Міністерство освіти і науки України
ДВНЗ “Прикарпатський національний університет імені Василя Стефаника”
Кафедра комп’ютерної інженерії та електроніки

Курс лекцій з дисципліни
“ПРИКЛАДНІ ПРОГРАМИ КОМП’ЮТЕРНОЇ ІНЖНЕРІЇ”

Затверджено
на засіданні кафедри радіофізики і електроніки
протокол № 4 від 8 листопада 2017 р

Розроблено:
доц. Голота В.І.

Івано-Франківськ 2017

Зміст

1. ОСНОВИ QT

Мета. Ознайомлення з основами QT і структурою його класів.

Вступ.

Бібліотека Qt призначена для створення графічного інтерфейсу користувача. Qt – повністю об'єктно-орієнтована бібліотека. Нова концепція міжоб'єктних комунікацій, на основі "сигналів і слотів", повністю замінює стару, не цілком надійну модель зворотних викликів. Також є можливість оброблення подій, наприклад натиснення клавіш клавіатури, переміщення миші і т. д. Система розширень (plug-ins) дозволяє створювати модулі, що розширюють функціональні можливості застосунків. Ці розширення користувачі програм можуть отримувати не лише від головних, але і від інших розробників.

Незважаючи на те, що бібліотека Qt спочатку створювалася для мови програмування C++, вона використовується і в інших мовах. Так, існують модулі для роботи з бібліотекою: Qt Jambi в Java, Qt# в C#, PerlQt в Perl, PyQt в Python, PHP і т. д.

Програми, реалізовані за допомогою Qt, можуть використати мову сценаріїв Qt Script. Ця технологія дозволяє користувачам застосунку розширити його можливості без зміни початкового коду і без перекомпонування самого застосунку.

Qt прекрасно документована, завдяки чому можна завжди отримати будь-яку інформацію про цю бібліотеку, використовуючи програму Qt Asisstant. Крім того, Qt – це бібліотека з відкритим початковим кодом (Open Source), і можна завжди його переглянути і детально розібратися у тому, як працює та або інша частина цієї бібліотеки.

І якщо бути гранично коротким, то бібліотеку Qt можна охарактеризувати в трьох словах: Простота + Швидкість + Потужність.

План.

1 Огляд ієрархії класів Qt
2 Філософія об'єктної моделі
3 Робота з Qt
4 Бібліотека контейнерів

1 Огляд ієрархії класів Qt

Перша програма на Qt

Як прийнято при вивченні мов програмування, спочатку пишеться програма для виведення вітання "Hello World" ("Привіт Світ"), результат виконання якої показаний на рис. 1.1.

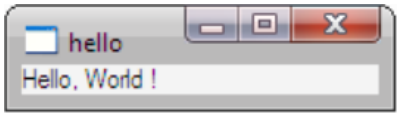


Рисунок 1.1 – Вікно програми "Hello World"

Роздрук 1.1. Програма "Hello World". Файл hello.cpp

```
1 #include <QtGui>          // Qt 4.8
//#include <QtWidgets>        Qt 5.3
2 int main(int argc, char** argv)
3 {
4     QApplication app(argc, argv);
5     QLabel lbl("Hello, World "!);
```

```
6     lbl.show();
7     return app.exec();
8 }
```

У першому рядку роздруку 1.1 підключається заголовковий файл `QtGui`, який є файлом модуля, що включає заголовкові файли для використовуваних в програмі класів: `QApplication` і `QLabel`. Звичайно, можна б обійтися і без модуля QtGui, а безпосередньо підключити заголовкові файли для підтримки класів `QApplication` і `QLabel`, але при більшій кількості класів різних модулів, задіяних в програмі, прочитуваність самої програми помітно б погіршала. Крім того, підключення модулів дає можливість прискорити компіляцію самої програми за рахунок заголовкових файлів (Precompiled Headers), що заздалегідь відкомпілювалися, у тому випадку, якщо ваш компілятор дозволяє це робити.

Тепер розберемо приклад. У четвертому рядку створюється об'єкт класу `QApplication`, який здійснює контроль і керування застосунком. Для його створення в конструктор цього класу необхідно передати два аргументи. Перший аргумент є інформацією про кількість аргументів в командному рядку, з яким відбувається звернення до програми, а другий – це вказівник на масив символьних рядків, що містять аргументи, по одному в рядку. Люба Qt-програма з графічним інтерфейсом повинна створювати тільки один об'єкт цього класу, і він має бути створений до використання операцій, пов'язаних з користувацьким інтерфейсом.

У п'ятому рядку створюється об'єкт класу `QLabel`. Після створення елементи керування Qt за замовчуванням невидимі, і для їх відображення викликається метод `show()` (у данному випадку п'ятий рядок – `lbl.show()`). Об'єкт класу `QLabel` є основним керуючим елементом застосунку, який дозволяє завершити роботу застосунку при закритті вікна елементу. Якщо раптом виявиться, що в створеному застосунку є відразу декілька незалежних один від одного елементів керування, то при закритті вікна останнього такого елементу керування завершиться і сам застосунок. Це правильно, інакше застосунок залишився б в пам'яті комп'ютера і використав би його ресурси.

Нарешті, в сьомому рядку рядку програми застосунок запускається викликом `app.exec()`. З його запуском приводиться в дію цикл обробки подій, визначений в класі `QCoreApplication`, який є базовим для `QApplication`. Цей цикл передає отримувані від системи події на обробку відповідним об'єктам. Він триває до тих пір, поки або не буде викликаний статичний метод `QCoreApplication::exit()`, або не закриється вікно останнього елементу керування. Після завершення роботи застосунку метод `QApplication::exec()` повертає значення цілого типу, що містить код, який інформує про його завершення.

Модулі Qt

Ієрархія класів Qt має чітку внутрішню структуру, яку важливо зрозуміти, щоб уміти добре і інтуїтивно орієнтуватися в цій бібліотеці.

Бібліотека Qt – множина класів (більше 500), які охоплюють велику частину функціональних можливостей операційних систем, надаючи розробникові потужні механізми, що розширюють і, в той же час, спрощують розробку застосунків. При цьому не порушується ідеологія операційної системи. Qt не є єдиним цілим, вона розбита на модулі (табл. 1.1).

Таблиця 1.1 – Модулі Qt

Бібліотека	Позначення у проектному файлі	Призначення
QtCore	core	Основний модуль, що складається з класів не пов'язаних з графічним інтерфейсом
QtGui	gui	Модуль базових класів для програмування графічного інтерфейсу

QtWidgets	widgets	Модуль, доповнюючий QtGui ”будівельним матеріалом” для графічного інтерфейсу у вигляді віджетів C++
QtQuick	quick	Є описовим середовищем для швидкого створення графічного ынтерфейсу
QtQml	qml	Модуль який має двигун для мови QML і JavaScript
QtNetwork	network	Модуль для програмування мережі
QtOpenGL	opengl	Модуль для програмування графіки OpenGL
QtSql	sql	Модуль для програмування баз даних
QtSvg	Svg	Модуль для роботи з SVG (масштабована векторна графіка)
QtXml	xml	Модуль підтримки XML, класи, які відносяться до SAX і DOM
QtXmlPatterns	xmlpatterns	Модуль підтримки XPath, XQuery, XSLT, XmlShemaValidator
QtScript	script	Модуль підтримки мови сценаріїв
QtScriptTools	scripttools	Модуль додаткових можливостей підтримки мови сценаріїв
QtMultimedia	multimedia	Модуль мультимедіа
QtMultimedia-Widgets	Multimedia-widgets	Модуль з віджетами для QtMultimedia
QtWebKit	webkit	Модуль для створення web застосунків
QtWebKit-Widgets	webkit	Модуль з віджетами WebKit
QPrintSupport	printsupport	Модуль для роботи з прінтером
QtTest	test	Модуль, що містить класи для тестування коду

Будь-яка Qt-програма так чи інакше повинна використати хоча б один з модулів, у більшості випадків це `QtCore`, `QtGui` і `QtWidget`, тому ці модулі визначені в програмі створення `make`-файлів (див. розділ 3) за замовчуванням. Для використання інших модулів у своїх проектах необхідно перерахувати їх в проектному файлі (див. розділ 3). Наприклад, щоб додати модулі, треба написати

```
QT += widgets opengl network sql
```

А щоб виключити модуль з проекту:

```
QT -= gui
```

Найбільш важливий з перерахованих в табл. 1.1 модулів – це `QtCore`, так як він є базовим для усіх інших модулів (рис. 1.2). Далі йдуть модулі, які безпосередньо залежать від `QtCore`, це – `QtNetwork`, `QtGui`, `QtSql` і `QtXml`. І, нарешті, модулі, залежні від тільки що згаданих модулів.

Для кожного модуля Qt надає окремий заголовковий файл, який містить заголовкові файли усіх класів цього модуля. Назва цього заголовкового файлу відповідає назві самого модуля. Наприклад, для включення QtGui модуля треба додати в програму рядок

```
#include <QtGui>          // Qt 4.8
#include <QtWidgets>      // Qt 5.3
```

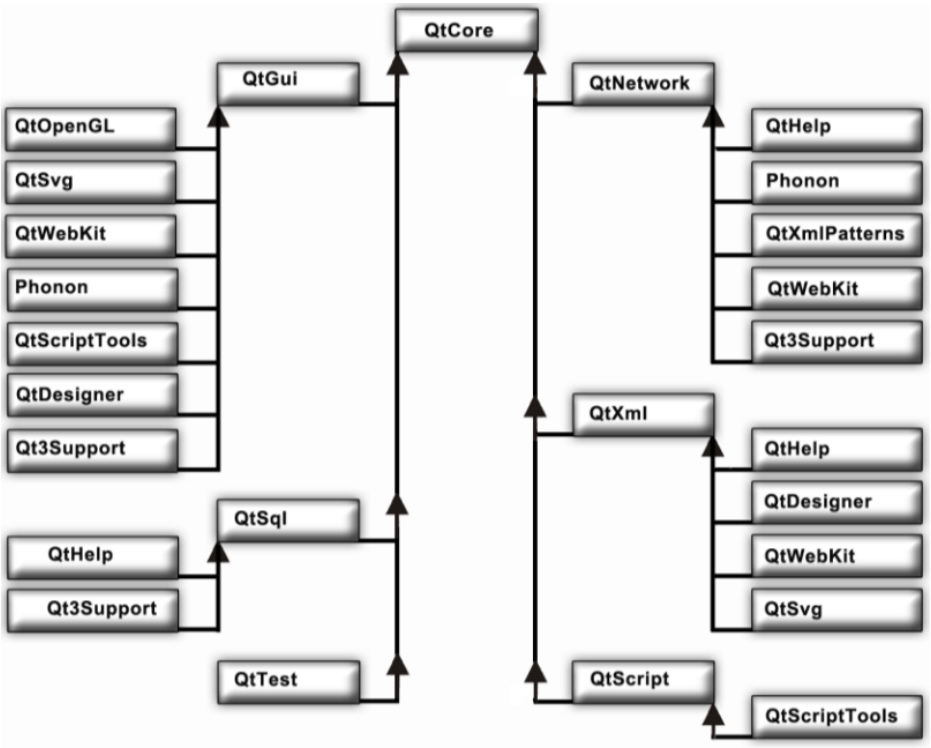


Рисунок 1.2 – Діаграма деяких модульних залежностей

Простір імен Qt

Простір імен Qt містить ряд типів перерахувань і констант, які часто застосовуються при програмуванні. Якщо необхідно отримати доступ до якої-небудь константи цього простору імен, то потрібно вказати префікс Qt (наприклад, не red, а Qt::red). Якщо все-таки хочеться опускати префікс Qt, то необхідно на початку файлу з початковим кодом додати наступну директиву:

```
using namespace Qt;
```

Модуль QtCore

QtCore є базовим модулем для застосунків і не містить класів, які відносяться до інтерфейсу користувача. При реалізації консольного застосунку можна обмежитися одним цим модулем. У модуль QtCore входять більше 200 класів, ось деякі з них:

• контейнерні класи `QList`, `QVector`, `QMap`, `QVariant`, `QString`;

• класи для введення і виведення `QIODevice`, `QTextStream`, `QFile`;

• класи процесу `QProcess` і для програмування багатопоточності `QThread`, `QWaitCondition`, `QMutex`;

• класи для роботи з таймером `QBasicTimer` і `QTimer`;

• класи для роботи з датою і часом `QDate` і `QTime`;

• клас `QObject`, що є наріжним каменем об'єктної моделі Qt;

• базовий клас подій `QEvent`;

• клас для збереження налаштувань застосунка `QSettings`;

• клас застосунку `QCoreApplication`, з об'єкту якого, якщо потрібно, можна запустити цикл подій.

Об'єкт класу застосунку `QCoreApplication` можна образно порівняти з посудиною, що містить об'єкти, приєднані до контексту операційної системи. Термін життя об'єкту класу `QCoreApplication` відповідає тривалості роботи усього застосунку, і він залишається

доступним у будь-який момент роботи програми. Об'єкт класу QCoreApplication повинен створюватися в застосунку тільки один раз. До завдань цього об'єкту можна віднести:
- керування подіями між застосунком і операційною системою;
- передачу і надання аргументів командного рядка.

Крім того, QCoreApplication можна успадкувати, для того, щоб перезаписати деякі методи, а також задіяти сам об'єкт для додаткових глобальних даних, використовуваних усередині застосунку. Такий підхід дозволяє уникнути небажаного використання шаблону проектування Singleton.

• класи підтримки анімації: QAbstractAnimation, QVariantAnimation і т. д.;

• класи для машини станів: QStateMachine, QState і т. д.;

• класи моделей інтерв'ю: QAbstractitemModel, QStringListModel, QAbstractProxyModel.

Модуль також містить механізми підтримки файлів ресурсів.

Модуль QtGui

Цей модуль надає класи інтеграції з віконною системою, з OpenGL і OpenGL ES. Він містить клас QWindow, який є елементарною областю з можливістю отримання подій користувацького введення, зміни фокусу і розмірів, а також дозволяючій робити графічні операції і малювання на своїй поверхні.

Модуль застосовується до класу QGuiApplication. Цей клас містить механізм циклу подій і має також можливості:

• отримання доступу до буфера обміну;

• ініціалізації необхідних налаштувань застосунку – наприклад, палітри для забарвлення елементів керування;

• керування формою курсора миші.

Модуль QtWidgets

Цей модуль містить в собі класи віджетів для програмування графічного інтерфейсу користувача. У цей модуль входять близько 300 класів. Ось деякі з них:

• клас QWidget – це базовий клас для усіх елементів керування бібліотеки Qt. За своїм зовнішнім виглядом це не що інше, як заповнений чотирикутник, але за цією зовнішньою простотою ховається великий потенціал непростих функціональних можливостей. Цей клас нараховує 254 методи і 53 властивості;

• класи для автоматичного розміщення елементів QVBoxLayout, QHBoxLayout;

• класи елементів відображення QLabel, QLCDNumber;

• класи кнопок QPushButton, QCheckBox, QRadioButton;

• класи елементів повзунків QSlider, QScrollBar;

• класи елементів введення QLineEdit, QSpinBox;

• класи елементів вибору QComboBox, QToolBox;

• класи меню QMainWindow і QMenu;

• класи вікон повідомлень і діалогових вікон QMessageBox, QDialog;

• класи для малювання QPainter, QBrush, QPen, QColor;

• класи для растрових зображень QImage, QPixmap;

• класи стилів QMotifStyle, QWindowsStyle та ін. Як окремому елементу так і усьому застосунку може бути присвоєний певний стиль, що змінює їх зовнішній вигляд;

• клас застосунку QApplication, який задає цикл подій.

Все, що було сказано вище про клас QCoreApplication, відноситься також і до класу QApplication, так як він є його нащадком. Об'єкт класу QApplication є центральним контрольним пунктом усіх Qt-застосунків, що мають користувацький інтерфейс. Цей об'єкт використовується для отримання подій клавіатури, миші, таймера і інших подій, на які

застосунок повинен реагувати відповідним чином. Наприклад, вікно навіть найпростішого застосунку може бути змінено за величиною або бути перекрито вікном іншого застосунку, і на усі подібні події потрібна правильна реакція.

Класс `QApplication` напряму успадкований від класу `QGuiApplication` і доповнює його наступними можливостями:

• установка стилю застосунка. Таким чином можна встановлювати стиль `Motif`, `Windows`, а також багато інших видів і поведінку (Look & Feel) застосунку, включаючи і свої власні;

• отримання вказівника на об'єкт робочого столу(desktop);

• отримання доступу до буфера обміну;

• керування глобальними маніпуляціями з мишею (наприклад, установка інтервалу подвійного клацання кнопкою миші) і реєстрація руху миші в межах і за межами вікна застосунку;

• видача попереджуючого звукового сигналу;

• забезпечення правильного завершення працюючого застосунку при завершенні роботи операційної системи;

• ініціалізація необхідних налаштувань додатку, наприклад палітри для забарвлення елементів керування.

Буває так, що застосунок може бути неактивним, а є необхідність привернути до себе увагу користувача. Для цієї мети клас `QApplication` надає статичний метод `alert()`. Його виклик приведе до пульсації значка додатку на панелі завдань в ОС Windows.

Модулі QtQuick і QtQML

Це альтернатива віджетам – модулі є набором технологій для швидкої розробки графічних інтерфейсів нового покоління на базі описової мови QML, мови програмування JavaScript і усіх інших можливостей бібліотеки Qt.

Модуль QtNetwork

Мережевий модуль надає інструментарій для програмування TCP- і UDP-сокетів (класи `QTcpSocket` і `QUdpSocket`), а також для реалізації програм-клієнтів, які використовують HTTP- і FTP- протоколи (класи `QHttp` і `QFtp`).

Модуль QtXml і Qtxm/Patterns

Цей модуль призначений для роботи з XML за допомогою SAX2- і DOM-інтерфейсів, які визначають класи Qt. А модуль QtXmlPatterns йде далі і надає підтримку для додаткових технологій XML – таких як: XPath, XQuery, XSLT і XmlShemaValidator.

Модуль QtSql

Цей модуль призначений для роботи з базами даних. У нього входять класи, які надають можливість для маніпулювання значеннями баз даних.

Модуль QtOpenGL

Модуль `QtOpenGL` дозволяє використовувати `OpenGL` в Qt-програмах для дво- і тривимірної графіки. Основним класом цього модуля є `QGLWidget`, який успадкований від `QWidget`.

Модуль QtWebKit

Цей модуль дозволяє дуже просто інтегрувати в застосунок зміст з Web. Можливо також розширювати елементи Web своїми власними віджетами.

Модулі QtWebKit і QtWebKitWidgets

Модуль `QtWebKit` дозволяє дуже просто інтегрувати в застосунок можливості Web. А модуль `QtWebKitWidgets` надає готові до інтеграції в застосунок елементи у вигляді віджетів з можливістю також розширювати елементи Web своїми власними віджетами.

Модулі QtMultimedia і QtMultimediaWidgets

Модуль `QtMultimedia` має усе необхідне для створення застосунків з підтримкою мультимедіа. Він підтримує як низький рівень, необхідний для детальнішої спеціалізованої реалізації, так і високий рівень, що робить можливим програвати відео- і звукові файли за допомогою всього декількох рядків програмного коду. Модуль `QtMultimediaWidgets` містить корисні елементи у вигляді віджетів, які дозволяють економити час для реалізації.

Модулі QtScript і QtScriptTools

Модуль `QtScript` надає можливості розширення і зміни вже написаних застосунків за допомогою мови сценаріїв JavaScript. Модуль `QtScriptTools` забезпечує засоби налагодження для програм сценаріїв.

Модуль QtSvg

Модуль підтримки графічного векторного формату SVG, що базується на XML. Цей формат надає можливість не лише для виведення одного кадру векторного зображення, але може бути використаний і для векторної анімації.

2 Філософія об'єктної моделі

Об'єктна модель Qt передбачає, що все побудовано на об'єктах. Фактично, клас `QObject` – основний, базовий клас. Переважна більшість класів Qt є його нащадками. Класи, що мають сигнали і слоти, мають бути успадковані від цього класу.

Примітка. При множинному успадкуванні важливо пам'ятати, що при визначенні класу ім'я класу `QObject` (або успадкованого від нього) повинне стояти першим, щоб MOC (Meta Object Compiler, метаоб'єктний компілятор) міг правильно розпізнати його. Інший порядок спричинить помилку при компіляції. У роздруку 2.1 показаний правильний порядок для множинного успадкування.

Роздрук 2.1. Порядок успадкування

```
class MyClass : public QObject, public AnotherClass {
...
};
```

Примітка. При множинному успадкуванні також важливо враховувати, що від класу `QObject` має бути успадкований тільки один з базових класів. Іншими словами, не можна успадковувати відразу від декількох класів, що успадковують клас `QObject`.

Клас `QObject` містить в собі підтримку:
- сигналів і слотів (signal/slot);
- таймера;
- механізму об'єднання об'єктів в ієрархії;
- подій і механізму їх фільтрації;
- організації об'єктних ієрархій;
- метаоб'єктної інформації;
- перетворення типів;
- властивостей.

Сигнали і слоти – це засоби, що дозволяють ефективно здійснювати обмін інформацією про події, які виробляються об'єктами.

Підтримка таймера дає можливість кожному з класів, успадкованих від класу `QObject`, не створювати додатково об'єкта таймера. Тим самим економиться час на розробку.

Механізм об'єднання об'єктів в ієрархічні структури дозволяє різко скоротити часові витрати при розробці застосунків, не думаючи про звільнення пам'яті створюваних об'єктів, так як об'єкти-предки самі відповідають за знищення своїх нащадків.

Механізм фільтрації подій дозволяє здійснити їх перехоплення. Фільтр подій може бути встановлений у будь-якому класі, успадкованому від класу `QObject`, завдяки чому можна змінювати реакцію об'єктів на події, що відбуваються, без зміни початкового коду класу.

Метаоб'єктна інформація включає інформацію про успадкування класів, що дозволяє визначати, чи є класи безпосередніми нащадками, а також взнати ім'я класу.

Для перетворення типів Qt надає шаблонну функцію `qobject_cast<T>()`, яка основується на метаінформації, що створюється метаоб'єктним компілятором MOC, для класів, успадкованих від `QObject`.

Властивості – це поля, для яких обов'язково повинні існувати методи читання. За їх допомогою можна отримати доступ до атрибутів об'єктів ззовні, наприклад з мови сценаріїв Qt Script. Властивості також широко використовуються у візуальному середовищі розробки призначеного для користувача інтерфейсу Qt Designer. Цей механізм реалізований в Qt за допомогою директив препроцесора. Задається властивість за допомогою макросу `Q_PROPERTY`. Визначення властивості в загальному вигляді виглядає таким чином:

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
          )
```

Першими задаються тип і ім'я властивості, другим – ім'я методу читання (`READ`). Визначення інших параметрів не є обов'язковим. Третій параметр задає ім'я методу запису (`WRITE`), четвертий – ім'я методу скидання значення (`RESET`), п'ятий (`DESIGNABLE`) є логічним (булевим) значенням, яке говорить про те, чи повинна властивість з'являтися в інспекторі властивостей Qt Designer. Шостий параметр (`SCRIPTABLE`) – також логічне значення, яке керує тим, чи буде властивість доступною для мови сценаріїв Qt Script. Останній, сьомий параметр (`STORED`) керує серіалізацією, тобто тим, чи буде властивість запам'ятовуватися під час збереження об'єкту.

Після ознайомлення з поняттям "властивість", як приклад, визначимо в класі властивість для керування режимом тільки читання (роздрук 2.2).

Роздрук 2.2. Визначення властивості для керування режимом тільки читання

```
class MyClass : public QObject {
Q_OBJECT
Q_PROPERTY(bool readOnly READ isReadOnly WRITE setReadOnly)

private:
    bool m_bReadOnly;

public:
    MyClass(QObject* pobj = 0) : QObject(pobj)
                               , m_bReadOnly(false)
    {
    }

public:
```

```
    void setReadOnly(bool bReadOnly)
    {
        m_bReadOnly = bReadOnly;
    }

    bool isReadOnly() const
    {
        return m_bReadOnly;
    }
}
```

Клас `MyClass`, показаний в роздруку 2.2, успадковується від класу `QObject`. Визначаємо атрибут `m_bReadOnly`, в якому запам'ятовуватимуться значення стану. Цей атрибут ініціалізувався в конструкторі значенням `false`. Для отримання і зміни значення атрибуту в класі `MyClass` визначені методи `isReadOnly()` і `setReadOnly()`. Ці методи реєструються в макросі `Q_PROPERTY`. Метод `isReadOnly()` служить для набуття значення, тому вказується в секції `READ`, а метод `setReadOnly()` – для зміни значення, тому пишеться в секції `WRITE`.

З програми можна змінити значення властивості таким чином:

```
pobj->setProperty("readOnly", true);
```

А так можна отримати поточне значення:

```
bool bReadOnly = pobj->property("readOnly").toBool();
```

Механізм сигналів і слотів

Елементи графічного інтерфейсу певним чином реагують на дії користувача і посилають повідомлення. Існує декілька варіантів такого рішення.

Стара концепція функцій зворотного виклику (callback functions), що лежить в основі X Window System, основана на використанні звичайних функцій, які повинні викликатися в результаті дій користувача. Застосування такої концепції значно ускладнює початковий код програми, роблячи його менш зрозумілим. Крім того, відсутня можливість робити перевірку типів значень, які повертаються, тому що в усіх випадках повертається вказівник на порожній тип `void`. Наприклад, для того, щоб зіставити код з кнопкою, необхідно передати у функцію вказівник на кнопку. Якщо користувач натискає на кнопку, функція буде викликана. Самі бібліотеки не перевіряють, чи були аргументи, передані у функцію, необхідного типу, а це часто є причиною збоїв. Інший недолік функцій зворотного виклику полягає в тому, що елементи графічного інтерфейсу користувача тісно пов'язані з функціональними частинами програми і це, у свою чергу, помітно ускладнює розробку класів незалежно один від одного. Одним з яскравих представників цієї концепції є бібліотека Motif.

Важливо пам'ятати, що Motif і Windows API призначені для процедурного програмування, і з реалізацією об'єктно-орієнтованих проектів, напевно, з'являться труднощі.

Для програмування в ОС Windows існують спеціальні бібліотеки класів мови C++, що полегшують програмування для цієї операційної системи. Найпопулярнішою бібліотекою є Microsoft Foundation Classes (MFC). Її можна, з великим застереженням, назвати об'єктно-орієнтованою, так як вона створювалася людьми, які не знали про існування самих елементарних принципів об'єктно-орієнтованого підходу. Одне з найфундаментальніших положень об'єктно-орієнтованого підходу – це інкапсуляція, яка забороняє залишати атрибути класів незахищеними (адже тоді об'єкти можуть читати і змінювати дані без відома об'єкта-власника), але, незважаючи на це, у багатьох MFC-класах ця вимога не дотримана. Сама бібліотека MFC є надбудовою, що надає доступ до функцій Windows, реалізованих на мові C, що заставляє розробників час від часу використати застарілі структури, що не вписуються в рамки концепції об'єктно-орієнтованого підходу. Цікаво зазначити, що сама Microsoft для реалізації широко відомої програми Microsoft Word не використовує MFC взагалі.

При використанні MFC для забезпечення зв'язків повідомлення і методів обробки використовуються спеціальні макроси – так звані карти повідомлень (роздрук 2.3). Вони дуже сильно захаращують початковий код програми, помітно знижуючи її читаність.

Роздрук 2.3. Фрагмент програми, реалізованої за допомогою MFC

```
class CPhotoStylerApp : public CWinApp {
public:
    CPhotoStylerApp();
public:
    virtual BOOL InitInstance();

    afx _ msg void OnAppAbout();
    afx _ msg void OnFileNew();

