін "динамічне" – змінюється у часі. Однак так можна описати практично будь-яку систему керування. Особливість динамічного програмування полягала у тому, що здійснювали розбивку процесу на фіксовані *проміжки* часу та оптимальні розв'язки на кожному з проміжків (етапі, кроці) підбирали так, щоб у *сукупності* отримати оптимальний розв'язок усього процесу.

Іншими словами, метод *динамічного програмування* є варіантом методу декомпозиції для розв'язування *багатоетапних* задач оптимізації, які мають задовольняти *принцип оптимальності для підзадач*: оптимальний розв'язок задачі має містити оптимальні розв'язки підзадач на цих етапах (тобто метод ДП можна застосувати до тих задач, в яких функція мети є *адитивною*, а процес розв'язування піддається розбиттю на етапи).

Термін "*програмування*" (*dynamic programming*) у цьому випадку прямо не пов'язаний з розробкою програм для комп'ютера, так само як, наприклад, і "*лінійне програмування*" (*linear programming*). Він означає щось інше, а саме – строго задану послідовність операцій (арифметичних, логічних) з відшукання оптимального розв'язків (фактично це універсальний алгоритм розв'язку задачі).

Згодом метод ДП знайшов широке застосування в *інформатиці*, де його інтерпретують як підхід до розв'язування задач, в яких унаслідок застосування методу декомпозиції виникають підзадачі, що *перекриваються*.

Отже, другою важливою властивістю (для неоптимізаційних задач – єдиною властивістю) застосування методу ДП є наявність незначної кількості різних підзадач: під час рекурсивного розв'язування задачі алгоритм багатократно використовує одні і ті ж підзадачі (задачу розбивають на підзадачі, що *перекриваються*).

Для задач, які розв'язують методом декомпозиції, рекурсивний алгоритм на кожному кроці використовує нові підзадачі.

Підзадачі, що *перекриваються*, виникають з *рекурентних* співвідношень, які пов'язують розв'язок початкової задачі із розв'язками підзадач меншого розміру. Метод ДП вимагає, щоб розв'язок кожної підзадачі було визначено тільки один раз і збережено у таблиці (масиві). Якщо підзадача під час рекурсивного розв'язування задачі трапляється знову, то програма бере готовий розв'язок з таблиці.

Отже, у сучасному розумінні поняття динамічного програмування окреслює цілком визначений метод проектування алгоритмів, який не обмежується тільки задачами оптимізації. Природно, цей метод не універсальний, він застосовний до цілком певного класу задач, а саме – до задач, які під час застосування методу декомпозиції породжують порівняно незначну кількість *підзадач* одного типу, що *перекриваються*. Між цими підзадачами не обхідно вивести (визначити) *функціональні залежності* (*рекурентні співвідношення*).

Щодо оптимізаційних задач, які розв'язують методом ДП, додатково необхідне виконання *принципу оптимальності* для підзадач.

Для кращого розуміння суті динамічного програмування, спочатку формалізуємо поняття *задачі* та *підзадачі*. Нехай початкова задача полягає у відшуканні деякої величини Z за вихідними даними $n_1, n_2, ..., n_k$. Тобто тут ідеться про функцію $Z(n_1, n_2, ..., n_k)$ від n_k змінних (параметрів), значення якої є відповіддю задачі.

Тоді підзадачами вважають задачі $Z(i_1, i_2, ..., i_k)$, при $i_1 < n_1$, $i_2 < n_2$, ..., $i_k < n_k$. Далі ми говоритимемо про *однопараметричне*, *двопараметричне* і *багатопараметричне* динамічне програмування, якщо $k = 1$, $k = 2$ і $k > 2$, відповідно.

Визначення чисел Фібоначчі є добрим прикладом застосування методу однопараметричного динамічного програмування. Зазначимо, що задача визначення чисел Фібоначчі не належить до класу задач оптимізації. Простота і прозорість рекурентного співвідношення цієї задачі дають змогу проілюструвати базові підходи до програмування методу ДП.

5.2.2. Реалізація однопараметричного методу ДП

Різні підходи до реалізації однопараметричного методу ДП спочатку продемонструємо на прикладі обчислення чисел Фібоначчі з таким рекурентним співвідношенням:

$$u(i)=\begin{cases}1, & i=\overline{1;2},\\ u(i-2)+u(i-1), & i=\overline{3;n}.\end{cases} \quad (19)$$

Реалізація формули (19) за допомогою рекурсивного процесу *буквально* (один до одного) повторює формулу, якщо "розгортати" її *від кінця до початку* (або *зверху-донизу*):

```
int fib(int n)
{ return (n<=2)? 1 : fib(n-2)+fib(n-1); }
```

Визначення чисел Фібоначчі слугує яскравим прикладом застосування методу декомпозиції (початкову задачу весь час *рекурсивно* ділять на дві підзадачі), унаслідок застосування якого виникають підзадачі, що *перекриваються*.

У цьому випадку рекомендують поєднувати рекурсивний процес з методом динамічного програмування, згідно з яким необхідно, щоб розв'язок кожної підзадачі визначали тільки один раз і зберігали у масиві, з якого потім обирають необхідний розв'язок.

Приклад 1. Скласти програму обчислення чисел Фібоначчі за допомогою рекурсивного процесу та методу ДП.

➢ *Програма*:

```
#include "windows.h"
#include <iostream>
#include<ctime>
using namespace std;
const int N=50;
//---------------------------------------------------------
// Функція fib(n) реалізує метод динамічного програму-
// вання (зверху-донизу) обчислення чисел Фібоначчі:
// F(n)=F(n-2)+F(n-1), де n>=3; F(1)=F(2)=1
//---------------------------------------------------------
```



```
int fib(int n)
{ static int F[N]; // Зберігає числа Фібоначчі
  if (F[N]) return F[n];
  return F[n]=(n<=2)? 1 : fib(n-2) + fib(n-1);
}
//--------------------------------------------------------
int main() {
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
 int n;
 cout<<"Обчислення n-го числа Фібоначчі (0<n<=N)" <<endl;
 cout<<"Введіть n (n<0 - завершення!)"<<endl;
 while(cout<<"n=", cin>>n, n>0 && n<=N) {
   t1=time(0);
   cout<<"Fib("<<n<<")="<<fib(n);
   t2=time(0); cout<<" Час обчислення="<<t2-t1<<endl;
 }
 cin>>n; return 0;
}
```

Результати тестування програми (рекурсивний процес):

```
E:\CPP_2012\Проекти_С++_2012\5_Д...
Обчислення n-го числа  Фібоначчі (0<n<=N
Введіть n (n<0 - завершення!)
n=1
Fib(1)=1 Час обчислення=0
n=5
Fib(5)=5 Час обчислення=0
n=25
Fib(25)=75025 Час обчислення=0
n=40
Fib(40)=102334155 Час обчислення=7
n=45
Fib(45)=1134903170 Час обчислення=81
n=0
```

Поєднання рекурсивного процесу з методом динамічного програмування називають методом *низхідного динамічного програмування* (НДП) або методом ДП з *функцією запам'ятовування*. Метод НДП дає змогу не дублювати обчислення потрібних підзадач чи уникнути обчислення непотрібних підзадач.

Для зберігання обчислених значень послідовності чисел Фібоначчі у функції `fib` використано одновимірний *статичний* масив `F[N]`, який дає змогу зберігати результати обчислень між викликами функції `fib`. Масив `F[N]` ініціалізується *нулями* тільки один раз під час першого виклику `fib`. Якщо `F[i]==0`, то це слугує ознакою того, що `F[i]` ще не обчислено.

Якщо *нуль* є одним зі значень рекурентного співвідношення, то використовувати його (нуль) як ознаку обчислених значень уже не можна. Отже, у цьому випадку у рамках парадигми процедурного програмування замість статичного масиву, зазвичай, використовують *глобальний* масив, який перед викликом рекурсивної функції ініціалізують необхідними початковими значеннями.

Поєднання ітераційного процесу з методом динамічного програмування називають методом *висхідного динамічного програмування* (ВДП). Для методу ВДП типовим є розв'язок *усіх* підзадач початкової задачі.

Приклад 2. Скласти програму обчислення чисел Фібоначчі за допомогою висхідного динамічного програмування.

➢ *Попередні міркування.* У програмі прикладу 1 змінимо тільки реалізацію функції `fib`.

Програма:

```
…
// Ітераційний процес (знизу-доверху)
int fib(int n)
{  static int F[N]; // Зберігає числа Фібоначчі
   F[1]=F[2]=1;
   for (int i=3; i<=n;i++) F[i]=F[i-2]+F[i-1];
  return F[n];
}
…
```


Результати тестування програми (ітераційний процес):

```
E:\CPP_2012\Проекти_C++_2012\5_Динамічне_Пpr...
Обчислення n-го числа  Фібоначчі (0<n<=N)
Введіть n (n<0 - завершення!)
n=1
Fib(1)=1 Час обчислення=0
n=5
Fib(5)=5 Час обчислення=0
n=25
Fib(25)=75025 Час обчислення=0
n=40
Fib(40)=102334155 Час обчислення=0
n=45
Fib(45)=1134903170 Час обчислення=0
n=0
```

◄

Оскільки у прикладі 2 функція `fib` повертає тільки значення `fib(n)`, то можна обійтися і без використання масиву `F[N]`, обмежившись запам'ятовуванням тільки двох послідовних елементів послідовності (19):

```
int fib(int n)
{ int a, b, i, u;
  for (u=a=b=1, i=2; i<n; i++, u=a+b, a=b, b=u);
  return u;
}
```

Якщо порівняти результати прикладів 1 і 2, то можна констатувати, що метод ВДП для достатньо великих значень n працює значно швидше, ніж метод НДП. Однак незалежно від способу реалізації методу ДП головним етапом у процесі розробки відповідних алгоритмів є етап виведення (придумування та обгрунтування) *рекурентного співвідношення*, яке зв'язуватиме розв'язок початкової задачі з розв'язками підзадач, що перекриваються.

Перейдемо до розгляду задач оптимізації, які можна розв'язати за допомогою методу однопараметричного ДП.

Приклад 3 (задача про *банкомат*). В обігу перебувають банкноти різних номіналів a_i $(i=\overline{1;k})$. Банкомат має видати суму *n* за допомогою мінімальної кількості банкнот або повідомити, що запитувану суму видати не можна. Вважатимемо, що запаси банкнот кожного номіналу необмежені. Записати математичну модель цієї задачі та вивести відповідне рекурентне співвідношення.

➢ Нехай x_i – кількість банкнот номіналу a_i $(i=\overline{1;k})$. Задача про банкомат моделюється такою задачею комбінаторної оптимізації:

$$\begin{cases} \sum\limits_{i=1}^{k} x_i \to \min, \\ \sum\limits_{i=1}^{k} a_i x_i = n, \\ 0 \le x_i \le \left\lfloor \dfrac{n}{a_i} \right\rfloor;\ x_i \in Z;\ i=\overline{1,k}, \end{cases} \tag{20}$$

де Z – множина цілих чисел.

Нехай $F(j)$ – *мінімальна* кількість банкнот, якими можна заплатити суму *j* ($j=\overline{0;n}$). Очевидно, що $F(0)=0$. Якщо суму *j* видати неможливо, вважатимемо, що $F(j)=\text{INF}=+\infty$. Виведемо рекурентне співвідношення для обчислення $F(j)$, вважаючи, що значення $F(0)$, $F(1)$, ..., $F(j-1)$ уже обчислено.

Якщо вдасться видати суму $j-a_i$, то потім можна просто додати одну банкноту номіналом a_i $(i=\overline{1;k})$, щоб отримати суму *j*. У цьому випадку мінімальна кількість банкнот, якими можна заплатити суму *j*, дорівнюватиме $F(j-a_i)+1$. З усіляких способів формування $j-a_i$ $(i=\overline{1;k})$ оберемо найкращий, тобто:

$$F(j)=\begin{cases} \min\left\{\text{INF};\ \min\limits_{1\le i\le k}(F(j-a_i)+1)\ \middle|\ j\ge a_i\right\}, & j=\overline{1;n}, \\ 0, & j=0. \end{cases} \tag{21}$$ ⮜

Після обчислення за рекурентним співвідношенням (21) необхідно ідентифікувати номінали банкнот, які беруть участь у формуванні суми *n*. Тепер розглядатимемо *всі* номінали банкнот і значення $n-a_i$ $(i=\overline{1;k})$. Якщо для деякого $m(m=\overline{1;k})$ виявиться, що $F(n-a_m)=F(n)-1$, то це означатиме, що банкнота номіналом a_m бере участь у формуванні суми *n*. Після цього розглядатимемо видачу "*нової*" суми *n* (тобто $n:=n-a_m$), і так продовжуватимемо доти, доки *n* не дорівнюватиме нулю.

Приклад 4. Скласти програму розв'язування задачі прикладу 3 за співвідношенням (21), застосувавши метод ВДП.

➢ *Попередні міркування*. Для збереження значень функції $F(j)$, де $j=\overline{0;n}$, використовуватимемо глобальний масив `F[N]`.

Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
const int N=50;
const int INF=1000000; // Умовна нескінченність
int F[N]; // Глобальний масив кількості банкнот
//----------------------------------------------------------
// Функція count(a[], k, n) реалізує метод динамічного
// програмування (знизу-доверху) видачі мінімуму банк-
// нот для формування суми n (0<n<N) за формулою (21).
//----------------------------------------------------------
int count(int a[], int k, int n) {
 int j, i;  F[0]=0; //  Заповнення елементів F[n+1]
 for(j=1; j<=n; ++j) {  F[j]=INF;
  for(i=0; i<k; ++i)
    if(j>=a[i] && F[j-a[i]]+1<F[j]) F[j]=F[j-a[i]]+1;
 }
 return F[n];
}
//----------------------------------------------------------
void print_Sum(int a[], int k, int n) {
 if(F[n]==INF) cout<<"Не вдається набрати "<<n;
```

```
else
 while(n>0)
  for(int i=0; i<k; ++i)
   if(n>=a[i] && F[n-a[i]]==F[n]-1) {
    cout<<a[i]<<"  ";  n-=a[i]; break; /* Вихід з for*/
   }
 cout<<endl;
}
//-----------------------------------------------------------
int main() { SetConsoleCP(1251); SetConsoleOutputCP(1251);
 int n,k;
 cout<<"Видача мінімуму банкнот для суми n (0<n<N)"<<endl;
 cout<<"Введіть кількість банкнот k (0<k<N): ";
 cin >>k; int* a = new int[k];
 cout<<"Введіть номінали "<<k<<" банкнот (через пропуски):";
 for (int i=0; i<k; ++i) cin>>a[i];
 while(cout<<"n=", cin>>n, n>0 && n<N) {
  cout<<"count("<<n<<")="<<count(a, k, n);
  cout<<"  Номінали: ";  print_Sum(a, k, n);
 }
 delete [] a; cin>>n; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\5_Динамічне_Прграмування\Видач...
Видача мінімуму банкнот для суми n (0<n<N)
Введіть кількість банкнот k (0<k<N): 4
Введіть номінали 4 банкнот (через пропуски):3 5 10 25
n=7
count(7)=1000000  Номінали: Не вдається набрати 7
n=11
count(11)=3  Номінали: 3  3  5
n=27
count(27)=6  Номінали: 3  3  3  3  5  10
n=44
count(44)=5  Номінали: 3  3  3  10  25
n=39
count(39)=5  Номінали: 3  3  3  5  25
n=23
count(23)=3  Номінали: 3  10  10
n=0
```


Приклад 5 (задача про *завантаження транспортного засобу*). Транспортний засіб вантажопідйомністю n одиниць завантажують k різними предметами. Для i-го предмета ($i=\overline{1,k}$) відомі величини: c_i – прибуток від завантаження одного предмета; a_i – вага одного предмета ($a_i < n$). Величини n і a_i ($i=\overline{1,k}$) вимірюють однаковими одиницями ваги. Необхідно так завантажити транспортний засіб цими предметами, щоб отримати *максимальний* прибуток. Записати математичну модель цієї задачі та вивести відповідне рекурентне співвідношення.

➤ Нехай x_i – кількість одиниць i-го предмета a_i $(i=\overline{1;k})$, завантажених на транспортний засіб. Визначення усіх x_i, які забезпечать максимальний сумарний прибуток від їхнього завантаження, приводить до математичної моделі задачі комбінаторної оптимізації:

$$\begin{cases} \sum\limits_{i=1}^{k} c_i x_i \to \max, \\ \sum\limits_{i=1}^{k} a_i x_i \le n, \\ 0 \le x_i \le \left\lfloor \dfrac{n}{a_i} \right\rfloor;\ x_i \in Z;\ i=\overline{1,k}. \end{cases} \qquad (22)$$

Система (22) є математичною моделлю *статичної* задачі ухвалення рішень. Однак не заборонено вважати, що завантаження різних предметів здійснюють *послідовно* за зростанням їхніх номерів. У цьому випадку отримуємо *багатокрокову* задачу ухвалення рішень (22) з *адитивною* цільовою функцією, для розв'язування якої можна застосувати метод ДП.

Нехай $F(j)$ – *максимальний* прибуток, отриманий від завантаження предметів сумарною вагою j ($j=0;n$). Очевидно, що $F(0)=0$. Якщо вагу j отримати неможливо, вважатимемо, що $F(j)=\text{INF}=-\infty$. Виведемо рекурентне співвідношення для обчислення $F(j)$, вважаючи, що значення $F(0)$, $F(1)$, …, $F(j-1)$ уже обчислені.

Якщо вдасться завантажити предмети сумарною вагою $j-a_i$, то потім можна просто додати предмет вагою a_i $(i=\overline{1;k})$, щоб отримати сумарну вагу *j*. У цьому випадку максимальний прибуток, який можна отримати від завантаження предметів сумарною вагою *j*, становитиме $F(j-a_i)+c_i$. З усіляких способів формування $j-a_i$ $(i=\overline{1;k})$ оберемо найкращий, тобто:

$$F(j)=\begin{cases}\max\left\{\text{INF}; \max\limits_{1\le i\le k}(F(j-a_i)+c_i)\mid j\ge a_i; F(j-a_i)\ge 0\right\}, & j=\overline{1;n},\\ 0, & j=0.\end{cases} \quad (23)$$

Після обчислення за рекурентним співвідношенням (23) необхідно вивести на консоль ваги та прибутки предметів, які завантажено на транспортний засіб, а також вивести сумарну вагу та сумарний прибуток. Тепер розглядатимемо *всі* предмети і значення $n-a_i$ $(i=\overline{1;k})$. Якщо для деякого $m(m=\overline{1;k})$ виявиться, що $F(n-a_m)=F(n)-c_m$, то це означатиме, що предмет вагою a_m і прибутком c_m завантажено на транспортний засіб. Після цього розглядатимемо формування "*нової*" ваги *n* (тобто $n:=n-a_m$), і так продовжуватимемо доти, доки *n* не дорівнюватиме нулю.

Приклад 6. Скласти програму розв'язування задачі прикладу 5 за співвідношенням (23), застосувавши метод ВДП.

➢ *Попередні міркування*. Для збереження значень функції $F(j)$, де $j=\overline{0;n}$, використовуватимемо глобальний масив `F[N]`.

Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
const int N=50;
const int INF=-1000000; // Умовна нескінченність
int F[N]; // Глобальний масив сумарних вартостей
//---------------------------------------------------------
```



```
// Функція cost(a[], c[], k, n) реалізує метод динамічного
// програмування (знизу-доверху) максимальної
// сумарної вартості завантажених предметів на транспорт-
// ний засіб (ТЗ) вагою n (0<n<N) за формулою (23).
// Масив a[k] містить ваги k предметів.
// Масив c[k] містить вартості k предметів.
// Якщо неможливо сформувати n, то повертається INF.
//---------------------------------------------------------
void cost(int a[], int c[], int k, int n) {
 int j, i;  F[0]=0; // Заповнення елементів F[n+1]
 for(j=1; j<=n; ++j) {
  F[j]=INF;
  for(i=0; i<k; ++i)
   if(j>=a[i] && F[j-a[i]]>=0 && F[j-a[i]]+c[i]>F[j])
       F[j]=F[j-a[i]]+c[i];
 }
}
//---------------------------------------------------------
void print_WC(int a[], int c[], int k, int n) { int j=n;
 while (F[j]==INF) --j; //Пошук ваги завантажених предметів
 int v=0, p=0;
 while(j>0)
  for(int i=0; i<k; ++i)
   if(j>=a[i] && F[j-a[i]]>=0 && F[j-a[i]]==F[j]-c[i]) {
     v+=a[i], p+=c[i];
     cout<<"Вага= "<<a[i]<<"; вартість= "<<c[i]<<endl;
     j-=a[i]; break; /* Вихід з for*/
   }
   cout<<"Сумарна вага= "<<v;
   cout<<"; сумарна вартість= "<<p<<endl;
}
//---------------------------------------------------------
int main() {
 SetConsoleCP(1251); SetConsoleOutputCP(1251); int n,k;
 cout<<"Завантаження ТЗ вагою n (0<n<N)" <<endl;

 cout<<"Введіть кількість предметів k (0<k<N): ";
```

```
 cin >>k; int* a=new int[k]; int* c=new int[k];
 cout<<"Введіть ваги "<<k<<" предметів (через пропуски):";
 for (int i=0; i<k; ++i) cin>>a[i];
 cout<<"Введіть прибуток "<<k;
 cout <<" предметів (через пропуски):";
 for (int i=0; i<k; ++i) cin>>c[i];
 while(cout<<"n=", cin>>n, n>0 && n<N) {
  cost(a, c, k, n);
  print_WC(a, c, k,  n);
 }
 delete [] a; delete [] c;
 cin>>n; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\5_Динамічне_Прграмування\За...
Завантаження ТЗ вагою n (0<n<N)
Введіть кількість предметів k (0<k<N): 3
Введіть ваги 3 предметів (через пропуски):7 9 12
Введіть прибуток 3 предметів (через пропуски):3 4 5
n=30
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 9; вартість= 4
Сумарна вага= 30; сумарна вартість= 13
n=32
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 9; вартість= 4
Вага= 9; вартість= 4
Сумарна вага= 32; сумарна вартість= 14
n=20
Вага= 7; вартість= 3
Вага= 12; вартість= 5
Сумарна вага= 19; сумарна вартість= 8
n=0
```


Приклад 7. Скласти програму розв'язування задачі прикладу 5 за співвідношенням (23), застосувавши метод НДП.