    DECLARE _ MESSAGE _ MAP()
};
BEGIN _ MESSAGE _ MAP(CPhotoStylerApp, CWinApp)
    ON _ COMMAND(ID _ APP _ ABOUT, OnAppAbout)
    ON _ COMMAND(ID _ FILE _ NEW, OnFileNew)
    ON _ COMMAND(ID _ FILE _ NEW, CWinApp::OnFileNew)
    ON _ COMMAND(ID _ FILE _ OPEN, CWinApp::OnFileOpen)
    ON _ COMMAND(ID _ FILE _ PRINT _ SETUP, CWinApp::OnFilePrintSetup)
END _ MESSAGE _ MAP()
```

Конструкції, подібні до показаної в роздруку 2.3, дуже незручні для сприйняття і ускладнюють аналіз коду програми. Нехай багато хто розповідає про зручність використання засобів для автоматичного створення подібного коду, але створені вони були не від хорошого життя. Так, непродуманість самої бібліотеки заставляє розробника при внесенні незначних змін модифікувати код самої програми відразу в декількох місцях. Наприклад, для того, щоб добавити в діалогове вікно текстове поле, необхідно провести цілий ряд операцій. По-перше, треба створити в класі діалогу атрибут, призначений для зберігання значень, що вводяться в текстовому полі. По-друге, потрібно задати ідентифікатор ресурсу текстового поля. По-третє, поставити ідентифікатор ресурсу і атрибут у методі `DoDataExchange()` у відповідність один з одним за допомогою методу `DDX_Text()`, після чого здійснюватиметься обмін даними між текстовим полем і атрибутом. По-четверте, цим обміном необхідно керувати, передаючи в методі `UpdateData()` значення булевого типу `true` або `false`. І лише за допомогою засобів автоматичного створення коду можна частково позбавитися від цієї проблеми, змусивши виконати ці зміни за вас і отримавши натомість інші недоліки, наприклад додаткове засмічення коду програми непотрібною інформацією і можливе неспівпадіння створеного коду із затвердженими для проекту вимогами для форматування і нотації.

У цій ситуації частина провини прихована в самій мові C++. Справа у тому, що C++ не створювався як засіб для написання користувацького інтерфейсу, і тому не надає відповідної підтримки. Наприклад, якби робота по передачі подій реалізовувалася засобами самої мови, тоді відпадала б необхідність у використанні подібного роду макросів. До теперішнього часу не вдавалося зробити нічого подібного, саме тому бібліотека Qt явилася "як грім серед ясного неба". На відміну від більшості інших бібліотек програмування, Qt розширює мову C++ додатковими ключовими словами.

Проблема розширення мови C++ розв'язана в Qt за допомогою спеціального препроцесора MOC (Meta Object Compiler, метаоб'єктний компілятор). Він аналізує класи на наявність спеціального макросу `Q_OBJECT` в їх визначенні і впроваджує в окремий файл усю необхідну додаткову інформацію. Це відбувається автоматично, без безпосередньої участі розробника. Подібна операція автоматичного створення коду не суперечить звичному процесу програмування на C++, адже стандартний препроцесор перед компіляцією самої програми теж створює проміжний код, що містить виконані команди препроцесора. Так само діє і MOC, записуючи усю необхідну додаткову інформацію в окремий файл, вміст якого може бути проігнорований. Макрос `Q_OBJECT` повинен розміщуватися зразу на наступному рядку після

ключового слова `class` з визначенням імені класу. Дуже важливо пам'ятати, що після макросу не повинно стояти крапки з комою. Впроваджувати макрос у визначення класу має сенс в тих випадках, коли створений клас використовує механізм сигналів і слотів або якщо йому потрібна інформація про властивості.

Механізм сигналів і слотів повністю заміщає стару модель функцій зворотного виклику, він дуже гнучкий і повністю об'єктно-орієнтований. Сигнали і слоти – це основа програмування з використанням Qt, що дозволяє з'єднати разом незв'язані один з одним об'єкти. Кожен успадкований від `QObject` клас здатний відправляти і отримувати сигнали. Ця особливість ідеально вписується в концепцію об'єктної орієнтації і не суперечить людському сприйняттю. Уявіть собі ситуацію: у вас дзвонить телефон, і ви реагуєте на це зняттям трубки. На мові сигналів і слотів подібну ситуацію можна описати таким чином: об'єкт "телефон" вислав сигнал "дзвінок", на який об'єкт "людина" відреагувала слотом "зняття трубки".

Використання механізму сигналів і слотів дає програмістові наступні переваги:

• кожний клас, успадкований від `QObject`, може мати будь-яку кількість сигналів і слотів;

• повідомлення, що посилаються за допомогою сигналів, можуть мати безліч аргументів будь-якого типу;

• сигнал можна сполучати з різною кількістю слотів. Сигнал, що відправляється, поступить до усіх приєднаних слотів;

• слот може приймати повідомлення від багатьох сигналів, що належать різним об'єктам;

• з'єднання сигналів і слотів можна робити у будь-якій точці застосунку;

• сигнали і слоти є механізмами, що забезпечують зв'язок між об'єктами.

Більше того, цей зв'язок може створюватися між об'єктами, які знаходяться в різних потоках;

• при знищенні об'єкту відбувається автоматичне роз'єднання усіх сигнало-слотових зв'язків. Це гарантує, що сигнали не вирушатимуть до неіснуючих об'єктів.

Не можна не згадати і про недоліки, пов'язані із застосуванням сигналів і слотів :

• сигнали і слоти не є частиною мови C++, тому потрібно запуск додаткового препроцесора перед компіляцією програми;

• відправлення сигналів відбувається трохи повільніше, ніж звичайний виклик функції при використанні механізму функцій зворотного виклику;

• необхідно успадковувати клас `QObject`;

• в процесі компіляції не робиться ніяких перевірок: чи є сигнал або слот у відповідних класах або ні; чи сумісні сигнал і слот один з одним і чи можуть вони бути сполучені разом. Про помилку можна буде дізнатися лише тоді, коли застосунок буде запущений. Уся ця інформація виводиться на консоль, тому, для того, щоб побачити її в Windows, в проектному файлі необхідно в секції `CONFIG` додати опцію `console` (для Linux ніяких додаткових змін проектного файлу не потрібно).

Сигнали

Сигнали (signals) оточують нас в повсякденному житті скрізь: дзвінок будильника, жест регулювальника, а також і в не повсякденному, наприклад індіанське сигнальне вогнище і т. д. У програмуванні з використанням Qt під цим поняттям маються на увазі методи, які в змозі пересилати повідомлення. Причиною для появи сигналу може бути повідомлення про зміну стану керуючого елементу, наприклад переміщення повзунка. На подібні зміни приєднаний об'єкт, що відстежує ці сигнали, може відповідно відреагувати, хоча це і не обов'язково. Цей дуже важливий момент говорить про те, що з'єднувані об'єкти можуть бути абсолютно незалежні і реалізовані окремо один від одного. Це дозволяє об'єкту, що відправляє сигнали, не турбуватися про те, що згодом відбуватиметься з цими сигналами. Об'єкт, що відправляє сигнали, може навіть і не здогадуватися, що їх приймають і обробляють інші об'єкти. Завдяки такому розділенню, можна розбити великий проект на компоненти, які розроблятимуться різними програмістами окремо, а потім з'єднуватися за допомогою сигналів і слотів разом. Це

робить код дуже гнучким і легко розширюваним; якщо один з компонентів застаріє або повинен буде реалізований інакше, то усі інші компоненти, що беруть участь в комунікації з цим компонентом, і сам проект в цілому, не зміняться. Новий компонент після розробки стане на місце старого і буде підключений до основної програми за допомогою тих же самих сигналів і слотів. Це робить бібліотеку Qt особливо привабливою для реалізації компонентно-орієнтованих застосувань. Проте не забувайте, що велика кількість взаємозв'язків приводить до виникнення сильно пов'язаних систем, в яких навіть незначні зміни можуть привести до непередбачуваних наслідків.

Сигнали визначаються в класі, як і звичайні методи, *тільки без реалізації*. З точки зору програміста вони є тільки прототипами методів, що містяться в заголовковому файлі визначення класу. Усю подальшу роботу з реалізації коду для цих методів бере на себе MOC. Методи сигналів не повинні повертати яких-небудь значень, і тому перед ім'ям методу завжди повинно стояти `void`.

Сигнал не обов'язково сполучати із слотом. Якщо з'єднання не сталося, то він просто не оброблятиметься. Подібне розділення відправляючих і отримуючих об'єктів виключає можливість того, що один з приєднаних слотів якимсь чином зможе завадити об'єкту, що відправив сигнали.

Бібліотека надає велику кількість вже готових сигналів для існуючих елементів керування. В основному, для вирішення поставлених завдань вистачає цих сигналів, але іноді виникає необхідність реалізації нових сигналів у своїх класах. Приклад визначеня сигналу показаний в роздруку 2.4.

Роздрук 2.4. Визначення сигналу

```
class MySignal {
Q_OBJECT
...
signals:
    void doIt();
...
};
```

Зверніть увагу на метод сигналу `doIt()`. Він не має реалізації, цю роботу приймає на себе MOC, забезпечуючи приблизно таку реалізацію:

```
void MySignal::doIt()
{
    QMetaObject::activate(this, &staticMetaObject, 0, 0);
}
```

Із сказаного стає ясно, що не має сенсу визначати сигнали як `private`, `protected` або `public`, так як вони грають роль викликаючих методів.

Вислати сигнал можна за допомогою ключового слова `emit`. З огляду на те, що сигнали відіграють роль викликаючих методів, конструкція відправки сигналу `emit doIt()` приведе до звичайного виклику методу `doIt()`. Сигнали можуть відправлятися з класів, які їх містять.

Наприклад, в роздруку 2.4 сигнал `doIt()` може відсилатися тільки об'єктами класу `MySignal`, і ніякими іншими. Щоб мати можливість відіслати сигнал програмно з об'єкту цього класу, слід додати метод `sendSignal()`, виклик якого змусить об'єкт класу `MySignal` відправляти сигнал `doIt()`, як це показано в роздруку 2.5.

Роздрук 2.5. Реалізація сигналу

```
class MySignal {
Q_OBJECT
public:
    void sendSignal()
    {
        emit doIt();
    }
signals:
```

```
    void doIt();
};
```

Сигнали також мають можливість висилати інформацію, що передається в параметрі. Наприклад, якщо виникла необхідність передати в сигналі рядок тексту, то можна реалізувати це, як в роздруку 2.6.

Роздрук 2.6. Реалізація сигналу з параметром

```
class MySignal : public QObject {
Q_OBJECT
public:
    void sendSignal()
    {
        emit sendString("Information");
    }
signals:
    void sendString(const QString&);
};
```

Слоти

Слоти (slots) – це методи, які приєднуються до сигналів. По суті, вони є звичайними методами. Найбільша їх відмінність полягає в можливості приймати сигнали. Як і звичайні методи, вони визначаються в класі як `public`, `private` або `protected`. Відповідно, перед кожною групою слотів повинно стояти: `private slots:`, `protected slots:` або `public slots:`. Слоти можуть бути і віртуальними.

Примітка. З'єднання сигналу з віртуальним слотом приблизно вдесятеро повільніше, ніж з невіртуальним. Тому не варто робити слоти віртуальними, якщо немає особливої необхідності.

Правда, є невеликі обмеження, що відрізняють звичайні методи від слотів. У слотах не можна використати параметри за замовчуванням, наприклад `slotMethod(int n=8)`, або визначати слоти як `static`.

Класи бібліотеки містять цілий ряд вже реалізованих слотів. Але визначення слотів для своїх класів – це часта процедура. Реалізація слоту показана в роздруку 2.7.

Роздрук 2.7. Реалізація слоту

```
class MySlot : public QObject {
    Q_OBJECT
public:
    MySlot();
public slots:
    void slot()
    {
        qDebug() << "I' m a slot";
    }
};
```

Усередині слоту викликом методу `sender()` можна дізнатися, від якого об'єкту був висланий сигнал. Він повертає вказівник на об'єкт типу `QObject`. Наприклад, в цьому випадку на консоль буде виведено ім'я об'єкту, що вислав сигнал:

```
void slot()
{
    qDebug() << sender()->objectName();
}
```

З'єднання об'єктів

З'єднання об'єктів здійснюється за допомогою статичного методу `connect()`, який визначений в класі `QObject`. У загальному вигляді, виклик методу `connect()` виглядає таким чином:

```
QObject::connect(const QObject*     sender,
                 const char*        signal,
                 const QObject*     receiver,
                 const char*        slot,
                 Qt::ConnectionType type = Qt::AutoConnection
                );
```

Йому передаються п'ять наступних параметрів:

• `sender` – вказівник на об'єкт, що відправляє сигнал;

• `signal` – це сигнал, з яким здійснюється з'єднання. Прототип (ім'я і аргументи) методу сигналу має бути поміщений в спеціальний макрос **`SIGNAL`**`(method());`

• `receiver` – вказівник на об'єкт, який має слот для оброблення сигналу;

• `slot` – слот, який викликається при отриманні сигналу. Прототип слоту має бути поміщений в спеціальний макрос **`SLOT`**`(method());`

• `type` – керує режимом обробки. Є три можливі значення:

`Qt::DirectConnection` – сигнал обробляється відразу викликом відповідного методу слота;

`Qt::QueuedConnection` – сигнал перетворюється в подію і ставиться в загальну чергу для обробки;

`Qt::AutoConnection` – це автоматичний режим, який діє таким чином: якщо об'єкт, що посилає сигнал, знаходиться в одному потоці з об'єктом, що приймає його, то встановлюється режим `Qt::DirectConnection`, інакше – режим `Qt::QueuedConnection`. Цей режим (`Qt::AutoConnection`) визначений в методі `connection()` за умовчанням.

Наступний приклад демонструє те, як може бути здійснене з'єднання об'єктів в програмі.

```
void main()
{
    ...
    QObject::connect(pSender, SIGNAL(signalMethod()),
                     pReceiver, SLOT(slotMethod())
                    );
    ...
}
```

Якщо виклик здійснюється з класу, успадкованого від `QObject`, тоді `QObject::` можна опустити:

```
MyClass::MyClass() : QObject()
{
    ...
    connect(pSender, SIGNAL(signalMethod()),
            pReceiver, SLOT(slotMethod())
           );
    ...
}
```

У випадку якщо слот міститься в класі, з якого робиться з'єднання, то можна скористатися скороченою формою методу `connect()`, опустивши третій параметр (`pReceiver`), що вказує на об'єкт-одержувач. Іншими словами, якщо в як об'єкт-одержувач повинен стояти вказівник `this`, його можна просто не вказувати:

```
MyClass::MyClass() : QObject()
{
    connect(pSender, SIGNAL(signalMethod()), SLOT(slot()));
}
void MyClass::slot()
```

```
{
   qDebug() << "I’ m a slot";
}
```

Іноді виникають ситуації, коли об’єкт не обробляє сигнал, а просто передає його далі. Для цього необов’язково визначати слот, який у відповідь на отримання сигналу (за допомогою `emit`) посилає свій власний. Можна просто з’єднати сигнали один з одним. Сигнал, що відправляється, повинен міститися у визначенні класу :

```
MyClass::MyClass() : QObject()
{
    connect(pSender, SIGNAL(signalMethod()), SIGNAL(mySignal()));
}
```

Відправку сигналів заблокувати можна на деякий час, викликавши метод `blockSignals()` з параметром `true`. Об’єкт "мовчатиме", поки блокування не буде знято тим же методом `blockSignals()` з параметром `false`.

За допомогою методу `signalsBlocked()` можна взнати поточний стан блокування сигналів.

Два вікна програми, показані на рис. 2.1, демонструють механізм сигналів і слотів у дії. Створюється за стосунок (роздруки 2.8-2.10), в першому вікні якого знаходиться кнопка натиснення, а в другому – віджет надпису. При клацанні на кнопці ADD (Додати) збільшується відображуване значення на 1. Як тільки значення буде дорівнювати п’яти, відбудеться вихід із застосунку.

Рисунок 2.1 – Програма-лічильник. Демонстрація роботи механізму сигналів і слотів

Роздрук 2.8. Файл main.cpp

```
#include <QtGui>          // Qt 4.8
//#include <QtWidgets>       Qt 5.3

#include "Counter.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QLabel      lbl("0");
    QPushButton cmd("ADD");
    Counter     counter;

    lbl.show();
    cmd.show();

    QObject::connect(&cmd, SIGNAL(clicked()),
                     &counter, SLOT(slotInc())
                    );

    QObject::connect(&counter, SIGNAL(counterChanged(int)),
                     &lbl, SLOT(setNum(int))
                    );

    QObject::connect(&counter, SIGNAL(goodbye()),
                     &app, SLOT(quit())
                    );
```

```
    return app.exec();
}
```

У основній програмі додатку (роздрук 2.8) створюється об'єкт надпису `lbl`, натискна кнопка `cmd` і об'єкт лічильника `counter` (опис якого показаний у роздруках 2.9-2.10). Далі сигнал `clicked()` з'єднується із слотом `slotInc()`. При кожному натисненні на кнопку викликається метод `slotInc()`, збільшуючи значення лічильника на 1. Він має бути в змозі повідомляти про подібні зміни, щоб елемент надпису відображав завжди тільки актуальне значення. Для цього сигнал `counterChanged(int)`, який передає в параметрі актуальне значення лічильника, з'єднується із слотом `setNum(int)`, здатним приймати це значення.

Примітка. При з'єднанні сигналів із слотами, передаючими значення, важливо стежити за співпадінням їх типів. Наприклад, сигнал, передаючий в параметрі значення типу `int`, не повинен з'єднуватися із слотом, приймаючим `QString`.

Нарешті, сигнал `goodbye()`, символізуючий кінець роботи лічильника, з'єднується із слотом об'єкту застосунку `quit()`, який завершує роботу застосунку, після натиснення кнопки ADD в п'ятий раз. Показаний застосунок складається з двох вікон, і після закриття останнього вікна його робота автоматично завершиться.

Роздрук 2.9. Файл `Counter.h`

```
#ifndef _ Counter _ h _
#define _ Counter _ h _

#include <QObject>

// ========================================================================
class Counter : public QObject {
    Q _ OBJECT
private:
    int m _ nValue;

public:
    Counter();

public slots:
    void slotInc();

signals:
    void goodbye        (   );
    void counterChanged(int);
};
#endif  //_ Counter _ h _
```

Як видно з роздруку 2.9, у визначенні класу лічильника є два сигнали: `goodbye()`, який повідомляє про кінець роботи лічильника, і `counterChanged(int)`, передаючий актуальне значення лічильника, а також слот `slotInc()`, збільшуючий значення лічильника на одиницю.

Роздрук 2.10. Файл `counter.cpp`

```
#include "Counter.h"
// ------------------------------------------------------------------------
Counter::Counter() : QObject()
                   , m _ nValue(0)
{
}

// ------------------------------------------------------------------------
void Counter::slotInc()
{
    emit counterChanged(++m _ nValue);
```

```
    if(m _ nValue == 5) {
        emit goodbye();
    }
}
```

У роздруку 2.10 метод слоту `slotInc()` відправляє два сигнали: `counterChanged()` і `goodbye()`. Сигнал `goodbye()` відправляється при значенні атрибуту `m_nValue`, рівному 5.
Слот, що не має параметрів, можна з'єднати з сигналом, що має параметри. Це зручно, коли сигнали поставляють більше інформації, ніж вимагається для об'єкту, отримуючого сигнал. В цьому випадку в слоті можна не вказувати параметри:

```
MyClass::MyClass() : QObject()
{
    connect(pSender, SIGNAL(signalMethod(int)), SIGNAL(mySignal()));
}
```

Якщо не має впевненості, чи пригодиться параметр сигналу в майбутньому, то краще визначити слот з параметром і проігнорувати його усередині слоту. Зате потім, коли виникне необхідність, не потрібно буде міняти прототип слота.

Роз'єднання об'єктів

Якщо є можливість з'єднання об'єктів, то повинна існувати і можливість їх роз'єднання. У Qt, при знищенні об'єкту, усі пов'язані з ним з'єднання знищуються автоматично, але в окремих випадках може виникнути необхідність в знищенні цих з'єднань "вручну". Для цього існує статичний метод `disconnect()`, параметри якого аналогічні параметрам статичного методу `connect()`. У загальному випадку цей метод має наступний вигляд:

```
QObject::disconnect(sender, signal, receiver, slot);
```

Наступний приклад демонструє, як роз'єднати об'єкти в програмі:

```
void main()
{
    ...
    QObject::disconnect(pSender, SIGNAL(signalMethod()),
                        pReceiver, SLOT(slotMethod())
                       );
    ...
}
```

Існують два скорочених, не статичних варіанти: `disconnect(signal, receiver, slot)` і `disconnect(receiver, slot)`.

Організація об'єктних ієрархій

Організація об'єктів в ієрархії знімає з розробника необхідність самому турбуватися про звільнення пам'яті від створених об'єктів.

Конструктор класу `QObject` виглядає таким чином:

```
QObject(QObject* pobj = 0);
```

У його параметрі передається вказівник на інший об'єкт класу `QObject` або успадкованого від нього класу. Завдяки цьому параметру існує можливість створення об'єктів ієрархій. Він є вказівником на об'єкт-предок. Якщо в першому параметрі передається значення рівне нулю або нічого не передається, то це означає, що у створюваного об'єкту немає предка, і він буде об'єктом верхнього рівня і знаходитиметься на верхівці об'єктної ієрархії. Об'єкт-предок задається в конструкторі при створенні об'єкту, але згодом його можна у будь-який момент виконання програми змінити на інший за допомогою методу `setParent()`.

Створені об’єкти за замовчуванням не мають імені. За допомогою методу `setObjectName()` можна присвоїти об’єкту ім’я. Ім’я об’єкту не має особливого значення, але може бути корисне при налагодженні програми. Для того, щоб взнати ім’я об’єкту, можна викликати метод `objectName()`.

Роздрук 2.11. Приклад створення об’єктної ієрархії

```
QObject* pobj1 = new QObject;
QObject* pobj2 = new QObject(pobj1);
QObject* pobj4 = new QObject(pobj2);
QObject* pobj3 = new QObject(pobj1);
pobj2 ->setObjectName("the first child of pobj1");
pobj3 ->setObjectName("the second child of pobj1");
pobj4 ->setObjectName("the first child of pobj2");
```

У першому рядку роздруку 2.11 створюється об’єкт верхнього рівня. При створенні об’єкту `pobj2` в його конструктор передається як предок, вказівник на об’єкт `pobj1`. Об’єкт `pobj3` має як предок `pobj1`, а об’єкт `pobj4` має предка `pobj2`. Отримана об’єктна ієрархія показана на рис. 2.2.

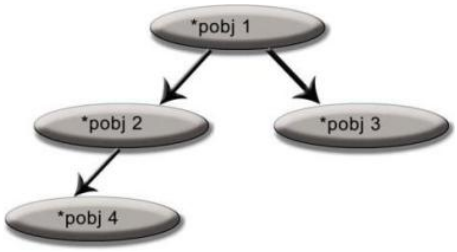


Рисунок 2.2 – Схема об’єктної ієрархії

При знищенні створеного об’єкту (при виклику його деструктора) усі приєднані до нього об’єкти-нащадки знищуються автоматично. Ця особливість рекурсивного знищення об’єктів значно спрощує програмування, так як не треба турбуватися про звільнення ресурсів пам’яті. Саме тому необхідно створювати об’єкти, а особливо об’єкти неверхнього рівня, динамічно, за допомогою оператора `new`, інакше вилучення об’єкту приведе до помилки при виконанні програми.

Попередження. Одна з найпоширеніших помилок програмістів, які пишуть мовою C++, при програмуванні з використанням бібліотеки Qt, – це самостійний контроль процесу виділення/звільнення пам’яті для об’єкту і нединамічне створення елементів керування. При програмуванні з Qt важливо пам’ятати, що усі об’єкти повинні створюватися в пам’яті динамічно, за допомогою оператора `new`. Виняток з цього правила можуть складати тільки об’єкти, що не мають предків.

Для отримання інформації про об’єктну ієрархію існують два методи: `parent()` і `children()`. За допомогою методу `parent()` можна визначити об’єкт-предок. Згідно рис. 2.2, виклик `pobj2→parent()` поверне вказівник на об’єкт `obj1`. Для об’єктів верхнього рівня цей метод поверне значення 0. Щоб вивести на консоль увесь ланцюг імен об’єктів-предків якого-небудь з об’єктів, можна поступити так, як показано в роздруку 2.12 (виконаємо це для об’єкту pobj4 з роздруку 2.11).

Роздрук 2.12. Виведення імен об’єктів предків

```
for(QObject* pobj = pobj4; pobj; pobj = pobj ->parent()) {
    qDebug() << pobj ->objectName();
}
```

І на екрані можна побачити:

```
the first child of pobj2
the first child of pobj1
```

Метод `children()` повертає константний вказівник на список об'єктів-нащадків. Для наведеного вище прикладу (роздрук 2.11, рис. 2.2), метод `pobj1→children()` поверне вказівник на список `QObjectList`, що містить два елементи : вказівники `pobj2` і `pobj3`.

Можна шукати потрібний об'єкт-нащадок за допомогою методу `findChild()`. У параметрі цього методу необхідно передати ім'я шуканого об'єкту. Наприклад, наступний виклик поверне вказівник на об'єкт `pobj4`:

```
QObject* pobj = pobj1 ->findChild<QObject*>("the first child of pobj2");
```

Для розширеного пошуку існує метод `findChildren()`, повертаючий список вказівників на об'єкти. Усі параметри методу не обов'язкові, може передаватися або стрічка імені, або регулярний вираз, а виклик методу без параметрів приведе до того, що він поверне список вказівників на усі об'єкти-нащадки. Пошук здійснюється рекурсивно. Наступний виклик поверне список вказівників на усі об'єкти, імена яких розпочинаються з букв `th`, в нашому випадку їх три:

```
QList<QObject*> plist = pobj1 ->findChildren<QObject*>(QRegExp("th*"));
```

Можна скористатися також глобальною шаблонною функцією `qFindChildren()`, яка повертає усі об'єкти нащадків вказаного типу:

```
QList<QObject*> plist = ::qFindChildren<QObject*>(pobj1);
```

у даному конкретному випадку ця функція поверне вказівники на об'єкти `pobj2`, `pobj3` і `pobj4`.

Для налагодження програми корисний метод `dumpObjectInfo()`, який показує наступну інформацію, яка відноситься до об'єкту :

• ім'я об'єкту;
• клас, від якого був створений об'єкт;
• сигнально-слотові з'єднання.

Уся ця інформація поступає в стандартний потік виведення `stdout`. При налагодженні можна скористатися і методом `dumpObjectTree()`, призначеним для відображення об'єктів-нащадків у вигляді ієрархії. Наприклад, виклик `dumpObjectTree()` для об'єкту `pobj1` з роздруку 2.11 покаже:

```
QObject::
    QObject::the first child of pobj1
        QObject::the first child of pobj2
    QObject::the second child of pobj1
```

Метаоб'єктна інформація

Кожний об'єкт, створений від класу `QObject` або від успадкованого від нього класу, має в розпорядженні структуру даних, яка називається метаоб'єктною інформацією (клас `QMetaObject`). У ній зберігається інформація про сигнали, слоти (включаючи вказівники на них), про сам клас і про успадкування. Отримати доступ до цієї інформації можна за допомогою методу `QObject::metaObject()`. Таким чином, для того, щоб взнати, наприклад, ім'я класу об'єкту, від якого він був створений, можна поступити таким чином:

```
qDebug() << pobj1 ->metaObject() ->className();
```

А для того, щоб порівняти ім'я класу з відомим, можна поступити так:

```
if(pobj1 ->metaObject() ->className() == "MyClass") {
    // Виконати які-небудь дії
}
```

Для отримання інформації про успадкування класів існує метод `inherits(const char*)`, який визначений безпосередньо в класі `QObject` і повертає значення `true`, якщо клас об'єкту успадкований від вказаного в цьому методі класу або створений від цього класу, інакше метод повертає значення `false`. Наприклад:

```
if(pobj ->inherits("QWidget")) {
    QWidget* pwgt = static _ cast<QWidget*>(pobj);
    // Виконати які-небудь дії з pwgt
}
```

Метаоб'єктну інформацію використовує і операція перетворення типів `qobject_cast<T>`. Таким чином, за допомогою методу `inherits()` приклад можна змінити:

```
QWidget* pwgt = qobject_cast<QWidget*>(pobj);
if(pwgt) {
    // Виконати які-небудь дії з pwgt
}
```

До метаоб'єктної інформації відноситься також і метод `tr()`, призначений для інтернаціоналізації програм.

3. Робота з Qt

Інтегроване середовище розробки IDE

Існує багато різних IDE (Integrated Development Environment, інтегроване середовище розробки), які можна дуже ефективно використати при розробці Qt-проектів. До них відносяться Microsoft Visual Studio, IBM Eclipse, QDevelop та ін. Але, мабуть, найкраще середовище розробки – це Nokia Qt Creator (рис. 3.1). Ця IDE включена в пакет поставки Qt SDK.

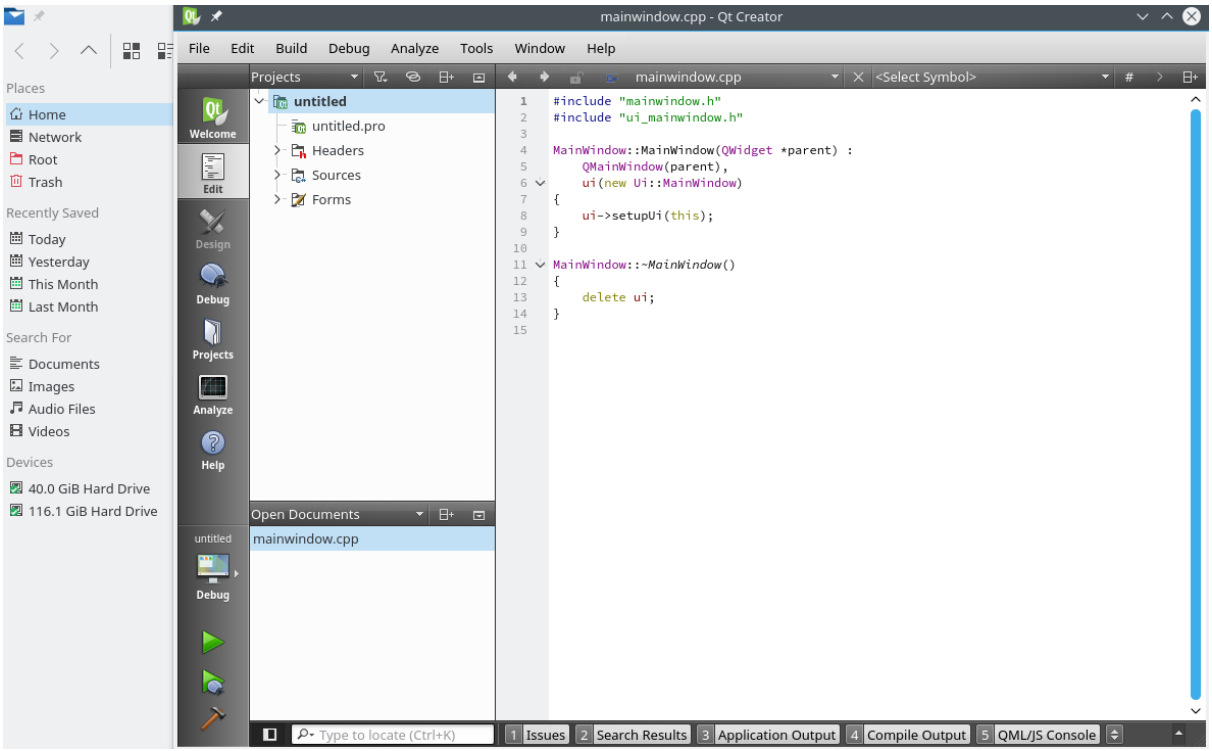


Рисунок 3.1 – Вікно інтегрованого середовища розробки Qt Creator (Qt 5.3)

Qt Assistant

Документація – це те, чим найчастіше користується розробник. У Qt вона містить більше 1500 HTML-сторінок і необхідний засіб, що забезпечує швидкий пошук потрібної інформації. Таким засобом і є програма Qt Assistant, яка за принципом роботи подібна до Web-переглядача. Qt Assistant надає можливість для пошуку тексту в усіх доступних документах про Qt. Для того, щоб встановити шлях до місця розташування іншої документації, можна скористатися параметром `-docPath`. На рис. 3.2 показано вікно програми Qt Assistant.

Розробники можуть за допомогою класу `QAssistantClient` інтегрувати Qt Assistant у свої програми.

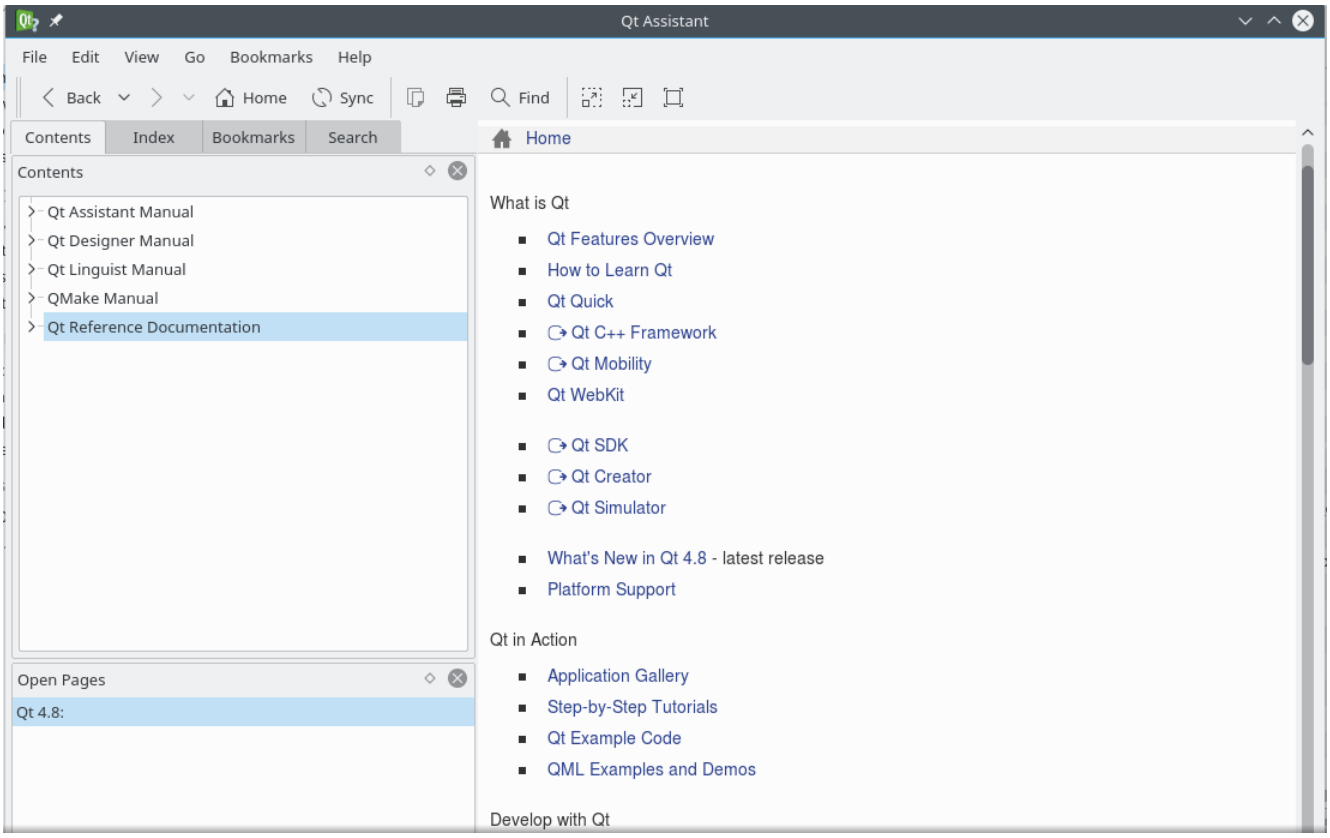


Рисунок 3.2 – Вікно програми Qt Assistant (Qt 5.3)

Работа з qmake

Ні один програміст для компіляції своєї програми кожного разу не задаватиме параметри для компонування і не передаватиме шлях до бібліотек “вручну”. Набагато зручніше створити `make`-файл (makefile), який візьме на себе усю роботу по налаштуванню компілятора і компонувача.

Створення `make`-файлів вручну вимагає від їх розробника досвіду і розуміння процесів компонування застосунку, причому залежно від платформи вид цих файлів відрізнятиметься. Раніше техніка створення подібних файлів була невід’ємною частиною програмування, але тепер багато що змінилося. І справа зовсім не в тому, що структура make-файлів стала простіша, швидше навпаки – вона стала складніша. Просто з’явилися спеціальні утиліти – генератори, які виконають цю роботу за розробника.

Утиліта `qmake` входить в поставку Qt починаючи з версії 3.0. Вона так само добре переносима, як і сам Qt. `qmake` інтерпретує файли проектів, які мають розширення `pro` і містять різні параметри, для створення `make`-файлів. Утиліта `qmake` здатна не лише створювати `make`-файли, але і самі `pro`-файли.

Припустимо, що в каталозі є початкові файли C++ і в ньому виконана наступна команда:

```
qmake –project
```

В результаті буде автоматичного створено pro-файл. Це зручно, так як на початку не потрібно вникати в усі тонкощі створення pro-файлів. Також це може бути корисним у тому випадку, якщо є велика кількість файлів, для яких вимагається створити pro-файл, – тоді відпаде необхідність вносити їх імена вручну. Створити з pro-файлу make-файл зовсім неважко, для цього треба просто виконати команду:

```
qmake file.pro –o Makefile
```

Як неважко здогадатися, file.pro – це ім'я pro-файлу, а Makefile – ім'я для створюваного платформонезалежного make-файлу.

Якщо виконати команду без параметрів, то утиліта qmake спробувала б знайти в поточному каталозі pro-файл і, у разі успіху, автоматично створила б make-файл. Таким чином, маючи в розпорядженні тільки початкові файли на C++, можна створити виконувану програму, виконавши усього лише три команди:

```
qmake -project
qmake
make
```

Звичайно, для серйознішої роботи потрібно буде змінювати вміст pro-файлів, що дозволить здійснювати тонкіше налаштування для розроблюваних проектів. Табл. 3.1 містить деякі опції pro-файлу, повний список яких можна отримати в офіційній документації Qt, яка поставляється разом з самою бібліотекою (для цього можна просто запустити програму Qt Assistant).

Таблиця 3.1 – Деякі опції для файлу проекту

Опція	Призначення
HEADERS	Список створених заголовкових файлів
SOURCES	Список створених файлів реалізації (з розширенням cpp)
FORMS	Список файлів з розширенням ui. Ці файли створюються програмою Qt Designer і містять опис інтерфейсу користувача у форматі XML
TARGET	Ім'я застосунку. Якщо це поле не заповнене, то назва програми відповідатиме імені проектного файлу
LIBS	Задає список бібліотек, які мають бути підключені для створення виконуваного модуля
CONFIG	Задає опції, які повинен використати компілятор
DESTDIR	Задає шлях, куди буде поміщений готовий виконуваний модуль
DEFINES	Тут можна передати опції для компілятора. Наприклад, це може бути опція поміщення налагоджувальної інформації для зневадника (debugger) у виконуваний модуль
INCLUDEPATH	Шлях до каталогу, де містяться заголовкові файли. Цією опцією можна скористатися у випадку, якщо вже є готові заголовкові файли і їх потрібно підключити в поточному проекті
DEPENDPATH	У цьому розділі вказуються залежності, необхідні для компіляції
SUBDIRS	Задає імена підкаталогів, які містять pro-файли
TEMPLATE	Задає різновид проекту. Наприклад: app – застосунок, lib – бібліотека, subdirs – підкаталоги
TRANSLATIONS	Задає файли перекладів, які використовуються в проекті

Розглянемо структуру наступного проектного файлу:

```
TEMPLATE = app
```

```
HEADERS += file1.h \
           file2.h
SOURCES += main.cpp  \
           file1.cpp \
           file2.cpp
TARGET = file
CONFIG += qt warn_on release
```

У першому рядку задається тип програми. В даному випадку це застосунок, тому `TEMPLATE = app` (якщо потрібно створити бібліотеку, то `TEMPLATE = lib`). У другому рядку, в `HEADERS`, перераховуються усі заголовкові файли, що належать проекту. У опції `SOURCES` перераховуються усі файли реалізації проекту. `TARGET` визначає ім'я програми, `CONFIG` – опції, які повинен використати компілятор відповідно до під'єднуваних бібліотек. Наприклад, в заданому файлі:

• `qt` вказує, що це Qt-застосунок і використовується бібліотека Qt;

• `warn_on` означає, що компілятор повинен видавати якомога більше попереджувальних повідомлень;

• `release` вказує, що додаток треба відкомпілювати в кінцевому варіанті, без налагоджувальної інформації.

Як видно з прикладу, програмі `qmake` не потрібно багато інформації, так як вона використовує файл локальної конфігурації, який визначений системною конфігурацією. Такий файл дуже важливий ще і тому, що один і той же виклик утиліти `qmake` створить різні `make`-файли залежно від того, на якій платформі вона була викликана. Це один з дуже важливих кроків для підтримки платформонезалежності самих проектних файлів.

Проектний файл може бути використаний для компіляції проектів, розташованих в різних каталогах. Цей файл виглядає приблизно так:

```
TEMPLATE = subdirs
SUBDIRS = Example1 Example2 .... ExampleN
```

Примітка. Для вилучення об'єктних файлів проекту служить опція `clean`, а для вилучення об'єктних файлів, створених проектом виконуваних модулів і створених `make`-файлів, існує опція `distclean`. Наприклад: `make distclean`.

Рекомендації для проекту з Qt

При реалізації файли класів краще всього розбивати на дві окремі частини. Частина визначення класу поміщається в заголовковий файл з розширенням `h`, а реалізація класу – у файл з розширенням `cpp`. Важливо пам'ятати, що в заголовковому файлі з визначенням класу повинна міститися директива препроцесора `#ifndef`. Директива призначена для того, щоб уникнути конфліктів у тому випадку, коли один і той же заголовковий файл включатиметься в початкові файли більше одного разу.

```
#ifndef _MyClass_h_
#define _MyClass_h_
class MyClass {
...
};
#endif //_MyClass_h_
```

За традицією заголовковий файл, як правило, має ім'я класу, який знаходиться в ньому. У заголовкових файлах, з метою швидшої компіляції, для вказівників на типи даних використовується попереднє оголошення для типу даних, а не пряме включення за допомогою директиви `#include`. На початку визначення класу міститься макрос `Q_OBJECT` для MOC; це необхідно, якщо ваш клас використовує сигнали і слоти, а в інших випадках, якщо у вас немає нужди в метаінформації, цим макросом можна знехтувати. Але потрібно враховувати ту

обставину, що через відсутність метаінформації не можна буде використати перетворення типу `qobject_cast<T>(obj)`.

```
class MyClass : public QObject {
Q_OBJECT
public:
    MyClass();
...
};
```

Основна програма має бути реалізована в окремому файлі. Цьому файлу прийнято давати ім'я `main.cpp`. Це зручно ще і тому, що проект може складатися з сотень файлів, і якщо дотримуватися такого правила, то знайти відправну точку усього проекту не буде складно.

Метаоб'єктний компілятор MOC

Метаоб'єктний компілятор (MOC, Meta Object Compiler), по суті є не компілятором, а препроцесором, який виконується в ході компіляції застосунку, створюючи, відповідно до визначення класу, додатковий код на мові C++. Це відбувається через те, що визначення сигналів і слотів в початковому коді програми недостатньо для компіляції. Сигнально-слотовий код має бути перетворений в код, зрозумілий для компілятора C++. Код зберігається у файлі з прототипом імені `moc_<filename>.cpp`.

Увага! Створені moc-файли не варто включати за допомогою команди препроцесора `#include` "main.moc" в кінець основного файла. Наприклад:

```
#include <QtGui>          // Qt 4.8
//#include <QtWidgets>       Qt 5.3

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    ...
    return app.exec();
}
#include "main.moc"
```

Краще, якщо вони будуть окремо відкомпільовані і приєднані компонувачем до основної програми. Хоча при написанні демонстраційних програм цим правилом можна знехтувати, щоб розмістити увесь код в одному файлі `main.cpp`.
При роботі з файлами проекту, про існування MOC можна і не здогадуватися, адже в цьому випадку керування MOC автоматизоване. Для створення moc-файлу "вручну" можна скористатися наступною командою:

```
moc –o proc.moc proc.h
```

Після її виконання MOC створить додатковий файл `proc.moc`. Для кожного класу, успадкованого від `QObject`, MOC надає об'єкт класу, успадкованого від `QMetaObject`. Об'єкт цього класу містить інформацію про структуру об'єкту, наприклад сигнально-слотові з'єднання, ім'я класу і структуру успадкування.

Компілятор ресурсів RCC

Майже кожна програма так чи інакше звертається до сторонніх ресурсів, таких як растрові зображення, файли перекладу і т. д. Це не є надійним і ефективним способом, так як ці ресурси можуть бути вилучені або бути недоступні з яких-небудь інших причин. Компілятор ресурсів дозволяє вбудовувати такі файли у виконуваний модуль, для того, щоб застосунок мав доступ до необхідних ресурсів в процесі його виконання. Існують спеціальні домовленості про іменування, завдяки яким можна однозначно звертатися до таких ресурсів. Усі необхідні для

використання файли (ресурси) мають бути описані в спеціальному файлі з розширенням qrc (Qt Resource Collection, колекція ресурсів Qt) разом з їх шляхами. Цей опис виконується в нотації XML. Наприклад:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/open.png</file>
    <file>images/quit.png</file>
</qresource>
</RCC>
```

Файл прикладу буде аналізуватися компілятором ресурсів – утилітою rcc, для створення з файлів open.png і quit.png одного початкового файлу C++, що містить усі їх дані, які компілюватимуться і компонуватимуться разом з іншими файлами проекту. Усі дані ресурсу зберігаються у файлі C++ у вигляді одного великого масиву.
Такий підхід забезпечує доступність усіх необхідних ресурсів, що дозволяє уникнути проблем неправильної установки необхідних для виконуваної програми файлів. Сам же qrc-файл має бути вказаний в pro-файлі в секції RESOURCES, для того, щоб утиліта qmake врахувала інформацію з файлу ресурсу. Наприклад:

```
RESOURCES = images.qrc
```

Можна скористатися файлом open.png, а точніше, вмістимим в ньому растровим зображенням, так:

```
plbl->setPixmap(QPixmap(":/images/open.png"));
```

У даному випадку всі растрові зображення розміщені у каталозі images. Але не завжди є можливість розміщувати файли ресурсів там де зручно. Ця незручність усовується за рахунок використання синонімів, які записуються в xml-файлі у тегу <file> за допомогою опції alis, наприклад:

```
<!DOCTYPE RCC><RCC version=”1.0”>
<qresource>
    <file alias=”open.png”>../../very/long/path/images/open.png</file>
    <file alias=”quit.png”>../../very/long/path/images/quit.png</file>
</qresource>
</RCC>
```

Тепер можна звертатися до файлів ресурсів наступним чином:

```
plbl→setPixm(QPixmap(”:/open.png”);
```

Без використання синонімів звернення до ресурсів виглядало б так:

```
plbl→setPixm(QPixmap(”:/very/long/path/images/open.png”);
```

Структура Qt-проекта

Ми розглянули окремо кожну утиліту для Qt-проекту. Тепер можна зібрати їх разом, щоб зрозуміти, як вони взаємодіють. Структура проекту Qt дуже проста, крім файлів початкового коду на мові C++ звичайно є файл проекту з розширенням pro. З цього файлу викликом утиліти qmake створюється make-файл. Цей make-файл містить в собі усі необхідні інструкції для створення готового виконуваного модуля (рис. 3.3).

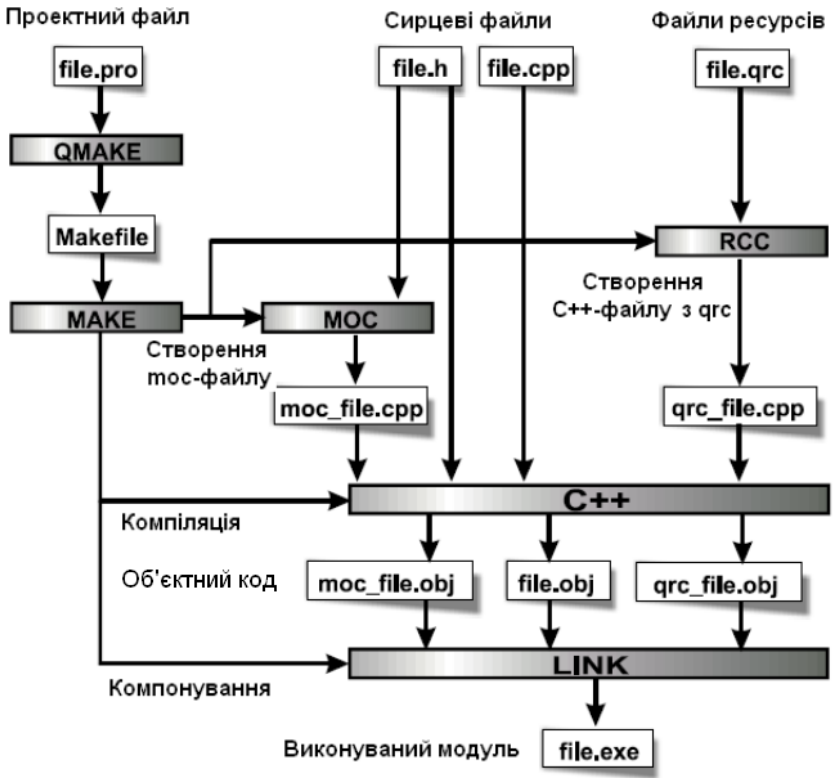


Рисунок 3.3 – Схема створення виконуваного модуля

У `make`-файлі міститься виклик MOC для створення додаткового коду C++ і необхідних заголовкових файлів. Якщо проект містить `qrc`-файл, то буде також створений файл C++, який містить дані ресурсів. Після цього усі початкові файли компілюються C++-компілятором у файли об'єктного коду, які об'єднуються компонувачем `link` в готовий виконуваний модуль.

Методи налагодження

Немає програм без помилок і дефектів (bug, помилка). В процесі розробки програм часто виникають проблеми з їх виявленням. Розробникам доводиться витрачати значний час на пошук і усунення помилок. До засобів пошуку помилок можна віднести:

• надання початкового коду для перегляду іншими розробниками (code review);

• створення класів для автоматизованих тестів.

Найпершим засобом, який виявляє помилки у програмі є зневажник (debugger). Зневадник надає оболонку, в якій можна відстежувати зміну даних під час виконання програми. Завдяки платформонезалежності Qt розробник може для налагодження програм використати будь-який з наявних зневадників, наприклад GDB або зневадник, вбудований в Microsoft Visual Studio. Почати можна із зневадника без графічного інтерфейсу користувача, оскільки виконання команд в діалоговому режимі допоможе зрозуміти його роботу, а також в майбутньому належним чином оцінити зневадники, які мають графічний інтерфейс. Тому розглянемо зневадник GDB, доступний як для Windows, так для Linux і Mac OS X.

Зневадник GDB (GNU Debugger)

GDB – цей простий засіб для налагодження програм в ОС Unix. З цим зневадником працюють з командного рядка, хоча є оболонки, для роботи в інтерактивному режимі (XXGDB, DDD і KDBG). Найідеальнішим середовищем для роботи із зневадником в інтерактивному режимі є IDE, у цьому випадку усе необхідне знаходиться "під рукою".

Інші методи налагодження

Одним із стандартних прийомів налагодження є вставлення в початковий код операторів виведення, що дозволяє побачити значення змінних і порівняти їх з очікуваними значеннями. Такий спосіб налагодження часто використовується розробниками. У Qt прикладом такого підходу є метод `QObject::dumpObjectInfo()`, який виводить на екран метаінформацію об'єкту.

Qt надає макроси і функції для налагодження, за допомогою яких можна вбудовувати в саму програму різного роду перевірки і виведення тестових повідомлень.

У заголовковому файлі `QtGlobal` містяться визначення двох макросів `Q_ASSERT()` і `Q_CHECK_PTR()`:

• `Q_ASSERT()` приймає як аргумент значення булевого типу і виводить попереджуавльне повідомлення, якщо це значення не рівне `true`;

• `Q_CHECK_PTR()` приймає вказівник і виводить попереджувальне повідомлення, якщо переданий вказівник дорівнює 0, а це означає, що або вказівник не ініціалізований або операція по виділенню пам'яті пройшла невдало.

Qt надає глобальні функції `qDebug()`, `qWarning()` і `qFatal()`, які також визначені в заголовковому файлі `QtGlobal`. Їх застосування подібне до функції `printf()`.

Як і в `printf()`, в ці функції передаються форматований рядок і різні параметри. У Microsoft Visual Studio ці функції виводяться у вікно зневадника, а в ОС Linux – в стандартний потік виведення помилок.

Примітка. Виклик функції `qFatal()` після виведення повідомлення зразу завершує роботу усього застосунку.

Якщо потрібно буде перенаправити потік виведення повідомлення, то потрібно створити і встановити свою власну функцію для керування виведенням. Встановлюється вона за допомогою функції `qInstallMsgHandler()`. Цій функції як аргумент передається адреса функції, яка керує повідомленнями і має наступний прототип:

```
void fct(QtMsgType type, const char *msg);
```

На місці `fct` має стояти ім'я функції. Перший аргумент є типом повідомлення, що приймає одне із значень перечислення `QtMsgType: QtDebugMsg, QtWarningMsg або QtFatalMsg`. Другий аргумент – це вказівник на саме повідомлення. Проілюструємо сказане фрагментом коду (роздрук 3.2).

Роздрук 3.2. Перенаправлення потоку виведення повідомлень

```
void  messageToFile (QtMsgType  type ,
                          const  QMessageLogContext &  context ,
                          const QSt ring&  msg
        QFile file("protocol.log");
        if (!file.open (QIODevice::WriteOnly | QIODevice::Text |
                        QIODevice::Append) )
       return;
   QTextStream  out(&file);
   switch(type ) {
      case  QtDebugMsg:
         out << "Debug: " << msg << ", " << context.file << endl;
         break;
      case QtWarningMsg:
         out << "Warning: " << msg << ", ", << context.file << endl;
         break;
      case  QtCriticalMsg:
         out << "Critical: " << msg << ", " << context.file <<endl;
         break;
      case QtFatalMsg:
         out << "Fatal: " << msg << ", " << context.file << endl;
         abort();
```

```
int main (int argc , char** argv)
{
   QApplication app(argc, argv);
   qinstallMessageHandler(messageToFile);
}
```

Тепер усі повідомлення `qDebug()`, `qWarning()` і `qFatal()` будуть записуватися у файл `protocol.log`, а не виводитися на консоль. Це дуже зручно для вивчення помилок і незрозумілої поведінки програми, які сталися на стороні тестера і користувачів. Тепер можна завжди попросити прислати файл `protocol.log` для більше поглибленого дослідження.

Для полегшення процесу налагодження рекомендується присвоювати усім об'єктам імена. Таким чином, ці об'єкти можна буде завжди знайти, викликавши метод `QObject::objectName()`. Це дозволить в процесі роботи програми скористатися методом `QObject::dumpObjectInfo()`, який дозволяє відобразити внутрішню інформацію об'єкту.

Також для налагодження можна скористатися встановленням фільтру подій для об'єкту класу `QCoreApplication`, у цьому випадку цей фільтр буде найпершим об'єктом, отримуючим і обробляючим події усіх об'єктів застосунку.

Найпростіший спосіб операції виведення в Qt – це використання об'єкту класу `QDebug`. Цей об'єкт дуже нагадує стандартний об'єкт потоку виведення в C++ `cout`. Наприклад, вивести повідомлення у зневаднику або на консолі за допомогою функції `qDebug()` можна таким чином:

```
qDebug() << "Test";
```

Ця функція створює об' єкт класу потоку `QDebug`, передаючи в його конструктор згаданий раніше аргумент `QtDebugMsg`. Можна поступити і так:

```
QDebug(QtDebugMsg) << "Test";
```

Важливо розуміти, що інформація виводиться за допомогою функції `qDebug()` як в налагоджувальних так і кінцевих компонуваннях. Якщо виводити інформацію потрібно тільки в налагоджувальній версії програми, а не в кінцевій, то можна реалізувати порожню функцію для керування виведення `dummyOutput()` і встановити її в `qinstallMessageHandler()`, як показано в роздруку 3.3:

Роздрук 3.3. Приховування інформації, яка виводиться функціями `qDebug()`, `qWarning()` і `qFatal()`.

```
void dumyOutput(QtMsgType, const QMessageLogContext&, const QString&)
{
   int main (int argc, char** argv)
   QApplication app(argc, argv);
   #ifndef QT_DEBUG
      qinstallMessageHandler(dummyOutput);
   #endif
}
```

При цьому потрібно використати замість функції `qDebug()` наступний запис:

```
qDebug() << ”Test1” << 123 << "Test2" << 456;
```

Тепер, в кінцевій версії програми, інформація налагодження, яка виводиться функціями `qDebug()`, `qWorning` і `qFatal()` буде прихована.

Глобальні визначення Qt

Qt містить в заголовковому файлі `QtGlobal` деякі макроси і функції, які можуть бути дуже корисні при написанні програм.

Шаблонні функції `qMax(a, b)` і `qMin(a, b)` використовуються для визначення максимального і мінімального з двох переданих значень.

```
int n = qMax<int>(3, 5); // n = 5
```

```
int n = qMin<int>(3, 5); // n = 3
```

Функція `qAbs(a)` повертає абсолютне значення:

```
int n = qAbs(- 5); // n = 5
```

Функція `qRound()` округлює передаване число до цілого:

```
int n = qRound(5.2);  // n = 5
int n = qRound(-5.2); // n = - 5
```

Функція `qBound()` повертає значення, що знаходиться між мінімумом і максимумом :

```
int n = qBound(2, 12, 7); // n = 7
```

А ось ще одна цікава функція. Справа у тому, що порівняння на точну рівність двох значень з плаваючою крапкою є однією з частих помилок в програмуванні. Функція `qFuzzyCompare()` сама відповідає за правильне порівняння. Вона приймає два значення типу `double` або `float` і повертає логічне значення `true`, якщо змінні вважаються рівними, інакше вона повертає значення `false`. Саме порівняння здійснюється у відносній манері, коли точність для порівняння збільшується зі зменшенням чисельних значень порівнюваних величин. Тому єдине значення, яке є складністю для цієї функції, – це нульове значення. Але є рішення і для цієї задачі. Треба просто зробити так, щоб порівнювані значення були або рівні, або більше 1.0. Наприклад:

```
double dValue1 = 0.0;
double dValue2 = myFunction();
if (qFuzzyCompare(1 + dValue1, 1 + dValue2)) {
    // Значення рівні
}
```

Примітка. Ця функція також може бути дуже корисною для написання модульних тестів.

У табл. 3.3 приведений список типів Qt, які можна використати при програмуванні.

Таблиця 3.3 – Таблиця цілих типів Qt

Тип Qt	Еквівалент C++	Розмір
qint8	signed char	8 біт
quint8	unsigned char	8 біт
qint16	short	16 біт
quint16	unsigned short	16 біт
qint32	int	32 біт
quint32	unsigned int	32 біт
qint64	__int64 або long long	64 біт
quint64	unsigned __int64 или unsigned long long	64 біт
qlonglong	То же саме, що і qint64	64 біт
qulonglong	То же саме, що і quint64	64 біт

Як видно з табл. 3.3, самими суперечливими є типи `qint64` і `quint64`. Давайте перевіримо правильність вказаних для них в таблиці значень бітів, а заразом їх мінімальні і максимальні значення:

```
qDebug() << "Number of bits for qint64 ="  << (sizeof(qint64) * 8);
qDebug() << "Minimum of qint64 = -"        << ~(~quint64(0) >> 1);
qDebug() << "Maximum of qint64 ="          << (~quint64(0) >> 1);
qDebug() << "Number of bits for quint64 =" << (sizeof(quint64) * 8);
qDebug() << "Minimum of quint64 ="         << 0;
qDebug() << "Maximum of quint64 ="         << ~quint64(0);
```

Результат виконання:

```
Number of bits for qint64 = 64
Minimum of qint64 = -9223372036854775808
Maximum of qint64 = 9223372036854775807
Number of bits for quint64 = 64
Minimum of quint64 = 0
Maximum of quint64 = 18446744073709551615
```

Інформація про бібліотеку Qt

Іноді буває дуже корисно отримати інформацію про саму бібліотеку, яка використовується комп'ютері. Наприклад, потрібно взнати в якому каталозі Qt зберігає свої файли розширень (plug-ins) або потрібно уточнити поточну версію Qt і т. д. За отримання такої інформації відповідає клас `QLibraryInfo` і надає для цього цілий ряд статичних методів. Покажемо їх застосування на прикладі (роздрук 3.4).

Роздрук 3.4. Використання статичних функцій класу `QLibraryInfo`

```
#inc1ude <QtCore>
int main(int argc, char** argv)
qDebug() << "Дата компонування (Bui1d date):"
 << QLibraryinfo::bui1dDate().toString("yyyy-MM-dd" );
qDebug() << "Ліцензія продукту (License Products):"
 << QLibraryinfo::licensedProducts();
qDebug() << "Ліцензія (Licensee):"
 << QLibraryinfo::licensee();
qDebug() << "Налаштувальна версія (Is Debug Build):"
 << QLibraryinfo::isDebugBuild();
qDebug() << "Розміщення (Locations):";
qDebug() << "Заголовки (Headers):"
 << QLibraryinfo::location(QLibraryinfo::HeadersPath);
qDebug() << "Бібліотеки (Libraries):"
 << QLibraryinfo::location(QLibraryinfo::LibrariesPath);
qDebug() << "Виконувані файли (Binaries):"
 << QLibraryinfo::location(QLibraryinfo::BinariesPath);
qDebug() << "Префікс шляху (Prefix):"
 << QLibraryinfo::location(QLibraryinfo::PrefixPath);
qDebug() << "Документація (Documentation):"
 << QLibraryinfo::location(QLibraryinfo::DocumentationPath);
qDebug() << "Файли розширень (Plugins):"
 << QLibraryinfo::location(QLibraryinfo::PluginsPath);
qDebug() << "Шлях до даних (Data):"
 << QLibraryinfo::location(QLibraryinfo::DataPath);
qDebug() << "Налаштування (Eettings): "
 << QLibraryinfo::location(QLibraryinfo::SettingsPath);
qDebug() << "Приклади (Examp1es): "
 << QLibraryinfo::location(QLibraryinfo::Examp1esPath) ;
```

Результат виведення програми для версії Qt 5.3.1, встановленої на комп'ютер з операційною системою Windows:

```
Bui1d date: "2014-05-20"
Build key: "Windows mingw debug fu11-config"
License Products: "OpenSource"
Licensee :  "Open  Source "
Is Debug Build: fa1se
Locations
  Headers:  "C:/Qt/5.3.1/inc1ude"
  Libraries: "C:/Qt/5.3.1/lib"
  Binaries: "C:/Qt/5.3.1/bin"
  Prefix "C:\Qt\5.3.1"
  Documentation: "C:/Qt/5.3.1/doc"
```

```
  Plugins: "C:/Qt/5.3.1/plugins"
  Data:  "C:/Qt/5.3.1"
  Settings: "C:/Qt/5.3.1"
Examples: "C:/Qt/5.3.1/examples"
```

4. Бібліотека контейнерів

Одне з найпоширеніших завдань в програмуванні полягає в організації оброблення груп елементів. Реалізація і налагодження програм з використанням такого роду структур займає у розробників багато часу. Для вирішення цієї проблеми бібліотека контейнерів надає набір часто використовуваних класів. Це дозволяє розробникові сконцентруватися на реалізації самого застосунку і не вникати в деталі реалізації використовуваних контейнерних класів.