➢ *Попередні міркування*. Для збереження значень функції $F(j)$, де $j=\overline{0;n}$, використовуватимемо глобальний масив `F[N]`. Під час обчислення цієї функції за умовою задачі необхідно застосовувати метод *низхідного* динамічного програмування, який передбачає організацію рекурсивного процесу. З метою економії часу та пам'яті масиви, в яких зберігатимуться значення ваг та прибутків предметів, також будуть глобальними.

Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
const int N=100;
const int K=10;
const int INF=-1000000; // Умовна нескінченність
int F[N]; // Глобальний масив сумарних вартостей
int a[K]; // Глобальний масив ваг предметів
int c[K]; // Глобальний масив прибутків предметів
//-------------------------------------------------------------
// Функція cost(k, n) реалізує метод динамічного
// програмування (зверху-донизу) визначення макси-
// мальної сумарної вартості завантажених предметів
// на транспортний засіб (ТЗ) вантажопідйомності
// n (0<n<N) за формулою (23).
// Параметр k - кількість предметів (0<k<=K).
//-------------------------------------------------------------
int cost(int k, int n) {
 int i, new_a, new_c, max_c;
 // Складна база рекурсії:
 // 1) n==0 - початкове значення F[0]=0;
 // 2) F[n]>0 - значення F[n] обчислено раніше;
 // 3) F[n]==INF - вагу n не можна накопичити (це зна-
 //    чення також обчислено раніше).
 // Отже, при F[n]==0 - необхідні рекурсивні обчислення.
 if(!n || F[n]>0 || F[n]==INF) return F[n];
```

```
// Організація рекурсивних обчислень
 max_c=INF;
 for(i=0; i<k; ++i) {
   new_a=n-a[i];
   if(new_a>=0) {
     new_c=cost(k, new_a);
     if(new_c>=0 && new_c +c[i]>max_c) max_c=new_c+c[i];
   }
 }
 F[n]=max_c; return F[n];
}
//-----------------------------------------------------------
// Формування і виведення оптимального розв'язку
void print_WC(int k, int n) { int j=n;
 while (F[j]==INF) --j; // Пошук ваги завантажених предметів
 int v=0, p=0; // Накопичення ваг і прибутків предметів
 while(j>0)
  for(int i=0; i<k; ++i)
   if(j>=a[i] && F[j-a[i]]>=0 && F[j-a[i]]==F[j]-c[i]) {
     v+=a[i], p+=c[i];
     cout<<"Вага= "<<a[i]<<"; вартість= "<<c[i]<<endl;
     j-=a[i]; break; /* Вихід з for*/
   }
 cout<<"Сумарна вага = "<<v;
 cout<<"; сумарна вартість = "<<p<<endl;
}
//-----------------------------------------------------------
int main() {
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
 int n,k,m;
 cout<<"Використання низхідного методу ДП!" <<endl;
 cout<<"Завантаження ТЗ вагою n (0<n<" <<N<<")"<<endl;
 cout<<"Введіть кількість предметів k (0<k<="<<K<<"): ";
 cin >>k; cout<<"Введіть ваги " << k;
 cout <<" предметів (через пропуски):";
 for (int i=0; i<k; ++i) cin>>a[i];
 cout<<"Введіть прибуток "<< k;
 cout <<" предметів (через пропуски):";
 for (int i=0; i<k; ++i) cin>>c[i];
```



```
 while(cout<<"n=", cin>>n, n>0 && n<N) { m=n;
 while(cost(k,m)==INF) m--; // Пошук ваги предметів
 print_WC(k, n);
 }
 cin>>n; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\5_Динамічне_Прграма...
Завантаження ТЗ вагою n (0<n<100)
Введіть кількість предметів k (0<k<=10): 3
Введіть ваги 3 предметів (через пропуски):7 9 12
Введіть прибуток 3 предметів (через пропуски):3 4 5
n=30
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 9; вартість= 4
Сумарна вага= 30; сумарна вартість= 13
n=20
Вага= 7; вартість= 3
Вага= 12; вартість= 5
Сумарна вага= 19; сумарна вартість= 8
n=47
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 7; вартість= 3
Вага= 12; вартість= 5
Сумарна вага= 47; сумарна вартість= 20
n=6
Сумарна вага= 0; сумарна вартість= 0
n=13
Вага= 12; вартість= 5
Сумарна вага= 12; сумарна вартість= 5
```

5.2.3. Реалізація двопараметричного методу ДП

Під час реалізації *однопараметричного* методу ДП для зберігання значень функції мети ми використовували одновимірний масив (вектор). Зрозуміло, що під час реалізації *двопараметричного* методу ДП використовуватимемо двовимірний масив (матрицю).

Різні підходи до реалізації двопараметричного методу ДП спочатку продемонструємо на прикладі обчислення кількості комбінацій C_n^i, де $0 \le i \le n$. Відомо, що $C_n^0 = C_n^n = 1$. Обчислення C_n^i, згідно з рекурентним співвідношення (3.37), реалізовано за допомогою рекурсивного та ітераційного процесів (для порівняння) у прикладі 3.29. Однак для методу ДП доцільно застосувати рекурентне співвідношення (3.35).

Приклад 7. Скласти програму обчислення кількості комбінацій C_n^i за співвідношенням (3.35), застосувавши метод ВДП.

➢ *Попередні міркування*. Для збереження значень функції C_n^i використовуватимемо двовимірний *динамічний* масив C.

Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
//-----------------------------------------------------------
int comb(int n, int i) {
 if(i>n || n<0 || i<0) return -1; // Помилки даних
 int **C, j, k;
 C = new int* [n+1]; // Формування динамічної матриці
 for (j=0; j<=n; j++) C[j] = new int[n+1];
 for (k=0; k<=n; k++) {
  C[k][0]=1; C[k][k]=1;
  for (j=1; j<k; j++)
    C[k][j]=C[k-1][j-1]+C[k-1][j];
 }
 k=C[n][i];
 for(j=0; j<=n; j++) delete [] C[j]; delete [] C;
 return k;}
```



```
//-----------------------------------------------------------
int main() {
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
 int i, n;
 cout<<"Комбінації з n по i (0<=i<=n)"<<endl;
 cout<<"Введіть n та i (n<0 - завершення!)"<<endl;
 while(cout<<"n=", cin>>n, cout<<"i=",cin>>i,
      i>=0 && n>=0 && i<=n)
  cout<<"comb("<<n<<","<<i<<")="<<comb(n, i)<<endl;
 cin>>n; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_C++_2012\5_Динамічне_Програ...
Комбінації з n по i (0<=i<=n)
Введіть n та i (n<0 - завершення!)
n=7
i=4
comb(7,4)=35
n=28
i=7
comb(28,7)=1184040
n=10025
i=125
comb(10025,125)=-1189447872
n=100
i=20
comb(100,20)=-474913254
n=100
i=50
comb(100,50)=-938977944
n=50
i=4
comb(50,4)=230300
n=-4
i=2
```

Додатковий коментар. Від'ємні значення, які проявилися під час тестування, є ознакою того, що відбулося переповнення розрядної сітки (помилка обчислень). ⊲

Для кращого розуміння роботи цього алгоритму виведемо на консоль значення допоміжного масиву C:

```
int comb(int n, int i) {
 if(i>n || n<0 || i<0) return -1; // Помилки даних
 int **C, j, k; C = new int* [n+1];
 for (j=0; j<=n; j++) C[j] = new int[n+1];
 for (k=0; k<=n; k++) {
  C[k][0]=1; C[k][k]=1;
  for (j=1; j<k; j++) C[k][j]=C[k-1][j-1]+C[k-1][j]; }
 cout<<"Тестування допоміжного масиву:"<<endl;
 for (k=0; k<=n; k++) {
  for (j=0; j<=k; j++) cout<<C[k][j]<<" "; cout<<endl;}
 k=C[n][i]; for (j=0; j<=n; j++) delete [] C[j];
 delete [] C;  return k;
}
```

Результати тестування програми з виведенням значень C:

```
E:\CPP_2012\Проекти_С++_2012\5_Динамічне_Пр...
Комбінації з n по i (0<=i<=n)
Введіть n та i (n<0 - завершення!)
n=6
i=3
Тестування допоміжного масиву:
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
comb(6,3)=20
n=
```


Аналізуючи результати тестування програми з виведенням значень допоміжного масиву C, можна відзначити, що наведений алгоритм має певні недоліки. Ми обчислюємо значення усіх елементів останнього рядка, хоча нам необхідний тільки один із них – C_6^3. Аналогічно, у попередньому рядку нам необхідно мати тільки два елементи – C_5^3 та C_5^2 і т. д. Окрім цього, ми використовуємо тільки половину елементів C.

Оскільки для обчислення кожного рядка масиву C ми використовуємо тільки елементи попереднього рядка, то можна обійтися одновимірним допоміжним масивом C.

Програма, яка використовує одновимірний масив C:

```
#include "windows.h"
#include <iostream>
using namespace std;
//---------------------------------------------------
int comb(int n, int i) {
 if(i>n || n<0 || i<0) return -1; // Помилки даних
 int *C, j, k, a, b;  C=new int [n+1];
 for (k=0; k<=n; k++) C[k]=0;
 for (k=0; k<=n; k++) { a = C[0]=C[k]=1;
  for (j=1; j<k; j++) { b = C[j]; C[j] = a+b; a = b; }
  for (j=0; j<=n; j++) cout<<C[j]<<" "; cout<<endl;
 }
 k = C[i]; delete [] C; return k;
}
//---------------------------------------------------
int main() {
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
 inti, n;
 cout<<"Комбінації з n по i (0<=i<=n)"<<endl;
 cout<<"Введіть n та i (n<0 - завершення!)"<<endl;
 while(cout<<"n=", cin>>n, cout<<"i=",cin>>i,
     i>=0 && n>=0 && i<=n)
  cout<<"comb("<<n<<","<<i<<")="<<comb(n, i)<<endl;
 cin>>n; return 0;
}
```

Результати тестування програми з виведенням значень допоміжного одновимірного масиву `C`:

```
E:\CPP_2012\Проекти_C++_2012\5_Динамічне_Прг...
Комбінації з n по i (0<=i<=n)
Введіть n та i (n<0 - завершення!)
n=5
i=3
1 0 0 0 0 0
1 1 0 0 0 0
1 2 1 0 0 0
1 3 3 1 0 0
1 4 6 4 1 0
1 5 10 10 5 1
comb(5,3)=10
n=6
i=5
1 0 0 0 0 0 0
1 1 0 0 0 0 0
1 2 1 0 0 0 0
1 3 3 1 0 0 0
1 4 6 4 1 0 0
1 5 10 10 5 1 0
1 6 15 20 15 6 1
comb(6,5)=6
n=
```

Програма обчислення кількості комбінацій C_n^l є прикладом застосування методу динамічного програмування до задачі, яка не є оптимізаційною. Далі здебільшого розглядатимемо задачі оптимізації.

Приклад 8 (задача про *максимальний наскрізний маршрут*). На площині маємо ділянку розміром $n \times m$ (n рядків і m стовпців). Кожній клітинці ділянки $(i;\ j)$, де $i = \overline{0; n-1}$; $j = \overline{0; m-1}$ приписана вага $w(i;\ j)$ – деяке невід'ємне ціле число ($w(i;\ j) \geq 0$). У лівому верхньому куті ділянки (клітинці $(0;\ 0)$) стоїть робот, який може

переміщатися на одну клітинку вниз або на одну клітинку вправо. Скласти рекурентне співвідношення, яке даватиме змогу обчислити найбільшу сумарну вагу клітинок маршруту переміщення робота від клітинки $(0;\ 0)$ до клітинки $(n-1;\ m-1)$. Іншими словами, необхідно сформувати максимальний наскрізний маршрут.

➤ Нехай $s(i;\ j)$ – *максимальна* вага, яку може назбирати робот під час свого переміщення від клітинки $(0;\ 0)$ до клітинки $(i;\ j)$. Оскільки для довільної клітинки верхнього рядка ($i = 0$) та лівого стовпця ($j = 0$) існує єдиний можливий шлях від клітинки $(0;\ 0)$, то для них максимальну вагу $s(i;\ j)$ обчислити дуже просто:

$$s(0; j) = s(0; j-1) + w(0; j-1);\ j = \overline{1; m},$$
$$s(i; 0) = s(i-1; 0) + w(i-1; 0);\ i = \overline{1; n}, \qquad (24)$$

де $s(0;\ 0) = w(0;\ 0)$.

Співвідношення (24) є межовими умовами для функції $s(i;\ j)$.

Побудуємо рекурентне співвідношення для випадків, коли $i > 0$ та $j > 0$ (тобто клітинка $(i;\ j)$ не розміщена у верхньому рядку чи у лівому стовпці). Тоді у клітинку $(i;\ j)$ робот може прийти або зліва (з клітинки $(i;\ j-1)$), або зверху (з клітинки $(i-1;\ j)$). З двох можливих значень сумарної ваги обираємо найбільше значення:

$$s(i; j) = \max\{s(i; j-1); s(i-1; j)\} + w(i; j);\ i = \overline{1; n};\ j = \overline{1; m}. \qquad (25)$$

≺

Після обчислення за рекурентними співвідношеннями (24), (25) необхідно вивести на консоль координат клітинок, які належатимуть максимальному наскрізному маршруту робота. Існує два варіанти розв'язання цієї проблеми: 1) з використанням спеціального допоміжного масиву предків клітинок; 2) без використання допоміжного масиву.

Розглянемо спочатку *перший варіант*. Нехай $p(i;\ j)$ містить позначку клітинки-попередника (`const L=1` – клітинка зліва; `const U=2` – клітинка зверху) для клітинки $(i;\ j)$. Очевидно, що $p(0;\ 0) = 0$ (перша клітинка маршруту немає попередників). У

решті клітинок першого рядка запишемо L, а першого стовпця – U. Решту клітинок масиву p заповнюватимемо одночасно зі заповненням масиву s:

```
if(s[i-1][j]>s[i][j-1])
  { // У клітинку (i; j) робот приходить зверху
    s[i][j]=s[i-1][j]+w[i][j]; p[i][j]=U;}
 else
  { // У клітинку (i; j) робот приходить зліва
    s[i][j]=s[i][j-1]+w[i][j]; p[i][j]=L; }
```

Після цілковитого заповнення обох масивів можна відновити маршрут, проходячи по ньому у зворотному порядку від кінцевої клітинки $(n-1; m-1)$ до початкової клітинки $(0; 0)$. Для цього достатньо завести допоміжний масив ar:

```
  int ar[K][2]; // K – верхня межа для n + m (n + m < K)
  i=n-1; j=m-1;
  ar[0][0]=i; ar[0][1]=j; k=1; // Остання клітинка
  while(p[i][j]) { // Додавання до маршруту клітинок
                   // від його закінчення до початку
    p[i][j]==L? j-- : i--;
    ar[k][0]=i; ar[k][1]=j; k++;
  } // while(p[i][j] …
```

Тепер, щоб отримати маршрут у прямому напрямі від початкової клітинки $(0; 0)$ до кінцевої клітинки $(n-1; m-1)$ необхідно переглянути масив ar у зворотному порядку:

```
 for(--k; k>0; k--) { // Виведення маршруту від його по-
                  // чаткової клітинки до передостанньої
   cout<<"("<<ar[k][0]<<",";
   cout<<ar[k][1]<<")->";
 }
   cout<<"("<<ar[k][0]<<",";      // Виведення останньої
   cout<<ar[k][1]<<")"<<endl;  // клітинки маршруту
```


Приклад 9. Скласти програму реалізації алгоритму визначення наскрізного маршруту найбільшої вартості на прямокутній ділянці $n \times m$ методом висхідного динамічного програмування з використанням масиву *предків клітинок* (перший варіант) згідно з рекурентними співвідношеннями (24), (25).

➢ *Програма:*

```
#include "windows.h"
#include <iostream>
using namespace std;
//-----------------------------------------------------------
//  Реалізація алгоритму (на базі методу динамічного
//  програмування) визначення наскрізного маршруту най-
//  більшої ваги на прямокутному полі n на m від
//  клітинки (0; 0) до клітинки (n-1; m-1)
//-----------------------------------------------------------
int main() {
 const int N = 10, L=1, U=2;
 const int K = 2*N;
 int n,     // Кількість рядків
     m,     // Кількість стовпців
     w[N][N], // Матриця ваг
     s[N][N], // Матриця максимальних сумарних ваг
     p[N][N]; // Матриця  предків клітинок
 int i, j, k;
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------- Ініціалізація w  ----------------------------
 cout << "Вводьте кількість рядків (n) ";
 cout << "і стовпців (m)."<<endl;
 cout<<"Якщо (n<=0||n>10||m<=0||m>10)";
 cout<<" - завершення тестувань."<<endl;
 while(cout<<"n, m:", cin>>n>>m,
       n>0 && m>0 && n<=10 && m<=10)
 {
  cout<<"Введення невід'ємних ваг у матрицю w."<<endl;
  for (i=0; i<n; i++) {
    cout<<"Для  "<<i<<"-го рядка: ";
    for (j = 0; j<m; j++) cin>>w[i][j];
  }
```

```
//------ Ініціалізація перших рядків s і p ------------
  s[0][0]=w[0][0]; p[0][0]=0;
  for (j=1; j<m; j++)
   { s[0][j]=s[0][j-1]+w[0][j]; p[0][j]=L; }
//-------- Базовий цикл ------------------------------
  for (i=1; i<n; i++) {
   s[i][0]=s[i-1][0]+w[i][0]; p[i][0]=U; // 0 стовпчик
   for (j=1; j<m; j++)
     if(s[i-1][j]>s[i][j-1])
       {s[i][j]=s[i-1][j]+w[i][j]; p[i][j]=U;}
     else
       {s[i][j]=s[i][j-1]+w[i][j]; p[i][j]=L; }
  }
  //------ Результати роботи алгоритму  ----------
  cout<<" Наскрізний маршрут максимальної ваги ";
  cout<<s[n-1][m-1]<<endl;
  int ar[K][2]; i=n-1; j=m-1;
  ar[0][0]=i; ar[0][1]=j; k=1; // Остання клітинка
  while(p[i][j]) { // Додавання до маршруту клітинок
                   // від його закінчення до початку
    p[i][j]==L? j-- : i--;
    ar[k][0]=i; ar[k][1]=j; k++;
  } // while(p[i][j] …
  for(--k; k>0; k--) { // Виведення маршруту від його по-
                    // чаткової клітинки до передостанньої
    cout<<"("<<ar[k][0]<<",";
    cout<<ar[k][1]<<")->";
  }
    cout<<"("<<ar[k][0]<<",";     // Виведення останньої
    cout<<ar[k][1]<<")"<<endl;    // клітинки маршруту
 } // while(cout<<"n, m:", …
 cin>>k;
return 0;
}
```


Результати тестування першого варіанта:

```
E:\CPP_2012\Проекти_C++_2012\5_Динамічне_Пргграмування\5_2_Наскріз...
Вводьте кількість рядків (n) і стовпців (m).
Якщо (n<=0||n>10||m<=0||m>10) - завершення тестувань.
n, m:3 3
Введення невід'ємних ваг у матрицю w.
Для  0-го рядка: 2 1 4
Для  1-го рядка: 2 0 3
Для  2-го рядка: 1 3 5
 Наскрізний маршрут максимальної ваги 15
(0,0)->(0,1)->(0,2)->(1,2)->(2,2)
n, m:4 5
Введення невід'ємних ваг у матрицю w.
Для  0-го рядка: 2 4 6 1 4
Для  1-го рядка: 0 2 4 5 2
Для  2-го рядка: 2 5 4 4 3
Для  3-го рядка: 3 2 0 3 5
 Наскрізний маршрут максимальної ваги 33
(0,0)->(0,1)->(0,2)->(1,2)->(1,3)->(2,3)->(3,3)->(3,4)
n, m:-2 0
```

Тепер розглянемо *другий варіант* (без використання допоміжного масиву предків клітинок) виведення на консоль координат клітинок, які належатимуть максимальному наскрізному маршруту робота.

Для цілковито заповненого масиву s попередника кожної клітинки $(i;\ j)$ максимального наскрізного маршруту робота можна визначити, порівнюючи значення $s(i;\ j)$ зі значенням елементів масиву s для її лівого (клітинка $(i;\ j-1)$) та верхнього сусідів (клітинка $(i-1;\ j)$). З цих клітинок-сусідів обираємо клітинку з найбільшим значенням s:

```
int ar[K][2]; i=n-1; j=m-1;
ar[0][0]=i; ar[0][1]=j; k=1; // Остання клітинка
while(i>0 && j>0) { // Додавання до маршруту клітинок
                    // від його закінчення до однієї з меж
s[i][j-1]>s[i-1][j]? j-- : i--;
ar[k][0]=i; ar[k][1]=j; k++; } // while
```

Цей алгоритм завершиться, якщо ми вийдемо на верхню чи ліву межу ділянки. Після цього необхідно буде рухатися клітинками цієї межі доти, доки не вийдемо на клітинку (0; 0) :

```
 while(i>0) {      // Додавання до маршруту клітинок
                   // лівої межі
   i--;  ar[k][0]=i; ar[k][1]=j; k++;
 } // while(i>0) …
 while(j>0) {      // Додавання до маршруту клітинок
                   // верхньої межі
  j--;   ar[k][0]=i; ar[k][1]=j; k++;
 } // while(j>0) …
```

Тепер, щоб отримати маршрут у прямому напрямі від початкової клітинки (0; 0) до кінцевої клітинки $(n-1;\, m-1)$ необхідно переглянути масив ar у зворотному порядку.

Приклад 10. Скласти програму реалізації алгоритму визначення наскрізного маршруту найбільшої вартості на прямокутній ділянці $n \times m$ методом висхідного динамічного програмування *без* використання масиву предків клітинок (другий варіант) згідно з рекурентними співвідношеннями (24), (25).

➤ *Програма*:

```
#include "windows.h"
#include <iostream>
using namespace std;
//----------------------------------------------------------
// Реалізація алгоритму (на базі методу ДП) визначення
// наскрізного маршруту найбільшої ваги на прямокутній ді-
// лянці n на m від клітинки (0; 0) до клітинки (n-1; m-1)
//----------------------------------------------------------
int main() {
 const int N = 10; const int K = 2*N;
 int n,     // Кількість рядків
     m,     // Кількість стовпців
     w[N][N], // Матриця ваг
     s[N][N]; // Матриця максимальних сумарних ваг
     int i, j, k;
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
```



```
//------ Ініціалізація w ----------------------------------
 cout<<"Вводьте кількість рядків (n) ";
 cout<<"і стовпців (m)."<<endl;
 cout<<"Якщо (n<=0||n>10||m<=0||m>10)";
 cout<<" - завершення тестувань."<<endl;
 while(cout<<"n, m:", cin>>n>>m,
    n>0 && m>0 && n<=10 && m<=10) {
  cout<<"Введення невід'ємних ваг у матрицю w."<<endl;
  for (i=0; i<n; i++) {
    cout<<"Для  "<<i<<"-го рядка: ";
    for (j = 0; j<m; j++) cin>>w[i][j];
 }
//----Ініціалізація першого рядка s -----------------------
  s[0][0]=w[0][0];
  for (j=1; j<m; j++) s[0][j]=s[0][j-1]+w[0][j];

//--------- Базовий цикл ----------------------------------
  for (i=1; i<n; i++) {
   s[i][0]=s[i-1][0]+w[i][0];
   for (j=1; j<m; j++)
     if(s[i-1][j]>s[i][j-1])
       s[i][j]=s[i-1][j]+w[i][j];
     else
       s[i][j]=s[i][j-1]+w[i][j];
  }
//------ Результати роботи алгоритму  ----------
  cout<<" Наскрізний маршрут максимальної ваги ";
  cout<<s[n-1][m-1]<<endl;
  int ar[K][2]; i=n-1; j=m-1;
  ar[0][0]=i; ar[0][1]=j; k=1; // Остання клітинка
  while(i>0 && j>0) { // Додавання до маршруту клітинок
                 // від його закінчення до однієї з меж
   s[i][j-1]>s[i-1][j]? j-- : i--;
   ar[k][0]=i; ar[k][1]=j; k++;
  }
  while(i>0) {     // Додавання до маршруту клітинок
                   // лівої межі
    i--;  ar[k][0]=i; ar[k][1]=j; k++;
  } // while(i>0) …
  while(j>0) {     // Додавання до маршруту клітинок
```

```
                    // верхньої межі
  j--;  ar[k][0]=i;
  ar[k][1]=j; k++;
 } // while(j>0) …
 for(--k; k>0; k--) { // Виведення маршруту від його по-
                 // чаткової клітинки до передостанньої
   cout<<"("<<ar[k][0]<<",";
   cout<<ar[k][1]<<")->";
 }
 cout<<"("<<ar[k][0]<<",";    // Виведення останньої
 cout<<ar[k][1]<<")"<<endl;   // клітинки маршруту
} // while(cout<<"n …
cin>>k; return 0;
}
```

Результати тестування другого варіанта:

```
E:\CPP_2012\Проекти_С++_2012\5_Динамічне_Прграмування\5_2_H...
Вводьте кількість рядків (n) і стовпців (m).
Якщо (n<=0||n>10||m<=0||m>10) - завершення тестувань.
n, m:4 5
Введення невід'ємних ваг у матрицю w.
Для  0-го рядка: 1 3 5 0 3
Для  1-го рядка: 0 1 2 4 1
Для  2-го рядка: 2 4 3 3 2
Для  3-го рядка: 3 1 2 4 3
 Наскрізний маршрут максимальної ваги 25
(0,0)->(0,1)->(0,2)->(1,2)->(1,3)->(2,3)->(3,3)->(3,4)
n, m: 4 4
Введення невід'ємних ваг у матрицю w.
Для  0-го рядка: 1 5 0 3
Для  1-го рядка: 2 2 1 6
Для  2-го рядка: 4 2 3 1
Для  3-го рядка: 2 3 1 4
 Наскрізний маршрут максимальної ваги 20
(0,0)->(0,1)->(0,2)->(0,3)->(1,3)->(2,3)->(3,3)
n, m:-4 0
```


5.3. Метод динамічного програмування розв'язування задач про наплічник

У прикладах 5 – 7 цього розділу поставлено та розв'язано задачу про завантаження *транспортного засобу* (ТЗ) вантажопідйомності n одиниць k різними предметами. Для i-го предмета ($i = \overline{1,k}$) відомі: c_i – прибуток від завантаження одного предмета; a_i – вага одного предмета ($a_i < n$). Величини n і a_i ($i = \overline{1,k}$) вимірюють однаковими одиницями ваги. Необхідно було так завантажити ТЗ цими предметами, щоб отримати *максимальний* прибуток.

Математичну модель цієї задачі відображає формула (22), а відповідне рекурентне співвідношення – формула (23) (див. приклад 5). Розв'язування задачі *комбінаторної оптимізації* (22) у прикладі 6 реалізовано методом висхідного ДП, а у прикладі 7 – низхідного ДП.

Задача про завантаження транспортного засобу є своєрідним переформулюванням класичної задачі комбінаторної оптимізації про *обмежений наплічник* (*Bounded Knapsack Problem* – англ.). У назві задачі відображено проблему укладання у наплічник предметів, які забезпечують найбільший сумарний прибуток. Відомо, що загальна вага усіх розміщених предметів ("*вантажопідйомність*") обмежена деякою величиною. Предмети у наплічнику можуть повторюватися.

Задача про наплічник має багато різновидів. Серед них найвідомішою є задача комбінаторної оптимізації про *0-1 наплічник* (*0-1 Knapsack Problem* – англ.), в якій у наплічник необхідно укласти найцінніші предмети, що не можуть повторюватися. Окремі предмети можуть не входити у наплічник взагалі. Математична модель цієї задачі має такий вигляд:

$$\begin{cases} \sum_{i=1}^{k} c_i x_i \to \max, \\ \sum_{i=1}^{k} a_i x_i \le n, \\ x_i \in \{0;1\};\ i = \overline{1,k}. \end{cases} \qquad (26)$$

Відомо чимало еквівалентних формулювань задач про наплічник. Наприклад, під час формулювання задачі оптимізації про 0-1 наплічник можна замість наплічника розглядати космічний апарат – супутник Землі, а предметами вважати наукові прилади. Тоді задача інтерпретується як відбір приладів для запуску на орбіту.

Для задачі комбінаторної оптимізації про *обмежений* наплічник дещо актуальнішими є інтерпретації щодо практичного застосування в логістиці, економіці підприємства, організації виробництва тощо. Ми уже згадували у цьому контексті задачу завантаження ТЗ. Також "*предметами*" можна вважати окремі замовлення (або варіанти випуску партій тих чи інших товарів), а "вагою" – собівартість замовлення.

Відомі й інші види задач про наплічник (необмежений, з мультивибором, мультиплікативний, багатовимірний тощо), які ми залишимо за межами нашого розгляду.

Окрім *класичних* постановок задачі про наплічник, в яких функцію мети *максимізують*, важливе практичне значення мають постановки задачі з *мінімізацією* функції мети. У цьому випадку математична модель задачі про *обмежений* наплічник матиме такий вигляд:

$$\begin{cases} \sum_{i=1}^{k} c_i x_i \to \min, \\ \sum_{i=1}^{k} a_i x_i \le n, \\ 0 \le x_i \le \left\lfloor \frac{n}{a_i} \right\rfloor; \; x_i \in Z; \; i = \overline{1, k}. \end{cases} \qquad (27)$$

Частковим випадком задачі (27) є задача про банкомат (див. приклад 3), в якій $c_i = 1$; $a_i \; (i = \overline{1; k})$ – номінали банкнот, n – сума грошей, що їх бажає отримати клієнт. У прикладі 3 банкомат мав видати суму n за допомогою мінімальної кількості банкнот або повідомити, що запитувану суму видати не можна. У постановці (27) маємо ширше трактування: видати клієнтові максимально можливу суму, що не перевищує n, мінімальною кількістю банкнот.

Приклад 11. Скласти рекурентне співвідношення розв'язування задачі комбінаторної оптимізації про 0-1 наплічник.

➤ Нехай $s(i; j)$ – *максимальна* вартість наплічника "*вантажопідйомності*" $j \; (j = \overline{1; n})$, в якому розміщено найцінніші предмети з перших i предметів $(i = \overline{1; k})$. Множину перших i предметів можна розбити на дві підмножини: ту, яка містить i-й предмет, і ту, яка його не містить. Можна встановити таке:

- максимальна вартість наплічника без i-го предмета дорівнюватиме $s(i-1; j)$;
- максимальна вартість наплічника з i-м предметом (у цьому випадку $j - a_i \ge 0$) дорівнюватиме $s(i-1; j - a_i) + c_i$.

Отже, під час розміщення i-го предмета передусім необхідно визначити, чи його можна у принципі розмістити у наплічнику, а потім порівнювати *максимальні* вартості наплічника за наявності чи відсутності цього предмета.

Опираючись на ці міркування, запишемо рекурентне співвідношення визначення максимальної вартості наплічника:

$$s(i; j) = \begin{cases} \max\{s(i-1; j); s(i-1; j-a_i)\} + c_i; & j - a_i \ge 0, \\ s(i-1; j); & j - a_i < 0, \end{cases} \qquad (28)$$

де $i = \overline{1; k}; \; j = \overline{1; n}$.

Початкові умови формулюються дуже просто:

$$s(0; j) = 0; \; j = \overline{1; n}, \; s(i; 0) = 0; \; i = \overline{1; k}, \; s(0; 0) = 0. \qquad (29)$$

➤

Для цілковито заповненого масиву s попередника кожного набору наплічника $(i; j)$ максимальної вартості можна визначити, порівнюючи значення $s(i; j)$ зі значенням $s(i-1; j)$. Якщо ці значення співпадають, то i-й предмет не входить у набір $(i; j)$. Організувавши рекурсивний процес перегляду масиву s згідно з формулою (28), одержимо перелік предметів оптимального набору.

Приклад 12. Скласти програму розв'язування задачі комбінаторної оптимізації про 0-1 наплічник методом висхідного динамічного програмування згідно з рекурентними співвідношеннями (28), (29).

➢ *Програма*:

```
#include "windows.h"
#include <iostream>
using namespace std;
const int N = 50;  // Найбільша вага наплічника
const int K = 10;  // Найбільша кількість предметів
//-----------------------------------------------------------
// Рекурсивна функція print забезпечує виведення номерів
// найцінніших предметів, завантажених у наплічник 0-1
//-----------------------------------------------------------
void print(int s[N][N], int a[N], int k, int n) {
 if(!s[k][n]) return; // Нічого не виводимо за 0 вартості
 else
  if(s[k-1][n]==s[k][n])
    print(s, a, k-1, n); // Набір без k-го предмету
  else {
     print(s, a, k-1,n-a[k]); // Набір з k-м предметом
     cout<<k<<" ";   }
} // void print ...
//----------------------------------------------------------
// Реалізація і тестування алгоритму (на базі методу вис-
// хідного ДП) визначення завантаження наплічника
// типу 0-1 деякою підмножиною найцінніших предметів
//----------------------------------------------------------
int main() { int k,      // Кількість предметів
 n, /* Вага наплічника */  a[N], // Вектор ваг предметів
 c[N],    // Вектор вартостей предметів
 s[N][N]; // Матриця максимальних сумарних вартостей
 int i,j;
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------- Ініціалізація w  ------------------------------
 cout<<"Вводьте кількість предметів (k)";
 cout<< " і вагу наплічника (n)."<<endl;
 cout<<"Якщо (k<=0||k>=K||n<=0||n>=N)";
 cout<<" - завершення тестувань."<<endl;
 while(cout<<"k, n: ", cin>>k>>n,
```



```
                      k>0 && n>0 && k<K && n<N) {
  cout<<"Введення ваг і вартостей предметів."<<endl;
  for (i=1; i<=k; i++) {
     cout<<"Вага та вартість "<<i<<"-го предмета: ";
     cin>>a[i]>>c[i];  }
//-------Ініціалізація нульового рядка s -------------
  for (i=0; i<=n; i++) s[0][i]=0;
//--------- Базовий цикл -----------------------------
  for (i=1; i<=k; i++) {
   for (j=0; j<=n; j++) {
     s[i][j]=s[i-1][j];
     if(j>=a[i] && (s[i-1][j-a[i]]+c[i]>s[i-1][j]))
       s[i][j]=s[i-1][j-a[i]]+c[i];   }  }
//------ Результати роботи алгоритму -----------------
  cout<<"Максимальна вартість наплічника  ";
  cout<<s[k][n]<<endl; cout<<"Предмети: "; print(s,a,k,n);
  cout<<endl;
 } // while(cout<<"k …
 cin>>k; return 0;
}
```

Результати тестування програми:

```
E:\CPP_2012\Проекти_С++_2012\5_Динамічне_Пр...
Вводьте кількість предметів (k) і вагу наплічника
Якщо (k<=0||k>=K||n<=0||n>=N) - завершення тестув
k, n: 4 5
Введення ваг і вартостей предметів.
Вага та вартість 1-го предмета: 2 12
Вага та вартість 2-го предмета: 1 10
Вага та вартість 3-го предмета: 3 20
Вага та вартість 4-го предмета: 2 15
Максимальна вартість наплічника  37
Предмети: 1 2 4
k, n: 0 0
```

◄

Приклад 13. Скласти програму розв'язування задачі комбінаторної оптимізації про 0-1 наплічник методом низхідного динамічного програмування згідно з рекурентними співвідношеннями (28), (29).

➢ *Попередні міркування*. Для збереження значень функції $s(i;\ j)$, де $i=\overline{0;k}$; $j=\overline{0;n}$, використовуватимемо глобальний масив s[N][N]. Під час обчислення цієї функції за умовою задачі необхідно застосовувати метод *низхідного* динамічного програмування, який передбачає організацію рекурсивного процесу. З метою економії часу та пам'яті масиви, в яких зберігатимуться значення ваг та прибутків предметів, також будуть глобальними.

Ознакою того, що значення функції $s(i;\ j)$ ще не обчислювалися на попередніх кроках, слугуватиме значення $s(i;\ j)=-1$.

Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
const int N = 50; // Найбільша вага наплічника
const int K = 10; // Найбільша кількість предметів
int s[N][N]; // Масив максимальних вартостей
int a[K];    // Масив ваг предметів
int c[K];    // Масив прибутків предметів
//-------------------------------------------------------
//  Реалізація алгоритму (на базі методу низхідного ДП)
//  визначення  завантаження наплічника типу 0-1
//  деякою підмножиною найцінніших предметів
//-------------------------------------------------------
int bag(int i, int j) {  int v = s[i][j], r;
  if(v>=0) return v; // Початкове чи обчислене значення
  v = bag(i-1,j);
  if(j>=a[i]) {r = bag(i-1,j-a[i])+c[i]; if (r>v) v = r;}
  s[i][j]=v; return v;
} // int bag ...
// Виведення номерів предметів оптимального набору
void print(int i, int j) {
  if(!s[i][j]) return; // Нічого не виводимо за 0 вартості
  else
```