Qt надає бібліотеку контейнерів Tulip, яка є складовою частиною ядра Qt, тому її розуміння дуже важливе для подальшого вивчення Qt. Ця бібліотека не лише подібна на STL (Standard Template Library, стандартна бібліотека шаблонів), але і сумісна з нею. В даному випадку розробник може вільно вибирати, чим йому краще скористатися: Tulip або STL. Краще вибрати першу бібліотеку, тому що Tulip є частиною Qt і активно використовується в ньому. Другим аргументом на користь застосування Tulip є те, вона відповідно до специфіки класів Qt оптимізована за продуктивністю і витратами пам'яті. Незайве буде зазначити, що використання в програмах шаблонних класів, на основі яких побудовані контейнери, значно збільшує розмір виконуваних програм. Це пов'язано з тим, що в кожному об'єктному файлі реалізації класу, який використовує контейнери, знаходиться створений компілятором код контейнерів з потрібними типами, і цей код може повторюватися. Бібліотека Tulip створювалася саме з урахуванням цих обставин і, крім того, оптимізована для значного зменшення розміру об'єктного коду.

Реалізація класів Tulip розміщена в модулі QtCore. В основі бібліотеки Tulip (як і в STL) лежать три поняття:

- контейнерні класи (контейнери);
- алгоритми;
- ітератори.

Їх взаємозв'язок показаний на рис. 4.1.

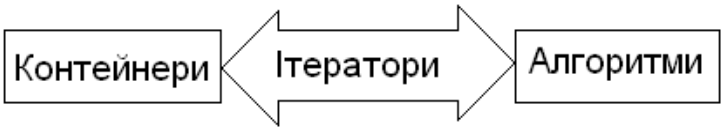


Рисунок 4.1 – Взаємозв'язок контейнерів, ітераторів і алгоритмів

Контейнерні класи

Контейнерні класи – це класи, які здатні зберігати в собі елементи різних типів даних. Майже усі контейнерні класи в Qt реалізовані як шаблонні і, таким чином, вони можуть зберігати дані будь-якого типу. Основна ідея шаблону полягає в створенні родового класу, який визначається при створенні об'єкту цього класу. Класи контейнерів можуть включати цілі серії інших об'єктів, які, у свою чергу, теж можуть бути контейнерами.

Щоб правильно підібрати контейнер для конкретного випадку, дуже важливо правильно розуміти відмінності різновидів контейнерів. Від цього значною мірою залежить швидкість роботи коду і ефективність використання пам'яті. Qt надає дві категорії різновидів класів контейнерів: послідовні (sequence containers) і асоціативні (associative containters).

Послідовні контейнери – це впорядковані колекції, де кожний елемент займає певну позицію. Позиція елементу залежить від місця його вставлення. До послідовних контейнерів

відносяться: *вектор* (vector), *список* (list), *стек* (stack) і *черга* (queue). Відповідно, Qt містить п'ять класів цієї категорії:

- `QVector<T> -вектор;`
- `QList<T>-список;`
- `QLinkedList<T>-двозв'язний список;`
- `QStack<T>-стек;`
- `QQueue<T> - черга.`

Асоціативні контейнери – це колекції, в яких позиція елементу залежить від його значення, тобто після занесення елементів в колекцію порядок їх слідування задаватиметься їх значеннями. До асоціативних контейнерів відносяться: *множина* (set), *словник* (map) і *хеш* (hash). Класи цієї категорії :

- `QSet<T> - множина;`
- `QMap<K, T> -словни;`
- `QMultiMap<K, T> - мультисловник;`
- `QHash<K, T> - хеш;`
- `QMultiHash<K, T> - мультихеш.`

В усіх контейнерах цих двох груп доступні операції, перераховані в таблиці. 4.1. Зверніть увагу на деякі винятки для класу `QSet<T>`.

Таблиця 4.1 – Оператори і методи, визначені в усіх контейнерних класах

Оператор, метод	Описання
`== і !=`	Оператори порівняння, дорівнює і не дорівнює
`=`	Оператор присвоєння
`[]`	Оператор індексації. Виняток становлять тільки класи QSet<T> і QLinkedList<T>, у яких цей оператор не визначений
`begin() і constBegin()`	Методи, що повертають ітератори, встановлені на початок послідовності елементів контейнера. Для класу QSet<T> повертаються тільки константні ітератори
`end() і constEnd()`	Методи, що повертають константні ітератори, встановлені на кінець послідовності елементів контейнера
`clear()`	Вилучення усіх елементів контейнері
`insert()`	Операція вставлення елементів в контейнер
`remove()`	Операція вилучення елементів з контейнера
`size() і count()`	Обидва методи ідентичні - повертають кількість елементів контейнера, але переважно застосовується перший, оскільки відповідає STL
`value()`	Повертає значення елементу контейнера. У QSet<T> цей метод не визначений
`empty() і isEmpty()`	Повертають true, якщо контейнер не містить жодного елементу. Обидва методи ідентичні, але переважно застосовується перший, оскільки відповідає STL

Примітка. Важливо не забувати про те, що класи, які успадковані від класу `QObject`, не мають доступного конструктора копіювання і оператора присвоєння, оскільки вони знаходяться в секції `private`. Отже, їх об'єкти не можуть зберігатися в контейнерах, тому потрібно зберігати в контейнерах не самі об'єкти, успадковані від класу `QObject`, а вказівники на них.

Увага! Якщо потрібно перевірити, чи містить контейнер елементи чи ні, то можна скористатися методом `empty()`, а не методом `size()`. Причина проста: метод `empty()` для усіх контейнерів виконується з постійною складністю, а `size()` може вимагати для контейнера списку лінійних витрат, що може істотно понизити швидкість алгоритму.

Ітератори

Для переміщення по елементах контейнера призначені ітератори. Вони дозволяють абстрагуватися від структури цих контейнерів. Наприклад, якщо потрібно використати інший тип контейнера, то для цього в коді потрібно замінити тип контейнера потрібним, а на коді, який використовує ітератори, це ніяк не відіб'ється.

Qt надає два стилі ітераторов:

• ітератори в стилі Java;

• ітератори в стилі STL.

Як альтернатива є варіант обходу елементів за допомогою ключового слова `foreach`.

Ітератори в стилі Java

Ітератори в стилі Java дуже прості у використанні. Вони були розроблені спеціально для програмістів, які не мають досвіду роботи з контейнерами STL. Основною їх відмінністю від останніх є те, що вони вказують не на сам елемент, а на двох його сусідів. Таким чином, спочатку ітератор вкаже на положення перед першим елементом контейнера, а з кожним викликом методу `next()` (див. табл. 4.2) буде переміщати вказівник на одну позицію вперед. Але насправді ітератори в стилі Java є об'єктами, а не вказівниками. Їх застосування у більшості випадків робить код компактнішим, ніж при використанні ітераторів в стилі STL:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
QListiterator<QString> it(list);
while (it.hasNext()) {
qDebug () << "Element:" << it.next();
```

У табл. 4.2 показані методи класу `QListrterator`, які застосовуються також до класів `QLinkedListiterator, QVectoriterator, QHashiterator, QMapiterator` і `QSet iterator`. Ці ітератори є константними, відповідно зміна значень елементів, їх вставлення і вилучення неможливі.

Таблиця 4.2 – Методи `QListiIterator, QLinkedListiterator, QVectorIterator, QHashIterator, QMapIterator, QSetIterator`

Метод	Описання
`toFront()`	Переміщає ітератор на початок списку
`toBack()`	Переміщає ітератор у кінець списку
`hasNext()`	Повертає значення true, якщо ітератор не знаходиться у кінці списку
`peekNext()`	Просто повертає наступне значення без зміни позиції ітератора
`hasPrevious()`	Повертає значення true, якщо ітератор не знаходиться на початку списку
`previous()`	Повертає значення попереднього елементу списку і повертає ітератор на попередню позицію
`peekPrevious()`	Просто повертає попереднє значення без зміни позиції ітератора
`findNext(const T&)`	Пошук заданого елементу в прямому напрямі
`findPrevious(const &T)`	Пошук заданого елементу у зворотному напрямі

Якщо необхідно виконувати зміни в процесі проходження ітератором елементів, то для цього слід скористатися ітераторами, що змінюються (mutable). Їх класи називаються аналогічно, але з додаванням фрагмента `Mutable`: `QMutableListIterator`, `QMutableHashIterator`, `QMutableLinkedListIterator`, `QMutableMapIterator` і `QmutableVectorIterator`. Метод `remove()` вилучає поточний елемент, а метод `insert()` вставляє елемент у поточну позицію. За допомогою методу `setvalue()` можна присвоїти елементу інше значення.

Давайте присвоїмо елементу списку "Boney M" значення "Rolling Stones":

```
QList<QString> list;
list << "Beatles" << "ABBA" << "Boney M";
QMutableListIterator<QString> it(list);
while(it.hasNext())
{
    if (it.next() == "Boney M")
    {
        it.setValue("Rolling Stones");
    }
    qDebug () << it.peekPrevious();
}
```

Основним недоліком ітераторів в стилі Java є те, що їх застосування, як правило, значно збільшує розмір створеного об'єктного модуля, порівняно з використанням ітераторів в стилі STL.

Ітератори в стилі STL

Ітератори в стилі STL трохи ефективніше від ітераторів Java-стилю і можуть використовуватися спільно з алгоритмами STL. Мабуть, для розробників на мові C++ – це найзручніший тип ітераторів. Ітератори в стилі STL можна уявити як деякі узагальнені вказівники, які посилаються на елементи контейнера.

Виклик методу `begin()` з об'єкту контейнера повертає ітератор, який вказує на перший його елемент, а виклик методу `end()` повертає ітератор, який вказує на кінець контейнера. Зверніть увагу: саме на кінець контейнера, а не на останній елемент, тобто на позицію, на якій міг би бути розміщений наступний елемент. Тобто, цей ітератор не вказує на елемент, а служить тільки для позначення досягнення кінця контейнера (рис. 4.2).

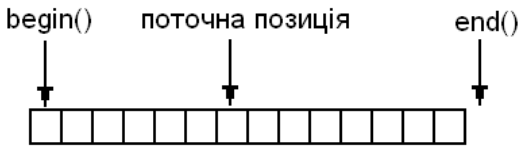


Рисунок 4.2 – Методи `begin()`, `end()` і поточна позиція

Оператори `++` і `--` об'єкту ітератора виконують переміщення на наступний або попередній елемент відповідно. Доступ до елементу, на який вказує ітератор, можна отримати за допомогою операції разіменування *. Наприклад:

```
QVector<QString> vec;
vec << "In Extremo" << "Blackmore's Night" << "Cultus Ferox";
QVector<QString>::iterator it = vec.begin();
for (; it != vec.end(); ++it)
{
    qDebug() << "Element:" << *it;
}
```

На екрані буде відображено:

```
Element: "In Extremo"
Element: "Blackmore's Night"
Element: "Cultus Ferox"
```

Зверніть увагу, що для збільшення ітератора `it` в циклі служить операція передінкрементації: `++it`, що дозволяє уникнути на кожній ітерації циклу збереження старого значення, яке скрито здійснюється при інкрементації, і це робить цикл ефективнішим.

При проходженні елементів у зворотному порядку за допомогою оператора -- необхідно пам'ятати, що він не симетричний до проходження за допомогою оператора ++. Тому цикл повинен в цьому випадку виглядати таким чином:

```
QVector<QString>::iterator it = vec.end();
for (;it != vec.begin();)
{
    --it;
    qDebug ()  <<  "Element :"  <<  *it;
}
```

На екрані буде відображено:

```
Element: "Cultus Ferox"
Element: "Blackmore's Night"
Element: "In Extremo"
```

Якщо потрібно тільки отримувати значення елементів, не змінюючи їх, то набагато ефективніше використати константний ітератор `const_iterator`. При цьому замість методів `begin()` і `end()` потрібно використовувати методи `constBegin()` і `constEnd()`. Таким чином, приклад матиме наступний вигляд:

```
QVector<QString> vec;
vec << "In Extremo" << "Blackmore's Night" << "Cultus Ferox";
QVector<QString>::const_iterator it = vec.constBegin();
for(; it != vec.constEnd(); ++it )
{
   qDebug() << "Element:" << *it;
}
```

Ці ітератори можна використати із стандартними алгоритмами STL, визначеними в заголовковому файлі `algorithm`. Наприклад, для сортування вектору за допомогою STL-алгоритму `sort()` можна поступити таким чином:

```
QVector<QString> vec;
vec << "In Extremo" << "Blackmore's Night" << "Cultus Ferox";
std::sort(vec.begin(), vec.end());
qDebug() << vec;
```

На екрані буде відображено:

```
QVector("Blackmore's Night", "Cultus Ferox", "In Extremo")
```

Qt має і свої власні алгоритми.

Ключове слово `foreach`

Звичайно, в мові C++ немає такого ключового слова, воно було створене штучно, за допомогою препроцесора, і є різновидом циклу, призначеного для перебору усіх елементів контейнера. Цей спосіб є альтернативою константному ітератору. Наприклад:

```
QList<QString> list;
list << "Subway to sally" << "Rammstain" << "After Forever";
foreach(QString str, list)
qDebug() << "Element:" << str;
```

У `foreach`, як і в циклах, можна використати ключові слова `break`, `continue`, а також вкладати цикли один в одного.

Увага! Qt робить копію контейнера при вході в цикл `foreach`, тому якщо мінятимуться значення елементів в циклі, то в оригінальному контейнері це ніяк не відобразиться.

Послідовні контейнери

Послідовні контейнери є впорядкованими колекціями, де кожний елемент займає певну позицію. Операції, доступні для усіх послідовних контейнерів, перераховані в табл. 4.3.

Таблиця 4.3 – Загальні методи послідовних контейнерів

Оператор/метод	Описання
+	Об'єднує елементи двох контейнерів
+=	Додає елемент в контейнер(те ж, що і <<)
<<	Додає елемент в контейнер
at()	Повертає вказаний елемент
back() і last()	Повертають посилання на останній елемент. Ці методи припускають, що контейнер не порожній. Обидва методи back() і last() ідентичні, але застосування першого прийнятніше, оскільки він відповідає STL
contains()	Перевіряє, чи міститься переданий як параметр елемент в контейнері
erase()	Вилучає елемент, розміщений на позиції ітератора, що передається як параметр
front() і first()	Повертає посилання на перший елемент контейнера. Методи припускають, що контейнер не порожній. Обидва методи front() і first() ідентичні, але застосування першого прийнятніше, так як він відповідає STL
indexOf()	Повертає позицію першого співпадіння знайденого в контейнері елементу відповідно до переданого в метод значення. Увага: в контейнері QLinkedList цей метод відсутній
lastindexOf()	Повертає позицію останнього співпадіння знайденого в контейнері елементу відповідно до переданого в метод значення. Увага: в контейнері QLinkedList цей метод відсутній
mid()	Повертає контейнер, який містить копії елементів, які задаються початковою позицією і кількістю
зор_back()	Вилучає останній елемент контейнера
pop_front()	Вилучає перший елемент контейнера
push_back() і append()	Методи додають один елемент в кінець контейнера. Обидва методи ідентичні, але застосування першого прийнятніше, так як він відповідає STL
push_front() і prepend()	Методи додають один елемент на початок контейнера. Обидва методи ідентичні, але застосування першого прийнятніше, оскільки він відповідає STL
replace()	Замінює елемент, який знаходиться на заданій позиції, значенням, переданим як другий параметр

Приклад:

```
QVector<QString> vec;
vec.append("In Extremo");
vec.append("Blackmore's Night");
vec.append("Cultus Ferox");
qDebug() << vec;
```

На екрані відобразиться:

```
QVector("In Extremo", "Blackmore's Night", "Cultus Ferox")
```

Вектор QVector<T>

Вектор – це структура даних, яка подібна на звичайний масив (рис. 4.3). Проте використання класу вектора надає деякі переваги в порівнянні із звичайним масивом – наприклад, можна отримати кількість його елементів або динамічно змінити його розмір. Крім того, цей контейнер, у порівнянні з іншими, найекономніше витрачає пам'ять.

Увага! Бажано не використовувати методи `push_front()`, `prepend()`, `pop_front()` і `remove()`, оскільки вставлення і вилучення з видобуванням елементів на початку вектора дуже неефективні і можуть понизити швидкодію алгоритму і програми в цілому. Це ж стосується і операції вставлення, тому також бажано не використати метод `insert()`. Якщо без цих методів не вдається обійтися, то потрібно розглянути вибір іншого типу контейнера, наприклад, `QLinkedList<T>`.

Рисунок 4.3 – Структура вектора

Для додавання елементів в кінець послідовного контейнера (див. табл. 4.3) можна скористатися методом `push_back()`. До елементів вектору можна звернутися як за допомогою оператора індексації [] (див. табл. 4.1), так і за допомогою ітератора. Наприклад:

```
QVector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);
qDebug() << vec;
```

На консолі буде відображено наступне:

```
QVector (10, 20, 30)
```

Розмір вектора можна задати в конструкторі при його створенні. За замовчуванням новостворений вектор матиме 0-й розмір, оскільки він не містить жодного елементу. Змінити його розмір можна або додавши до нього елементи, або викликавши метод `resize()` (табл. 4.4).

Таблиця 4.4 – Деякі методи контейнера QVector<T>

Метод	Описання
`data()`	Повертає вказівник на дані вектора (тобто на звичайний масив)
`fill()`	Присвоює одне і те ж значення усім елементам вектора
`reserve()`	Резервує пам'ять для кількості елементів відповідно до переданого значення
`resize()`	Встановлює розмір вектора відповідно до переданого значення
`toList()`	Повертає об'єкт QList з елементами, що містяться у векторі
`toStdVector()`	Повертає об'єкт std::vector з елементами, що містяться у векторі

Масив байтів QByteArray

Цей клас дуже подібний на `QVector<T>`, але різниця полягає в тому, що це не шаблонний клас, і в ньому допускається зберігання тільки елементи, які мають розмір в один байт. Об'єкти типу `QByteArray` можна використати скрізь, де потрібно проміжне зберігання даних. Кількість елементів масиву можна задати в конструкторі, а доступ до них отримувати за допомогою оператора []:

```
QByteArray arr(3);
arr[0] = arr[l] = OxFF;
arr[2 ] = 0x0;
```

До даних об'єктів класу `QByteArray` можна також застосувати операцію стискання і розтискання. Це досягається за допомогою двох глобальних функцій: `qCompress()` і `qUncompress()`. Приклад стискання і розтискання даних:

```
QByteArray a = "Test Data";
QByteArray aCompressed = qCompress(a);
qDebug() << qUncompress(aCompressed);
```

На екрані з'явиться: `"Test Data"`.

Трапляється, що потрібно перетворювати бінарні дані в текстові. Наприклад, потрібно записати растрове зображення в текст XML-файлу. Клас `QByteArray` надає для цього два методи: `toBase64()` і `fromBase64()`. З назв методів видно, що бінарні дані перетворюватимуться у формат Base64. Цей формат був спеціально розроблений для передачі бінарних даних в текстовій формі. Покажемо приклад таких перетворень до звичайного рядка тексту:

```
QByteArray a = "Test Data";
QByteArray aBase 64 = a.toBase64();
qDebug() << aBase64;
```

На екрані буде виведено: `"VGVzdCBEYXRh"`.

Тепер проведемо зворотне перетворення за допомогою статичного методу `fromBase64()`:

```
qDebug() << QByteArray::fromBase64(aBase64);
```

На екрані буде виведено: `"Test Data"`.

Щоб бінарні дані займали менше місця в текстовому файлі, їх можна перед кодуванням в Base64 стиснути.

Масив бітів QBitArray

Цей клас керує бітовим (чи булевим) масивом. Кожне із значень, яке зберігається, займає тільки один біт. Значення запаковуються у байти за допомогою класу `QByteArray`. Цей тип використовується для зберігання великої кількості змінних типу bool.

Для операцій з бітами цей клас надає методи: для читання `testBit()` і для записування `setBit()`. Разом з цими методами існує також оператор [], за допомогою якого можна звертатися до кожного біту окремо:

```
QBitArray bits(3);
bits[0] = bits[l] = true;
bits[2] = false;
```

Списки QList<T> і QLinkedList<T>

Список – це структура даних, яка є впорядкованим набором пов'язаних один з одним елементів. Перевага списків перед векторами і чергами полягає в тому, що вставлення і вилучення елементів у будь-якій позиції відбувається ефективніше, оскільки для виконання цих операцій змінюється тільки мінімальна кількість вказівників, виняток становить тільки вставлення елементу у центр списку. Але є і недолік – списки погано пристосовані для пошуку заданого елементу за індексом, для цього краще використати вектори.

Увага! Потрібно як менше опитувати кількість елементів списку викликом методу `size()`, оскільки при кожному виклику здійснюватиметься їх підрахунок, а це може суттєво вплинути на швидкість програми. Якщо потрібно вияснити, порожній список або ні, використовуються тільки методи `empty()` або `isEmpty()`.

Списки реалізує клас `QList<T>`. У загальному вигляді цей клас є масивом вказівників на елементи (рис. 4.4).

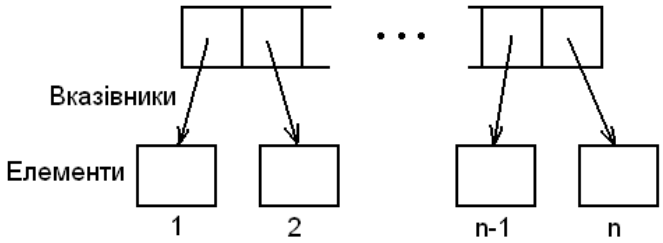


Рисунок 4.4 – Структура списку

Специфічні операції для списків показані в табл. 4.5.

Таблиця 4.5 – Деякі методи контейнера QList<T>

Метод	Описання
`move()`	Переміщає елемент з однієї позиції на іншу
`removeFirst()`	Вилучає перший елемент списку
`removeLast()`	Вилучає останній елемент списку
`swap()`	Міняє місцями два елементи на вказаних позиціях
`takeAt()`	Повертає елемент на вказаній позиції і вилучає його
`takeFirst()`	Вилучає перший елемент і повертає його
`takeLast()`	Вилучає останній елемент і повертає його
`toSet()`	Повертає контейнер QSet<T> з даними, що містяться в об'єкті QList<T>
`toStdList()`	Повертає стандартний список STL std::list<T> з елементами, що містяться в об'єкті QList<T>
`toVector()`	Повертає об'єкт вектор QVector<T> з елементами, що містяться в об'єкті QList<T>

Якщо не планується змінювати значення елементів, то, з міркувань ефективності, не рекомендується використовувати оператор індексації []. Замість цього краще скористатися методом `at()`, оскільки цей метод повертає константне посилання на елемент.

Одна з найпоширеніших операцій – обхід списку для послідовного отримання значень кожного елементу списку. Наприклад:

```
QList<int> list;
list << 10 << 20 << 30;
QValueList<int>::iterator it = list.begin();
while (it != list.end())
{
   qDebug() << "Element:" << *it;
   ++it ;

}
```

На консолі буде відображено наступне:

```
Element: 10
Element: 20
Element: 30
```

У класі `QList<T>` не досить ефективно працює вставлення елементів, тому при роботі з великими списками і при частому вставленні елементів, ефективніше використати двозв'язні списки `QLinkedList<T>`. Хоча цей контейнер витрачає більше пам'яті, ніж `QList<T>`, як це видно з його структури (рис. 4.5), зате операції вставлення і вилучення зводяться до перевизначення чотирьох вказівників, незалежно від позиції елементу, що вставляється або вилучається.

Примітки. В цьому класі не визначені операції для індексного доступу `[]` і `at()`.

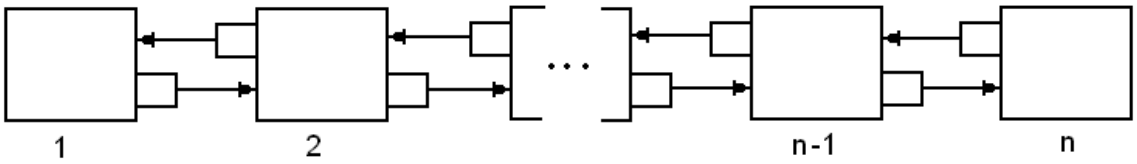


Рисунок 4.5 – Структура двозв'язного списку

Стек QStack<T>

Стек реалізує структуру даних, працюючу за принципом LIFO (Last In First Out, останнім прийшов – першим пішов), тобто із стека першим вилучається елемент, який був вставлений останнім (рис. 4.6).

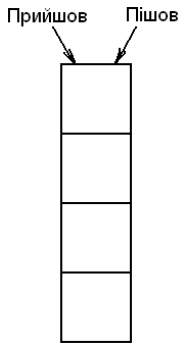


Рисунок 4.6 – Принцип роботи стеку

Клас `QStack<T>` є реалізацією стекової структури. Цей клас успадкований від класу `QVector<T>`. Процес вставлення елементів в стек звичайно називається заштовхуванням (pushing), а видобування – виштовхуваннмя (poping). Кожна операція заштовхування збільшує розмір стека на 1, а виштовхування – зменшує на 1. Для цих операцій в класі `QStack<T>` визначені методи `push()` і `pop()`. Метод `top()` повертає посилання на елемент верхівки стеку. Наступний приклад демонструє використання класу стека.