```
   if(s[i-1][j]==s[i][j])
     print(i-1, j); // Набір без i-го предмету
   else { print(i-1, j-a[i]); // Набір з i-м предметом
          cout<< i <<" ";  }
} // void print ...
int main() {
 int n,       // Вага  наплічника
     k;      // кількість предметів
 int i,j, m;
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------ Ініціалізація w  ------------------------------
  cout<<"Вводьте кількість предметів (k) ";
  cout<<"" "і вагу (n)."<<endl;
  cout<<"Якщо (k<=0||k>=K||n<=0||n>=N)";
  cout<<" - завершення тестувань."<<endl;
  while(cout<<"k, n: ",cin>>k>>n,k>0 && n>0 && k<K && n<N)
  {
   cout<<"Введення ваг і вартостей предметів."<<endl;
   for (i=1; i<=k; i++) {
     cout<<"Вага та вартість "<<i<<"-го предмета: ";
     cin>>a[i]>>c[i];
   }
//--------Ініціалізація нульового рядка s -------------
   for (j=0; j<=n; j++) s[0][j]=0;
//--------Ініціалізація нульового стовпця s ------------
   for (i=1; i<=k; i++) s[i][0]=0;
//--------Ініціалізація решти елементів s -------------
   for (i=1; i<=k; i++)
     for (j=1; j<=n; j++) s[i][j]=-1;
   m = bag(k, n); // Виклик базової функції
//------ Результати роботи алгоритму  ----------
   cout<<"Максимальна вартість наплічника  ";
   cout<<m<<endl;
   cout<<"Предмети: ";
   print(k, n);  cout<<endl;
  } // while(cout<<"k ...
 cin>>k; return 0;
}
```

◄

5.4. Метод динамічного програмування на графах

5.4.1. Базові поняття теорії графів

У багатьох задачах математики, інформатики та технічних дисциплін виникає необхідність відображення відношень між деякими об'єктами. *Графи* – це природна модель для таких відношень. Найкращою ілюстрацією може слугувати карта автошляхів. Міста – це вершини графа, а автошляхи – ребра. Іноді графи є орієнтованими (подібно до вулиць з одностороннім рухом) або навантаженими (кожен шлях має відстань і відповідну вартість проїзду).

Нехай задано непорожню скінченну множину V і множину E невпорядкованих пар *різних* елементів з множини V. *Простим графом* G називають пару множин V і E, тобто $G = (V, E)$.

Елементи множини V називають *вершинами* графа G, а невпорядковані пари різних вершин – *ребрами*. Кількість вершин позначають як $|V|$, а кількість ребер – як $|G|$. Якщо пара вершин a і b є ребром, то її позначають $[a, b]$. Вершини a і b є *кінцями* ребра. Зауважимо, що $[a, b]$ і $[b , a]$ позначають одне і те ж ребро.

На рисунках ребра зображають прямою чи кривою лінією без стрілок, а вершини позначають літерами або числами. Оскільки E – *множина*, то у простому графі довільну пару вершин може з'єднувати не більше одного ребра.

У деяких випадках розглядають графи, у яких дві вершини може з'єднувати декілька ребер. Виникає поняття *мультографа* – пари (V, E), де V – непорожня скінченна множина, а E – *сім'я* невпорядкованих пар *різних* елементів з множини V.

Термін "сім'я" означає, що елементи E (ребра) можуть повторюватися. Ребра, які з'єднують одну й ту ж пару вершин, називають *кратними* (або *паралельними*) ребрами.

Наступне узагальнення полягає у тому, що окрім кратних ребер допускають наявність *петель* – ребер, які з'єднують вершину саму з собою.

Псевдографом називають пару (V, E), де V – непорожня скінченна множина, а E – *сім'я* невпорядкованих пар елементів з множини V (не обов'язково різних).

Три типи графів, які введені вище, називають *неорієнтованими графами*, причому:

- *псевдограф* – може мати петлі і кратні ребра;
- *мультиграф* – може мати кратні ребра (без петель);
- *простий граф* – без петель і кратних ребер.

Вершини a і b в неорієнтованому графі називають *суміжними*, якщо $[a, b]$ є ребром. Два ребра називають *суміжними*, якщо вони мають спільну вершину. Вершина v і ребро e називають *інцидентними*, якщо вершина v є кінцем ребра e. Отже, суміжність відображає зв'язок між однорідними елементами графа, а інцидентність – між неоднорідними елементами.

Степінь вершини v (позначають $deg(v)$) у неорієнтованому графі – це кількість ребер, інцидентних вершині v, причому петлю враховують двічі. Якщо $deg(v) = 0$, то вершину v називають *ізольованою*; якщо $deg(v) = 1$, то вершину v називають *висячою*.

Послідовність ребер $[v_0, v_1], [v_1, v_2], [v_2, v_3], \ldots, [v_{n-1}, v_n]$ неорієнтованого графа називають *шляхом* довжини n, що з'єднує вершини v_0 та v_n (ребро враховують стільки разів, скільки воно входить у шлях). Якщо при цьому $v_0 = v_n$, то шлях називають *циклом*.

Якщо усі ребра шляху/циклу *різні*, то його називають *простим* шляхом/циклом. Шлях з крайніми вершинами v_0 та v_n позначають $<v_0, v_n>$ і називають $<v_0, v_n>$-шляхом.

Граф $G' = (V', E')$ називають підграфом графа $G = (V, E)$, якщо $V' \subseteq V$ і $E' \subseteq E$.

Підграф неорієнтованого графа називають *зв'язним*, якщо у ньому для кожної пари вершин знайдеться хоча б один шлях, що їх з'єднує. Відношення "з'єднані шляхом" є *відношенням еквівалентності* на множині вершин. Класи еквівалентності називають *компонентами зв'язності* графа (слово "*зв'язності*" у терміні іноді опускають). Неорієнтований граф *зв'язний*, якщо він складається з єдиної компоненти зв'язності.

Граф називають *деревом*, якщо він зв'язний і не містить простих циклів. Граф, який не містить простих циклів і складається з k компонент, називають *лісом* з k дерев.

Увага! З означень випливає, що дерева і ліси є простими графами.

Теорема 1. Для графа G, що має n вершин і m ребер (тобто $|V| = n$; $|G| = m$), такі твердження *еквівалентні*:

1. G – дерево.
2. G – зв'язний граф і $m = n - 1$.
3. G – не містить простих циклів і $m = n - 1$.
4. G – зв'язний граф, проте вилучення довільного ребра робить його незв'язним.
5. Будь-які дві вершини G з'єднані єдиним простим шляхом.
6. G – не містить простих циклів і при з'єднанні ребром двох несуміжних вершин у новому графі буде єдиний простий цикл.

Наслідок (з теореми 1). У лісі з k дерев: $m = n - k$.

Каркасом (*з'єднувальним деревом*) простого зв'язного графа G називають його підграф, який є деревом і містить усі вершини G.

Орієнтованим графом (або *орграфом*) називають пару множин V і E, де V – непорожня скінченна множина, а E – множина *упорядкованих* пар елементів з множини V.

Елементи множини V називають *вершинами* (або *вузлами*), а множини E – *дугами* (або *орієнтованими ребрами*). Якщо пара вершин a і b є дугою, то її позначають (a, b). Вершину a називають *початковою*, а вершину b – *кінцевою*. Дугу (a, a) називають *петлею*. Зауважимо, що орграф може мати петлі (проаналізуйте означення).

На рисунках дугу позначають прямою чи кривою лінією зі стрілкою, яка вказує на кінцеву вершину. Дуга – це впорядкована пара вершин, причому дуги (a, b) і (b, a) є *різними* дугами.

Орієнтованим мультографом називають пару множин V і E, де V – непорожня скінченна множина, а E – *сім'я* упорядкованих пар елементів з множини V.

Елементи E (дуги) в орієнтованому мультиграфі можуть повторюватися, тоді їх називають *кратними* дугами. Кратні дуги з'єднують одну й ту ж пару вершин та *однаково напрямлені*.

Граф, який має є і ребра, і дуги, називають *змішаним*. Його зводять до орграфа заміною кожного ребра $[a, b]$ дугами (a, b) і (b, a).

В орграфі *напівстепенем входу* вершини v називають кількість дуг з *кінцевою* вершиною v (позначають $deg^-(v)$). *Напівстепенем виходу* вершини v називають кількість дуг, для яких вершина v є *початковою* (позначають $deg^+(v)$). Степінь вершини v визначають так: $deg(v) = deg^+(v) + deg^-(v)$.

Послідовність дуг $(v_0, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$ орієнтованого графа називають *орієнтованим шляхом* (або просто *шляхом*), що з'єднує вузли v_0 та v_n (кінець довільної дуги шляху, окрім останньої, є початком наступної дуги). Довжиною шляху називають кількість дуг, з яких він складається.

В орграфі шлях однозначно визначають упорядкованою послідовністю вершин $p = <v_0, v_1, \ldots, v_{n-1}, v_n>$. У цьому випадку кажуть, що для вершин v_0 і v_n існує шлях p від v_0 до v_n (або вершина v_n є *досяжною* з вершини v_0) і позначають $v_0 \xrightarrow{p} v_n$.

Якщо в орієнтованому шляху $v_0 = v_n$, то цей шлях називають *орієнтованим циклом* (або просто *циклом*). Зауважимо, що петля є спеціальним типом циклу. Орграф, що не має циклів, називають *ациклічним* орграфом.

Якщо усі дуги орієнтованого шляху/циклу в орграфі – *різні*, то його називають *простим* орієнтованим шляхом/циклом.

Лема 1. В ациклічному орграфі є хоча б одна вершина, в яку не входить жодна дуга.

Теорема 2. Вершини ациклічного орграфа можна занумерувати так, що кінець кожної дуги матиме більший номер, ніж номер початку дуги.

Для орграфа поняття зв'язності вводять по-різному, залежно від того, чи враховують напрям дуг. Орієнтований підграф називають *сильнозв'язним,* якщо в ньому для кожної пари вершин u і v знайдеться шлях від u до v та шлях від v до u.

Будь-який орграф можна розбити на *компоненти сильної зв'язності*, які визначають як класи еквівалентності відношення "v є досяжною з u і u є досяжною з v".

Орієнтований граф *сильнозв'язний*, якщо він складається з єдиної компоненти сильної зв'язності.

Орієнтований граф називають *слабкозв'язним*, якщо існує шлях між будь-якими двома різними вершинами в його неорієнтованому варіанті (тобто *без урахування* напряму дуг). Очевидно, що сильнозв'язний орграф є водночас і слабкозв'язним.

Орієнтований граф називають *орієнтованим* (або *кореневим*) *деревом*, яке росте з вершини v_0 (*кореня* дерева), якщо:

- орграф утворює дерево в його неорієнтованому варіанті;
- v_0 – єдина вершина орграфа, в яку не заходить жодна дуга, а у всі інші вершини заходить тільки одна дуга.

Якщо (a, b) є дугою орієнтованого дерева, то вершину a називають *предком* (або *батьком*), а вершину b – *нащадком* (або *сином*). Вершини орієнтованого дерева, які не мають синів, називають *листками*. Вершини (окрім кореня), які мають синів, називають *внутрішніми*.

Орієнтований ліс – це ліс, що складається з орієнтованих дерев. *З'єднувальним орієнтованим деревом* (або *орієнтованим каркасом*) орграфа називають орієнтоване дерево, що містить усі вершини цього орграфа. *З'єднувальним орієнтованим лісом* орграфа називають орієнтований ліс, що містить усі вершини цього графа.

Часто у реальних задачах ребро $[u, v]$ неорієнтованого графа чи дугу (u, v) орграфа *навантажують* дійсним числом – *вагою* $w(u, v)$, яка відображає кількісну характеристику ребра/дуги (наприклад: відстань, швидкість, ціна, пропускна здатність, дохід, прибуток тощо). Ваги, зазвичай, є невід'ємними числами. У деяких випадках ваги мають від'ємні значення (витрати, штрафи тощо).

Граф, у якому кожне ребро/дуга має вагу, називають *навантаженим* (у деяких джерелах – *зваженим*) графом.

Найчастіше для зберігання навантаженого графа $G = (V, E)$ у пам'яті комп'ютера використовують *матрицю ваг*. Для цього нумерують вершини графа G числами 1, 2, …, n, де $n = |V|$, і розглядають матрицю $(w_{ij})_{n \times n}$, в якої $w_{ij} = w(i, j)$, якщо ребро $[i, j]$ чи дуга $(i, j) \in G$. Якщо ж ребро $[i, j]$ чи дуга $(i, j) \notin G$, то $w_{ij} = 0$.

Для графів зі значною кількістю вершин передусім постає задача про існування зв'язків між *усіма* його вершинами (задача

встановлення факту *зв'язності графа*). Цю задачу розв'язують шляхом *виокремлення* із графа деякого каркасу.

Інша задача полягає у виокремленні таких ребер/дуг, які *мінімізують/максимізують* певну адитивну кількісну характеристику графа. Цю задачу розв'язують виокремленням із навантаженого графа деякого каркасу мінімальної/максимальної сумарної ваги (суми ваг усіх ребер/дуг дерева). Такий каркас просто називають *мінімальним/максимальним каркасом.*

Розглянемо деякі задачі, які розв'язують у рамках теорії виокремлення каркасів у простих графах. Першою сформулюємо задачу, з якої історично розпочався розвиток цієї теорії.

Приклад 14 (*поширення чуток*). Розглядають населений пункт, у якому кожен мешканець зустрічає інших мешканців і спілкується з ними. Чи може у цьому населеному пункті поширитися чутка?

➢ Нехай кожному мешканцю відповідає вершина графа. З'єднують дві вершини ребром у тому випадку, якщо відповідні мешканці спілкуються між собою. За умови зв'язності отриманого графа на поставлене запитання можна відповісти позитивно. Граф буде зв'язним, якщо для нього існує каркас. Отже, необхідно перевірити, чи у графа можна виокремити *каркас*. ≺

Приклад 15 (*мережа автошляхів*). Заплановано будівництво мережі автошляхів, які сполучатимуть задану кількість населених пунктів. Вартість будівництва автошляху між будь-якими двома пунктами відома. Необхідно побудувати мережу автошляхів так, щоб вартість будівництва була *мінімальною* і ця мережа сполучала між собою будь-які два пункти.

➢ Побудуємо простий граф, вершини якого відповідають населеним пунктам і точкам перетинання автошляхів, а ребра – автошляхам. Кожне ребро отримує вагу, яка дорівнює вартості будівництва цієї ділянки автошляху. Проект зводиться до виокремлення у графі *мінімального каркасу*. ≺

Задачу прикладу 15 можна сформулювати і так: як прокласти телевізійний кабель між мікрорайонами міста для організації кабельного телебачення; як прокласти труби для забезпечення газом населених пунктів району тощо.

5.4.2. Постановка задачі про найкоротші шляхи

Для практичних застосувань важливою є задача відшукання в орієнтованому графі *оптимальних* шляхів (зазвичай, *найкоротших шляхів*) між вершинами графа. Однак у деяких задачах необхідно відшуковувати і *найдовші* шляхи. Визначимо такі задачі:

1. Від заданої *початкової* вершини графа s виокремити найкоротші шляхи до *всіх* інших вершин графа.
2. Від *усіх* вершин графа виокремити найкоротші шляхи до деякої *кінцевої* вершини графа t.
3. Від *заданої* вершини графа v_i виокремити найкоротший шлях до деякої *іншої* вершини графа v_j.
4. Виокремити найкоротші шляхи між *усіма парами* вершин графа.

Алгоритм, який розв'язує першу задачу, дає змогу розв'язати й інші три задачі. Розв'язування задачі 2 зводиться до розв'язування задачі 1, якщо умовно поміняти напрям усіх дуг. Якщо відшукаємо найкоротші шляхи від початкової вершини v_i до *всіх* інших вершин графа, то одночасно відшукаємо і найкоротший шлях від вершини v_i до вершини v_j (задача 3).

Задачу 4 можна розв'язати або багатократним застосуванням алгоритму задачі 1, де на кожнім кроці за початкову вершину s беруть інші вершини графа, або однократним застосуванням *спеціального* алгоритму.

Вимога розгляду орієнтованих графів не звужує області застосування відповідних алгоритмів, оскільки довільне ребро $[i;j]$ неорієнтованого графа довжиною d_{ij} можна замінити парою дуг (i,j) та (j,i), кожна з яких матиме довжину d_{ij}.

Уточнимо деякі визначення. Отже маємо орієнтований граф $G=(V,E)$ з ваговою функцією $w: E \to R$. *Довжиною* шляху p від вершини v_0 до вершини v_n (позначення $p = <v_0, v_1, \dots, v_{n-1}, v_n>$ або $v_0 \xrightarrow{p} v_n$) називають суму довжин дуг, які належать до цього шляху, тобто $w(p) = \sum_{i=1}^{n} w(v_{i-1}, v_i) = \sum_{i=1}^{n} d_{i-1,i}$.

Довжину найкоротшого шляху з u в v (позначення $\delta(u,v)$) визначають так:

$$\delta(u,v) = \begin{cases} \min\limits_{p} w(p), \forall u \xrightarrow{p} v, \\ \infty, \quad \text{інакше}. \end{cases} \tag{30}$$

Очевидно, що довжина найкоротшого шляху від s до v не перевищує довжину будь-якого шляху від s до v, тобто:

$$\delta(s,v) \le w(s,v). \tag{31}$$

У деяких застосуваннях ваги дуг можуть бути *від'ємними*. У цьому випадку важливо, чи є цикли від'ємної довжини. Якщо з вершини s можна наблизитися до такого циклу, то потім його можна обходити завгодно довго, і вага шляху усе зменшуватиметься, отож для вершин цього циклу найкоротших шляхів не існує (рис. 1).

У цьому випадку вважатимемо, що вага найкоротшого шляху дорівнює $-\infty$. На рис. 1 поблизу кожної вершини проставлено вагу найкоротшого шляху від вершини s. Якщо ж циклів від'ємної ваги немає, то будь-який цикл можна викинути, не подовжуючи шляху.

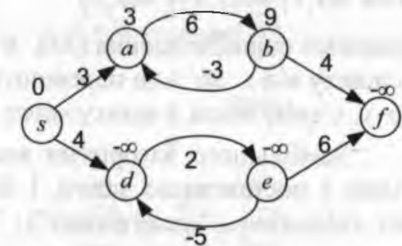


Рис. 1. Цикл від'ємної ваги

Усі алгоритми виокремлення найкоротшого (або найдовшого) шляху в графі є *багатокроковими процедурами ухвалення рішень* в умовах визначеності, оскільки при досягненні певної вершини графа обирається наступна дуга цього шляху. Під час реалізації цієї *процедури* користуються *принципом оптимальності* (або принципом Беллмана[1]), який для навантаженого орграфа формулюють таку лему.

[1] Вчений, який вперше сформулював принцип оптимальності.

Лема 2. Відрізки найкоротших шляхів є найкоротшими.

➢ Нехай $w\left(v_0 \xrightarrow{p} v_n\right) = \delta(v_0; v_n)$. Доведемо, що $\forall i, j$ $(0 \le i < j \le n)$: $w\left(v_i \xrightarrow{q} v_j\right) = \delta(v_i; v_j)$. Міркуватимемо від супротивного. Нехай шлях q – не найкоротший. Замінивши шлях q від v_i до v_j на коротший шлях, ми тим самим зменшимо довжину шляху від v_0 до v_n (*протиріччя*). ◄

Лема 3. Нехай $w\left(s \xrightarrow{p} v\right) = \delta(s; v)$. Тоді

$$\delta(s,v) = \delta(s,u) + w(u,v), \text{ якщо } (u; v) \in p; \tag{32}$$

$$\delta(s,v) \le \delta(s,u) + w(u,v), \text{ якщо } (u, v) \in E. \tag{33}$$

➢ Доведемо співвідношення (32). Запис $(u; v) \in p$ у (32) означає, що $(u; v)$ – остання дуга шляху p. За лемою 2: $w\left(s \xrightarrow{q} u\right) = \delta(s; u)$, отож $\delta(s,v) = \delta(s,u) + w(u,v)$.

Доведемо співвідношення (33). Згідно з (31), довжина найкоротшого шляху від s до v не перевищує довжину будь-якого шляху від s до v, у тому числі й шляху через дугу (u, v). ◄

Здебільшого алгоритми виокремлення найкоротших шляхів згідно з постановкою задачі 1 базуються на принципі *релаксації* (від *relaxation* – "полегшення"). Згідно з цим принципом кожній вершині $v \in V$ приписують число $d[v]$ – верхню оцінку довжини найкоротшого шляху від s до v. Початкові значення верхніх оцінок найкоротшого шляху $\forall v \in V\ (v \ne s): d[v] = +\infty;\ d[s] = 0$. *Релаксація* дуги $(u, v) \in E$ полягає у такому:

значення $d[v]$ зменшується до величини $d[u] + w(u, v)$, якщо $d[u] + w(u, v) < d[v]$.

Теорема 3. Після виконання довільної послідовності релаксацій дуг для кожної вершини $v \in V$ виконується таке:

$$\delta(s,v) \le d[v] \le d[u] + w(u,v). \tag{34}$$

Якщо ж для деякої вершини v маємо $\delta(s,v) = d[v]$, то ця рівність залишиться вірною і при наступних релаксаціях дуг.

➢ Права частина нерівності (34) є очевидною (відповідно до визначень оцінки найкоротшого шляху та релаксації дуги).

Доведемо ліву частину нерівності (34). Після ініціалізації значення $d[v]$ – нескінченні при $v \ne s$, отож слугують оцінкою зверху для чого завгодно, а $d[s] = 0$ (теж правильно). Якщо ж для дуги (u, v) здійснено релаксацію, то $d[v] = d[u] + w(u, v) \ge \delta(s,u) + w(u,v)$. Згідно з (33): $\delta(s,u) + w(u,v) \ge \delta(s,v)$. Отже, $d[v] \ge \delta(s,v)$.

У процесі релаксації значення $d[v]$ можуть тільки зменшуватися, а після досягнення $d[v] = \delta(s,v)$ далі зменшуватися нікуди. ◄

На рис. 2 наведено два приклади релаксації: в одному випадку оцінка найкоротшого шляху зменшується, в іншому – ні. Над вершинами зазначено оцінки найкоротших шляхів.

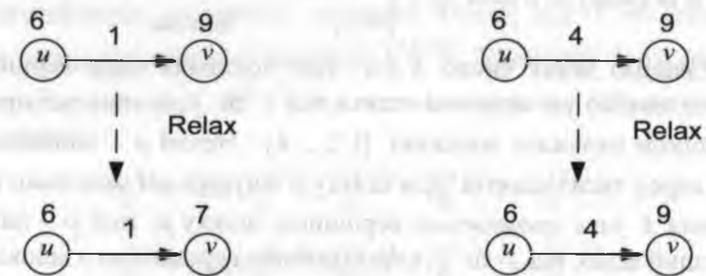


Рис. 2. Приклади релаксації дуг

Теорема 4. Нехай останньою дугою найкоротшого шляху від s до v є дуга (u, v). Якщо у деякий момент до релаксації дуги (u, v) виконувалася рівність $d[u] = \delta(s,u)$, то після релаксації дуги (u, v) виконуватиметься рівність $d[v] = \delta(s,v)$.

➢ Рівність $d[u] = \delta(s,u)$, за теоремою 2, збережеться і після релаксації дуги (u, v). Тоді $d[v] \le d[u] + w(u, v) = \delta(s,u) + w(u, v) = \delta(s,v)$ (остання рівність за співвідношенням (32)).

Однак за теоремою 3: $\delta(s,v) \le d[v]$, отож $d[v] = \delta(s,v)$, що і треба було довести. ◄

5.4.3. Алгоритм Флойда-Уоршолла

Алгоритм Флойда-Уоршолла дає змогу визначити найкоротші шляхи між усіма можливими парами вершин орграфа з *додатними* вагами дуг. Допускаются дуги з від'ємними довжинами, однак без циклів з їхньою сумарною від'ємною довжиною.

Дано зважений орієнтований граф $G = (V, E)$, в якому вершини пронумеровані від 1 до n. Довжини дуг цього орграфа відображає квадратна матриця D порядку n, у якій елемент $d_{ij} > 0$ відображає довжину дуги (i, j). Якщо дуги (i, j) не існує, то $d_{ij} = \infty$. Домовимося, що $d_{ii} = 0$.

Необхідно визначити квадратну матрицю найкоротших відстаней R порядку n, в якій $r_{ij} = \begin{cases} \delta(i, j), \exists i \xrightarrow{p} j, \\ +\infty, \quad \text{інакше.} \end{cases}$

Розглянемо деяке число $k \le n$. Для довільної пари вершин $i, j \in V$ розглянемо усі можливі шляхи від i до j, в яких усі проміжні вершини належать множині $\{1, 2, ..., k\}$. Нехай p – мінімальний шлях серед таких шляхів. Для шляху p існують дві можливості:

1) вершина k *не є проміжною* вершиною шляху p, тоді p – найкоротший шлях від i до j з проміжними вершинами з множини $\{1, 2, ..., k-1\}$;
2) вершина k – *проміжна* вершина шляху p.

У другому випадку вершина k розбиває шлях p на два шляхи: $i \xrightarrow{q} k$ та $k \xrightarrow{r} j$. Проміжні вершини обох шляхів q та r належать множині $\{1, 2, ..., k-1\}$. За лемою 2: $w(i \xrightarrow{q} k) = \delta(i, k)$; $w(k \xrightarrow{r} j) = \delta(k, j)$.

Нехай $r_{ij}^{(k)}$ – довжина найкоротшого шляху від i до j, в якому всі проміжні вершини належать множині $\{1, 2, ..., k\}$. Якщо $k = 0$, то проміжні шляхи відсутні. Отже, $r_{ij}^{(0)} = d_{ij}$.

Опираючись на ці міркування, запишемо рекурентне співвідношення визначення мінімальних шляхів між усіма парами вершин орієнтованого графа:

$$r_{ij}^{(k)} = \begin{cases} d_{ij}; & k = 0, \\ \min\{r_{ij}^{(k-1)}, r_{ik}^{(k-1)} + r_{kj}^{(k-1)}\}; & k = \overline{1, n}. \end{cases} \quad (35)$$

Згідно з методом динамічного програмування необхідно побудувати послідовність матриць $R^{(k)} = \left(r_{ij}^{(k)}\right)$, $k = \overline{0; n}$, між якими встановлені зв'язки згідно з рекурентним співвідношенням (35).

Матриця $R^{(n)} = \left(r_{ij}^{(n)}\right) = R$, оскільки проміжними вершинами можуть бути довільні вершини з множини $\{1, 2, ..., n\}$.

Для довільного $k > 0$ у матриці $R^{(k)}$, згідно з (35), модифікація елементів залежить тільки від k-го рядка і k-го стовпця попередньої матриці $R^{(k-1)}$. Елементи k-го рядка і k-го стовпця матриці $R^{(k-1)}$ під час застосування (35) не змінюються, справді:

$$r_{ik}^{(k)} = \min\{r_{ik}^{(k-1)}, r_{ik}^{(k-1)} + r_{kk}^{(k-1)}\} = r_{ik}^{(k-1)};$$

$$r_{kj}^{(k)} = \min\{r_{kj}^{(k-1)}, r_{kk}^{(k-1)} + r_{kj}^{(k-1)}\} = r_{kj}^{(k-1)}.$$

Звідси випливає, що матрицю $R^{(k)}$ можна записати поверх матриці $R^{(k-1)}$, $R^{(k-1)}$ – поверх матриці $R^{(k-2)}$ і т. д. Зрештою це означає, що для збереження результатів обчислень достатньо однієї матриці. Якщо квадратну матрицю D з початковими даними не використовують далі у програмі для будь-яких інших цілей, то для збереження результатів обчислень, зазвичай, обирають власне D.

Водночас з модифікацією матриці D формують і квадратну матрицю попередників S порядку n з *нульовими* початковими значеннями для її елементів. Ознакою наявності мінімального шляху,

що містить шляхи $i \xrightarrow{q} k$ та $k \xrightarrow{r} j$, слугуватиме $s_{ij} = k$. Відповідна рекурентна формула виглядає так:

$$s_{ij}^{(k)} = \begin{cases} 0; & k = 0, \\ k, & r_{ik}^{(k-1)} + r_{kj}^{(k-1)} < r_{ij}^{(k-1)};\ k = \overline{1, n}. \end{cases} \qquad (36)$$

Найкоротший шлях від вершини i до вершини j визначають за правилом:

- Найкоротшу відстань $\delta(i, j)$ отримують з матриці D.
- Проміжні вершини шляху від вершини i до вершини j рекурсивно визначають за матрицею S. Нехай $s_{ij} = k$, тоді отримуємо шлях $i \xrightarrow{q} k$ та $k \xrightarrow{r} j$. Якщо далі $s_{ik} = 0$ та $s_{kj} = 0$, то весь шлях відшукано (визначено усі проміжні вершини). У протилежному випадку ще повторюють процедуру визначення проміжних вершин для шляхів $i \xrightarrow{q} k$ і/або $k \xrightarrow{r} j$.

Алгоритм Флойда-Уоршолла може давати правильні відповіді за наявності у матриці відстаней дуг з від'ємними значеннями, однак без циклів сумарної від'ємної довжини.

Цикл сумарної від'ємної довжини обов'язково викликає появу *від'ємних* значень на *головній діагоналі* матриці D. Це легко довести.

Алгоритм Флойда-Уоршолла послідовно змінює відстані між усіма парами вершин (i, j), у тому числі і тими, в яких $i = j$. Як відомо, початкова відстань між парою вершин (i, i) дорівнює *нулю*. Отже, $d_{ii} < 0$ може відбутися тільки за наявності такої вершини k, що $d_{ik} + d_{ki} < d_{ii} = 0$. А це еквівалентно наявності циклу сумарної від'ємної довжини, що проходить через вершину i.

Отож для відшукання циклів сумарної від'ємної довжини необхідно після завершення роботи алгоритму визначити вершину m, для якої $d_{mm} < 0$, та вивести найкоротші шляхи між усіма парами вершин (m, j).

Приклад 16. Скласти програму реалізації алгоритму Флойда-Уоршолла.

➢ *Попередні міркування*. Для збереження значень матриць відстаней та попередників використовуватимемо глобальні масиви `d[N][N]` і `s[N][N]`, відповідно. Для зручності користувача дозволимо вводити нульові значення за відсутності дуги між вершинами графа. Під час ініціалізації матриць функцією `init` ці нульові значення перетворюються у значення умовної нескінченності `UN`.

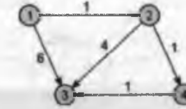
Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
#define N 20 // Верхня межа порядку матриць
#define UN 10000 // Умовна нескінченність
int d[N][N];      // Матриця відстаней
int s[N][N];      // Матриця попередників
//-----------------------------------------------------------
// Ініціалізація матриць
void init(int n) {
 for(int i=1; i<=n; i++)
  for (int j=1; j<=n; j++)
    { s[i][j]=0; if(i!=j && !d[i][j]) d[i][j]=UN; }
}
//-----------------------------------------------------------
// Реалізація алгоритму Флойда-Уоршолла
void min_P(int n) {
for (int k=1; k<=n; k++)
 for (int i=1; i<=n; i++)
  for (int j=1; j<=n; j++)
   if (d[i][k] + d[k][j]<d[i][j])
     {  d[i][j]=d[i][k]+ d[k][j]; s[i][j] = k;  }
}
//-----------------------------------------------------------
// Розрахунок найкоротшого шляху між вершинами і та j
void path(int i, int j) {
 if(s[i][j])
  {path(i, s[i][j]); cout<<s[i][j]<<"->"; path(s[i][j], j); }
 else return;
}
int main() {
```

```
int n;        // Кількість вершин
//------- Введення відстаней ----------------------------
SetConsoleCP(1251); SetConsoleOutputCP(1251);
cout<<"Кількість вершин n="; cin>>n;
cout<<"Введення відстаней у матрицю d";
cout<<" (0 - немає шляху):"<<endl;
for (int i = 1; i <= n; i++) {
   cout<<"від  "<<i<<"-ої вершини: ";
   for (int j = 1; j <= n; j++) cin>>d[i][j];
}
init(n); // Ініціалізація матриць
min_P(n); // Виконання алгоритму
cout<<"Матриця найменших відстаней між пунктами \n";
for (int i=1; i<=n; i++)  {
  for (int j=1; j<=n; j++)
   if(d[i][j]!=UN) cout<<d[i][j]<<" " ; else cout<<"N ";
  cout<<endl; // "N" на консолі позначає нескінченність
}
cout<<"Матриця попередників \n";
for (int i=1; i<=n; i++)
{for (int j=1; j<=n; j++) cout<<s[i][j]<<" "; cout<<endl; }
cout<<endl;
cout<<"****** Найкоротші шляхи ******\n";
int st; // Початкова вершина шляхів
while(cout<<"Старт (0 - завершення програми): ",
                    cin>>st,st) {
 for (int j=1; j<=n; j++) {
  if(d[st][j] == UN)
    cout<<st<<"->"<<j<<": немає шляху!\n";
  else {
    cout<<"d("<<st<<","<<j<<")="<<d[st][j]<<": ";
    cout<<st<<"->"; path(st, j); cout<< j<<endl;
  }
 }
}
cin>>n; return 0;
}
```