```
QStack<QString> stk;
stk.push("Era");
stk.push("Corvus Corax");
stk.push("Gathering");
while (!stk.empty())
{
  qDebug() << "Element:" << stk.pop();
}
```

На консолі буде відображено наступне:

```
Element:"Gathering"
Element:"Corvus Corax"
Element:"Era"
```

Черга QQueue<T>

Черга реалізує структуру даних, працюючу за принципом FIFO (First In First Out, першим прийшов – першим пішов), тобто з черги вилучається перший елемент (рис. 4.7). Реалізована черга в класі `QQueue<T>`, який успадкований від класу `QList<T>`.

Наступний приклад демонструє принцип використання черги:

```
QQueue<QString> que;
que.enqueue("Era");
que.enqueue("Corvus Corax");
que.enqueue("Gathering");
while( !que.empty() )
{
  qDebug() << "Element:" <<que dequeue();
}
```

На консолі буде відображено наступне:

```
Element:"Era"
Element:"Corvus Corax"
Element:"Gathering"
```

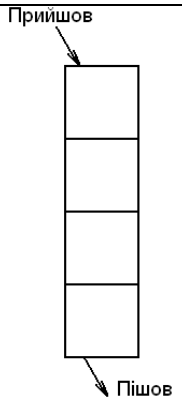


Рисунок 4.7 – Принцип роботи черги

Асоціативні контейнери

Завдання асоціативних контейнерів полягає у збереженні відповідностей ключа і значення. Це дозволяє звертатися до елементів не за індексом, а за допомогою ключа. Для усіх контейнерів цього типу (за деякими винятками для контейнера `QSet<T>`) доступні методи, перераховані в таблиці. 4.6.

Таблиця 4.6 – Загальні методи асоціативних контейнерів

Метод	Описання
`contains()`	Повертає значення true, якщо контейнер містить елемент із заданим ключем. Інакше повертається значення false
`erase()`	Вилучає елемент з контейнера відповідно до переданого ітератора
`find()`	Шукає елемент за значенням. У разі успіху повертає ітератор, що вказує на цей елемент, а у разі невдачі ітератор вказує на метод end()
`insertMulti()`	Вставляє в контейнер новий елемент. Якщо елемент вже є в контейнері, то створюється новий елемент. Цей метод відсутній в класі QSet<T>
`insert()`	Вставляє в контейнер новий елемент. Якщо елемент вже є в контейнері, він заміщається новим елементом. Цей метод відсутній в класі QSet<T>
`key()`	Повертає перший ключ відповідно до переданого в цей метод значення. Цей метод відсутній в класі QSet<T>
`keys()`	Повертає список усіх ключів, що знаходяться в контейнері. Цей метод відсутній в класі QSet<T>
`take()`	Вилучає елемент з контейнера відповідно до переданого ключа і повертає копію його значення. Цей метод відсутній в класі QSet<T>
`unite()`	Додає елементи одного контейнера в інший
`values()`	Повертає список усіх значень, що знаходяться в контейнері

Словник QMap<K, T> і QMultiMap<K, T>

“Програмні” словники подібні на звичайні словники, які використовуються в повсякденному житті. Вони зберігають елементи одного і того ж типу, індексовані ключовими значеннями. Головна особливість словника в тому, що він дозволяє швидко отримати значення, яке асоціюється із заданим ключем. Ключі мають бути унікальними (рис. 4.8), за винятком мультисловника, який допускає дублікати (рис. 4.9).

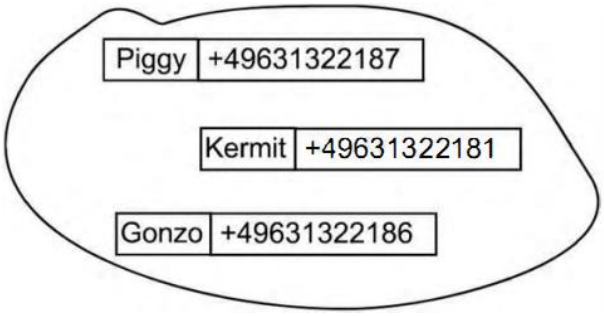


Рисунок 4.8 – Словник

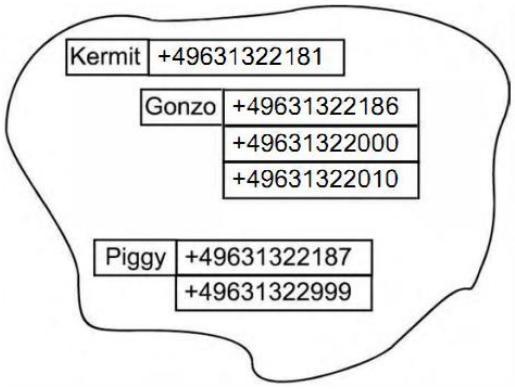


Рисунок 4.9 – Мультисловник

У контейнери цього типу заносяться елементи разом з ключами, за якими їх можна знайти і які можуть мати значення любого типу. У випадку із словником `QMap<K, T>` необхідно стежити за тим, щоб не було внесено два різних елементи з однаковим ключем, адже тоді один з цих елементів неможливо буде знайти. Тобто, кожен ключ словника `QMap<K, T>` має бути унікальний.

При створенні об'єкту класу `QMap<K, T>` потрібно передати його розмір в конструктор. Цей розмір не є, як це прийнято в інших контейнерних класах, розміром, що обмежує максимальну кількість елементів, а є кількістю позицій. Кількість позицій має бути більшою від кількості елементів, які планується зберігати, інакше пошук елементів в словнику буде недостатньо ефективним. Бажано, щоб це значення було простим числом, тоді у цьому випадку розміщення елементів буде більш зручним. Табл. 4.7 містить деякі з методів класу `QMap<K,T>`.

Таблиця 4. 7 – Деякі методи контейнера QMap<K, T>

Метод	Описання
`lowerBound ()`	Повертає ітератор, що вказує на перший елемент із заданим ключем
`toStdMap ()`	Повертає стандартний словник STL з елементами, що знаходяться в об'єкті QMap<T>
`upperBound ()`	Повертає ітератор, що вказує на останній елемент із заданим ключем

Одним з найчастіших способів звернення до елементів словника є використання ключа в операторі []. Але можна обійтися і без нього, оскільки ключ і значення можна отримати за допомогою методів ітератора `key()` і `value()`, наприклад:

```
QMap<QString, QString> mapPhonebook;
mapPhonebook["Piggy"]  "+49 631322187";
mapPhonebook["Kermit"] "+49 631322181"
mapPhonebook["Gonzo"]  "+49 631322186";
QMap<QString ,  QString>::iterator it = mapPhonebook.begin();
for (;it != mapPhonebook.end(); ++it ) {
   qDebug() << "Name:" << it.key() << "Phone:" << it.va1ue();
}
```

На консолі буде відображено наступне:

```
Name: Gonzo  Phone :+49  631322186
Name: Kermit Phone :+49  631322181
Name: Piggy  Phone :+49  631322187
```

Особливу увагу потрібно звернути на використання оператора [], який може застосовуватися як для вставлення, так і для отримання значення елементу. Але потрібно бути обережним, оскільки задання ключа для відсутнього елемент приведе до створення такого

елементу. Щоб уникнути цього, потрібно перевіряти існування елементу, прив'язаного до ключа. Подібну перевірку можна зробити за допомогою методу `contains()`. Наприклад:

```
if ( mapPhonebook.contains("Kermit") )
{
  qDebug() << "Phone:" << mapPhonebook["Kermit"];
}
```

На практиці трапляється так, що потрібно занести зразу декілька телефонних номерів для однієї і тієї ж людини, – наприклад, її домашній, робочий і мобільний телефони. Для цього звичайний словник `QMap<K, T>` вже не підходить, і потрібно скористатися мультисловником `QMultiMap<K, T>`. Приклад такого словника показаний на рис. 4.9, додамо до нього програмний код і взнаємо телефонні номери для Piggy:

```
QMultiMap<QString, QString> mapPhonebook;
mapPhonebook.insert("Kermit", "+49 631322181");
mapPhonebook.insert("Gonzo" , "+49 631322186");
mapPhonebook.insert("Gonzo" , "+49 631322000");
mapPhonebook.insert("Gonzo" , "+49 631322010");
mapPhonebook.insert("Piggy" , "+49 631322187");
mapPhonebook.insert("Piggy" , "+49 631322999");
QMu1tiMap<QString, QString>::iterator it = mapPhonebook.find("Piggy");
for(; it != mapPhonebook.end() && it.key() == "Piggy"; ++it)
{
  qDebug() << it.va1ue ();
}
```

Хеш QHash<K, T> і QMultiHash<K, T>

Функціональність хешів дуже подібна на функціональність словника `QMap<K, T>`, з тією лише різницею, що замість сортування за ключем цей клас використовує хеш-таблицю. Такий підхід дозволяє шукати ключові значення набагато швидше, ніж це робить словник `QMap<K, T>`.

Так само як і у випадку із словником `QMap<K, T>`, потрібно бути обережним при використанні оператора індексації [], оскільки задання ключа, для якого елемент не існує, приведе до створення елементу. Тому важливо перевіряти існування елементу, прив'язаного до ключа за допомогою методу `contains()` контейнера.

Якщо потрібно в хеші `QHash<K, T>` розмістити об'єкти власних класів, то необхідно реалізувати оператор порівняння == і спеціалізовану функцію `qHash()` для власного класу. Ось приклад реалізації оператора порівняння :

```
inline bool operator== (const MyClass& mc1, const MyClass& mc2)
{
   return (mc1.firstName() == mc2.firstName()
   && mc1.secondName() == mc2.secondName()
};
```

Функція `qHash()` повинна повертати число, яке має бути унікальним для кожного елементу кешу. Наприклад:

```
inline uint qHash(const MyClass& mc)
{
  return  qHash(mc.firstName()) ^ qHash(mc.secondName());
}
```

Клас `QMultiHash<K, T>` успадкований від `QHash<K, T>`. Він дозволяє розміщувати значення з однаковими ключами і, в цілому, подібний на клас `QMultiMap<K, T>`, але враховує специфіку свого батьківського класу. Методи, властиві тільки для цих контейнерів, показані в табл. 4.8.

Таблиця 4.8 – Деякі методи контейнерів `QHash<K, T>` і `QMul tiHash<K, T>`

Метод	Описання
`capacity()`	Повертає розмір хеш-таблиці
`reserve()`	Задає розмір хеш-таблиці
`squeeze()`	Зменшує розмір внутрішньої хеш-таблиці для зменшення використовуваного обсягу пам'яті

Множина QSet<T>

Як зауважив математик Георг Кантор, "Множина – це багато чого, яке подумки сприймається як єдине". Це "єдине", в контексті Tulip, є не що інше, як контейнер `QSet<T>`, який записує елементи в деякому порядку і дозволяє дуже швидкого переглянути значень і виконати з ними операції, характерні для множин, – такі як: об'єднання, перетин і різниця. Необхідним умовою є унікальність ключів.

Клас `QSet<T>` базується на використанні хеш-таблиці `QHash<K, T>`, але є її виродженим варіантом, оскільки з ключами не зв'язуються ніякі значення. Головне завдання цього класу полягає у зберіганні ключів. Контейнер `QSet<T>` можна використати як неврегульований список для швидкого пошуку даних. Приклад множин показаний на рис. 4.10, де показані дві множини з трьох елементів кожна. Операції, які можна виконувати з множинами, показані на рис. 4. 11.

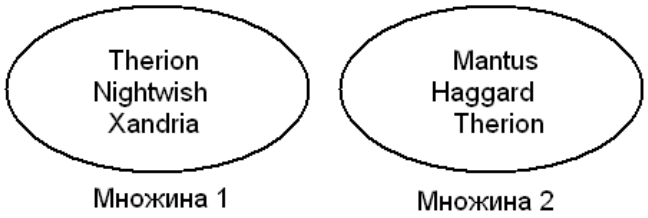


Рисунок 4.10 – Дві множини

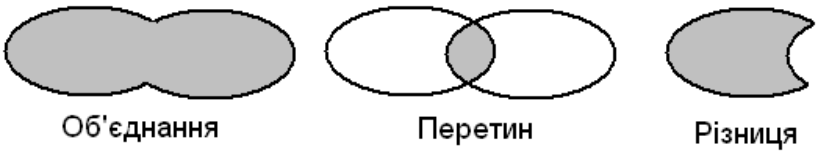


Рисунок 4.11 – Деякі операції над множинами

Створимо дві множини і запишемо в них елементи відповідно до рис. 4.10.

```
QSet<QString> set1;
QSet<QString> set2;
set1 << "Therion" << "Nightwish" << "Xandria";
set2 << "Mantus" << "Haggard" << "Therion";
```

Об'єднаємо ці дві множини, а для того, щоб елементи множин залишилися без змін, використаємо проміжну множину `setResult`:

```
QSet<QString> setResult = set1;
setResult.unite(set2);
qDebug() << "Об'єднання" << setResult.toList();
```

На екрані появиться наступне:

```
Об'єднання = ("Xandria", "Haggard", "Mantus", "Nightwish", "Therion")
```

Тепер зробимо операцію перетину:

```
setResult = set1;
setResult.intersect(set2);
qDebug() << "Перетин set1 з set2 =" << setResult.toList();
```

Оскільки дві множини мають тільки один однаковий елемент, то на екрані буде виведено:

```
Перетин set1 з set2 = ("Therion")
```

І останньою виконаємо операцію різниці двох множин:

```
setResult = set1;
setResult.subtract(set2);
qDebug() << "Різниця set1 з set2 =" << setResult.toList();
```

Множина `set1` відрізняється від множини `set2` двома елементами, тому на екрані повинно відобразитися:

```
Різниця set1 з set2 ("Xandria","Nightwish")
```

В табл. 4.9 показані методи для контейнера `QSet<T>`.

Таблиця 4.9 – Деякі методи контейнера `QSet<T>`

Метод	Описання
`intersect()`	Вилучає елементи множини, не присутні в переданій множині
`reserve()`	Задає розмір хеш-таблиці
`squeeze()`	Зменшує обсяг внутрішньої хеш-таблиці для зменшення використовуваного обсягу пам'яті
`subtract()`	Вилучає усі елементи множини, присутні в переданій множині
`toList()`	Повертає об'єкт контейнера QList<T>, який містить елементи з об'єкту контейнера QSet<T>
`unite()`	Об'єднує елементи множин

Алгоритми

Алгоритми визначені в заголовковому файлі `QtAlgorithms` і надають операції, які застосовуються до контейнерів, – наприклад: сортування, пошук, перетворення даних і т. д. Необхідно зазначити, що алгоритми реалізовані не як методи контейнерних класів, а як шаблонні функції, що дозволяє використати їх як для будь-якого контейнерного класу Tulip, так і для звичайних масивів. Наприклад, для копіювання елементів з одного масиву в інший можна задіяти алгоритм `qCopy()`:

```
QString values[] = {"Xandria", "Therion" , "Nightwish" , "Haggard" };
const int n = sizeof(values) / sizeof(QString);
QString copyOfValues[n];
qCopy (values, values + n, copyOfValues);
```

При копіюванні контейнерів важливо переконатися, що цільовий контейнер має розмір, достатній для розміщення копії. У показаному прикладі задано розмір цільового контейнера такий, як контейнер-джерело.

У табл. 4.1 перераховані усі алгоритми, які надає Tulip. Qt надає тільки найосновніші алгоритми, але якщо їх виявиться недостатньо, завжди можна скористатися алгоритмами STL.

Таблиця 4.10 – Алгоритми

Алгоритм	Описання
`qBinaryFind()`	Двійковий пошук заданих значень
`qCopy()`	Копіювання елементів, починаючи з першого
`qCopyBackward()`	Копіювання елементів, починаючи з останнього
`qCount()`	Підрахунок елементів контейнера
`qDeleteAll()`	Вилучення всіх елементів. Необхідно, щоб елементи контейнера не були константними вказівниками
`qEqual()`	Порівняння. Необхідно, щоб для розміщених об'єктів був визначений оператор ==
`qFill()`	Присвоює всім елементам контейнера задане значення
`qFind()`	Пошук заданих значень

qLowerBound()	Знаходження першого елементу із значенням, більшим або рівним заданому
qUpperBound()	Знаходження першого елементу із значенням, строго більшим заданого
qSort()	Сортування елементів
qStableSort()	Сортування елементів із збереженням порядку слідування рівних елементів
qSwap()	Взаємообмін двох значень місцями

Сортування

Сортування здійснює функція-алгоритм `qSort()`. Для сортування необхідно, щоб до типів елементів контейнера можна було застосувати оператори порівняння, оскільки вони потрібні для прийняття рішення самим алгоритмом. Наприклад, для `QString` ці оператори доступні. Виконаємо сортування для списку з елементами `QString`:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
qSort(list);
qDebug() << "Sorted list=" << list;
```

На екран буде виведено:

```
Sorted list= ("Anubis", "Mantus", "Within Temptation")
```

Можна так само задати межі і умову для сортування, наприклад:

```
qSort(list.begin(), list.end(), caseLessThan);
```

В цьому випадку виконується сортування за алфавітним порядком з врахуванням великих і малих букв в межах від початку до кінця списку. Для відключення врахування великих і малих букв треба скористатися третім параметром `caseInsensitiveLessThan`.

Для сортування чисел у спадаючому порядку як третій параметр потрібно використати `QGreater<T>`, например:

```
QList<int> list;
list << 1 << 2 << 3 << 4 << 5 << 6;
qSort(list.begin(), list.end(), qGreater<int>());
qDebug() << "Sorted list=" << list;
```

На екран буде виведено:

```
Sorted list = (6, 5, 4, 3, 2, 1)
```

Пошук

За пошук елементів відповідає функція-алгоритм `qFind()`. Ця функція повертає ітератор, встановлений на перший знайдений елемент або на `end()`, якщо елемент не знайдений.

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
QList<QString>::iterator it =
    qFind(list.begin(), list.end(),"Anubis");
if (it != list.end())
{
  qDebug() << "Found=" << *it;
{
else {
   qDebug() << "Not Found";

}
```

На екран буде виведено:

```
Found=Anubis
```

Порівняння

Іноді виникає необхідність в порівнянні вмісту контейнерів різних типів. Це можна зробити за допомогою функції-алгоритму `qEqual()`. Як і у випадку сортування, для елементів контейнера мають бути застосовані оператори порівняння.

```
QList<QString> list;
list << "Within Temptation" << "Anubs" << "Mantus";
QVector<QString> vec(3);
vec[O] = "Within Temptation";
vec[l]= "Anubis";
vec[2]= "Mantus";
qDebug() << "Equal=" << qEqual(list.begin(), list.end(),vec.begin());
```

На екран буде виведено:

```
Equal=true
```

Якщо змінити в одному з контейнерів яку-небудь стрічку, наприклад `mantus` на `mantux`, то функція `qEqual()` поверне значення `false`.

Заповнення значеннями

В деяких випадках потрібно присвоїти значення елементам якої-небудь частини контейнера. Для цього існує алгоритм `qFill()`. Присвоїмо усім елементам списку значення `Beatles`:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
qFill(list.begin(), list.end(), "Beatles");
qDebug() << list;
```

На екран буде виведено:

```
("Beatles ",  "Beatles", "Beatles ")
```

Стрічки

Практично всі застосунки оперують текстовими даними. У Qt реалізований клас `QString`, об'єкти якого можуть зберігати стрічки у форматі Unicode, де кожний символ займає два байти. Принцип зберігання даних аналогічний класу `QVector`, єдина відмінність полягає в тому, що елементи завжди є символьного типу `QChar`, тобто стрічка – це контейнер для зберігання символів. Клас `QString` надає цілу серію методів і операторів, які дозволяють виконувати із стрічками різні операції, наприклад: з'єднувати, шукати підстрічки, перетворювати їх у верхній або нижній регістр і т. п.

Стрічки можна порівнювати за допомогою операторів порівняння ==, !=, <, >, <= і >=. Результат порівняння залежить від регістра символів, наприклад:

```
QString str = "Lo";
bool b1 (str "Lo");l1 b true
bool b2 = (str != "LO"); // b2  true
```

За допомогою методу `isEmpty()` можна дізнатися, чи не порожня стрічка. Того ж результату можна добитися, перевіривши довжину стрічки методом `length()`. У класі `QString` є відмінності між порожніми і нульовими стрічками – таким чином, стрічка, створена за допомогою конструктора за замовчуванням, є нульовою стрічкою. Наприклад:

```
QString strl = "";
QString str2;
```

```
str1.isNull(); // false
str2.isNull(); // true
```

Об'єднання стрічок є однією з найпоширеніших операцій. Виконати його можна різними способами: скажімо, за допомогою операторів `+=` і `+` або викликом методу `append()`. Наприклад:

```
QString strl "Lo";
QString str2 = "stris";
QString str3 = strl + str2; // str3 = "Lostris"
Str1.append(str2); //str1 = "Lostris"
```

Для заміни деякої частини стрічки інший клас `QString` надає метод `replace()`. Наприклад:

```
QString str = "Lostris";
str.replace("stris", "gic"); // str = "Logic"
```

Для перетворення даних стрічки у верхній або нижній регістр використовуються методи `toLower()` или `toUpper()`. Например:

```
QString str1 = "LoStRiS";
QString str2 = str1.toLower(); // str2 = ”lostris”
QString str3 = str1.toUpper(); // str3 = ”LOSTRIS”
```

За допомогою методу `setNum()` можна конвертувати числові значення в стрічкові. Того ж результату можна добитися викликом статичного методу `number()`. Наприклад:

```
QString str = QString::number(35.123);
```

Аналогічного результату можна добитися також і за допомогою текстового потоку Qt. Наприклад:

```
QString str;
QTextStream(&str) << 35.123;
```

Перетворення із стрічкового в числове значення робиться методами, що містять у своєму імені назву типу. У цих методах другим параметром можна передавати посилання на змінну булевого типу для отримання інформації про те, чи успішно була проведена операція. Наприклад:

```
bool ok;
QString str = "234";
double d = str.toDouble(&ok);
int n = str.toInt(&ok);
```

Стрічка може бути розбита на масив стрічок за допомогою методу `split()` класу `QStringList`. Наступний приклад створить список з двох рядків: `Ringo` і `Star`:

```
QString str = "Ringo Star";
QStringList list = str.split(" ");
```

Список стрічок об'єднується в одну стрічку за допомогою методу `join()`. Наприклад, об'єднати список з двох елементів (`Ringo` і `Star`) в одну стрічку, розділивши їх знаком пропуску, можна таким чином:

```
str = list.join("  ");
```

Регулярні вирази

Для роботи з регулярними виразами Qt має клас `QRegExp`. Регулярні вирази – це потужний засіб аналізу і оброблення стрічок. Вони містять в собі шаблон, призначений для пошуку в стрічці. Це дозволяє швидко і гнучко видобувати текст, який співпав з шаблоном. Але потрібно

зауважити, що робота з регулярними виразами виконується повільніше за методи, визначені в класі `QString`, і тому їх застосування має бути обгрунтованим. Табл. 4.11 містить основні шаблонні символи, підтримувані класом `QRegExp`.

Таблиця 4. 11 – Шаблони регулярних виразів

Символ	Описання	Приклад
.	Любий символ	a.b
$	Має бути кінець стрічки	abc$
[]	Любий символ із заданого набору	[abc]
-	Визначає діапазон символів у групі	[0-9A-Za-z]
*	Повторення попереднього символу нуль або більше разів	a*b
+	Повторення попереднього символу один або більше разів	a+b
?	Повторення попереднього символу нуль або один разів	a?b
{n}	Повторення попереднього символу n разів	a{3}b
{n,}	Повторення попереднього символу мінімум n разів	a{3,}b
{,n}	Повторення попереднього символу до n разів	a{,3}b
{n,m}	Повторення попереднього символу від n до m разів	a{2,3}b
|	Вибір одного з двох символів	ac|bc
\b	Границя слова	a\b
\B	В цьому місці немає границі слова	a\Bd
( )	Пошук і зберігання в пам'яті групи знайдених символів	(ab|ac)ad
\d	Любе число	
\D	Все, крім числа	
\s	Любий тип пропусків	
\S	Все, крім пропусків	
\w	Люба буква, цифра і знак підкреслення	
\W	Все, крім букв	
\A	Початок стрічки	
\b	Ціле слово	
\B	Не слово	
\Z	Кінець стрічки (співпадає з символом кінця стрічки або перед символом переведення каретки)	
\z	Кінець стрічки (співпадає тільки з кінцем стрічки)	

Для того, щоб знайти один з декількох символів, потрібно помістити їх у квадратні дужки. Наприклад `[ab]` співпадатиме з `a` або `b`. Щоб не писати усі символи підряд, можна вказати діапазон – наприклад `[A-Z]` співпадає з будь-якою буквою у верхньому регістрі, [a-z] – з будь-якою буквою в нижньому регістрі, `a[0-9]` – з будь-якою цифрою. Можна поєднувати такі записи – наприклад `[a-z7]` співпадатиме з будь-якою буквою в нижньому регістрі і з цифрою 7.

Також можна виключати символи, поставивши перед ними знак ^. Наприклад `[^0-9]` відповідатиме усім символам, крім цифр.

Вказані в таблиці. 4.11 величин у фігурних дужках називаються межами. Межі дозволяють точно задати кількість разів, яку символ повинен повторюватися в тексті. Наприклад, `a{4,5}` співпадатиме з текстом, якщо буква `a` зустрінеться в ньому не менше чотирьох, але не більше п'яти разів підряд. Так, в наступному фрагменті задано регулярний

вираз для IP-адреси – ним можна скористатися, наприклад, для того, щоб перевірити стрічку на наявність в ній IP-адреси:

```
QRegExp reg("[0-9]{1,3)}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}");
QString str("this is an ip-address 123.222.63.1 1ets check it");
qDebug() << (str.contains(reg) > 0); // true
```

Зверніть увагу, що для задання символу крапки в регулярному виразі перед нею стоїть зворотна коса риска (\), а відповідно до правил мови C++ вона має екрануватися (подвоюватися). Якби косої риски не було, то крапка мала б у відповідності з табл. 4.11 значення «Будь-який символ», і регулярний вираз розпізнавав би, наприклад, рядок 1z2y3x4, як IP-адресу, що є не правильно.

Шаблони можна комбінувати за допомогою символу |, задаючи галуження в регулярному виразі. Регулярний вираз з двома гілками співпадає з підстрічкою, якщо співпадає одна з гілок. Наприклад:

```
QRegExp exp("(.com|.ua)");
int n1 = rxp.indexin("www.auk.ua"); // n1=7 (співпадіння в 7-й позиції)
int n2 = rxp.indexin("www.auk.uk"); // n2=-1(співпадінь не знайдено)
```

Вказані в таблиці. 4.11 символи із зворотною косою рискою дозволяють значно спростити регулярні вирази. Наприклад, регулярний вираз `[a-zA-Z0-9 ]` ідентичний виразу `\w`.

Для визначення правильності введення адреси електронної пошти (E-mail) можна використати наступний регулярний вираз, вказаний в об'єкті `regEmail`:

```
QRegExp regEmail("([a-zA-Z0-9_\\-\\.]+)@([a-zA-Z0-9_.-])+\\.([a-zA-Z]{2,4}|
 [0-9]{1,3})");
QString strEmail1 = "Max.Abcd@google.com";
QString strEmail2 = "Max.Abcd#google.com";
QString strEmail3 = "Max.Abcd@google";
bool b1 regEmai1.exactMatch(strEmail1);  //bl = true
bool b2 regEmai1.exactMatch(strEmail2);  //b2 = fa1se
bool b3 regEmai1.exactMatch(strEmail3);  //b3 = fa1se
```

Довільний тип QVariant

Об'єкти класу QVariant можуть містити дані різного типу, включно з контейнерами. До цих типів відносяться: `int`, `unsigned int`, `double`, `bool`, `QString`, `QStringList`, `QImage`, `QPixmap`, `QBrush`, `QCo1or`, `QRegExp` та ін. Важливо враховувати ту обставину, що часте застосування цього типу може негативно відбитися на швидкості програми і ефективності використання пам'яті, а також може значно знизити читаність самої програми. Тому об'єкти класу QVariant не слід використати без особливої на те необхідності.

Для створення об'єктів класу `QVariant` необхідно передати в конструктор змінну потрібного типу. Наприклад:

```
QVariant v1(34);
QVariant v2(true);
QVariant v2("Lostris");
```

Метод `type()` дозволяє взнати тип записаних в об'єкт QVariant даних. Цей метод повертає цілочисельний ідентифікатор типу. Щоб перетворити його в стрічку, слід передати його в статичний метод `typeToName()`. Того ж результату можна добитися і викликом методу `typeName()`, який повертає інформацію про тип у вигляді стрічки:

```
QVariant v(5.0);
qDebug() << QVariant::typeToName(v.type()); // =>double
```

Щоб отримати дані з об'єкту `QVariant` потрібного типу, існує серія спеціальних методів `toT()`, де T – це ім'я типу. Метод `toT()` створює новий об'єкт типу T і копіює дані з об'єкту QVariant в потрібний об'єкт. Наприклад:

```
QVariant v2(23);
int a= v2.toInt() + 5; // a = 28
```

Примітка. З огляду на те, що `QVariant` реалізований в `Qtcore`, відповідні методи `toT()` для класів `QColor`, `Qimage` і `QPixmap` та ін., що знаходяться в модулі `QtGui`, не надаються.

Замість методів `toT()` для перетворення до потрібного типу можна використати також шаблонний метод `value<T>()`. Приклад перетворення об'єкту QVariant до цілого типу:

```
QVariant v2(23);
int a= v2.value<int>() + 5;  //a = 28
```

або наприклад для об'єкту `QPixmap`

```
QPixmap pix(":/myimg.png"); // створюємо об'єкт QPixmap
QVariant vPix = pix;        // конвертуємо його в QVariant, неявним чином
QPixmap::operator QVariant()
QPixmap pix2 = vPix.value<QPixmap>(); // отримання об'єкту QPixmap з QVariant
```

Модель спільного використання даних

Для забезпечення ефективності у багатьох класах Qt намагаються уникнути копіювання даних – замість цього використовується посилання на потрібні дані (рис. 4.12). Цей принцип дістав назву *спільного використання даних* (shared data). У Qt застосовується модель неявних спільних даних. У такій моделі виклик конструктора копіювання або оператора присвоєння не спричинить копіювання даних, а тільки збільшить лічильник посилань на ці дані на 1. Відповідно, при вилученні елементу лічильник посилань зменшиться на 1. Якщо значення лічильника посилань стає рівним 0, то дані знищуються. Копіювання даних відбувається тільки при змінах – відповідно, значення лічильника посилань при цьому зменшується.

На рис. 4.12 на першому кроці створюються два об'єкти, і оскільки дані їм не були присвоєні, то вони обидва вказують на `shared_null` (спільний нуль). На другому кроці першому об'єкту присвоюються дані, і лічильник посилань стає рівним одиниці. На третьому кроці другому об'єкту присвоюється перший об'єкт, і вони тепер обидва вказують на одні і ті ж дані, лічильник посилань при цьому збільшується на одиницю. На четвертому кроці змінюються дані першого об'єкту, що призводить до створення для нього окремої копії, а лічильник посилань старих даних зменшується на одиницю, оскільки на один об'єкт, який використовує ці дані, стало менше. Якби п'ятим кроком змінили дані другого об'єкту, то після створення копії для нових даних лічильник посилань старих даних зменшився б до значення 0, і це привело б до звільнення пам'яті і знищення старих даних.

Проілюструємо зображену на рис. 4.12 ситуацію програмним кодом:

```
QString  str1 ;       // Посилається на  shared  null
QString  str2 ;       // Посилається на  shared  null
str1 = "Нова стрічка" // Посилання на дані, лічильник посилань = 1
str2 = str1;          //str1 і str2 вказують на одні і ті ж дані,
                      // лічильник посилань = 2
str1 += " добавлення";  // копіювання даних для str
```

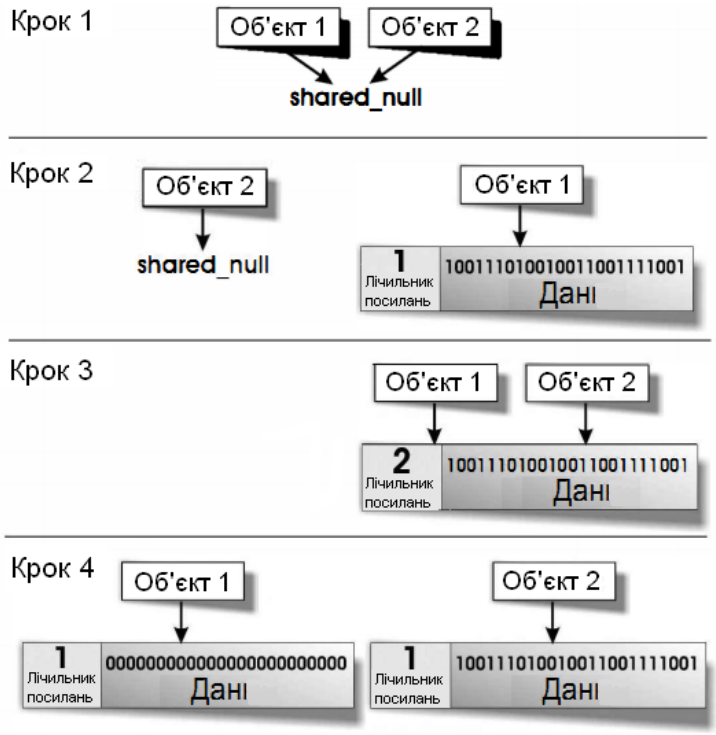


Рисунок 4.12 – Чотири кроки використання спільних даних

Висновки

• Бібліотека Qt не є монолітною бібліотекою, а розбита на окремі модулі: `QtCore`, `QtGui`, `QtNetwork`, `QtOpenGL`, `QtSql`, `QtXml` і `QtSvg`. Кожен модуль має своє призначення, наприклад програмування інтерфейсу користувача, графіки, баз даних та ін. Класи модулів надають розробникові механізми, що розширюють можливості програмістів і, в той же час, спрощують створення застосунків.

• Верхівкою модульної ієрархії є модуль QtCore, який дозволяє реалізовувати застосунки без графічного інтерфейсу користувача (консольні застосунки).

• Об’ єкт класу `QCoreApplication` має бути створений в застосунку тільки один раз.

• Для реалізації застосунків з графічним інтерфейсом користувача потрібний модуль `QtGui`.

• Клас `QApplication` є стержнем будь-якого Qt-застосунку з графічним інтерфейсом. Об’єкт цього класу також не повинен створюватися в застосунку більше одного разу. Цей об’єкт використовується для отримання подій клавіатури, миші та ін.

• Сигнали і слоти можуть бути з’єднані один з одним, причому сигнал може бути з’єднаний з великою кількістю слотів. Слот, у свою чергу, також може бути з’єднаний з багатьма сигналами. У разі коли слот не робить нічого, окрім відправки отриманого сигналу далі, то можна взагалі обійтися без нього, а просто з’єднати сигнали один з одним. Методи сигналів мають бути позначені у визначенні класу спеціальним словом `signals`, а слоти – `slots`. При цьому слоти є звичайними методами мови C++ і в їх визначенні можуть бути присутніми модифікатори `public`, `protected`, `private`. Реалізацію коду для сигналів бере на себе MOC. Сигнал відправляється за допомогою ключового слова `emit`. Клас, що містить сигнали і слоти, має бути успадкований від класу `QObject` або від класу, успадкованого від цього класу. Сигнально-слотові з’єднання завжди можна вилучити (від’єднати), скориставшись методом `disconnect()`, але це потрібно дуже рідко, так як при вилученні об’єкту автоматично знищуються усі його з’єднання. Об’єкти з’єднуються за допомогою статичного методу `QObject::connect()`.

• `QObject` – клас, що по суті є основним класом при програмуванні з використанням Qt. Конструктор класу `QObject` має два параметри: перший використовується для створення

об’єктних ієрархій, а другий – для присвоєння об’єкту імені. Властивості об’єктів важливі, так як дозволяють отримувати інформацію про клас і про об’єкт в процесі виконання програми. Усі об’єкти класу `QObject` або успадкованих від нього класів повинні створюватися динамічно оператором `new`, а про звільнення пам’яті створеної об’єктної ієрархії програміст може не турбуватися.

• Оскільки концепцію сигналів і слотів, а також інформацію про успадкування неможливо було реалізувати засобами самої мови C++, був створений спеціальний препроцесор, MOC (метаоб’єктний компілятор), завдання якого – створювати для заголовкових файлів додаткові `cpp`-файли, що підлягають компіляції і приєднання їх об’єктного коду до виконуваного коду програми. Для того, щоб MOC міг розпізнати класи, що потребують подібної переробки, такий клас повинен містити макрос `Q_OBJECT`.

• Утиліта `qmake`, бере на себе усю роботу із створення з файлів проекту `make`-файлів для будь-якої платформи. Препроцесор MOC при запуску створює додатковий код підтримки сигналів і слотів.

• Контейнер – це об’єкт, призначений для зберігання і керування елементами, які містяться в ньому. Він забезпечує виділення і звільнення пам’яті, а також відповідає за додавання і вилучення елементів. Контейнерні класи поділяються на послідовні і асоціативні. До послідовних контейнерів відносяться вектор, список, стек і черга, до асоціативних: множина, словник і хеш.

• Для проходження по елементах контейнера використовуються ітератори. Qt надає ітератори в стилях Java і STL. Як альтернативу для проходження по елементах контейнера можна також використати макрос `foreach`. За допомогою алгоритмів можна виконувати такі операції над вмістом контейнерів, як сортування, пошук і багато що інше.

• Клас QString є реалізацією стрічок і містить цілий ряд методів для виконання різного роду операцій з ними.

• Регулярні вирази є потужним механізмом для перевірки стрічок на відповідність шаблону.

• Об’єкти класу `QVariant` можуть містити дані різного типу, включаючи також і контейнери.

Запитання

1. Призначення модулів бібліотеки Qt.
2. Призначення модуля `QtCore`.
3. Призначення модуля `QtGui`.
4. Особливості об’єктної моделі Qt.
5. Механізм сигналів і слотів в Qt.
6. Створення і відправлення сигналів.
7. Особливості створення слотів для приймання сигналів.
8. З’єднання і роз’єднання об’єктів сигналів і слотів.
9. Організація об’єктів в ієрархії в Qt.
10. Метаоб’єктна інформація в Qt.
11. Засоби для розроблення проектів в Qt.
12. Метаоб’єктний компілятор і компілятор ресурсів
13. Бібліотека контейнерів. Оператори і методи контейнерів.
14. Послідовні контейнери.
15. Асоціативні контейнери.
16. Алгоритми для роботи з контейнерами.
17. Регулярні вирази Qt.
18. Довільний тип QVariant.
19. Модель спільного використання даних.

Література

1. Шлее М. Qt 5.3. Профессиональное программирование на C++. – СПб.: БХВ-Петербург, 2015. – 928 с.
2. Lee Zhi Eng. Qt 5 GUI Progranning Book. Birmingham, UK: Pack Publishing, 1026. – 283.

2. ЕЛЕМЕНТИ КЕРУВАННЯ

Мета. Вивчення елементів керування в QT.

Вступ. До елементів керування відносяться елементи відображення, кнопки, прапорці, перемикачі, поля введення, елементи вибору,

План.
1. Розміри і координати елеметів
2. Клас Qwidget
3. Розміри і координати віджету
4. Механізм закулісного зберігання
5. Установка фону віджета
6. Зміна вказівника миші
7. Стек віджетів.
8. Рамки
9. Віджет оглядової прокрутки

1. Розміри і координати елементів

Практично будь-яка програма має графічний інтерфейс користувача (GUI, Graphical User Interface). Віджети (widgets) – це «будівельний матеріал» для його створення. Віджет – це не просто область, яка відображається на екрані, це компонент, здатний виконувати різні дії, – наприклад, реагувати на сигнали і події або відправляти сигнали іншим віджетів. Qt надає повний арсенал віджетів: від кнопок меню до діалогових вікон, необхідних для створення професійних додатків. Якщо не вистачає цих віджетів, то можна створити свої власні, успадковуючи класи вже існуючих. Ієрархія, показана на рис. 1, містить класи віджетів.

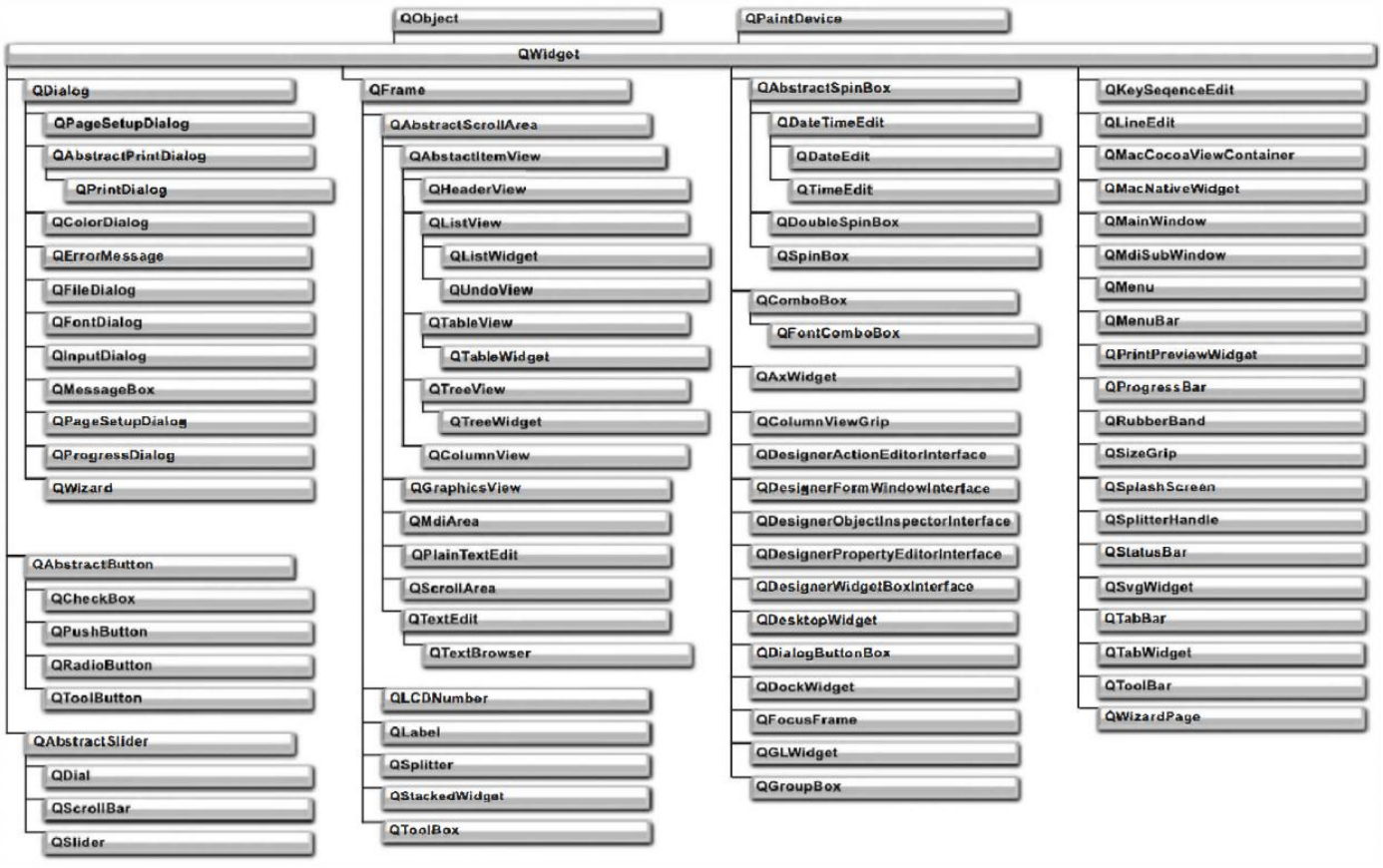


Рисунок 1 – Ієрархія класів віджетів

2. Клас QWidget

Клас `QWidget` є фундаментальним для всіх класів віджетів. Його інтерфейс містить 254 методи, 53 властивості і масу визначень, необхідних кожному з віджетів, наприклад, для зміни розмірів, розташування, обробки подій та ін. Сам клас `QWidget`, як видно з рис. 1, успадкований від класу QObject, а значить, може використовувати механізм сигналів/слотів і механізм об'єктної ієрархії. Завдяки цьому віджети можуть мати нащадків, які відображаються всередині предка. Це дуже важливо, тому що кожен віджет може служити контейнером для інших віджетів, – тобто в Qt немає поділу між елементами керування і контейнерами. Віджети в контейнерах можуть виступати в ролі контейнерів для інших віджетів, і так до нескінченності. Наприклад, діалогове вікно містить кнопки Ok і Cancel (Скасувати) – отже, воно є контейнером. Це зручно ще й тому, що якщо віджет-предок стане недоступним або невидимим, то віджети-нащадки автоматично візьмуть його стан. Віджети без предка називаються віджетами верхнього рівня (top-1eve1 widgets) і мають своє власне вікно. Всі віджети без винятку можуть бути віджетами верхнього рівня. Позиція віджетів-нащадків всередині віджета-предка може змінюватися методом `setGeometry()` вручну або автоматично, за допомогою спеціальних класів компонування (Layouts). Для відображення віджету на екрані викликається метод `show()`, а для приховання - метод `hide()`.

*Примітка***.** Не забувайте, що після створення віджета верхнього рівня, щоб показати його на екрані, потрібно викликати метод `show()`, інакше і його вікно, і віджети-нащадки будуть невидимими.

Клас `QWidget` і більшість успадкованих від нього класів мають конструктор з двома параметрами:

```
QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0)
```

З визначення видно, що не обов'язково передавати параметри в конструктор, так як вони дорівнюють нулю за замовчуванням. А це означає, що якщо конструктор викликається без аргументів, то створений віджет стане віджетом верхнього рівня. Другий параметр `Qt::WindowsFlags` служить для задання властивостей вікна, і з його допомогою можна керувати зовнішнім виглядом вікна і режимом відображення (щоб вікно не перекривався іншими вікнами і т. д.). Щоб змінити зовнішній вигляд вікна, необхідно в другому параметрі конструктора передати значення модифікаторів, об'єднані з типом вікна (рис. 2) побітовою операцією АБО, позначеної символом |. Аналогічного результату можна домогтися викликом методу `setWindowFlags()`. Наприклад:

```
wgt.setWindowFlags(Qt::Window | Qt::WindowTitleHint |
Qt::WindowStaysOnTopHint);
```

На рис. 2 зображені деякі з варіантів застосування цих значень.

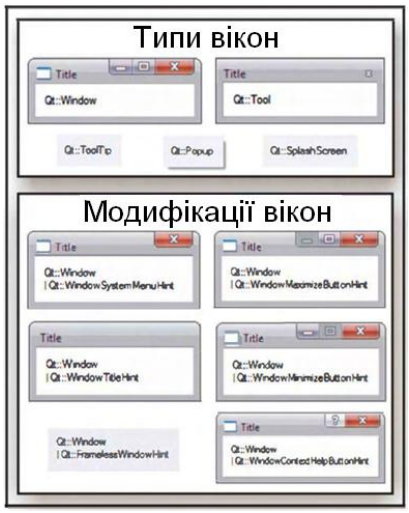


Рисунок 2 – Вид вікон віджетів верхнього рівня

*Примітка***.** Значення `Qt::WindowStaysOnTopHint` не змінює зовнішній вигляд вікна, а лише рекомендує, щоб вікно завжди знаходилося на передньому плані і не перекривався іншими вікнами.

За допомогою слот-методу `setWindowTitle()` встановлюється напис заголовка вікна. Але це має сенс тільки для віджетів верхнього рівня. Наприклад:

```
wgt.setWindowTitle("My Window");
```

Слот `setEnabled()` встановлює віджет в доступний (enabled) або недоступний (disabled) стан. Параметр `true` відповідає доступному, а `false` – недоступному стану. Щоб дізнатися, в якому стані знаходиться віджет, викликається метод `isEnabled()`.

При створенні власних класів віджетів важливо, щоб віджет був в змозі обробляти події. Наприклад, для обробки подій миші необхідно перезаписати хоча б один з наступних методів: `mousePressEvent()`, `mouseMoveEvent()`, `mouseReleaseEvent()` або `mouseDoubleClickEvent()`.

3. Розміри і координати віджета

Віджет являє собою прямокутну область (рис. 3). Існує цілий ряд методів, за допомогою яких можна дізнатися місцезнаходження віджета і його розміри. Методи `size()`, `height()` і `width()` повертають розміри віджета. При цьому, якщо виклики `height()` і `width()` повернуть значення висоти і ширини цілого типу, відповідно, то виклик методу `size()` поверне об'єкт класу `QSize`, який зберігає ширину і висоту віджета.

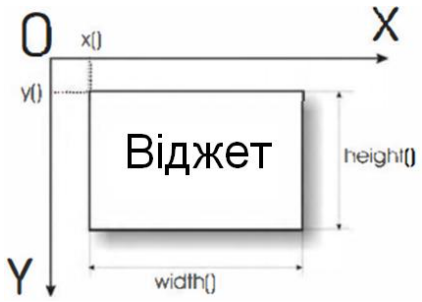


Рисунок 3 – Віджет в області екрана (або предка)

Методи `x()`, `y()` і `pos()` служать для визначення координат віджета. Перші два методи повертають цілі значення координат по осях X і Y, а метод `pos()`– об'єкт класу QPoint, який зберігає обидві координати.

Метод `geometry()` повертає об'єкт класу `QRect`, що описує стан і розміри віджета. Положення віджета можна змінити методом `move()`, а його розміри - методом `resize()`. Наприклад:

```
pwgt->move(5, 5 );
pwgt->resize(260, 330 );
```

Одночасно змінити і положення, і розміри віджета можна, викликавши метод `setGeometry()`. Перший параметр цього методу задає координату лівого верхнього кута віджета по осі X, другий – по осі Y, третій задає ширину, а четвертий – висоту. Наприклад, наступний виклик еквівалентний двом раніше пріведеіним викликам `move()` і `resize()`:

```
pwgt->setGeometry (5, 5, 2 60, 330);
```

4. Механізм закулісного зберігання

Техніка закулісного зберігання (Backing Store) полягає в запам'ятовуванні в пам'яті комп'ютера растрових зображень для всіх віджетів вікна в будь-який час, що дозволяє дуже швидко поміщати потрібну частину збереженої області без виклику системою подій малювання (PaintEvent). При цьому не грає ролі, наскільки складне малювання самого віджета. Виклик події малювання для віджета виконується тільки в тих випадках, коли це дійсно необхідно, – наприклад, для зміни фону. Ця техніка дозволяє значно підвищити продуктивність.

5. Установка фону віджета

Віджету можна задати фон, причому це може бути колір або растрове зображення. Для заповнення суцільним кольором або растровим зображенням необхідно спочатку створити об'єкт палітри, а потім викликом методу `setPalette()` встановити його у віджеті.

Віджет має важливе властивість `autoFillBackground`, яке за замовчуванням дорівнює `false`. Внаслідок цього всі нащадки віджета не заповнюються фоном і, відповідно, невидимі. Встановивши цю властивість рівною `true`, можна змусите віджет заповнювати фон автоматично, що зробить його видимим. Наприклад:

```
wgt.setAutoFillBackground(true);
```

У роздруку 1 створюється віджет верхнього рівня `wgt`, який передається двом іншим віджетів (вказівники `pwgt1` і `pwgt2`) в якості предка.

Роздрук 1. `Файл main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QWidget* pwgt1 = new QWidget(&wgt);
    QPalette pal1;
    pal1.setColor(pwgt1->backgroundRole(), Qt::blue);
    pwgt1->setPalette(pal1);
    pwgt1->resize(100, 100);
    pwgt1->move(25, 25);
    pwgt1->setAutoFillBackground(true);
    QWidget* pwgt2 = new QWidget(&wgt);
    QPalette pal2;
    pal2.setBrush(pwgt2->backgroundRole(), QBrush(QPixmap(":/stone.jpg")));
    pwgt2->setPalette(pal2);
    pwgt2->resize(100, 100);
    pwgt2->move(75, 75);
```

```
    pwgt2->setAutoFillBackground(true);
    wgt.resize(200, 200);
    wgt.show();
    return app.exec();
}
```

Першому віджету методом `setPalette()` передається об'єкт палітри, який встановлює в ньому суцільний колір фону (блакитний). Після зміни його розмірів (методом `resize()`) і переміщення в області віджету-предка (методом `move()`) за допомогою методу `setAutioFillBackround()` властивості `autoFillBackground` присвоюється значення `true`, щоб віджет став видимим.

З другим віджетом-нащадком (вказівник `pwgt2`) виконуються ті ж операції, що і з першим. Різниця полягає в тому, що в другому віджеті в якості фону встановлюється растрове зображення з файлу `stone.jpg` за допомогою об'єкта палітри `pal2`. В результаті один віджет заповнений суцільним кольором, а інший – растровим зображенням (рис. 4).

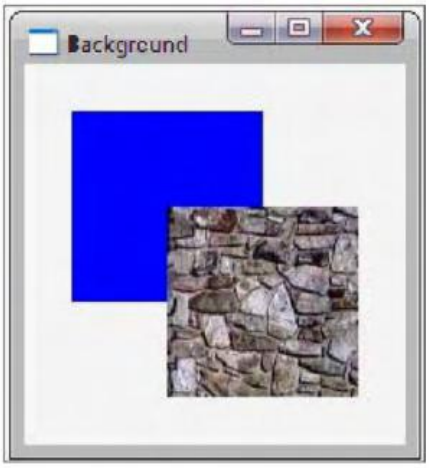


Рисунок 4 – Віджети з фоном

6. Зміна вказівника миші

Клас вказівника (курсора) миші `QCursor` визначено у файлі `QCursor`. Вказівник являє собою невелику растрову картинку, що інформує користувача про позицію миші на екрані. Залежно від місця розташування, зовнішній вигляд вказівника може змінюватися. У більшості випадків це стрілка, але, скажімо, при попаданні на межі вікна він може перетворитися в двонаправлену стрілку, інформуючи користувача, що розміри вікна можуть бути змінені.

Встановити зображення вказівника можна методом `setCursor()`, передавши йому одне із зазначених в табл. 1 значень.

Таблиця 1 – Значення `CursorShape` простору імен Qt

Значення	Вид	Опис
ArrowCursor		Стандартний вказівник стрілки, він з'являється поверх більшості віджетів. Служить для вказівки, вибору або переміщення об'єкта
UpArrowCursor		Стрілка, що показує вгору. Застосування цього вказівника залежить від конкретної ситуації
CrossCursor		Хрестоподібний вказівник і Використовується для виділення прямокутних областей. Може з'явитися над будь-яким віджетом, допускає цю операцію

WaitCursor		Вказівник очікування, з'являється над будь-яким віджетом або позицією при виконаний її операци і в фоновому режимі
IbeamCursor		I-образний текстовий вказівник (I-beam cursor), являє собою верти локальні лінію. Появляється над текстом для його зміни, вибору і переміщення
PointingHandCursor		Вказівник у вигляді руки, з'являється над гіпертекстовими посиланнями
ForbiddenCursor		Вказівник неможливості входу, з'являється над об'єктом-приймачем при проведенні операції перетягування, сигналізуючи про те, що приймаюча сторона не в змозі прийняти перетаскування
WhatsThisCursor		Вказівник з питанням, співаючи вляется поверх більшості віджетів для отримання контекстно-залежної допомоги
SizeVerCursor		Вказівник зміни вертикального розміру вікна, з'являється поверх регульованої межі вікна
SizeHorCursor		Вказівник зміни горизонтального розміру вікна, з'являється поверх регульованої межі вікна
SizeBDiagCursor		Вказівник зміни розмірів вікна по діагоналі, з'являється поверх регульованої межі вікна
SizeFDiagCursor		Вказівник зміни розмірів вікна з іншої діагоналі, з'являється поверх регульованої межі вікна
SizeAllCursor		Вказівник для зміни місця розташування вікна, сигналізує про готовність вікна бути переміщеним
SplitVCursor		Вказівник зміни висоти для розділених віджетів, з'являється над кордоном між двома розділеними віджетами. Поділ віджетів описано в розділі 6
SplitHCursor		Вказівник зміни ширини для розділених віджетів, з'являється над кордоном між двома розділеними віджетами. Поділ віджетів описано в розділі 6
OpenHandCursor		Вказівник у вигляді розціпленої руки, сигналізує про можливість переміщення частин зображення у видимій області
ClosedHandCursor		Вказівник у вигляді стислої руки, сигналізує про готовність переміщення частин зображення у видимій області
BlankCursor	Пустий вказівник	Порожній вказівник, говорить про неможливість використання миші

Примітка. Виклик статичного методу `QGuiApplication::setOverrideCursor()` встановлює зображення вказівника для всієї програми. Це може знадобитися, наприклад, для інформування користувача про те, що додаток виконує інтенсивну, тривалу за часом операцію і не в змозі реагувати на команди. В цей момент всі віджети повинні відображати курсор миші у вигляді пісочного годинника, для чого викликається метод `QGuiApplication::setOverrideCursor(Qt::WaitCursor)`. Коли додаток знову буде в змозі виконувати команди користувача, викликом статичного методу `QGuiApplication::restoreOverrideCursor()` вказівнику миші повертається його колишній вигляд.

У класі `QCursor` міститься метод `pos()`, який повертає поточну позицію вказівника миші щодо лівого верхнього кута екрану. За допомогою методу `setpos()` можна переміщати вказівник миші.

Щоб створити власне зображення вказівника миші, потрібні два растрових зображення типу `QBitmap`. Ці зображення повинні мати однакові розміри, а одне з них являти собою бітову маску. У тих місцях, на яких маска буде мати колір `color1`, буде намальовано саме зображення вказівника, а в місцях, де маска буде мати колір `color0` – зображення буде прозоре.

Більш простий спосіб – це використання об'єкта класу `QPixmap`. Результат виконання програми (роздрук 2), показаний на рис. 5, демонструє цю можливість.

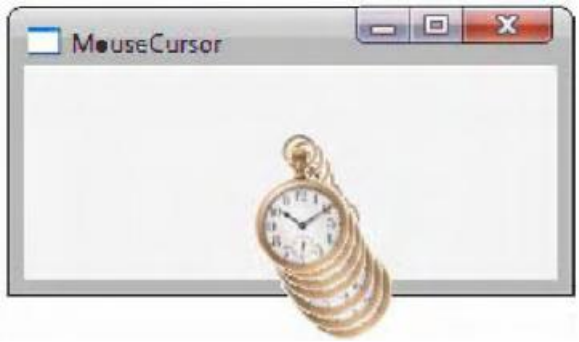


Рисунок 5 – Використання власного зображення для вказівника миші

Роздрук 2. Зміна вказывника миші

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPixmap      pix(":/clock.png");
    QCursor      cur(pix);
    wgt.setCursor(cur);
    wgt.resize(180, 100);
    wgt.show();
    return app.exec();
}
```

Уроздруку 2 спочатку створюється віджет `wgt`, потім – об'єкт растрового зображення `pix`, в конструктор якого передається ім'я `png`-файлу, що представляє собою растрове зображення та бітову маску. Для створення вказівника миші об'єкт растрового зображення передається в конструктор класу `QCursor` і за допомогою методу `setCursor()` встановлюється в віджеті.

7. Стек віджетів

Клас `QStackedWidget` успадкований від класу `QFrame` і являє собою віджет, який показує в окремо взятий проміжок часу тільки одного зі своїх нащадків. Віджети додаються в стек за допомогою методу `addWidget()`. Він приймає вказівник на віджети і повертає присвоєний віджету ідентифікаційний номер. Вилучення віджетів із стека здійснюється викликом методу `removeWidget()`, в який передається вказівник на віджет. Передаючи вказівник на віджет в слот `setCurrentWidget()` або ідентифікаційний номер віджета в слот `setCurrentindex()`, можна зробити його видимим. Ідентифікаційний номер віджета можна дізнатися викликом методу `indexOf()`, передавши йому вказівник на віджет.

8. Рамки

Клас `QFrame` успадкований від класу `QWidget` і розширює його можливістю відображення рамки. Цей клас є базовим для великого числа класів віджетів. Стиль рамки може бути різним, і встановлюється він за допомогою методу `setFrameStyle()`, якому передаються прапори форми і прапори тіней рамки. Значення з'єднуються один з одним побітовою операцією | (АБО).

Існують три прапори тіней (табл. 5.2): `QFrame::Raised`, `QFrame::Plain` і `QFrame::Sunken`. З їх допомогою досягається ефект увігнутості або опуклості рамки.

Таблиця 2 – Приклади рамок

Прапорці	Вигляд	Прапорці	Вигляд
Box | Plain		HLine | Plain	
Box | Raised		HLine | Raised	
Box | Sunken		HLine | Sunken	
Panel | Plain		VLine | Plain	
Panel | Raised		VLine | Raised	
Panel | Sunken		VLine | Sunken	
WinPanel | Plain		StyledPanel | Plain	
WinPanel | Raised		StyledPanel | Raised	
WinPanel | Sunken		StyledPanel | Sunken	

Налаштувавши вигляд рамки можна скориставшись однією з п'яти основних форм (див. табл. 5.2): `QFrame::Box`, `QFrame::Panel`, `QFrame::WinPanel`, `QFrame::HLine` або `QFrame::VLine`. Якщо потрібно, щоб рамка взагалі не відображалася, то тоді в метод `setFrameStyle()` передається значення `QFrame::NoFrame`.

Методом `setContentsMargin()` класу `QWidget` встановлюється відстань від рамки до вмісту віджета, а методами `setLineWidth()` і `setMidLineWidth()` можна змінювати товщину самої рамки.

```
QFrame pfrm = new QFrame;
pfrm->setFrameStyle(QFrame::Box | QFrame::Sunken);
pfrm->setLineWidth(3);
```

У цьому прикладі створюється віджет рамки, в якому методом setFrameStyle() встановлюється потрібний стиль рамки, а методом `setLineWidth()` – її товщина.

9. Віджет оглядової прокрутки

Базовий клас для оглядової прокрутки `QAbstractScrollArea` успадкований від класу `QFrame` і являє собою вікно для перегляду тільки частини інформації. Сам віджет оглядової прокрутки реалізує клас `QScrollArea`.

Цей віджет може розміщувати віджети нащадків, а якщо хоча б один з них вийде за межі вікна перегляду, то автоматично з'являються вертикальна і/або горизонтальна смуги прокрутки. З їх допомогою можна переміщати частини віджету в область перегляду.

Якщо ви хочете, щоб смуги прокрутки було видно завжди, то потрібно передати в методи управління поведінкою смуг значення `Qt::ScrollBarAlwaysOn`. Наприклад:

```
QScrollArea sa;
sa.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
sa.setVerticalScrol lBarPolicy(Qt::ScrollBarAlwaysOn);
```

Як видно з рис. 6, віджет оглядової прокрутки є сукупністю відразу декількох віджетів, які працюють разом. Вказівники на ці віджети можна отримати позначеними на малюнку методами. Здійснити доступ до віджету області перегляду можна за допомогою методу `QAbstractScrollArea::viewport()`. Методи `verticalScrollBar()` і `horizontalScrollBar()` повертають вказівники на віджети вертикальної і горизонтальної смуги прокрутки (класу `QScrollBar`) відповідно. Метод `cornerWidget()` повертає вказівник на віджет, що знаходиться в правому нижньому кутку.

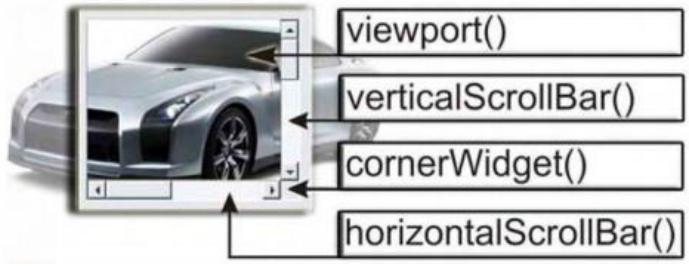


Рисунок 6 – Структура віджета оглядової прокрутки

Встановити віджет у віджеті оглядової прокрутки можна за допомогою методу `setWidget()`, передавши йому вказівник на нього. Ця операція автоматично зробить переданий віджет нащадком віджета області перегляду. Вказівник встановленого віджета завжди можна отримати методом `widget()`. Вилучення віджета з `QScrollArea` здійснюється викликом методу `removeChild()`.

На рис. 7 показаний результат роботи програми застосування видовий прокрутки (роздрук 3). Програма дозволяє переміщати частини зображення в видиму область вікна.