Результати тестування програми

1) Приклад з російської Вікіпедії (алгоритм Флойда)

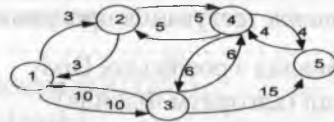


E:\CPP_2012\Проекти_С++_2012\5_Алгоритми_На_Мас...

```
Кількість вершин n=4
Введення відстаней у матрицю d (0 - немає шляху):
від  1-ої вершини: 0 1 6 0
від  2-ої вершини: 0 0 4 1
від  3-ої вершини: 0 0 0 0
від  4-ої вершини: 0 0 1 0
Матриця найменших відстаней між пунктами
0 1 3 2
N 0 2 1
N N 0 N
N N 1 0
Матриця попередників
0 0 4 2
0 0 4 0
0 0 0 0
0 0 0 0

****** Найкоротші шляхи ******
Старт (0 - завершення програми): 1
d(1,1)=0: 1->1
d(1,2)=1: 1->2
d(1,3)=3: 1->2->4->3
d(1,4)=2: 1->2->4
Старт (0 - завершення програми): 2
2->1: немає шляху!
d(2,2)=0: 2->2
d(2,3)=2: 2->4->3
d(2,4)=1: 2->4
Старт (0 - завершення програми): 3
3->1: немає шляху!
3->2: немає шляху!
d(3,3)=0: 3->3
3->4: немає шляху!
Старт (0 - завершення програми):
```

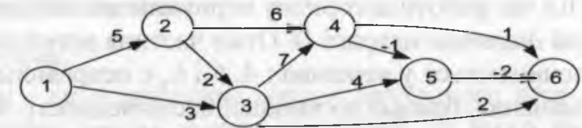
2) Приклад 3.6 з [2, с. 44]



```
E:\CPP_2012\Проекти_C++_2012\5_Алгоритми_На_М...
Кількість вершин n=5
Введення відстаней у матрицю d (0 - немає шляху):
від  1-ої вершини: 0 3 10 0 0
від  2-ої вершини: 3 0 0 5 0
від  3-ої вершини: 10 0 0 6 15
від  4-ої вершини: 0 5 6 0 4
від  5-ої вершини: 0 0 0 4 0
Матриця найменших відстаней між пунктами
0 3 10 8 12
3 0 11 5 9
10 11 0 6 10
8 5 6 0 4
12 9 10 4 0
Матриця попередників
0 0 0 2 4
0 0 4 0 4
0 4 0 0 4
2 0 0 0 0
4 4 4 0 0

****** Найкоротші шляхи ******
Старт (0 - завершення програми): 1
d(1,1)=0: 1->1
d(1,2)=3: 1->2
d(1,3)=10: 1->3
d(1,4)=8: 1->2->4
d(1,5)=12: 1->2->4->5
Старт (0 - завершення програми): 4
d(4,1)=8: 4->2->1
d(4,2)=5: 4->2
d(4,3)=6: 4->3
d(4,4)=0: 4->4
d(4,5)=4: 4->5
Старт (0 - завершення програми):
```


3) Приклад 3.4 з [2, с. 41]



```
E:\CPP_2012\Проекти_C++_2012\5_Алгоритми_На_Маси...
Кількість вершин n=6
Введення відстаней у матрицю d (0 - немає шляху):
від  1-ої вершини: 0 5 3 0 0 0
від  2-ої вершини: 0 0 2 6 0 0
від  3-ої вершини: 0 0 0 7 4 2
від  4-ої вершини: 0 0 0 0-1 1
від  5-ої вершини: 0 0 0 0 0 -2
від  6-ої вершини: 0 0 0 0 0 0
Матриця найменших відстаней між пунктами
0 5 3 10 7 5
N 0 2 6 5 3
N N 0 7 4 2
9997 9997 9997 0 -1 -3
9998 9998 9998 9998 0 -2
N N N N 9999 0
Матриця попередників
0 0 0 3 3 3
0 0 0 0 4 5
0 0 0 0 0 0
6 6 6 0 0 5
6 6 6 6 0 0
0 0 0 0 4 0

****** Найкоротші шляхи ******
Старт (0 - завершення програми): 2
2->1: немає шляху!
d(2,2)=0: 2->2
d(2,3)=2: 2->3
d(2,4)=6: 2->4
d(2,5)=5: 2->4->5
d(2,6)=3: 2->4->5->6
Старт (0 - завершення програми):
```

Під час роботи алгоритму отримали два *від'ємні* значення на *головній діагоналі* матриці *D*. Отже частина результатів щодо шляхів, які починаються у вершинах 4, 5 і 6, є неправильною (про це свідчать значення, близькі до умовної нескінченності: 9997, 9998, 9999):

```
E:\CPP_2012\Проекти_С++_2012\5_Алгоритми_На_Маси...
Кількість вершин n=6
Введення відстаней у матрицю d (0 - немає шляху):
від  1-ої вершини: 0 5 3 0 0 0
від  2-ої вершини: 0 0 2 6 0 0
від  3-ої вершини: 0 0 0 7 4 2
від  4-ої вершини: 0 0 0 0-1 1
від  5-ої вершини: 0 0 0 0 0 -2
від  6-ої вершини: 0 0 0 0 0 0
...
****** Найкоротші шляхи ******
Старт (0 - завершення програми): 4
d(4,1)=9997: 4->5->6->1
d(4,2)=9997: 4->5->6->2
d(4,3)=9997: 4->5->6->3
d(4,4)=0: 4->4
d(4,5)=-1: 4->5
d(4,6)=-3: 4->5->6
Старт (0 - завершення програми): 5
d(5,1)=9998: 5->6->1
d(5,2)=9998: 5->6->2
d(5,3)=9998: 5->6->3
d(5,4)=9998: 5->6->4
d(5,5)=0: 5->5
d(5,6)=-2: 5->6
Старт (0 - завершення програми): 6
6->1: немає шляху!
6->2: немає шляху!
6->3: немає шляху!
6->4: немає шляху!
d(6,5)=9999: 6->4->5
d(6,6)=0: 6->6
Старт (0 - завершення програми):
```

◄

5.5. Застосування жадібного методу

5.5.1. Загальні положення

Жадібний метод – це модифікація методу вичерпного перебирання до розв'язування деяких задач *оптимізації*, за якого на кожному кроці визначається розв'язок, який є локально *оптимальним*, *кінцевим* і *задовольняє* усім *обмеженням* задачі.

На відміну від алгоритмів методу динамічного програмування, алгоритми жадібного методу оптимально розв'язують кожну підзадачу як *окрему*, *незалежну* задачу. Від розв'язку кожної окремої підзадачі не залежать оптимальні розв'язки інших підзадач. Алгоритм жадібного методу на кожному кроці робить *локально-оптимальний* вибір у надії, що результуючий розв'язок також виявиться оптимальним. Це не завжди так, однак для багатьох задач алгоритми жадібного методу насправді дають оптимальні розв'язки.

Задачі оптимізації, які допускають застосування жадібного методу, обов'язково задовольняють *принципу жадібного вибору* та мають *оптимальну підструктуру*.

До задачі оптимізації можна застосувати принцип *жадібного вибору*, якщо послідовність локально оптимальних розв'язків дає глобально оптимальний розв'язок. У типовому випадку доведення оптимальності має таку схему:

- спочатку доводять, що жадібний вибір на першому кроці не буде перешкодою до відшукання глобального оптимального розв'язку: для будь-якого розв'язку існує інший розв'язок, *узгоджений* з жадібним, що дає не гірший результат;
- далі доводять, що підзадача, яка виникла після жадібного вибору на першому кроці, аналогічна початковій, і міркування закінчується за індукцією.

Іншими словами, жадібний алгоритм ніколи не переглядає свої попередні вибори для здійснення наступного вибору, на відміну від динамічного програмування.

Задача має *оптимальну підструктуру*, якщо оптимальний розв'язок усієї *задачі* містить оптимальні розв'язки для кожної *підзадачі*. Зазначимо, що цією властивістю володіють і задачі оптимізації, до яких застосовують метод динамічного програмування.

Необхідність властивості оптимальної підструктури задачі для обох методів може призводити до прихованих пасток під час спроби застосування алгоритму жадібного методу. Пояснимо це на прикладі задачі про банкомат (приклад 3).

У реальних грошових системах більшості країн світу номінали банкнот підібрано так, що жадібний метод дійсно дає оптимальний (мінімальний щодо кількості банкнот) розв'язок задачі про видачу банкоматом довільної суми. Однак проблема полягає у тому, що у банкоматі у фіксований момент часу можуть бути відсутні окремі номінали банкнот. Візьмемо винятково учбову ситуацію.

Нехай у банкоматі залишилися банкноти номіналом 2, 5 і 10 гривень. Очевидно, що за такого набору номіналів сформувати суму в 1 чи 3 гривні неможливо жодним методом.

Жадібний метод не зможе сформувати суму 6, 8, 11 гривень і т. д. Наприклад, для видачі суми 6 гривень жадібний метод на першому кроці обирає банкноту з *найбільшим* допустимим номіналом 5 гривень. Сформувати залишок в 1 гривню не вдається.

Метод динамічного програмування без проблем справляється з видачею цих сум грошей: 6 = 2 + 2 +2; 8 = 2 + 2 + 2 + 2; 11 = 2 + + 2 + 2 + 5 і т. д.

Отже, жадібний метод передбачає побудову розв'язку за допомогою виконання послідовності кроків, на кожному з яких отримують *часткові оптимальні* розв'язки початкової задачі доти, доки не буде отримано повний розв'язок.

При цьому на кожному кроці вибір розв'язку повинен бути *допустимим* (задовольняти усі обмеження задачі), *локально оптимальним* (бути найкращим локальним вибором серед усіх варіантів, які доступні на конкретному кроці), *остаточним* (частковий оптимальний розв'язок не може змінюватися на наступних кроках алгоритму).

Далі у цьому параграфі розглянемо *класичні* алгоритми, які базуються на жадібному методі. Це алгоритми побудови мінімального каркаса в неорієнтованому простому графі (алгоритми Пріма та Краскала) та алгоритми побудови найкоротших шляхів в орієнтованому графі (алгоритми Дейкстри та Беллмана-Форда).

5.5.2. Виокремлення мінімального каркаса в простих графах

Нехай задано зв'язний простий граф $G = (V, E)$ і вагову функцію $w : E \to R$. Опишемо *жадібний метод* виокремлення мінімального каркаса для цього графа.

Нехай підмножина $A \subseteq E$ є водночас і *підмножиною* деякого *мінімального каркаса*. Шуканий каркас будують поступово:

> до початково *порожньої* множини A на кожному кроці методу додають деяке ребро $[u, v]$ так, щоб множина $A \cup \{[u, v]\}$ теж була *підмножиною мінімального каркаса* (ребро $[u, v]$ називають *безпечним* ребром для A).

Безпечні ребра на будь-якій ітерації циклу забезпечують виконання *вимоги*, щоб A залишалася підмножиною деякого мінімального каркаса (для порожньої множини вимога справджується).

Далі опишемо правило відшукання безпечних ребер. Почнемо з визначень. Нехай $S \subseteq V$ – підмножина вершин графа G, які уже з'єднані ребрами підмножини $A \subseteq E$, а $V \setminus S$ – містить решту вершин V (очевидно, що $(V \setminus S) \cap S = \varnothing$).

Перерізом $(S, V \setminus S)$ неорієнтованого графа $G = (V, E)$ називають підмножину таких ребер з E, що один з його кінців лежить у S, а інший – у $V \setminus S$ (рис. 3).

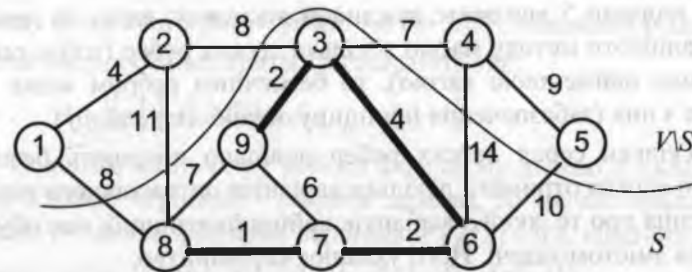


Рис. 3. Переріз $(S, V \setminus S)$ неорієнтованого графа G

Переріз *узгоджений* з множиною ребер A, якщо жодне ребро з A не перетинає цей переріз. У множині ребер, що належать перерізу, виокремлюють ребра *найменшої* ваги (назвемо їх *легкими* ребрами). На рис. 3 переріз узгоджений з A; ребро [3, 4] – *легке* ребро.

Теорема 5. Нехай A – підмножина ребер деякого мінімального каркаса $T \subseteq G$, $(S, V \setminus S)$ – переріз G, узгоджений з A. Якщо $[u, v]$ – легке ребро цього перерізу, то $[u, v]$ є безпечним для A.

➢ Нехай T не містить ребра $[u, v]$, оскільки у цьому випадку твердження теореми є очевидним. Покажемо, що існує інший мінімальний каркас T^*, що містить $A \cup \{[u, v]\}$ (тоді $[u, v]$ є безпечним).

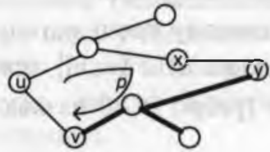


Рис. 4. Замикання p

Каркас T зв'язний, отож містить єдиний шлях p з u до v (рис. 4); ребро $[u, v]$ замикає p у цикл. Оскільки u і v належать різним підмножинам перерізу $(S, V \setminus S)$, узгодженого з A, то шлях p має принаймні одне ребро $[x, y]$, що перетинає переріз і не лежить в A.

Додавши до дерева T ребро $[u, v]$ і видаливши з нього ребро $[x, y]$, одержимо новий каркас $T^* = (T \cup \{[u, v]\}) - \{[x, y]\}$. Залишилося довести, що T^* – мінімальний каркас.

Оскільки $[u, v]$ – легке ребро перерізу $(S, V \setminus S)$, то вилучене з T ребро $[x, y]$ має не меншу вагу, ніж вага ребра $[u, v]$. Отже, вага каркаса T^* могла тільки *зменшитися*. Проте каркас T – *мінімальний*. Відповідно, вага нового каркаса T^* дорівнює вазі T. Отож ребро $[u, v]$, що міститься у T^*, є *безпечним*. ➤

З теореми 5 випливає важливий висновок: якщо на деякому кроці жадібного методу маємо декілька легких ребер (тобто ребер з однаковою найменшою вагою), то безпечним ребром може бути будь-яке з них (забезпечення принципу *жадібного вибору*).

Оскільки серед легких ребер довільно обирають безпечне ребро, то можна отримати декілька варіантів оптимального розв'язку. Рішення про те, який з варіантів найприйнятніший, має обумовлюватися змістом задачі. Його ухвалює керівництво.

Для реалізації жадібного методу виокремлення мінімального каркаса в простих графах найчастіше використовують алгоритми *Пріма* (R. C. Prim) та *Краскала* (J. B. Kruskal). Обидва алгоритми однаково використовують загальну схему жадібного методу, однак по-різному обирають безпечні ребра.

5.5.3. Алгоритм Пріма

Алгоритм Пріма будує мінімальний каркас за допомогою *одного* дерева A, формування якого починається з довільної кореневої вершини r. На кожній ітерації циклу до дерева долучають нову вершину, додаючи *легке* ребро. Якщо таких ребер є декілька, то обирають будь-яке з них. Згідно з теоремою 5, ці ребра є безпечними для A.

Алгоритм завершує роботу після того, як усі вершини графа належатимуть до дерева A. Оскільки алгоритм розширює дерево на одну вершину за ітерацію, то загальна кількість цих ітерацій дорівнюватиме n, де n – кількість вершин графа.

Під час роботи алгоритму для кожної вершини $i \in V$ підтримують три величини:

- $d[i]$ – *пріоритетності* вершини;
- $p[i]$ – *предка* вершини;
- $z[i]$ – *позначки* належності вершини дереву A (1 – належить; 0 – не належить).

Величина $d[i]$ дорівнює мінімальній вазі серед ваг усіх ребер, що з'єднують i з вершинами S (тобто з вершинами дерева A). Якщо таких ребер ще немає, то $d[i] = +\infty$. Величина $p[i]$ указує на вершину S, до якої прямує ребро вагою $d[i]$. Множину ребер дерева A під час роботи алгоритму визначають так:

$$A = \{[p[i], i]: d[i] = d_{p[i], i};\ z[i] = 1\}.$$

Оскільки на початку роботи алгоритму дерево A складається тільки з однієї вершини r (кореня дерева), то зручно вважати, що предком r є деяка фіктивна вершина 0 ($p[r] = 0$), відстань до якої дорівнює нулю (тобто $A = \{[0, r]: d[r] = 0;\ z[r] = 1\}$).

Решта вершин $i \in V$ $(i \neq r)$ на початку роботи алгоритму матиме такі значення величин: $d[i] = +\infty$; $p[i] = 0$; $z[i] = 0$.

Вершину $j \in V$, для якої $z[j] = 0$, називають вершиною з *тимчасовою* позначкою. На кожній ітерації алгоритму серед вершин з тимчасовими позначками визначають деяку вершину $k \in V$ з

найменшим пріоритетом і долучають її до дерева A (тобто вершина k отримує *постійну* позначку: $z[k]=1$). Після цього перевизначають значення $d[j]$ та $p[j]$ тих вершин з тимчасовими позначками, для яких існує ребро $[k, j]$.

Якщо після завершення роботи алгоритму існує вершина $l\ (l \neq r)$, що $p[l]=0$, то це означатиме, що вершина l – ізольована (таких вершин може бути і декілька).

Приклад 17. Скласти програму реалізації алгоритму Пріма.

➢ *Попередні міркування*. Оскільки матриця ваг ребер W для простого графа (неорієнтованого, без петель) є *симетричною*, то для зручності користувача організуємо введення ваг ребер тільки для верхньої частини матриці (над головною діагоналлю). Значення елементів матриці під головною діагоналлю формуватимемо програмно. За відсутності ребра між вершинами графа вводять нульові значення. Очевидно, що за цих умов нульовими мають бути і елементи головної діагоналі (відсутність петель).

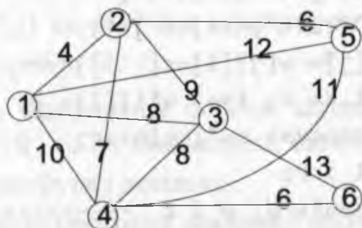
Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
//-------------------------------------------------------
//  Реалізація алгоритму Пріма
//-------------------------------------------------------
int main() {
  const int N = 20, UN = 10000; // UN - нескінченість
  int n,    // Кількість вершин
  w[N][N],  // Матриця ваг
  r,        // Корінь дерева
  d[N],     // Вектор пріоритетів вершин
  z[N],     // Вектор позначок вершин
  p[N];     // Вектор предків
  SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------ Ініціалізація w i r -------------------------
  cout<<"Кількість вершин n="; cin>>n;
  cout<<"Ваги ребер (0 - немає ребра):"<<endl;  int c;
```



```
  for (int i = 1; i <= n; i++)
   for (int j = i+1; j <= n; j++) {
     cout<<"Вага ["<<i<<","<<j<<"]=";
     cin>>c;  w[i][j]= w[j][i]=c;    }
  for (int i = 1; i <= n; i++) w[i][i]= 0;
  cout<<"Стартова вершина r="; cin>>r;
  (r<1 || r>n)? r=1 : r;
//--------Ініціалізація d, p i z  --------------------
  for(int i=1; i<=n; i++) { p[i]=z[i]=0; d[i] = UN; }
  d[r] = 0; z[r]=1;
//--------- Базовий цикл -----------------------------
  for (int i=1; i<=n; i++)  {
    int k=r, j, m_d = UN;
    // Визначення мінімального пріоритету серед вершин
    // з тимчасовою позначкою (ТП)
    for (j=1; j<=n; j++)
      if (!z[j] && m_d>d[j]) {m_d = d[j]; k = j; }
    z[k] = 1; // Вершина k отримує постійну позначку
    // Перевизначення мінімальних пріоритетів
    // для вершин з ТП
    for (j=1; j<=n; j++)
     if (w[k][j] && !z[j] && d[j] > w[k][j])
       { d[j] = w[k][j]; p[j] = k; }
  } // for (int i = 1; ...
//------ Результати роботи алгоритму Пріма -----------
  cout<<"****** Каркасне дерево ******\n\n";
  int total=0; // Якщо total==0 після наступного циклу,
  for(int i=1;i<=n; i++)  // то r - ізольована вершина!
   if(p[i]) {
          cout<<"d["<<p[i]<<","<<i<<"]= "<<d[i]<<endl;
          total+=d[i];
   }
  cout<<"Сумарна довжина="<<total<<endl; cin>>n; return 0;
}
```

Результати тестування програми (приклад 2.5 з [5, с. 22])




```
E:\CPP_2012\Проекти_C++_2012\5_Алгоритми_На_Масивах\...
Кількість вершин n=6
Ваги ребер (0 - немає ребра):
Вага [1,2]=4
Вага [1,3]=8
Вага [1,4]=10
Вага [1,5]=12
Вага [1,6]=0
Вага [2,3]=9
Вага [2,4]=7
Вага [2,5]=6
Вага [2,6]=0
Вага [3,4]=8
Вага [3,5]=0
Вага [3,6]=13
Вага [4,5]=11
Вага [4,6]=6
Вага [5,6]=0
Стартова вершина r=1
****** Каркасне дерево ******

d[1,2]= 4
d[1,3]= 8
d[2,4]= 7
d[2,5]= 6
d[4,6]= 6
Сумарна довжина=31
```


5.5.4. Алгоритм Краскала

Алгоритм Краскала ототожнює мінімальний каркас з ациклічним зв'язаним *підграфом* з $|V|-1$ ребрами, сума ваг яких є мінімальною. Згідно з теоремою 1, такий підграф є *деревом*.

На початку роботи алгоритму мінімальний каркас є *порожнім*. Алгоритм базується на ідеї *жадібного* методу: на кожній ітерації до каркаса додають *найлегше* ребро, не допускаючи при цьому утворення циклів (тобто, якщо чергове найлегше ребро при додаванні до каркаса утворюватиме цикл, то його просто пропускають і додають до каркаса наступне найлегше ребро і т. д.). На завершенні алгоритму отримують єдине дерево, яке згідно з побудовою буде мінімальним.

Отже, алгоритм Краскала будує мінімальний каркас за допомогою послідовності *підграфів* $G_1 \to G_2 \to ... \to G_i \to ... \to G_{n-1}$, які завжди *ациклічні*, однак на проміжних ітераціях не завжди зв'язані:

- G_1 – підграф, що містить одне (*найлегше*) ребро графа;
- G_2 – підграф, що містить два (*найлегші*) ребра графа;
- G_i – ліс, що містить *i найлегших* ребер, які не утворюють циклів (може містити один або декілька підграфів, побудованих на попередніх ітераціях);
- G_{n-1} – шуканий мінімальний каркас.

Вважатимемо, що граф відображають списком ребер графа, які попередньо посортовані за неспаданням значень їхніх ваг. Перше (*найлегше*) ребро цього списку e_1 обов'язково має входити у мінімальний каркас. Доведемо *від протилежного*.

Нехай T – мінімальний каркас, який не містить e_1. Зрозуміло, що граф $T \cup \{e_1\}$ має цикл. Видалимо у $T \cup \{e_1\}$ будь-яке ребро циклу, відмінне від e_1. Отримаємо каркас з e_1, який є *легшим*, ніж Т (*протиріччя*).

Отже, після приєднання до порожнього каркаса ребра e_1 одержали *перший фрагмент* мінімального каркаса.

Оскільки у графі немає паралельних ребер, то додавання до каркаса наступного найлегшого ребра не може спричинити циклу. Отож друге за вагою ребро також має входити у мінімальний каркас. Воно може або "приєднатися" до вже існуючого першого фрагмента або дати початок *другому* (поки що однореберному) *фрагменту*.

Для решти ребер необхідно здійснювати аналіз на можливість їхнього додавання до мінімального каркаса:

- кінці ребра не належать жодному фрагменту (ребро необхідно додати до мінімального каркаса, утворивши при цьому новий фрагмент, якому слід присвоїти черговий номер);
- тільки один з кінців ребра належить деякому фрагменту (ребро необхідно додати до мінімального каркаса, приєднуючи його до цього фрагмента);
- кінці ребра належать різним фрагментам з номерами k_1 і k_2, де $k_1 < k_2$ (ребро необхідно додати до мінімального каркаса, об'єднавши їх в єдиний фрагмент з номером k_1);
- кінці ребра належать одному фрагменту; ребро не можна додавати до мінімального каркаса, оскільки це спричинить появу циклу.

Побудова мінімального каркаса завершиться після додавання до нього ребер у кількості $n-1$. До цього моменту всі фрагменти об'єднаються в один. Якщо всі m ребер графа переглянуті, а мінімальний каркас містить менше, ніж ($n-1$)-не ребро, то це означатиме, що початковий граф є незв'язаним графом.

Теорема 6. Алгоритм Краскала будує мінімальний каркас.

➤ Нехай T_K – каркас, отриманий за алгоритмом Краскала, а T_m – мінімальний каркас. Вважатимемо, що $e_i = [u, v]$ – *перше* таке ребро, що $e_i \in T_K$; $e_i \notin T_m$ (припущення $e_i \notin T_K$; $e_i \in T_m$ означало б, що e_i утворює цикл в T_K, а в T_m – не утворює, а цього не може бути для *першого* неспівпадіння ребер у каркасах).

Оскільки T_m – дерево, то у ньому існує єдиний шлях p з u до v (див. рис. 4). Очевидно, що ребро $[u, v]$ замикатиме p у цикл. Тоді має існувати ребро $e_k = [x; y] \in T_m$; $e_k \notin T_K$.

Згідно з припущенням, *першим* ребром, яке в одному каркасі перебуває, а в іншому – ні, є ребро e_i. Отже, e_k – наступне таке ребро, а це означає, що $|e_i| \le |e_k|$.

Видаливши з дерева T_m ребро e_k і додавши до нього ребро e_i, одержимо новий каркас T'_m (тобто $T'_m = (T_m / e_k) \cup e_i$). Залишилося довести, що T'_m – *мінімальний* каркас.

Оскільки $|e_i| \le |e_k|$, то $|T'_m| \le |T_m|$. Згідно з припущенням T_m – мінімальний каркас, отож $|T'_m| = |T_m|$. З останньої рівності випливає, що $|e_i| = |e_k|$. Отже, T'_m – теж *мінімальний* каркас.

Повторюємо процедуру порівняння ребер для T'_m і T_K і т. д. Унаслідок цього одержуємо послідовність мінімальних каркасів, що збігається до T_K. ⋞

Приклад 18. Скласти програму реалізації алгоритму Краскала.

➤ *Попередні міркування*. Оскільки матриця ваг ребер W для простого графа (неорієнтованого, без петель) є *симетричною*, то для зручності користувача організуємо введення ваг ребер тільки для верхньої частини матриці (над головною діагоналлю). Значення елементів матриці під головною діагоналлю формуватимемо програмно. За відсутності ребра між вершинами графа вводять нульові значення. Очевидно, що за цих умов нульовими мають бути й елементи головної діагоналі (відсутність петель).

Оскільки алгоритм на кожному кроці відшукатиме *найлегше* ребро, то під час ініціалізації базових величин усі відсутні ребра з початковою *нульовою* вагою (окрім елементів головної діагоналі) будуть замінені ребрами з *умовно-нескінченною* вагою (позначення UN). Це даватиме змогу їх ідентифікувати та вилучати з розгляду під час роботи алгоритму.

Програма:

```
#include "windows.h"
#include <iostream>
using namespace std;
//---------------------------------------------------------
//  Реалізація алгоритму Краскала, який відшукує
//  мінімальне каркасне дерево.
//---------------------------------------------------------
int main() {
  const int N = 20, UN = 10000;
  int n,    // Кількість вершин
  w[N][N],  // Матриця ваг
  z[N];  // Вектор включень вершин
  //Допоміжні змінні
  int c, c1, d1=0, d2=0, i, j, k, l, m, m_d, p, v=0;
  SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------ Ініціалізація w i r  ------------------------
  cout<<"Кількість вершин n="; cin>>n;
  cout<<"Ваги ребер (0 - немає ребра):"<<endl;
  for (i = 1; i <= n; i++)
   for (j = i+1; j <= n; j++) {
     cout<<"Вага ["<<i<<","<<j<<"]=";
     cin>>c;  w[i][j]= w[j][i]=c;
   }
  for (i = 1; i <= n; i++) w[i][i]=UN;
//------ Уточнення w;  ініціалізація z -----------------
  for (i = 1; i <= n; i++) {
   z[i]=0;
   for (j = i+1; j <= n; j++)
      if(!w[i][j]) w[i][j]= w[j][i]=UN;
  }
  cout<<"****** Каркасне дерево ******\n\n";
  int total=0;
//-------- Базовий цикл -----------------------------
```