Рисунок 7 – Приклад віджету оглядової прокрутки

Роздрук 3. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
```

```
    QApplication app(argc, argv);
    QScrollArea  sa;
    QWidget*     pwgt = new QWidget;
    QPixmap      pix(":/img.jpg");
    QPalette pal;
    pal.setBrush(pwgt->backgroundRole(), QBrush(pix));
    pwgt->setPalette(pal);
    pwgt->setAutoFillBackground(true);
    pwgt->setFixedSize(pix.width(), pix.height());
    sa.setWidget(pwgt);
    sa.resize(350, 150);
    sa.show();
    return app.exec();
}
```

У роздруку 3 створюється віджет оглядової прокрутки sa. Потім створюється звичайний віджет (вказівник `pwgt`) і об'єкт растрового зображення `pix`. Об'єкт растрового зображення ініціалізується файлом `img.jpg`, який потім встановлюється викликом методу `setPalette()` в якості заднього фону віджета. Для того щоб віджет було видно, метод `setAutoFillBackground()` включає режим автоматичного заповнення фону. Розміри віджета (вказівник `pwgt`) приводяться у відповідність з розмірами растрового зображення викликом методу `setFixedSize()`. Потім віджет оглядової прокрутки `sa` викликом методу `setWidget()` додає в своє вікно створений нами віджет.

Висновки

• У центрі створення призначеного для користувача інтерфейсу стоїть поняття віджет (елемент управління). Клас `QWidget` є базовим для всіх елементів управління. Все, з чого в основному складається інтерфейс користувача в додатках Qt, – це об'єкти класу `QWidget` і успадкованих від нього класів.

• Поділ між віджетами-контейнерами і просто віджетами відсутня – будь-який віджет може використовуватися в якості контейнера для інших віджетів. До основних операцій з віджетами, крім показу (`show`) і приховування (`hide`), відносяться методи, що дозволяють змінювати їх розміри і розташування.

• Позиція і розміри віджетів всередині віджета-предка можуть при використанні класів компонування встановлюватися автоматично.

• Віджети верхнього рівня мають своє власне вікно, яке можна по-різному декорувати, – наприклад, змінювати рамку вікна.

• У кожному віджеті можна встановлювати вказівники миші. Qt надає ряд зумовлених зображень таких вказівників, які можна використовувати для установки. Є також можливість створювати свої власні вказівники з растрових зображень.

• Клас `QFrame` успадкований від класу `QWidget` і являє собою прямокутник з рамкою, стиль якої можна міняти.

• Клас `QStackedWidget` показує в окремо взятий проміжок часу тільки одного з нащадків. Цим властивістю користуються тоді, коли є багато віджетів і потрібно, щоб тільки один з них в певний час був видимим.

• Віджет оглядової прокрутки надає вікно для перегляду тільки частини інформації. Цей віджет застосовується для відображення інформації, розміри якої перевищують виділену для неї область перегляду.

Запитання

1. Ієрархія класів віджетів.
2. Клас `QWidget`.
3. Класи для отримання розмірів і координат віджета.

4. Класи для зміни вказівника миші.
5. Класи відображення рамки.
6. Віджет оглядової прокрутки.

3. УПРАВЛІННЯ АВТОМАТИЧНИМ РОЗМІЩЕННЯМ ЕЛЕМЕНТІВ

Мета. Вивчення засобів автоматичного розміщення елементів.

Вступ.

Класи компонування віджетів (Layouts) є однією з сильних сторін Qt (не треба плутати компоновку віджетів і компоновку додатка – це зовсім різні речі). По суті, це контейнери, які після зміни розмірів вікна автоматично приводять у відповідність розміри і координати віджетів, які перебувають в ньому. Хоча вони нічого не додають до функціональної частини самої програми, проте, вони дуже важливі для зовнішнього вигляду вікон програми. Компонування визначає розташування різних віджетів відносно один одного.

Звичайно, можна вручну розміщувати віджети в областях програми, але це суттєво ускладнює розробку. Адже тоді, щоб заново впорядкувати елементи, потрібно буде відловлювати і обробляти зміна розмірів вікна програми. Подібні ситуації добре відомі програмістам на мові Visual Basic, які змушені писати для цього складні методи.

Ще один з недоліків розміщення вручну полягає в тому, що якщо додаток підтримує кілька мов, то, оскільки слова в різних мовах мають різну довжину, необхідний механізм, який міг би в процесі роботи програми динамічно поправляти і змінювати розміри і координати віджетів, – інакше частини тексту іншою мовою можуть виявитися «відрізаними». Класи компонування бібліотеки Qt виконують цю непросту роботу. Більш того, класи компоновки можуть інвертувати напрямок розміщення елементів, що може бути корисно для пишучих справа наліво.

Qt надає так звані менеджери компонування, що дозволяють організувати розміщення віджетів на поверхні іншого віджета. Основу їх роботи визначає можливість кожного віджета повідомляти про те, скільки йому необхідно місця, чи може він бути розтягнутий по вертикалі і/або горизонталі і т. п.

План.

1. Менеджери компонування (layout managers)
2. Горизонтальне і вертикальне розміщення
2.1. Клас QBoxLayout
2.2. Горизонтальне розміщення QHBoxLayout
2.3. Вертикальне розміщення QVBoxLayout
2.4. Вкладені розміщення
2.5. Табличне розміщення QGridLayout
3. Порядок проходження табулятора
4. Розділювачі QSplitter

1. Менеджери компонування (layout managers)

Менеджери компонування надають можливості для горизонтального, вертикального і табличного розміщення не тільки віджетів, але і вбудованих компоновок. Це дозволяє конструювати досить складні розміщення.

Примітка. Створення свого власного класу компонування – явище дуже рідкісне, так як практично всі задачі розміщення можна вирішити стандартними класами розміщення, наданими Qt. Але якщо знадобиться створити власний менеджер компонування, то можна успадкувати клас `QLayout`, реалізувавши методи `additem()`, `count()`, `setGeometry()`, `takeAt()` і `itemAt()`.

Від класу `QLayout` успадковані класи `QGridLayout` і `QBoxLayout` (рис. 1). Клас `QGridLayout` керує табличним розміщенням, а від `QBoxLayout` успадковані два класи `QHBoxLayout` і `QVBoxLayout` для горизонтального і вертикального розміщення. За

замовчуванням між віджетами залишається невелика відстань. Ця відстань необхідна для їх візуального розділення. Задати його можна за допомогою методу `setSpacing()`, передавши в нього потрібне значення в пікселях. Методом `setMargin()` можна встановити відступ віджетів від межі компонування – звичайно типовими значеннями можуть бути 5 або 10 пікселів. Рис. 2 ілюструє суть цих методів на прикладі горизонтального розміщення.

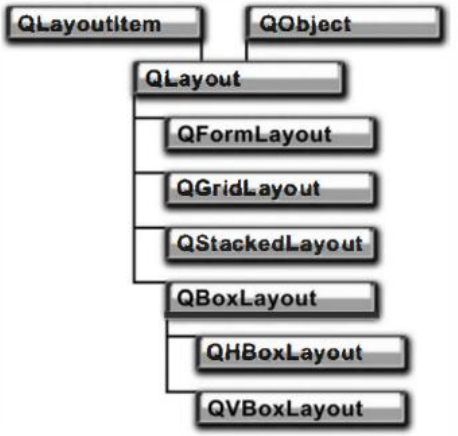


Рисунок 1 – Ієрархія класів менеджерів компонування

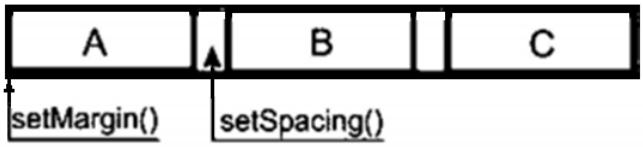


Рисунок 2 – Розміщення віджетів по горизонталі

За допомогою методу `addWidget()` виконується додавання віджетів в компоновку, а за допомогою методу `addLayout()` можна додавати вбудовані менеджери компоновки. Якщо знадобиться вилучити будь-який віджет з компонування, то слід скористатися методом `removeWidget()`, передавши йому вказівник на цей віджет.

Примітка. Об'єктна ієрархія віджетів і об'єктів розміщення відокремлені один від одного. Віджети це нащадки інших віджетів, а об'єкти розміщення – нащадки інших об'єктів розміщення.

Об'єкти розміщень відповідають за правильне розміщення віджетів і присвоєння їм потрібних віджетів-предків. Тобто, не потрібно турбуватися про те, щоб привласнювати об'єкти предків, так як це буде зроблено автоматично.

2. Горизонтальне і вертикальне розміщення

Для горизонтального або вертикального розміщення можна скористатися класом `QBoxLayout` або успадкованими від нього класами `QHBoxLayout` і `QVBoxLayout`. Класи `QHBoxLayout` і `QVBoxLayout`, успадковані від `QBoxLayout`, відрізняються від нього тим, що в їх конструктор не передається параметр, що говорить про спосіб розміщення – горизонтальний або вертикальний, так як порядок розміщення закладений уже в самому класі. Ці класи самі виконують горизонтальне або вертикальне розміщення: зліва направо або зверху вниз.

2.1. Клас QBoxLayout

Об'єкт класу `QBoxLayout` може управляти як горизонтальним, так і вертикальним розміщенням. Для того щоб задати спосіб розміщення, першим параметром конструктора повинно бути одне з наступних значень:

- `LeftToRight` – горизонтальне розміщення, заповнення здійснюється зліва направо;
- `RightToLeft` – горизонтальне розміщення, заповнення виконується справа наліво;
- `TopToBottom` – вертикальне розміщення, заповнення здійснюється зверху вниз;
- `BottomToTop` – вертикальне розміщення, заповнення виконується знизу вгору.

Цей клас розширює клас `QLayout` методами вставки на задану позицію: віджета `insertWidget()`, вбудованої компонування – `insertLayout()`, відстані між віджетами - `insertSpacing()` і фактора розтягування – `insertStretch()`.

До компонування за допомогою методу `addSpacing()` можна додати задану відстань між двома віджетами. Клас `QBoxLayout` визначає свій власний метод `addWidget()` для додавання віджетів в компоновку з можливістю вказівки, в додатковому параметрі, фактора розтягування (за замовчуванням цей параметр дорівнює нулю). Демонстрація цієї можливості показу на на рис. 3, а текст відповідної програми приведений в роздруку 1.

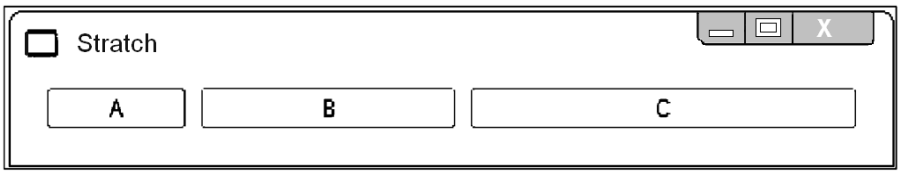


Рисунок 3 – Кнопки з факторами розтягнень

Роздрук 1. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
    //Layout setup
    QBoxLayout* pbxLayout = new QBoxLayout(QBoxLayout::LeftToRight);
    pbxLayout->addWidget(pcmdA, 1);
    pbxLayout->addWidget(pcmdB, 2);
    pbxLayout->addWidget(pcmdC, 3);
    wgt.setLayout(pbxLayout);
    wgt.resize(450, 40);
    wgt.show();
    return app.exec();
}
```

У роздруку 1 створюються три кнопки A, B і C, які поміщаються в компоновку за допомогою методу `QBoxLayout::addWidget()`, причому в другому параметрі цього методу вказується параметр розтягування. При створенні об'єкта компонування в конструктор передається параметр `QBoxLayout::LeftToRight`, який задає горизонтальне розміщення елементів зліва направо. Виклик методу `QWidget::setLayout()` встановлює компонування в віджеті `wgt`.

Можливо, що наведений у роздруку 1 приклад зробить шокуючий ефект, оскільки створювані кнопки (вказівники `pcmdA, pcmdB i pcmdC`) не мають об'єкта-предка, а це значить, що не буде кому подбати про звільнення пам'яті, виділеної для цих віджетів. Так що ж все-таки

відбувається? Невже програма і справді містить помилку, яка може привести до витоку пам'яті (memory leak)?

Насправді турбуватися нема про що, так як за присвоєння віджета-предка відповідає сама компоновка. При виклику методу `setLayout()` всім поміщеним в компоновку віджетів буде присвоєно віджет предка – в нашому випадку це `wgt`.

Фактори розтягування можна самостійно додавати в компонування, для чого існує метод `addStretch()`. У цьому випадку фактор розтягування образно можна порівняти з пружиною, яка знаходиться між віджетами і може мати різну пружність відповідно до заданого параметру. На рис. 4 показаний приклад додавання фактора розтягування між двома віджетами для розташування віджетів по краях вікна.

Рисунок 4 – Додавання фактора розтягування між віджетами A і B

Результат виконання програми (роз друк 2), показаний на рис. 5, демонструє додавання фактора розтягування і являє собою невелику модифікацію попередньої програми (див. роз друк 1), виконання якої показано на рис. 3, – тільки замість однієї з кнопок тут додається фактор розтягування.

Рисунок 5 – Додавання фактора розтягування між кнопками A і B

Роздрук 2. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    //Layout setup
    QBoxLayout* pbxLayout = new QBoxLayout(QBoxLayout::LeftToRight);
    pbxLayout->addWidget(pcmdA);
    pbxLayout->addStretch(1);
    pbxLayout->addWidget(pcmdB);
    wgt.setLayout(pbxLayout);
    wgt.resize(350, 40);
    wgt.show();
    return app.exec();
}
```

У роздруку 2 створюється віджет класу `QWidget` і дві кнопки A і B. Після створення об'єкта компонування для горизонтального розміщення в нього викликом методу `QLayout::addWidget()` додається перша кнопка (вказівник `pcmdA`). Потім виклик методу `QBoxLayout::addStretch()` додає фактор розтягування, після чого додається друга кнопка (вказівник `pcmdB`).

2.2. Горизонтальне розміщення QHBoxLayout

Об'єкти класу `QHBoxLayout` впорядковують всі віджети тільки в горизонтальному порядку - зліва направо. Його застосування аналогічно використанню класу `QBoxLayout`, але передавати в конструктор додатковий параметр, що задає горизонтальний порядок розміщення, не потрібно. Вікно програми, яка впорядковує віджети за допомогою об'єкта класу `QHBoxLayout` (роздрук 3), показано на рис. 6.

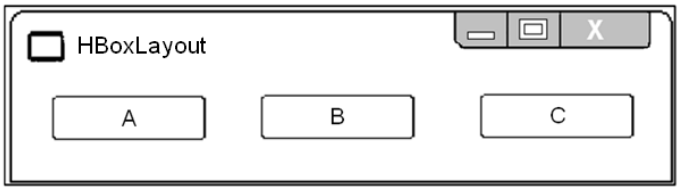


Рисунок 6 – Розміщення кнопок по горизонталі

Роздрук 3. Файл main.cpp

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->setMargin(10);
    phbxLayout->setSpacing(20);
    phbxLayout->addWidget(pcmdA);
    phbxLayout->addWidget(pcmdB);
    phbxLayout->addWidget(pcmdC);
    wgt.setLayout(phbxLayout);
    wgt.show();
    return app.exec();
}
```

У роздруку 3 створюються три кнопки A, B і C (вказівники `pcmdA`, `pcmdB` і `pcmdC`). Потім створюється об'єкт класу `QHBoxLayout` для горизонтального розміщення дочірніх віджетів. Метод `QLayout::setmargin()` встановлює товщину рамки в 10 пікселів. Метод QLayout::setSpacing() задає відстань між віджетами рівне 20 пікселям. Три виклику методу `QLayout::addWidget()` додають віджети кнопок в компоновку.

2.3. Вертикальне розміщення QVBoxLayout

Компонування `QVBoxLayout` успадкована від `QBoxLayout` і впорядковує всі віджети тільки по вертикалі – зверху вниз. В іншому вона нічим не відрізняється від класів `QBoxLayout` і `QHBoxLayout`. Якщо замінити в роздруку 3 ім'я класу `QHBoxLayout` на `QVBoxLayout`, то в результаті вийде програма, вікно якої показано на рис. 7.

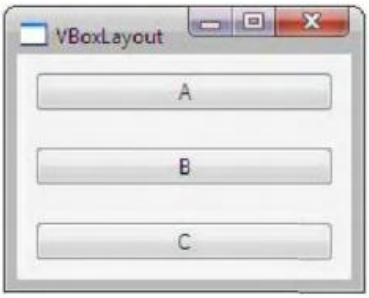


Рисунок 7 – Розміщення кнопок по вертикалі

2.4. Вкладені розміщення

Розміщуючи одну компоновку всередині іншої, можна створювати розміщення практично будь-якої складності. Для організації вкладених розміщень існує метод `addLayout()`, в який другим параметром передається фактор розтягування для компонування, яке додається. На рис. 8 показаний приклад вкладеного розміщення двох менеджерів компонування (роз друк 4). У компоновку `QVBoxLayout` поміщається компоновка `QHBoxLayout`.

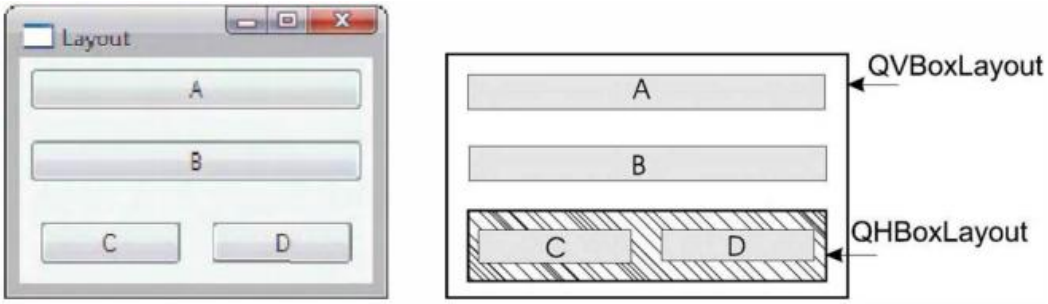


Рисунок 8 – Вкладені розміщення

Роздрук 4. Файл Layout.cpp

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
    QPushButton* pcmdD = new QPushButton("D");
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->setMargin(5);
    phbxLayout->setSpacing(15);
    phbxLayout->addWidget(pcmdC);
    phbxLayout->addWidget(pcmdD);
    pvbxLayout->setMargin(5);
    pvbxLayout->setSpacing(15);
    pvbxLayout->addWidget(pcmdA);
    pvbxLayout->addWidget(pcmdB);
    pvbxLayout->addLayout(phbxLayout);
    wgt.setLayout(pvbxLayout);
    wgt.show();
    return app.exec();
}
```

У роздруку 4 наводиться фрагмент програми, вікно якої показано на рис. 8. Програма буде читатися набагато краще, якщо спочатку створити всі віджети, а потім об'єкти менеджерів компонування. Використовуючи метод для установки товщини рамки `setmargin()`, ми встановлюємо її рівною п'яти для обох компоновок, а за допомогою методу `setSpacing()` встановлюємо відстань між віджетами в 15 пікселів, також для обох компоновок. У горизонтальну компоновку ми додаємо віджети кнопок `pcmdC` і `pcmdD`. Потім віджети кнопок `pcmdA` і `pcmdB` по черзі передаються в метод `QLayout::addWidget()` вертикальної компоновки `pvbxLayout`, після чого за допомогою методу `QBoxLayout::addLayout()` в неї передається об'єкт горизонтальної компоновки `pvbxLayout`. Виклик методу `QWidget::setLayout()` встановлює вертикальну компоновку `pvbxLayout` у віджеті `wgt`.

2.5. Табличне розміщення QGridLayout

Для табличного розміщення використовується клас `QGridLayout`, за допомогою якого можна швидко створювати складні за структурою розміщення. Таблиця складається з комірок, позиції яких задаються рядками і стовпцями.

Примітка. Якщо потрібна таблиця, яка складається з двох стовпців, то можна скористатися класом `QFormLayout`, – це допоможе реалізувати більш компактний код, ніж з використанням класу `QGridLayout`. Подібна ситуація зустрічається часто, наприклад, при створенні діалогів, в яких в першому стовпці стоять пояснюють написи, а в другому – віджети для введення інформації. Додавання віджетів здійснюється викликом методу `addRow()`, в який передаються відразу два віджету: віджет надпису і функціональний віджет.

Додати віджет в таблицю можна за допомогою методу `addWidget()`, передавши йому позицію комірки, в яку поміщається віджет. Іноді необхідно, щоб віджет займав відразу кілька позицій, чого можна досягти тим же методом `addWidget()`, вказавши в додаткових параметрах кількість рядків і стовпців, які буде займати віджет. В останньому параметрі можна задати вирівнювання, наприклад, по центру:

```
playout->addWidget(widget, 17, 1, Qt::AlignCenter);
```

Фактор розтягування встановлюється методами `setRowStretch()` і `setColumnStretch()`, але не для кожного віджета окремо, а для рядків або стовпчиків. Відстань між віджета ми також встановлюється для стовпців або рядків методом `setSpacing()`. Приклад, показаний на рис. 9 (роздрук 5), розміщує чотири кнопки: A, B, C і D в таблиці розміром 2 на 2 комірки.

Рисунок 9 – Розміщення кнопок в табличному порядку

Роздрук 5. Файл main.cpp

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
```

```
    QPushButton* pcmdD = new QPushButton("D");
    //Layout setup
    QGridLayout* pgrdLayout = new QGridLayout;
    pgrdLayout->setMargin(5);
    pgrdLayout->setSpacing(15);
    pgrdLayout->addWidget(pcmdA, 0, 0);
    pgrdLayout->addWidget(pcmdB, 0, 1);
    pgrdLayout->addWidget(pcmdC, 1, 0);
    pgrdLayout->addWidget(pcmdD, 1, 1);
    wgt.setLayout(pgrdLayout);
    wgt.show();
    return app.exec();
}
```

У лістингу 5 створюється компонування табличного розміщення `pgrdLayout`. Метод `setmargin()` встановлює відступ від країв в 5 пікселів. Виклик методу `setSpacing()` встановить відстань в 15 пікселів між віджетами. Віджети кнопок додаються в компоновку викликом методу `addWidget()`, останні два параметри якого вказують рядок і стовпець, в яких повинен бути розташований віджет. Застосунук (роздрук 6), виконання якого показано на рис. 10, демонструє застосування табличного розміщення на прикладі калькулятора. У прикладі задіяні класи стеку `QValueStack` і регулярного виразу `QRegExp`.

Рисунок 10 – Результат виконання програми, використовує табличний розміщення

Роздрук 6. Файл main.cpp

```
#include <QApplication>
#include "Calculator.h"
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    Calculator   calculator;
    calculator.setWindowTitle("Calculator");
    calculator.resize(230, 200);
    calculator.show();
    return app.exec();
}
```

У програмі, показаній в роздруку 6, створюється віджет калькулятора `calculator` (див. роздруку 7-11). Після зміни розміру методом `resize()` виклик методу `show()` відображає калькулятор на екрані.

У визначенні класу `Calculator`, показаного у роздруку 7, описуються атрибути: `m_plcd` – вказівник на віджет електронного індикатора, `m_stk` – стек для проведення операцій обчислення і `m_strDisplay` – рядок, в яку будуть записуватися символи натиснутих користувачем кнопок. Метод `createButton()` призначений для створення кнопок калькулятора, а метод `calculate()` – для обчислення виразів, які перебувають у стеку `m_stk`. Слот `slotButtonClicked()` викликається при натисканні на будь-яку з кнопок калькулятора.

Роздрук 7. Файл `Calculator.h`

```
#pragma once
#include <QWidget>
#include <QStack>
class QLCDNumber;
class QPushButton;
class Calculator : public QWidget {
    Q_OBJECT
private:
    QLCDNumber*     m_plcd;
    QStack<QString> m_stk;
    QString         m_strDisplay;
public:
    Calculator(QWidget* pwgt = 0);
    QPushButton* createButton(const QString& str);
    void         calculate   (                  );
public slots:
    void slotButtonClicked();
};
```

При створенні електронного індикатора (роздрук 8) в його конструктор передається кількість сегментів, яка дорівнює 12. Прапор `QLCDNumber::Flat`, переданий в метод `setSegmentStyle()`, задає сегментам індикатора плоский стиль. Метод `setMinimumSize()` перезадає мінімально можливі розміри віджета індикатора. У масиві `aButtons` визначаються написи для кнопок калькулятора. Віджет електронного індикатора поміщається в компоновку викликом методу `addWidget()`, перші два параметра якого задають його розташування, а останні два – кількість займаних ним рядків і стовпців у табличному компонуванні.

Кнопка CE, після свого створення методом `createButton()`, поміщається в компоновку методом `addWidget()` на позиції (1,3) (тобто на перетині другого рядка і четвертого стовпця – які нумеруються з нуля). Всі інші віджети кнопок створюються і поміщаються в компоновку в циклі за допомогою методів `createButton()` і addWidget().

Роздрук 8. Файл `Calculator.cpp`. Конструктор `calculator`

```
#include <QtWidgets>
#include "Calculator.h"
Calculator::Calculator(QWidget* pwgt/*= 0*/) : QWidget(pwgt)
{
    m_plcd = new QLCDNumber(12);
    m_plcd->setSegmentStyle(QLCDNumber::Flat);
    m_plcd->setMinimumSize(150, 50);
    QChar aButtons[4][4] = {{'7', '8', '9', '/'},
                            {'4', '5', '6', '*'},
                            {'1', '2', '3', '-'},
                            {'0', '.', '=', '+'}
                           };
    //Layout setup
    QGridLayout* ptopLayout = new QGridLayout;
    ptopLayout->addWidget(m_plcd, 0, 0, 1, 4);
    ptopLayout->addWidget(createButton("CE"), 1, 3);
    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
           ptopLayout->addWidget(createButton(aButtons[i][j]), i + 2, j);
        }
```

```
    }
    setLayout(ptopLayout);
}
```

Метод `createButton()`, показаний у роздруку 9, отримує рядок з написом і створює натискну кнопку. Після цього викликом методу `setMinimumSize()` для кнопки встановлюються мінімально можливі розміри, а сигнал `clicked()` з'єднується зі слотом `slotButtonClicked()` викликом `connect()`.

Роздрук 9. Файл `Calculator.cpp`. Метод `createButton()`

```
QPushButton* Calculator::createButton(const QString& str)
{
    QPushButton* pcmd = new QPushButton(str);
    pcmd->setMinimumSize(40, 40);
    connect(pcmd, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    return pcmd;
}
```

Метод `calculate()` (роз друк 10) призначений для обчислення виразу, який міститься у стеку `m_stk`. Змінна `dOperand2` отримує зняте з верхівки стеку значення, перетворене до типу `double`. Стрічкова змінна `strOperation` отримує символ операції. Змінна `dOperand1` із стеку отримує останнє значення, яке також перетвориться до типу `double`. В операторах `if` символ операції порівнюється з чотирма допустимими і, в випадку збігу, виконується необхідна операція, результат якої зберігається у змінній `dResult`. Після цього викликом методу `dislay()` значення змінної `dResult` відображається на електронному індикаторі (вказівник `m_plcd`).

Роздрук 10. Файл `Calculator.cpp`. Метод `calculate()`

```
void Calculator::calculate()
{
    double  dOperand2    = m_stk.pop().toDouble();
    QString strOperation = m_stk.pop();
    double  dOperand1    = m_stk.pop().toDouble();
    double  dResult      = 0;

    if (strOperation == "+") {
        dResult = dOperand1 + dOperand2;
    }
    if (strOperation == "-") {
        dResult = dOperand1 - dOperand2;
    }
    if (strOperation == "/") {
        dResult = dOperand1 / dOperand2;
    }
    if (strOperation == "*") {
        dResult = dOperand1 * dOperand2;
    }
    m_plcd->display(dResult);
}
```

У слоті `slotButtonClicked()` (роздрук 11) здійснюється перетворення віджета, який вислав сигнал, до типу `QPushButton`, після чого змінній `str` присвоюється текст надпису на кнопці. Якщо напис дорівнює CE, то виконується операція скидання – очищення стеку і увстановлення значення індикатора в 0. Якщо була натиснута цифра або крапка, то виконується її додавання в кінець стрічки `m_strDisplay`, вона відображається індикатором з подальшою актуалізацією. При натисканні будь-якої іншої кнопки вважається, що була натиснута кнопка операції. Якщо в стеку знаходиться менше двох елементів, то відображене число і операція заносяться у стек. Інакше у стек заноситься відображуване значення і викликається метод `calculate()` для обчислення виразу, якй знаходиться у стеку. Після цього стек очищається за допомогою методу `clear()` і в нього записується значення результату, яке відображається

індикатором, і наступна операція. Якщо виконується операція =, то вона не буде додаватися у стек.

Роздрук 11. Файл `Calculator.cpp`. Метод `slotButtonClicked()`

```
void Calculator::slotButtonClicked()
{
    QString str = ((QPushButton*)sender())->text();

    if (str == "CE") {
        m_stk.clear();
        m_strDisplay = "";
        m_plcd->display("0");
        return;
    }
    if (str.contains(QRegExp("[0-9]"))) {
        m_strDisplay += str;
        m_plcd->display(m_strDisplay.toDouble());
    }
    else if (str == ".") {
        m_strDisplay += str;
        m_plcd->display(m_strDisplay);
    }
    else {
        if (m_stk.count() >= 2) {
            m_stk.push(QString().setNum(m_plcd->value()));
            calculate();
            m_stk.clear();
            m_stk.push(QString().setNum(m_plcd->value()));
            if (str != "=") {
                m_stk.push(str);
            }
        }
        else {
            m_stk.push(QString().setNum(m_plcd->value()));
            m_stk.push(str);
            m_strDisplay = "";
            m_plcd->display("0");
        }
    }
}
```

3. Порядок проходження табулятора

Користувач може взаємодіяти з віджетами за допомогою миші і клавіатури. В останньому випадку для вибору потрібного віджета використовується клавіша табуляції – <Tab>, при натисканні якої відбувається перехід фокуса згідно з встановленим порядком від одного віджета до іншого. Іноді виникає необхідність у зміні цього порядку, який за замовчуванням відповідає черговості встановлення дочірніх віджетів у віджеті предка. На рис. 11 цифрами зображений порядок зміни фокуса за допомогою табулятора. Якщо відобразиться діалогове вікно з трьома кнопками фокус буде встановлено на кнопці С і після натискання на клавішу табуляції він перейде на кнопку В, а потім – на кнопку А.