```
 for (m=1; m<n; m++)  {
   m_d = UN;
   for (i = 1; i <= n; i++)
    for (j = 1; j <= n; j++)
     if(w[i][j]!=UN && w[i][j]<m_d && w[i][j]!=-1) {
      c=0; for(p=1;p<=n;p++) if(z[p]==i||z[p]==j) c++;
       if(c>=2) {
     for(p=1;p<=n;p++) if(w[i][p]!=UN && p!=j) d1=p;
     for(p=1;p<=n;p++) if(w[j][p]!=UN && p!=i) d2=p;
     if(w[d1][d2]==-1) continue;
    }
  m_d = w[i][j]; l=i; k=j;
  }
  cout<<"d["<<l<<","<<k<<"]= "<<w[l][k]<<endl;
  total+=w[l][k];
  w[l][k]=w[k][l]=-1; c=c1=0;
  for (i = 1; i <= n; i++)
   { if(z[i]==l) c++; if(z[i]==k) c1++; }
  if(!c) z[++v]=l; if(!c1) z[++v]=k;
 } // for (m = 1; ...
//------ Сумарна довжина ребер дерева ----------
 cout<<"Сумарна довжина="<<total<<endl;
 cin>>n; return 0;
}
```

5.5.5. Алгоритм Дейкстри

Нехай маємо навантажений орієнтований граф $G = (V, E)$, в якому довжини усіх дуг *невід'ємні*. Алгоритм Дейкстри розв'язує задачу про визначення *найкоротших* шляхів у G від деякої початкової вершини $s \in V$ (старту) до всіх інших вершин, досяжних з s.

Під час роботи алгоритму для кожної вершини $i \in V$ підтримують три величини:

- $d[i]$ – *пріоритетності* вершини;
- $p[i]$ – *предка* вершини;
- $z[i]$ – *позначки* вершини.

Величина $d[i]$ дорівнює *верхній оцінці* довжини найкоротшого шляху від s до i. Початкові значення верхніх оцінок вершин $\forall i \in V\ (i \neq s): d[i] = +\infty;\ d[s] = 0$.

У процесі роботи алгоритму Дейкстри підтримується множина $S \subseteq V$, яка складається з вершин i, для яких $\delta(s,i)$ уже знайдено (тобто $d[i] = \delta(s,i)$). Якщо $i \in S$, то $z[i]=1$ (кажуть, що вершина i має *постійну* позначку). Якщо ж $i \in V \setminus S$, то $z[i]=0$ (кажуть, що вершина i має *тимчасову* позначку). Початкові значення позначок вершин $\forall i \in V\ (i \neq s): z[i] = 0;\ z[s] = 1$.

Величина $p[i]$ указує на вершину S, що є початком дуги, яка направлена до вершини i з верхньою оцінкою $d[i]$. Початкові значення предків вершин $\forall i \in V: p[i] = 0$.

Нехай Q позначає підграф з множиною вершин S та множиною дуг A, які їх зв'язують, тобто

$$Q = (S, A), \text{ де } S = \{i \in V:\ z[i] = 1\},\ A = \{(p[i], i) \in E : i \in S\}.$$

На початку роботи алгоритму Дейкстри $S = \{s\}$, $A = \{(0, s)\}$, де 0 – фіктивна вершина підграфа Q.

На кожній ітерації алгоритму Дейкстри серед вершин з тимчасовими позначками визначають деяку вершину $k \in V \setminus S$ з *найменшим* пріоритетом і включають її разом з дугою $(p[k], k)$ у підграф Q (тобто вершина k отримує *постійну* позначку: $z[k]=1$).

Після цього перевизначають значення $d[j]$ та $p[j]$ тих вершин з тимчасовими позначками, для яких існує дуга (k, j). Перевизначення цих величин полягає у *релаксації* дуги (k, j):

якщо $d[k] + w(k, j) < d[j]$, то $d[j] := d[k] + w(k, j);\ p[j] := k$.

Теорема 7. Під час роботи алгоритму Дейкстри підграф Q відображає орієнтоване дерево, що росте зі стартової вершини s. Після завершення алгоритму підграф Q буде *деревом найкоротших шляхів* від старту s до будь-якої, *досяжної* зі старту, вершини $i \in V$. Множина дуг Q при цьому виглядатиме так:

$$A = \{(p[i], i):\ d[i] = \delta(s,i);\ z[i]=1;\ i \in V\}.$$

➤ Доведення теореми можна відшукати у [24, стор. 490 – 492]. ≺

Якщо після завершення роботи алгоритму існує вершина $l \in V\ (l \neq s)$, що $d[l] = +\infty$, то це означатиме, що від вершини s до вершини l не існує шляху (таких вершин може бути і декілька). Зрозуміло, що ознакою недосяжності вершини l ще можуть слугувати ознаки $p[l] = 0$ чи $z[l] = 0$.

Алгоритм завершує роботу після того, як усі досяжні з s вершини графа належатимуть до дерева Q. Оскільки алгоритм розширює дерево на одну вершину за ітерацію, то загальна кількість цих ітерацій дорівнюватиме n, де n – кількість вершин графа.

Можлива ситуація, за якою зі стартової вершини s не виходить жодна дуга (тобто від s неможливо прокласти шлях до будь-якої іншої вершини $i \in V$). Очевидно, що потреба застосування алгоритму Дейкстри у цій ситуації відсутня.

З метою правильної обробки цієї ситуації зробимо таке. Релаксацію дуг вигляду (s, i), де $i \in V$, виконаємо перед *базовим* циклом перегляду та релаксації дуг. Якщо відбудеться релаксація хоча б однієї дуги (s, i), то значення спеціальної ознаки `s_Path` встановимо рівним одиниці (`s_Path = 1`). У протилежному випадку – `s_Path = 0` (у цьому випадку алгоритм Дейкстри не застосовуватимемо, а обмежимося простим повідомленням про ситуацію).

Приклад 19. Скласти програму реалізації алгоритму Дейкстри.

➢ *Програма*:

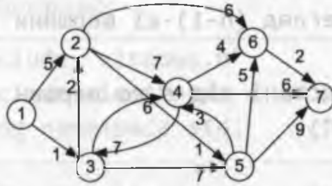
```
#include "windows.h"
#include <iostream>
using namespace std;
//-----------------------------------------------------------
// Реалізація алгоритму Дейкстри, який відшукує найкорот-
// ші шляхи від вершини графа s (стартової) до всіх інших.
//-----------------------------------------------------------
int main() {
 const int N = 20, UN = 10000; // UN - нескінченність
 int n,     // Кількість вершин
    w[N][N], // Матриця ваг
    s,      // Стартова вершина
    d[N],   // Вектор найкоротших відстаней до s
    p[N],   // Вектор предків
    z[N];   // Вектор позначок
 int i, j, k, m_d; // Допоміжні змінні
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------ Ініціалізація w і s ---------------------------
 cout<<"Кількість вершин n="; cin>>n;
 cout<<"Введення відстаней у матрицю w";
 cout<<" (0 - немає шляху):"<<endl;
 for (i = 1; i <= n; i++) {
   cout<<"від  "<<i<<"-ої вершини: ";
   for ( j = 1; j <= n; j++) cin>>w[i][j];
 }
 while(cout<<"Стартова вершина s=", cin>>s,s>=1 && s<=n){
//-------- Ініціалізація d, p, z та w[s][i] ------------
  int s_Path=0; // Ознака наявності дуг від вершини s
  for(int i=1; i<=n; i++) {
    p[i]=0; z[i] = 0;
    w[s][i]? (p[i]=s, d[i]=w[s][i], s_Path=1) : d[i]=UN;
  }
  d[s] = 0; z[s] = 1;
```



```
  if(s_Path) {// Дуги від s наявні -> реалізація алгоритму
//------- Базовий цикл перегляду та релаксації дуг -----
   for (i=1; i<n; i++)  { // Перегляд (n-1)-єї вершини
    m_d = UN;
    // Визначення мінімальної відстані від s до вершин
    // з тимчасовою позначкою (ТП)
    for (j=1; j<=n; j++)
     if (!z[j] && m_d>d[j]) {m_d = d[j]; k = j; }
    z[k] = 1; // Вершина k отримує постійну позначку
    // Перевизначення мінімальних відстаней
    // від вершини s до вершин з ТП
    for (j=1; j<=n; j++)
     if (w[k][j] && !z[j] && d[j] > d[k] + w[k][j] )
       { d[j] = d[k] + w[k][j];  p[j] = k;  }
   } // Кінець базового циклу
//------ Результати роботи алгоритму Дейкстри --------
   cout<<"****** Найкоротші шляхи ******\n";
   for(int i=1;i<=n;i++) {
    cout<<"до вершини "<<i<<": ";
    if(d[i]==UN) cout<<" немає шляху!"<<endl;
    else {
           int arr[N], k=1; int j=i;
           while(p[j]) { arr[k]=p[j]; k++; j=p[j]; }
           for(--k; k>0; k--) cout<<arr[k]<<"->";
           cout<<i; cout<<"; d="<<d[i]<<endl;
         } // else
   } // for
  }  // if(s_Path) …
  else cout<<"Немає шляхів від вершини "<<s<<endl;
 } // while(cout<<"Стартова вершина s="…
 cin>>n; // Пауза
 return 0;
}
```

Результати тестування програми

Приклад 3.1 з [5, с. 35]



```
E:\CPP_2012\Проекти_C++_2012\5_Алгоритми_На_М...
Кількість вершин n=7
Введення відстаней у матрицю w (0 - немає шляху):
від  1-ої вершини: 0 5 1 0 0 0 0
від  2-ої вершини: 0 0 0 7 1 6 0
від  3-ої вершини: 0 2 0 6 7 0 0
від  4-ої вершини: 0 0 0 0 0 4 6
від  5-ої вершини: 0 0 0 3 0 5 9
від  6-ої вершини: 0 0 0 0 0 0 2
від  7-ої вершини: 0 0 0 0 0 0 0
Стартова вершина s=1
****** Найкоротші шляхи ******
до вершини 1: 1; d=0
до вершини 2: 1->3->2; d=3
до вершини 3: 1->3; d=1
до вершини 4: 1->3->4; d=7
до вершини 5: 1->3->2->5; d=4
до вершини 6: 1->3->2->6; d=9
до вершини 7: 1->3->2->6->7; d=11
Стартова вершина s=2
****** Найкоротші шляхи ******
до вершини 1:  немає шляху!
до вершини 2: 2; d=0
до вершини 3:  немає шляху!
до вершини 4: 2->5->4; d=4
до вершини 5: 2->5; d=1
до вершини 6: 2->6; d=6
до вершини 7: 2->6->7; d=8
Стартова вершина s=7
Немає шляхів від вершини 7
Стартова вершина s=
```


5.5.6. Алгоритм Беллмана-Форда

Нехай маємо навантажений орієнтований граф $G = (V, E)$, в якому довжини дуг можуть бути *від'ємними* числами. Алгоритм Беллмана-Форда розв'язує задачу про визначення *найкоротших* шляхів у G від деякої стартової вершини $s \in V$ до всіх інших вершин, досяжних з s, якщо в графі G немає циклу від'ємної ваги, досяжного зі стартової вершини.

Під час роботи алгоритму для кожної вершини $i \in V$ підтримують дві величини:

- $d[i]$ – *пріоритетності* вершини;
- $p[i]$ – *предка* вершини.

Величина $d[i]$ дорівнює *верхній оцінці* довжини найкоротшого шляху від s до i. Початкові значення верхніх оцінок вершин $\forall i \in V\ (i \neq s): d[i] = +\infty;\ d[s] = 0$.

Величина $p[i]$ указує на вершину, що є початком дуги, направленої до вершини i з верхньою оцінкою $d[i]$. Початкові значення предків вершин $\forall i \in V: p[i] = 0$.

Алгоритм Беллмана-Форда:

1) $\forall i \in V$ ініціалізують величини $d[i]$, $p[i]$ (задають початкові значення);
2) у базовому циклі, що містить $(|V| - 1)$-ну ітерацію, на кожній ітерації послідовно переглядають і піддають *релаксації* кожну дугу графа;
3) послідовно переглядають кожну дугу графа (u, v) з метою перевірки виконання умови: $d[v] \le d[u] + w(u, v)$. Якщо ця умова виконується для *усіх* дуг, то алгоритм Беллмана-Форда повертає правильні результати (у графі G немає циклу від'ємної ваги, досяжного зі стартової вершини), у протилежному випадку – результати недостовірні.

Доведемо, що алгоритм Беллмана-Форда працює правильно.

Лема 4. Нехай у графі G немає циклів від'ємної ваги, досяжних з s. Після завершення роботи алгоритму Беллмана-Форда рівність $d[v] = \delta(s, v)$ виконуватиметься для усіх вершин v, досяжних з s.

➤ Нехай $p = \langle s = v_0, v_1, \ldots, v_k = v \rangle$ – найкоротший шлях з s у v, де $k \le |V| - 1$. Доведемо за індукцією, що після i-ої ітерації *базового* циклу буде виконано рівність $d[v_i] = \delta(s, v_i)$. Для $i = 0$ це очевидно, оскільки $d[s] = \delta(s, s) = 0$ при вході у цикл.

Нехай після $(i - 1)$-ої ітерації було $d[v_{i-1}] = \delta(s, v_{i-1})$. Під час i-ої ітерації відбудеться релаксація ребра (v_{i-1}, v_i), після чого, за теоремою 4, установиться рівність $d[v_i] = \delta(s, v_i)$.

Оскільки всього виконується $|V| - 1$ ітерація, то, за теоремою 4, рівність $d[v] = \delta(s, v)$ виконуватиметься для усіх вершин v, досяжних з s. ≺

Теорема 8. Якщо у графі G немає циклів від'ємної довжини, досяжних з s, то алгоритм Беллмана-Форда повертає правильні результати. Якщо ж у графі G є цикли від'ємної ваги, досяжні з s, то алгоритм не гарантує правильних результатів.

➤ Якщо у графі G немає циклів від'ємної ваги, досяжних з s, то, за лемою 4, після виконання алгоритму Беллмана-Форда рівність $d[v] = \delta(s, v)$ виконуватиметься для усіх вершин v, досяжних з s. Тоді, згідно з формулою (34), для довільної дуги (u, v) виконуватиметься умова: $d[v] \le d[u] + w(u, v)$. А це означає, що алгоритм поверне правильні результати.

Нехай у графі є цикл від'ємної ваги $\langle v_0, v_1, \ldots, v_k = v_0 \rangle$, досяжний з вершини s. У цьому випадку існує хоча б одна вершина $i \in V$, для якої $d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$. Доведемо *від протилежного*.

Нехай $d[v_i] \le d[v_{i-1}] + w(v_{i-1}, v_i)$, $i = \overline{1, k}$. Додаючи усі k нерівностей і скорочуючи на величину $\sum d[v_i]$ в обох частинах, одержимо $0 \le \sum_{i=1}^{k} w(v_{i-1}, v_i)$, що *суперечить* циклу з від'ємною вагою.

Виходить, що для деякого i маємо $d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$, а це означає, що алгоритм поверне недостовірні результати. ≺

Як і в алгоритмі Дейкстри, з метою уникнення зайвих обчислень (за відсутності дуг від стартової вершини) початкову релаксацію дуг вигляду (s, i), де $i \in V$, виконуватимемо перед *базовим* циклом перегляду та релаксації дуг.

Приклад 20. Скласти програму реалізації алгоритму Беллмана-Форда.

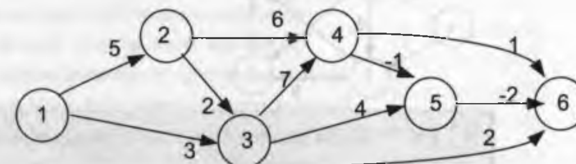
➤ *Програма*:

```
#include "windows.h"
#include <iostream>
using namespace std;
int main() {
 const int N = 20, UN = 10000; // UN - нескінченність
 int n,         // Кількість вершин
     w[N][N], // Матриця ваг
     s,         // Стартова вершина
     d[N],      // Вектор оцінок / відстаней до s
     p[N];      // Вектор предків
 bool Neg_Cycle=false; // Відсутність від'ємного циклу
 SetConsoleCP(1251); SetConsoleOutputCP(1251);
//------ Ініціалізація w i s ------------------
 cout<<"Кількість вершин n="; cin>>n;
 cout<<"Введення відстаней у матрицю w";
 cout<<" (0 - немає шляху):"<<endl;
 for (int i = 1; i <= n; i++)
 {
   cout<<"від "<<i<<"-ої вершини: ";
   for (int j = 1; j <= n; j++) cin>>w[i][j];
 }
 while(cout<<"Стартова вершина s=", cin>>s,s>=1 && s<=n){
//--------Ініціалізація d i p ---------------------
  int s_Path=0; // Відсутність дуг від вершини s
  for(int i=1;i<=n; i++)
  {
    p[i]=0;
    w[s][i]? (p[i]=s, d[i]=w[s][i], s_Path=1) : d[i] = UN;
  }
  d[s] = 0; // Оцінка для стартової вершини
```

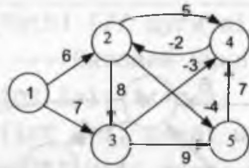
```
  if(s_Path) {// Дуги від s наявні -> реалізація алгоритму
//--------- Базовий цикл перегляду та релаксації дуг ----
   for(int i=1; i<=n-1; i++)
    for(int u=1; u<=n; u++)
     for(int v=1; v<=n; v++)
      if(w[u][v] && d[v]>d[u]+w[u][v])
       { d[v]=d[u]+w[u][v]; p[v]=u; } // Релаксація дуг
//--------Виведення результатів--------------------
   for(int u=1; (u<=n) && (!Neg_Cycle); u++)
    for(int v=1; (v<=n) && (!Neg_Cycle); v++)
     if(w[u][v] && d[v]>d[u]+w[u][v]) Neg_Cycle=true;
   if(Neg_Cycle) cout<<"Існує від'ємний цикл!"<<endl;
   else {   // * Правильні результати *
    cout<<"****** Найкоротші шляхи ******\n";
    for(int i=1;i<=n;i++) {
     if(d[i]) { // Вилучення фіктивної петлі
      cout<<"до вершини "<<i<<": ";
      if(d[i]==UN) cout<<"Немає шляху"<<endl;
      else { // Шлях наявний
       int arr[N], k=1; int j=i;
       while(p[j]) // Збереження вершин шляху в arr
        { arr[k]=p[j];  k++;  j=p[j]; }
       for(--k; k>0; k--) cout<<arr[k]<<"->";
       cout<<i; cout<<"; d="<<d[i]<<endl;
      }  // Шлях наявний
     }   // if(d[i])
    }    // for * Найкоротші шляхи *
   }     // else * Правильні результати *
  }      // if(s_Path)
  else cout<<"Немає шляхів від вершини "<<s<<endl;
 }       // while(cout<<"Стартова вершина s ...
 cin>>n;  // Пауза
 return 0;
}
```


Результати тестування програми

1) Приклад 3.4 з [5, с. 41]

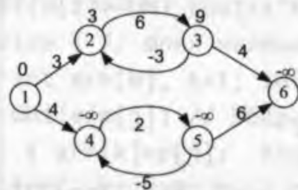



```
E:\CPP_2012\Проекти_C++_2012\5_Алгоритм...
Кількість вершин n=6
Введення відстаней у матрицю w (0 - немає шляху):
від  1-ої вершини: 0 5 3 0 0 0
від  2-ої вершини: 0 0 2 6 0 0
від  3-ої вершини: 0 0 0 7 4 2
від  4-ої вершини: 0 0 0 0 -1 1
від  5-ої вершини: 0 0 0 0 0 -2
від  6-ої вершини: 0 0 0 0 0 0
Стартова вершина s=1
****** Найкоротші шляхи ******
до вершини 2: 1->2; d=5
до вершини 3: 1->3; d=3
до вершини 4: 1->3->4; d=10
до вершини 5: 1->3->5; d=7
до вершини 6: 1->3->6; d=5
Стартова вершина s=2
****** Найкоротші шляхи ******
до вершини 1: Немає шляху
до вершини 3: 2->3; d=2
до вершини 4: 2->4; d=6
до вершини 5: 2->4->5; d=5
до вершини 6: 2->4->5->6; d=3
Стартова вершина s=5
****** Найкоротші шляхи ******
до вершини 1: Немає шляху
до вершини 2: Немає шляху
до вершини 3: Немає шляху
до вершини 4: Немає шляху
до вершини 6: 5->6; d=-2
Стартова вершина s=6
Немає шляхів від вершини 6
```



2) Рис. 25.7 з [24, с. 495]

```
E:\CPP_2012\Проекти_С++_2012\5_Алгорит...
Кількість вершин n=5
Введення відстаней у матрицю w (0 - немає шляху):
від  1-ої вершини: 0 6 7 0 0
від  2-ої вершини: 0 0 8 5 -4
від  3-ої вершини: 0 0 0 -3 9
від  4-ої вершини: 0 -2 0 0 0
від  5-ої вершини: 0 0 0 7 0
Стартова вершина s=1
****** Найкоротші шляхи ******
до вершини 2: 1->3->4->2; d=2
до вершини 3: 1->3; d=7
до вершини 4: 1->3->4; d=4
до вершини 5: 1->3->4->2->5; d=-2
Стартова вершина s=
```



3) Рис. 3.1 з [5, с. 30]

```
E:\CPP_2012\Проекти_С++_2012\5_Алг...
Кількість вершин n=6
Введення відстаней у матрицю w (0 - немає шляху
від  1-ої вершини: 0 3 0 4 0 0
від  2-ої вершини: 0 0 6 0 0 0
від  3-ої вершини: 0 -3 0 0 0 0
від  4-ої вершини: 0 0 0 0 2 0
від  5-ої вершини: 0 0 0 -5 0 0
від  6-ої вершини: 0 0 0 0 0 0
Стартова вершина s=1
Існує від'ємний цикл!
```


? Запитання для самоперевірки

1. *Дайте визначення точки глобального мінімуму.*
2. *Дайте визначення точки локального мінімуму*
3. *Запишіть задачу математичного програмування.*
4. *Охарактеризуйте задачу дискретної оптимізації.*
5. Охарактеризуйте задачу комбінаторної оптимізації.
6. Охарактеризуйте задачі, що перекриваються.
7. Що розуміють під принципом оптимальності?
8. *Охарактеризуйте метод динамічного програмування.*
9. *Охарактеризуйте висхідний метод динамічного програмування.*
10. *Охарактеризуйте низхідний метод динамічного програмування.*
11. Запишіть задачу про обмежений наплічник.
12. Запишіть задачу про 0-1 наплічник.
13. Що розуміють під релаксацією вершин графа?
14. Опишіть алгоритм Пріма.
15. Опишіть алгоритм Краскала.
16. Опишіть алгоритм Дейкстри.
17. Опишіть алгоритм Флойда-Уоршолла.
18. Опишіть алгоритм Форда-Беллмана.

Завдання для програмування

Завдання 5.1. Скласти програми розв'язання таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи), використавши метод динамічного програмування:

1. Дано послідовність матриць $A_1, A_2, \ldots A_n$. Необхідно мінімізувати кількість скалярних операцій для їхнього перемножування.
2. Дано дві послідовності чисел. Необхідно визначити найдовшу однакову підпослідовність.
3. Дано послідовність чисел. Необхідно визначити найдовшу зростаючу підпослідовність.
4. Команди A і B проводять серію деякої гри доти, доки одна з них не отримає n перемог (нічиї в грі не передбачені правилами). Імовірність перемоги команди A одна і та ж у кожній грі та дорівнює $p\ (0 \le p \le 1)$. Нехай $P(i, j)$ – імовірність перемоги команди A в серії ігор, якщо A до перемоги потрібно ще виграти i ігор, а $B - j$. Визначити імовірність перемоги команди A в серії ігор (базою має слугувати рекурентне співвідношення обчислення $P(i, j)$).
5. Визначити транзитивне замикання орграфа з n вершинами за допомогою алгоритму Уоршолла ([24, с. 520]).

6. Визначити мінімальну тріангуляцію багатокутника ([24, с. 305]).
7. Визначити мінімальну редакційну відстань ([24, с. 310]).
8. Визначити оптимальні терміни заміни обладнання ([5, с. 51]).
9. Визначити оптимальний розподіл інвестицій між агентами ([5, с. 54]).
10. Розв'язати бітонічну евклідову задачу комівояжера ([24, с. 309]).
11. Визначити оптимальне розбиття абзацу на рядки ([24, с. 310]).
12. Організація вечірки ([24, с. 311]).
13. Алгоритм Вітербі ([24, с. 312; варіант *а*]).
14. Алгоритм Вітербі ([24, с. 312; варіант *б*]).
15. Визначити оптимальну траєкторію підйому літака.

Завдання 5.2. Скласти програми розв'язання таких задач (задачу обрати згідно з номером студента у списку студентів підгрупи), використавши жадібний метод:

1. Визначити оптимальне пакування неперервного наплічника.
2. Дано *n* точок на координатній прямій $x_1, x_2, ... x_n$. Необхідно накрити всі ці точки найменшою кількістю відрізків одиничної довжини.
3. Побудувати оптимальний префіксний код Хаффмена ([24, с. 320]).
4. Розв'язати задачу про вибір заявок ([24, с. 313]).
5. Узагальнення алгоритму Хаффмена для тернарних кодів ([24, вправа 17.3-6]).
6. Розв'язати задачу про розклад ([24, с. 332]).
7. Розв'язати задачу про решту з одного долара ([24, с. 332; варіант *а*]).
8. Розв'язати задачу про решту з одного долара ([24, с. 333; варіант *б*]).
9. Розв'язати задачу про розклад за допомогою модифікованого алгоритму ([24, задача 17-3]).
10. Розв'язати задачу про побудову другого за величиною каркаса ([24, задача 24-1]).
11. Розв'язати задачу про побудову каркаса в розрідженому графі ([24, задача 24-2]).
12. Визначити найкоротші шляхи в ациклічному орграфі з використанням модифікованого алгоритму Дейкстри.
13. Визначити найдовші шляхи в ациклічному орграфі з використанням модифікованого алгоритму Дейкстри.
14. Розв'язати систему обмежень на різниці ([24, с. 499]).
15. Розв'язати модифіковану систему обмежень на різниці ([24, вправа 25.5-10]).

БІБЛІОГРАФІЧНИЙ СПИСОК

1. *Алексеев Е. Р.* Программирование на Microsoft Visual C++ и Turbo C++ Explorer / Е. Р. Алексеев; под общ. ред. О. В. Чесноковой. – М. : НТ Пресс, 2007.
2. *Архангельский А. Я.* Программирование в C++ Builder 5 / А. Я. Архангельский. – М. : БИНОМ, 2000.
3. *Ахо А.* Структуры данных и алгоритмы / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М. : Вильямс, 2003.
4. *Бартіш М. Я.* Дослідження операцій. Частина 1. Лінійні моделі : підручник / М. Я. Бартіш, І. М. Дудзяний. – Львів : Видавничий центр ЛНУ імені Івана Франка, 2007.
5. *Бартіш М. Я.* Дослідження операцій. Частина 2. Алгоритми оптимізації на графах : підручник / М. Я. Бартіш, І. М. Дудзяний. – Львів : Видавничий центр ЛНУ імені Івана Франка, 2007.
6. *Бартіш М. Я.* Дослідження операцій. Частина 3. Ухвалення рішень і теорія ігор : підручник / М. Я. Бартіш, І. М. Дудзяний. – Львів : Видавничий центр ЛНУ імені Івана Франка, 2009.
7. *Бобровский С. И.* Самоучитель программирования на языке C++ в системе Borland C++ Builder 4.0. / С. И. Бобровский. – М. : ДЕСС, 1999.
8. *Вайсфельд Мэтт.* Объектно-ориентированный подход: Java, .Net, C++ / Мэтт Вайсфельд. – М. : КУДИЦА-ОБРАЗ, 2005.
9. *Гамма Э.* Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб. : Питер, 2006.
10. *Гилберт С.* Самоучитель Visual C++ 6 в примерах / С. Гилберт, Б. Маккарти. – К. : ТИД ДС, 2003.
11. *Глушаков С. В.* Язык программирования C++ / С. В. Глушаков, А. В. Коваль, С. В. Смирнов. – Харьков : Фолио, 2003.
12. *Грэхем Р.* Конкретная математика. Основания информатики / Р. Грэхем, Д. Кнут, О. Паташник. – М. : Мир, 1998.
13. *Дейл Н.* Программирование на C++ / Н. Дейл, Ч. Уимз, М. Хедингтон. – М. : ДМК, 2000.
14. *Джамса К.* Учимся программировать на языке C++ / К. Джамса. – М. : Мир, 1999.
15. *Динман М. И.* C++. Освой на примерах / М. И. Динман. – СПб. : BXB-Петербург, 2006.

16. *Довбуш Г. Ф.* Visual C++ на примерах / Г. Ф. Довбуш, А. Д. Хомоненко. – СПб. : БХВ –Петербург, 2007.
17. *Домнин Л. Н.* Элементы теории графов / Л. Н. Домнин. – Пенза, 2004.
18. *Дудзяний І. М.* Програмування мовою Visual Basic .NET : навч. посібник / І. М. Дудзяний. – Львів : Видавничий центр ЛНУ імені Івана Франка, 2006.
19. *Зеленяк О. П.* Практикум программирования на Turbo Pascal. Задачи, алгоритмы и решения / О. П. Зеленяк. – К. : ДиаСофт, 2001.
20. *Калверт Ч.* Borland C++ Builder 3. Самоучитель / Ч. Калверт. – К. : ДиаСофт, 1999.
21. *Каррано Ф. М.* Абстракция данных и решения задач на C++. Стены и зеркала / Ф. М. Каррано, Дж. Причард. – М. : Вильямс, 2003.
22. *Кетков Ю. Л.* Практика программирования: Visual Basic, C++ Builder, Delphi / Ю. Л. Кетков, А. Ю. Кетков – СПб. : БХВ – Санкт-Петербург, 2002.
23. *Киммел П.* Borland C++ Builder 5 / П. Киммел. – СПб. : BHV – Санкт-Петербург, 1999.
24. *Кормен Т.* Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест. – М. : МЦНО, 1999.
25. *Костів О. В.* Методи розробки алгоритмів : тексти лекцій / О. В. Костів, С. А. Ярошко. – Львів : Видавничий центр ЛНУ імені Івана Франка, 2002.
26. *Крупник А.* Самоучитель C++ / А. Крупник. – СПб. : Питер, 2005.
27. *Культин Н. Б.* C++ Builder в задачах и примерах / Н. Б. Культин. – СПб. : БХВ – Петербург, 2005.
28. *Культин Н. Б.* Самоучитель C++ Builder / Н. Б. Культин. – СПб. : БХВ – Петербург, 2004.
29. *Левитин Ананий В.* Алгоритмы: введение в разработку и анализ / Ананий В. Левитин. – М. : Вильямс, 2006.
30. *Макконнелл Дж.* Основы современных алгоритмов : учеб. пособие / Дж. Макконнелл. – М. : Техносфера, 2004.
31. *Мейерс С.* Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов / С. Мейерс. – СПб. : Питер, 2006.
32. *Мухомортов В. В.* Объектно-ориентированное программирование, анализ и дизайн / В. В. Мухомортов, В. Ю. Рылов. – Новосибирск : Новософт, 2002.

33. *Наконечний С. І.* Математичне програмування : навч. посібник / С. І. Наконечний, С. С. Савіна. – К. : КНЕУ, 2003.
34. *Нікольський Ю. В.* Дискретна математика : підручник / Ю. В. Нікольський, В. В. Пасічник, Ю. М. Щербина. – К. : BHV, 2007.
35. *Павловская Т. А.* C++. Программирование на языке высокого уровня / Т. А. Павловская. – СПб. : Питер, 2003.
36. *Павловская Т. А.* C++. Объектно-ориентированное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2005.
37. *Павловская Т. А.* C/C++. Структурное программирование : практикум / Т. А. Павловская, Ю. А. Щупак. – СПб. : Питер, 2002.
38. *Прата Стивен.* Язык программирования C++. Лекции и упражнения / Стивен Прата. – СПб. : ДиаСофтЮП, 2003.
39. *Пышкин Е. В.* Основные концепции и механизмы объектно-ориентированного программирования / Е. В. Пышкин. – СПб. : БХВ – Петербург, 2005.
40. *Романов Е. Л.* Практикум программирования на C++ / Е. Л. Романов. – СПб. : БХВ – Петербург, 2004.
41. *Саттер Г.* Новые сложные задачи на C++ / Г. Саттер. – М. : Вильямс, 2005.
42. *Седжвик Р.* Фундаментальные алгоритмы на C++. Алгоритмы на графах / Р. Седжвик. – СПб. : ДиаСофтЮП, 2002.
43. *Седжвик Р.* Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск / Р. Седжвик. – СПб. : ДиаСофт-ЮП, 2002.
44. *Секунов Н. Ю.* Самоучитель Visual C++ / Н. Ю. Секунов. – СПб. : ВХВ - Петербург, 2002.
45. *Сергеев А. П.* Программирование в Microsoft Visual C++ 2005. Самоучитель / А. П. Сергеев, А. Н. Терен. – М. : Вильямс, 2006.
46. *Страуструп Б.* Язык программирования C++. Специальное издание / Б. Страуструп. – М. : БИНОМ, 2011.
47. *Топп У.* Структуры данных в C++ / У. Топп, У. Форд. – М. : БИНОМ, 2000.
48. *Уилсон М.* C++. Практический подход к решению проблем программирования / М. Уилсон. – М. : Вильямс, 2006.
49. *Ускова О. Ф.* Программирование алгоритмов обработки данных / О. Ф. Ускова. – СПб. : БХВ – Петербург, 2003.
50. *Франка П.* C++: учебный курс / *П.* Франка. – СПб. : Питер, 1999.
51. *Холзнер С.* Visual C++ 6 : учебный курс / С. Холзнер. – СПб. : Питер, 2007.

52. *Хомоненко А. Д.* Программирование на C++ / А. Д. Хомоненко. – СПб. : КОРОНА принт, 1999.
53. *Хортон А.* Visual C++ 2005 : базовый курс / А. Хортон. – М. : Вильямс, 2007.
54. *Шамис В.* Borland C++ Builder 5: учебный курс / В. Шамис. – СПб. : Питер, 2002.
55. *Шаповаленко В. А.* Чисельне обчислення функцій, характеристик матриць і розв'язування нелінійних рівнянь та систем рівнянь : навч. посібник / В. А. Шаповаленко, Л. М. Буката, О. Г. Трофименко. – Одеса : ВЦ ОНАЗ, 2010. – Ч. 1.
56. *Шеферд Дж.* Программирование на Microsoft Visual C++ .NET. Мастер-класс / Дж. Шеферд. – СПб. : Питер, 2005.
57. *Шилдт Г.* Самоучитель C++. 3-е издание / Г. Шилдт. – СПб. : BHV – Санкт-Петербург, 1998.
58. *Шилдт Г.* C++: базовый курс. 3-е издание / Г. Шилдт. – М. : Вильямс, 2010.
59. *Шилдт Г.* Справочник программиста по C/C++ / Г. Шилдт. – М. : Вильямс, 2001.
60. *Элджер Дж.* C++: библиотека программиста / Дж. Элджер. – СПб. : Питер, 1999.
61. *Якушев Д.* Философия программирования на языке C++ / Д. Якушев. М. : Бук-пресс, 2006.

ПРЕДМЕТНИЙ ПОКАЖЧИК

A ... Z

ANSI C 31
ANSI 15
ASCII 15
C++/CLI 36
CLR 36
Common Language Infrastructure 43
Common Language Runtime 36
Common Types System 102
CTS 102
ECMA 43
European Association for Standardizing Information and Computer Systems 43
FCL 36
Framework Class Library 36
IDE 24
IL 43
Integrated Development Environment 24
ISO/ANSI 36
Just-In-Time Compiler 44
Manager Extension for C++ 45
Metadata System 43
Microsoft Intermediate Language 43
MSIL 43
.NET Framework 36
Object Browser 46
Portable Executable 43
UNIX 31
VES 43
Virtual Execution System 43
Windows Forms 36

А

Алгоритм
– Беллмана-Форда 451
– бінарного пошуку в масиві 321
– вибору сортування масиву 308
– включення сортування масиву 309
– Дейкстри 446
– Евкліда визначення НСД 222
– Ератосфена визначення простих чисел 226
– Краскала 441
– обміну сортування масиву 308
– поділу відрізка навпіл 182
– послідовного пошуку в масиві 321
– Пріма 437
– Флойда-Уоршолла 424

Алфавіт мови 51

Б

Бібліотека
– С-функцій 21
– класів C++ 21
– стандартних функцій 21

Блок 11
Буфер 32

В

Визначення сутності 11, 21, 54
Визначення синонімів типів 66
Вираз 67
Вказівники
– на об'єкти 263
– на масиви 264
– на функції 263, 271, 280, 281, 282
– типу `void` 263

Г

Глибина рекурсії 208
Графи
– ациклічні 417
– зважені 418
– неорієнтовані 415
– орієнтовані 416

– прості 414

Д

Ділення з остачею 219
Директива передпроцесора 11
Директиви передпроцесора
– `define` 27, 28
– `elif` 29
– `else` 29
– `endif` 29
– `if` 29
– `ifdef` 29, 30
– `ifndef` 29, 30
– `include` 16, 25, 26
– `pragma` 30
Динамічний розподіл пам'яті 265

З

Задача
– відбору у розвідку 203
– визначення сусіднього розміщення елементів масиву 313
– злиття двох упорядкованих масивів 320
– Йосипа 201
– комбінаторної оптимізації 374
– лінійного програмування 374
– математичного програмування 373
– нелінійного програмування 374
– оптимізації 368
– параметричного програмування 374
– переміщення елементів масиву 316
– про обмежений наплічник 407
– про 0-1 наплічник 409
– стохастичного програмування 374
– умовної оптимізації 371
Заголовок 11
Змінна 60
Змінні
– автоматичні (`auto`) 63, 64
– зовнішні (`extern`) 64, 65
– зовнішня циклу 137
– локальні 62, 64, 65
– область дії 62, 84
– регістрові (`register`) 63
– статичні (`static`) 63, 64, 65
– час життя (існування) 62, 64

І

Ідентифікатор 52
Ініціалізація
– змінної 60
– константи 61
Інструкція 11
Інструкції
– `break` 134
– `goto` 123
– `continue` 134
– `return` 123
– `using` 17
– виразу 122
– вибору 128
– галуження 125
– циклу `do … while` 135
– циклу `for` 136
– циклу `while` 133

К

Ключові слова 52
Код
– керований 36, 37, 43
– некерований 36, 37, 38
Коментар 12
Консольне застосування 15
Константа 61
Конструкції
– вибору 128
– галуження 125
– циклу `do … while` 135
– циклу `for` 136
– циклу `while` 133

Л

Лексема 10, 52
Літерал 53
Літерали
– числові 57, 58, 59
– рядки символів 57
– символьні 57, 58, 59
Логічна структура консольної програми 10

М

Маніпулятори
– без параметрів 34, 35
– з параметрами 34, 35
Маніфест 43
Масиви
– багатовимірні 260
– ініціалізація 254
– логічний рівень 252
– одновимірні 252
– фізичний рівень 252
Метод
– вибору сортування масиву 308
– включення сортування масиву 309
– вичерпного перебирання 184
– гілок і меж 185
– грубої сили 184
– декомпозиції 186
– динамічного програмування 186, 378, 382
– жадібний 184, 433
– зменшення розрізу задачі 185, 198
– обміну сортування масиву 308
– перетворення 186
– послідовного поліпшення розв'язку 185
– пошуку з поверненням 184
Механізм форматування CLR 105
Модифікатори
– `const` 13
– `endl` 19
Модуль
– виконавчий 21, 23, 42
– вихідний (початковий) 21, 22
– об'єктний 21, 23, 42

Н

Найбільший спільний дільник (НСД) 217, 222
Найменше спільне кратне (НСК) 218
Налагодження програми 76

О

Область дії об'єкта 124
Оголошення сутності 11, 21, 53
Оператор 53
Оператори
– арифметичні 67
– виведення даних у вихідний потік (<<) 18
– визначення розміру 70
– витягання даних з потоку (>>) 19
– логічні 70
– логічні порозрядні 70
– постфіксні 69
– префіксні 69
– присвоювання 67
– розширення області видимості 17
– явного перетворення типу 70

П

Перелік конструкцій та інструкцій 122
Потік 32
Потоки
– вихідні 33
– вхідні 33
– двоспрямовані 33
– стандартні 33
Пріоритет
– назв 62
– операторів 68

Програми
– багатомодульних проектів 288, 300, 301, 306, 322
– визначення НСД і НСК 222
– визначення простих чисел 226, 228
– визначення сусіднього розміщення елементів масиву 313
– визначення характеристик математичного об'єкта з формулюванням обмежень на вхідні дані 82
– визначення характеристик математичного об'єкта з формулюванням обмежень на вхідні дані та функцією користувача 91, 97, 99
– виокремлення та опрацювання цифр числа 233, 234, 235
– ділення з остачею 220, 221
– злиття двох упорядкованих масивів 320
– з галуженням 81, 121
– з галуженням та функцією користувача 95
– з конструкцією вибору 130
– з простим повторенням 105, 138, 139, 140, 144, 145, 146
– з рекурентними співвідношеннями 147, 148, 155, 156, 159
– з рекурентними співвідношеннями функціональними 187, 188, 189, 192, 194, 198, 200, 202, 204, 210, 211, 212, 214
– лінійні 15, 18, 19, 49, 79, 89
– лінійні з функцією користувача 86, 94
– обробки масивів 255, 256, 257, 259, 261, 267, 269, 272, 274, 276, 278, 303, 304
– обробки рядків 327, 329, 332, 333, 340, 341, 343, 345, 348, 349, 352, 354, 356, 357, 358, 359, 360
– обчислення визначених інтегралів 174, 175, 176, 179, 279
– обчислення добутків 151, 152, 212, 215, 216
– обчислення значень многочлена за схемою Горнера 305
– обчислення максимуму трьох чисел 126
– обчислення рядів степеневих 169
– обчислення рядів числових 159, 162, 163, 164
– обчислення сум 149, 150, 152, 216
– переміщення елементів масиву 316
– пошуку в масиві 321
– розв'язування задач жадібним методом 438, 444, 448, 453
– розв'язування задач методом динамічного програмування 380, 382, 384, 385, 388, 391, 394, 401, 404, 410, 412, 424
– розв'язування рівняння з однією змінною 182
– розкладання натурального числа на прості множники 231
– сортування масиву 309
Проект 37, 40
Простір назв
– `std` 17
– визначення 17
– глобальний 18
– складених модулів 46, 47
Прототип функції 85

Р

Рекурентні співвідношення
– задачі Йосипа 202
– задачі відбору у розвідку 204, 205
– найпростіші 153, 154
– обчислення біноміальних коефіцієнтів 193, 194
– обчислення степеня 191, 198
– обчислення факторіала 155, 187, 214
– обчислення чисел Фібоначчі 155, 189, 190
Рішення 37
Роздільне компілювання модулів 285
Рядки символів
– ініціалізація 326
– у стилі C 325

С

Складений модуль 46
Специфікація програми 75
Спільна система типів 102
Список параметрів функції 84
Стандартний потік
– введення (`cin`) 19
– виведення (`cout`) 18
Стек 207
Сутність програми 11
Сутність-об'єкт 13
Сутність-суб'єкт 13, 14

Т

Теорема
– Абеля 167
– Вейєрштрасса 369
– Д'Аламбера 161
– Лейбніца 162
Тип даних 54, 102
Типи даних
– базові 54
– вбудовані 54
– логічний 55
– перетворення 72
– посилання 103
– похідні 54
– раціональний 55
– розширений символьний 54
– символьний 54
– стандартні 54
– структурні 103
– фундаментальні 54
– цілий 55
Тіло функції 84

Ф

Файл
– PE-файл 43
– визначення 32
– двійковий (бінарний) 32
– реалізації 21
– специфікацій 21
– текстовий 32
Фізична структура програми 10, 21, 22
Фрагмент програми
– з конструкцією вибору 131, 132
– обчислення максимуму двох чисел 125
– обчислення факторіала 133, 135, 136
Функція 12, 84
Функції
– визначення літер кирилиці 339
– головна 12, 16, 17
– класифікації символів 351
– обробки рядків 330
– стандартні 21

Ч

Час життя об'єкта 124
Числа
– псевдовипадкові 141
– Фібоначчі 148, 155, 189, 380

Навчальне видання

Дудзяний Ігор Михайлович

Програмування мовою С++

Частина 1

Парадигма процедурного програмування

Навчальний посібник

Редактор *І. Лоїк*
Технічний редактор *С. Сеник*
Коректор *Ю. Бурка*
Комп'ютерне макетування *І. Дудзяний*
Обкладинка *В. Роган*

Формат 60×84/16. Ум. друк. арк. 13,8. Тираж 300 прим. Зам.С-452

Львівський національний університет імені Івана Франка
вул. Університетська, 1, м. Львів, 79000

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції : Серія ДК № 3059 від 13.12.2007 р.

Віддруковано у книжковій друкарні "Коло"
вул. Бориславська, 8, м. Дрогобич, Львівська обл., 82100

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців, виготівників і розповсюджувачів видавничої продукції : Серія ДК № 498 від 20.06.2001 р.

НБ ПНУС

788344