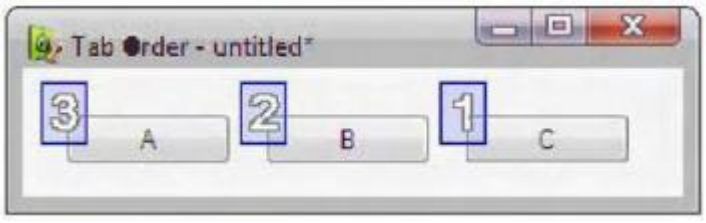


Рисунок 11 – Порядок зміни фокуса

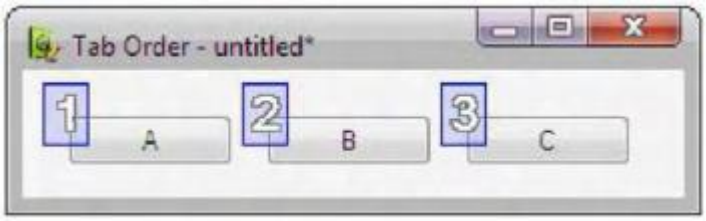


Рисунок 12 – Змінений порядок зміни фокуса

Змінити порядок зміни фокуса можна за допомогою статичного методу `QWidget::setTabOrder()`, який отримує як параметри два вказівники на віджети. Наступні виклики змінять порядок проходження табулятора, показаний на рис. 11, на більш логічний порядок, показаний на рис. 12:

```
QWidget:: setTabOrder (A, B);
QWidget:: setTabOrder (B, C);
```

4. Розділювачі QSplitter

Розділювачі призначені для одночасного перегляду різних частин текстових або графічних об'єктів. У деяких випадках застосування розділювачів більш бажане, ніж розміщення за допомогою класів компонування, так як з'являється можливість зміни розмірів віджетів. Конкретний приклад – це всім відома програма Провідник (Windows Explorer) з ОС Windows. Розділювачі реалізовані в класі `QSplitter`, визначення якого знаходиться в заголовковому файлі `QSplitter`. За допомогою віджета розділювача можна розташовувати віджети як горизонтально, так і вертикально. Між віджетами відображається риса розділювача, яку можна переміщати за допомогою миші, тим самим змінюючи розміри віджетів.

Якщо необхідно, щоб віджети розділювача були проінформовані про зміну розмірів, то тоді потрібно викликати метод `setOpaqueResize()`, передавши йому значення `true`. Приклад, показаний на рис. 13, розділяє два віджету класу QTextEdit (роз друк 12).

Роздрук 12. Файл main.cpp

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QSplitter  spl(Qt::Vertical);
    QTextEdit* ptxt1 = new QTextEdit;
    QTextEdit* ptxt2 = new QTextEdit;
    spl.addWidget(ptxt1);
    spl.addWidget(ptxt2);
    ptxt1->setPlainText("Line1\n"
                        "Line2\n"
                        "Line3\n"
                        "Line4\n"
                        "Line5\n"
                        "Line6\n"
                        "Line7\n"
                       );
    ptxt2->setPlainText(ptxt1->toPlainText());
    spl.resize(200, 220);
    spl.show();
    return app.exec();
}
```

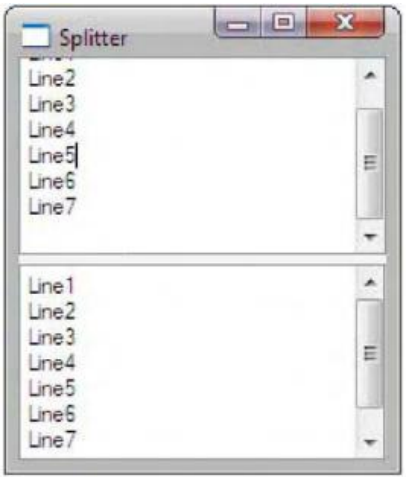


Рисунок 13 – Розділювач

У Роздруку 12 у конструктор класу `QSlitter` передається прапор `Qt::Vertical` і, тим самим, створюється віджет вертикального роздільник.

Примітка. Для створення горизонтального розділювача в конструктор потрібно передати значення `QSplitter::Horizontal`. Того ж ефекту можна досягти, якщо передавати прапори `QSplitter::Horizontal` і `QSplitter::Vertical` в метод `setOrientation()`.

Після створення віджетів класу `QTextEdit` вони додаються у віджет розділювач, для чого їх вказівники передаються в метод `QSplitter::addWidget()`. Текст у віджетах `QTextEdit` встановлюється методом `setPlainText()`.

Висновки.

• Об'єкти компонування віджетів при зміні розмірів вікна автоматично виконують в ньому розміщення віджетів. Qt надає менеджери компонування, які мають можливості горизонтального, вертикального і табличного розміщення. Ці способи можна комбінувати для створення складних розміщень. Класи менеджерів компонування базуються відразу на двох класах: `QObject` і `QLayoutItem`. Об'єкти компонування дозволяють встановити фактор розтягування для керування співвідношенням розмірів віджетів. Крім того, реалізовані методи для встановлення відстаней між віджетами і налаштування розміру відступу віджетів від межі компонування.

• Віджет розділювача надає можливість одночасного перегляду вмісту віджетів. При перетягуванні розділювача змінюються розміри віджетів, розміщених в розділювачі.

Запитання.

1. Менеджери компонування і їх класи.
2. Клас QboxLayout і його методи.
3. Горизонтальне розміщення і клас QHBoxLayout.
4. Вертикальне розміщення і клас QVBoxLayout
5. Вкладені розміщення.
6. Табличне розміщення і клас QGridLayout.
7. Порядок проходження табулятора і ого зміна.
8. Розділювачі і клас QSplitter.

4. ЕЛЕМЕНТИ ВІДОБРАЖЕННЯ

Мета. Вивчення елементів відображення.

Вступ.

Елементи відображення не приймають активної участі в діях користувача і використовуються для інформування його про те, що відбувається. Ця інформація може носити як текстовий, так і графічний характер (картинки, графіка).

План.

1. Написи
2. Індикатор процессу
3. Електронний індикатор

1. Написи

Віджет *Написи* служить для показу стану застосунку або пояснюючого тексту і є текстовим полем, текст якого не підлягає зміні з боку користувача. Інформація, яка відображається цим віджетом, може змінюватися тільки самим застосунком. Таким чином, програма може повідомити користувачу про свой змінений стані, але користувач не може змінити цю інформацію в самому віджеті. Клас віджета написи `QLabel` визначено в заголовковому файлі `QLabel`.

Віджет написи успадкований від класу `QFrame` і може мати рамку. Видима їм інформація може бути текстового, графічного або анімаційного характеру, для її передачі використовуються слоти `setText()`, `setPixmap()` і `setMovie()`.

Розташуванням тексту можна керувати за допомогою методу `setAligment()`. Метод використовує велику кількість прапорів, деякі з них наведені в табл. 1. Зверніть увагу, що значення не перетинаються, і це дозволяє комбінувати їх один з одним за допомогою логічної операції | (АБО). Наочним прикладом служить значення `AlignCenter`, складене з значень `AlignVCenter` і `AlignHCenter`.

Таблиця 1 – Значення прапорів `AlignmettFlag` простору імен Qt

Константа	Значення	Опис
`AlignLeft`	`0x0001`	Розташування тексту зліва
`AlignRight`	`0x0002`	Розташування тексту справа
`AlignCenter`	`0x0004`	Центрування тексту по горизонталі
`AlignJustify`	`0x0008`	Розтягування тексту по всій ширині
`AlignTop`	`0x0010`	Розташування тексту вгорі
`AlignBottom`	`0x0020`	Розташування тексту внизу
`AlignVCenter`	`0x0040`	Центрування тексту по вертикалі
`AlignCenter`	`AlignVCenter | AlignHCenter`	Центрування тексту по вертикалі і горизонталі

Як видно з рис. 1, віджет написи може відображати не тільки звичайний текст, а й текстову інформаці в форматі HTML (HyperText Markup Language, мова гіпертекстової розмітки). У цьому прикладі (роздрук 1) використовувався HTML для виведення тексту, таблиць і растрового зображення.

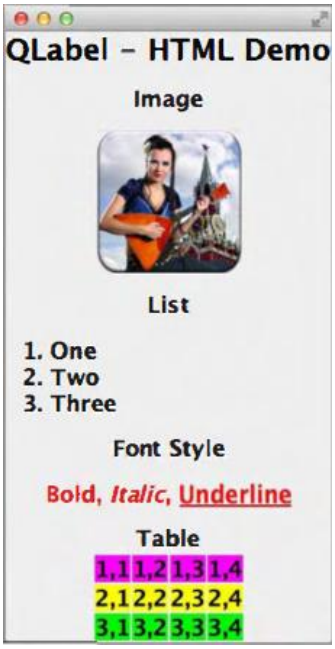


Рисунок 1 – Відображення інформації виджетом написи в форматі HTML

Роздрук 1. Файл `main.cpp`

```
#include <QtWidgets>

int  main (int  argc ,  char**  argv)
{
   QApplication app (argc ,  argv);

   QLabel  lbl("<Hl><CENTER>QLabel - HTML  Demo</CENTER></Hl>"
        "<H2><CENTER>Image</CENTER><H2 >"
        "<CENTER><IMG BORDER=\"0\" SRC=\":/Balalaika.png\"></CENTER>"
        "<H2><CENTER>List</CENTER><H2>"
        "<OL><LI>One</LI>"
             <LI>Two</LI>"
             <LI>Three</LI>"
        "</OL>"
        "<H2><CENTER>Font Style</CENTER><H2>"
        "<CENTER><FONT COLOR=RED>"
        <B>Bold</B>, <I>Italic</I>, <U>Underline</U>"
        "</FONT></CENTER>"
        "<H2 ><CENTER>Table</CENTER></H2>"
        "<CENTER> <TABLE>"
           <TR BGCOLOR=#ffO Off>"
               <TD>l,l</TD><TD>1,2</TD><TD>1,3</TD><TD>1,4</TD>"
           </TR>"
           <TR BGCOLOR=YELLOW>"
               <TD>2,1</TD><TD>2,2</TD><TD>2,3</TD><TD>2,4</TD>"
           </TR>"
           <TR BGCOLOR=#OOfOOO>"
               <TD>3,1</TD><TD>3,2</TD><TD>3,3</TD><TD>3,4</TD>"
           </TR>"
        "</TABLE> </CENTER>"
```

```
        );
lbl.show();
return app.exec();
}
```

У роздруку 1 при створенні віджету написи `lbl` першим параметром в конструктор передається текст у форматі HTML. Його можна передати і після створення цього віджета за допомогою методу-слота `setтext()`. Другий параметр конструктора опущений, а так як за замовчуванням він дорівнює 0, то це робить його віджетом верхнього рівня.

Наступний приклад (роздрук 2), показаний на рис. 2, демонструє можливість відображення інформації графічного характеру у віджеті напису без використання формату HTML.

Рисунок 2 – Відображення графічної інформації віджетом напису

Роздрук 2. Файл `main.cpp`

```
#include  <QtWidgets>

int main (int argc, char** argv)
{
   QApplicat ion app (argc, argv);
   QPixmap pix;
   pix.load(":/mira.jpg");

   QLabel lbl;
   lbl.resize(pix.size());
   lbl.setPixmap(pix);

   lbl.show();
   return app.exec();
}
```

Як видно з роздруку 2, спочатку створюється об'єкт растрового зображення `QPixmap`. Після цього викликом методу `load()` в нього завантажується з ресурсу файл `mira.jpg`.

Наступним кроком є створення самого віджету написи – об'єкта `lbl` класу `QLabel`. Потім викликом методу `resize()` його розміри приводяться у відповідність з розмірами растрового зображення. І, нарешті, виклик методу `setpixmap()` встановлює в віджеті саме растрове зображення. За допомогою методу `setвuddy()` віджет напису може асоціюватися з будь-яким іншим віджетом. Якщо текст напису містить знак & (амперсанд), то символ, перед яким він стоїть, буде підкресленим. При натисканні цього символу спільно з клавішею <Alt> фокус

перейде до віджету, встановленому методом `setBuddy()`. На рис. 3 показані такі віджети, а в роздруку 3 показаний текст відповідної програми.

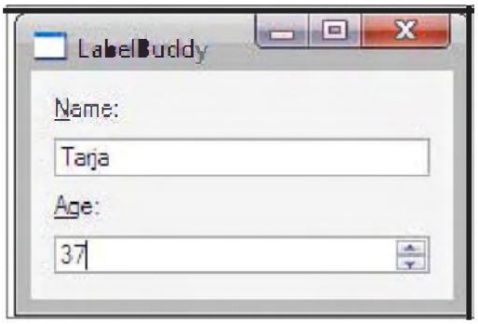


Рисунок 3 – Використання знаку &

Роздрук 3. Файл main.cpp

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget wgt;
    QLabel* plblName = new QLabel("&Name:");
    QLineEdit* ptxtName = new QLineEdit;
    plblName->setBuddy(ptxtName);

    QLabel* plblAge = new QLabel("&Age:");
    QSpinBox* pspbAge = new QSpinBox;
    plblAge->setBuddy(pspbAge);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plblName);
    pvbxLayout->addWidget(ptxtName);
    pvbxLayout->addWidget(plblAge);
    pvbxLayout->addWidget(pspbAgeb);
    wgt.setLayout(pvbxLayout);

    wgt.show();
    return app.exec();
}
```

У роздруку 3 віджет `wgt` класу `QWidget` є віджетом верхнього рівня, так як за замовчуванням його конструктор присвоює покажчику на віджет-предок значення 0. Віджети не володіють здатністю самостійного розміщення віджетів-нащадків, тому пізніше в ньому здійснюється установка компонування для вертикального розміщення `QVBoxLayout`. У віджеті напису Name (Ім'я) в тексті символ `N` визначений як символ для швидкого доступу. Потім створюється віджет однострічкового текстового поля. Далі, виклик методу `setbuddy()` пов'язує віджет напису із створеним текстовим полем, використовуючи покажчик на поле як аргумент. Аналогічно відбувається створення віджету напису Age (Вік), поля для введення віку (класу `QSpinBox`) і їх зв'язування.

У всіх віджетах є можливість оброблення подій клавіатури і миші. Цим можна скористатися, наприклад, для створення гіпертекстового посилання, яке при натисканні викличе певну HTM-сторінку.

Однак можна вчинити і простіше. Справа в тому, що клас `QLabel` надає підтримку для гіпертекстових посилань і при натисканні на посилання відправляє сигнал `linkActivated()`, який можна з'єднати зі слотом, з якого відбудеться виклик сторінки.

Але є й інший, ще більш простий спосіб, він полягає в тому, щоб перевести віджет `QLabel` в стан, коли він сам зможе відкривати посилання в Web-переглядачі (що буде досягатися не явним викликом статичного методу `QDesktopServices::openurl()`).

Для цього потрібно просто викликати метод `setOpenExternalLinks()` з параметром `true`. Наприклад:

```
QLabel *plbl =
new QLabel("<A HREF=\"http://www.bhv.ru\">www.bhv.ru</A>");
lbl.setOpenExternalLinks(true);
```

2. Індикатор процесу

Індикатор процесу (progress bar) – це віджет, який демонструє процес виконання операції і заповнюється зліва направо. Повне заповнення індикатора інформує про завершення операції. Цей віджет необхідний в тому випадку, коли програма виконує тривалі дії, – віджет дає користувачеві зрозуміти, що програма не зависла, а знаходиться в роботі. Він також показує, скільки вже зроблено і скільки ще належить зробити. Клас `QProgressBar` віджета індикатора процесу визначено в заголовковому файлі `QProgressBar`. Звичайно індикатори процесу розташовуються в горизонтальному положені, але це можна змінити, передавши в слот `setOrientation()` значення `Qt::Vertical`, після цього він буде розташований вертикально.

Наступний приклад демонструє використання індикатора процесу. При натисканні кнопки Step (Крок) збільшується значення індикатора на один крок. Натискання кнопки Reset (Скинути) скидає значення індикатора. В основній програмі, показаній у роздруку 4, створюється віджет, показаний на рис. 4.

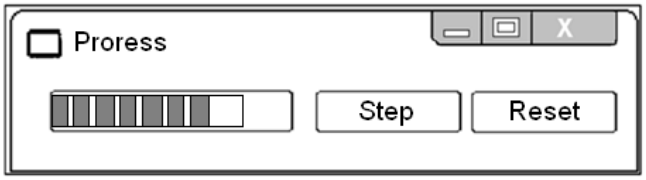


Рисунок 4 – Приклад індикатора процесу

Роздрук 4. Файл `main.cpp`

```
#include <QApplication>
#include "Progress.h"

int main(int argc, char** argv)
{
   QApplication app(argc, argv);
   Progress progress;

   progress.show();
   return  app.exec();
}
```

У роздруку 5 показаний файл `Progress.h`, який містить визначення класу `Progress`, успадкованого від `QWidget`. Клас містить два атрибути: покажчик на віджет індикатора процесу і ціле значення, що задає номер кроку. У класі визначено два слоти: `slotStep()` і `slotReset()`. Перший призначений для нарощування кроку на одиницю, а другий – для скидування індикатора процесу в нульове положення.

Роздрук 5. Файл `Progress.h`

```
#pragma once
```

```
#include <QWidget>
class QProgressBar;
//  ==========================
class Progress: puЬlic QWidget{
   Q_OBJECT
private:
   QProgressBar* m_pprb;
   int m_nStep;
public:
   Progress(QWidget* pobj = 0);
public slots:
   void  slotStep();
   void  slotReset();
};
```

У конструкторі класу (роздрук 6) атрибуту `m_nStep` присвоюється значення 0. Після створення об'єкта індикатора `m_pprb` викликом методу `setRange()` задається кількість кроків, рівна 5, а метод `setMinimumWidth()` встановлює мінімальну довжину віджета індикації процесу, – в нашому випадку забороняється йому мати довжину менше двохсот пікселів. Виклик методу `setAlignment()` з параметром `Qt::AlignCenter` переводить індикатор в режим відображення відсотків в центрі (див. табл. 1). Потім створюються дві кнопки: Step (Крок) і Reset (Скинути), які з'єднуються із слотами `slotStep()` і `slotReset()`. У слоті `slotstep()` значення атрибута `m_nstep` збільшується на 1 і передається в слот `QProgressвat::setValue()` об'єкта індикатора процесу. Слот `slotReset()` встановлює значення атрибута `m_nStep` рівним 0 і, викликавши слот `QProgressвar::reset()`, повертає індикатор в початковий стан. Для розміщення віджетів-нащадків горизонтально і зліва направо необхідно встановити в віджеті `Progress` об'єкт класу компонування `QVBoxLayout`, попередньо додавши в нього, в потрібній черговості, віджети-нащадки.

Роздрук 6. Файл `Progress.cpp`

```
#include  <QtWidgets>
#include  "Progress .h"
// ----------------------------------------
Progress::Progress(QWidget* pwgt/*=0*/)
   : QWidget(pwgt)
   , m_nStep(0)
{
   m_pprb = new QProgressBar;
   m_pprb->setRange(0, 5);
   m_pprb->setMiniтumWidth(200);
   m_pprb->setAlignment(Qt::AlignCenter);
   QPushButton* pcmdStep   = new QPushButton("&Step");
   QPushButton*  pcmdReset = new QPushButton("&Reset");
   QObject::connect(pcmdStep, SIGNAL(clicked()), SLOT(slotStep()));
   QObject::connect(pcmdReset, SIGNAL(clicked()), SLOT(slotReset()));
   //Layout  setup
   QHBoxLayout* phbxLayout = new QHBoxLayout;
   phbxLayout->addWidget(m_pprb);
   phЬxLayout->addWidget(pcmdStep);
   phЬxLayout->addWidget(pcmdReset);
   setLayout(phbxLayout);
//----------------------------
void  Progress :: slotStep ()
{
   m_pprb->setValue(++m_nStep);
}
//----------------------------
void  Progress :: slotReset ()
{
   m_nStep=0;
   m_pprb->reset();
```

```
}
```

3. Електронний індикатор

Клас `QLCDNumber` віджета електронного індикатора визначено в заголовковому файлі `QLCDNumber`. За зовнішнім виглядом цей віджет являє собою набір сегментних покажчиків – як, наприклад, на електронному годиннику. За допомогою міні-електронного індикатора відображаються цілі числа. Допускається використання крапки, яку можна відображати між позиціями сегментів як окремий символ, викликаючи метод `setSmallDecimalPoint()` і передаючи в нього `true` або `false` відповідно. Кількість сегментів, які відображаються, можна задати в конструкторі або за допомогою методу `setNumDigits()`. У тому випадку, коли для відображення числа не вистачає сегментів індикатора, відсилається сигнал `overflow()`.

За замовчуванням стиль електронного індикатора відповідає стилю `Outline`, але його можна змінити, передавши методу `setSegmentStyle()` одне з наступних значень: `QLCDNumer::Outline`, `QLCDNuer::Filled` або `QLCDNuer::Flat`. У табл. 2 показаний зовнішній вигляд віджету для кожного з перерахованих стилів.

Таблиця 2 – Стилі електронного індикатора

Константа	Зовнішній вигляд
`Outline`	
`Flat`	
`Filled`	

Електронний індикатор можна включати в режимі відображення двійкової, вісімкової, десяткової або шістнадцяткової системи числення. Режим відображення змінюється за допомогою методу `setMode()`, в який передається одне з наступних значень: `QLCDNumber::Bin` (двійкова), `QLCDNumber::Oct` (вісімкова), `QLCDNumber::Dec` (десяткова) або `QLCDNumber::Hex` (шістнадцяткова). Також для зміни режиму відображення можна скористатися слотами `setBinMode()`, `setOctMode()`, `setDecMode()` і `setHexMode()` відповідно. Наступний приклад (роздрук 7) демонструє електронний індикатор, який відображає шістнадцяткові значення (рис. 5).

У роздруку 7 створюються віджети електронного індикатора (покажчик `plcd`) і лічильника (вказівник `pspb`). Потім викликом методу `setSegmentStyle()` змінюється стиль сегментних покажчиків. Електронний індикатор викликом методу `setMode()` і з параметром `QLCDNwer::Hex` перемикається в режим шістнадцяткового відображення. У елемента лічильника методом `setFixedHeight()` встановлюється незмінна висота, рівна 30. Після цього сигнал `valueChanged()` віджета лічильника з'єднується із слотом `display()` електронного індикатора.

Рисунок 5 – Приклад електронного індикатора

Роздрук 7. Файл main.cpp

```
#include  <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget wgt;
    QLCDNunber* plcd = new QLCDNunber;
    QSpinBox* pspb = new QSpinBox;

    plcd->setSegmentStyle(QLCDNunber::Filled);
    plcd->setMode(QLCDNunber::Hex);
    pspb->setFixedHeight(30);

    QObject::connect(pspb, SIGNAL(valueChanged (int)),
        plcd, SLOT(display (int))
        ) ;

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plcd);
    pvbxLayout->addWidget(pspb);
    wgt.setLayout(pvbxLayout);

    wgt.resize(250,150);
    wgt.show();
    return app.exec();
}
```

Висновки.

• Віджет написи містить інформацію, призначену тільки для перегляду і не підлягає зміні зі сторони користувача. Цей віджет може відображати не тільки простий текст, а й текст у форматі HTML. Крім текстової може відображатися і графічна інформація.

• Віджет індикатора процесу – елемент для оповіщення користувача про режим роботи програми. Він незамінний для ілюстрації процесу виконання тривалих операцій.

• Віджет електронного індикатора за зовнішнім виглядом є набором сегментних покажчиків, як на електронному годиннику. Стиль віджета можна змінювати і використовувати різні режими відображення чисел: в двійковій, десятковій, шістнадцятковій і вісімковій системах числення.

Запитання.

1. Клас `QLabel` і можливості відображення тестової і графічної інформації.
2. Індикатор процесу і клас `QProcess`.
3. Клас `QLCDNumber` і можливості відображення чисел.

5. КНОПКИ, ПРАПОРЦІ І ПЕРЕМИКАЧІ

Мета. Вивчення можливостей використання кнопок, прапорців і перемикачів.

Вступ.

Кнопки – це один з найважливіших і найбільш часто вживаних елементів інтерфейсу користувача. Навіть якщо програмістом в коді програми не створюється жодної кнопки, все одно в правому верхньому куті вікна застосунка присутні керуючі кнопки. Кнопка може бути натиснута (on) або відтиснута (off).

План.

1. Клас QAbstractButton

1. Клас QAbstractButton

Клас `QAbstractButton` – базовий для всіх кнопок. У застосунках використовуються три основні види кнопок: кнопки, які натискаються (`QPushButton`), які звичайно називаються просто кнопками, прапорці (`QCheckBox`) і перемикачі (`QRadioButton`). В класі `QAstractButton` реалізовані методи і можливості, властиві всім кнопокам.

Встановлення тексту і зображення.

Всі кнопки можуть містити текст, який можна передати як в конструкторі першим параметром, так і встановити за допомогою методу `setтext()`. Для отримання тексту в класі `QAstractButton` визначено метод `text()`.

Растрове зображення встановлюється на кнопці за допомогою методу `seticon()`. Після встановлення зображення викликом методу `seticonSize()` можна змінити його максимальний розмір, який займає зображення на кнопці (зображення меншого розміру не розтягуються). Для отримання поточного максимального розміру зображення визначено метод `iconSize()`. І, нарешті, для того щоб кнопка повернула встановлене в ній зображення, потрібно викликати метод `icon()`.

Взаємодія з користувачем.

Для взаємодії з користувачем клас `QAstractButton` надає наступні сигнали:

- `clicked()` – відправляється при натисканні кнопкою миші;
- `pressed()` – відправляється при натисканні на кнопку миші;
- `released()` – відправляється при відпуску кнопки миші;
- `toggled()` – відправляється при зміні стану кнопки, що має статус вимикача

Опитування стану.

Для опитування поточного стану кнопок в класі `QAbstractButton` визначені три методи:

- `isDown()` – повертає значення `true`, якщо кнопка знаходиться в натиснутому стані. Змінити поточний стан може або користувач, натиснувши на кнопку, або виклик методу `setDown()`.
- `isChecked()` – повертає значення `true`, коли кнопка перебуває у включеному стані. Змінити поточний стан може або користувач, натиснувши на кнопку, або виклик методу `setChecked();`
- кнопка доступна, тобто реагує на дії користувача, якщо метод `isEnabled()` повертає значення `true`. Змінити поточний стан можна викликом методу `setEnabled()`.

Кнопки.

Віджет кнопки можна зустріти в будь-якому застосунку, – наприклад, майже завжди там є кнопки Ok або Cancel (Скасувати) – без них не обходиться жодне діалогове вікно.

Іноді такий віджет називають «командною кнопкою». Він являє собою прямокутний елемент керування і використовується, як правило, для виконання певної операції при натисканні на нього. Клас `QPushButton` віджета натискної кнопки визначено в заголовковому файлі `QPushButton`.

Створити натискну кнопку можна наступним чином:

```
QPushButton * pcmd = new QPushButton ("My Button");
```

Перший параметр (типу стрічка) задає напис кнопки. Звичайно, в конструктор можна передавати також і віджет предка, але так робити недоцільно, оскільки це зробить менеджер.

При натисканні кнопки відправляється сигнал `clicked()`. Кнопка вважається натиснутою, якщо користувач клацнув на ній кнопкою миші або натиснув на клавішу <Enter>, за умови, що кнопка була у фокусі. Для створення такої кнопки, можна скористатися методом `setDefault()`.

Приклад, показаний на рис. 1, демонструє різні варіанти натискання кнопок:

• **Normal Button** (Звичайна кнопка) відповідає тій кнопці, яку використовується у більшості випадків. Після відпускання кнопка завжди повертається в своє початкове положення;

• **Toggle Button** (Вимикач) може перебувати в двох станах: натиснутому або не натиснутому, які відповідають положенням «включено» або «вимкнено».

• **Flat Button** (Плоска кнопка) за своїми функціональними можливостями ідентична звичайної кнопці. Різниця лише в зовнішньому вигляді. Наприклад, завдяки тому, що контури цієї кнопки не видно, нею можна скористатися для розміщення «секретної кнопки» діалогового вікна;

• нарешті, остання кнопка **Pixmap Button** (Кнопка із зображенням) є кнопкою, що містить растрове зображення.

У роздруку 1 показано текст програми, вікно якого показано на рис. 1. Спочатку створюється віджет звичайної кнопки (покажчик `pcmdNormal`). Кнопка-вимикач (покажчик `pcmdToggle`) створюється так само, як і звичайна кнопка, але для неї викликом методу `setCheckable()` з параметром `true` встановлюється режим вимикача. Виклик методу `setChecked()` з параметром `true` переводить цю кнопку у включений стан.

Рисунок 1 – Приклади натискних кнопок

Потім створюється віджет плоскої кнопки (покажчик `pcmdFlat`) як звичайної кнопки. Викликом методу `setFlat()` з параметром `true` їй надається плоский вид.

Для створення кнопки з растровим зображенням спочатку створюється об'єкт растрового зображення `pix`, в який з ресурсів завантажується файл `ChordsMaestro.png`, зазначений в конструкторі. Потім створюється кнопка (покажчик `pcmdPix`), після чого об'єкт растрового зображення передається в метод `setIcon()` для встановлення на кнопці. Слот `setIconSize()`

задає розміри растрового зображення, в даному випадку вони відповідають оригінальним розмірами зображення, які повертає метод `size()` об'єкта `QPixmap`.

Роздрук 1. Файд main.cpp

```
#include <QtWidgets>

int main(int argc, char** argv)
{
  QApplication app(argc, argv);

  QWidget wgt;
  QPushButton* pcmdNormal new QPushButton("&Normal Button");

  QPushButton* pcmdToggle new QPushButton("&Toggle Button");
  pcmdToggle->setCheckaьle(true);
  pcmdToggle->setChecked(true);

  QPushButton* pcmdFlat = new QPushButton("&Flat Button");
  pcmdFlat->setFlat(true);

  QPixmap pix(":/ChordsMaestro .png");
  QPushButton* pcmdPix = new QPushButton("&Pixmap Button");
  pcmdPix->seticon(pix);
  pcmdPix->seticonSize(pix.size());

  //Layout setup
  QVBoxLayout* pvbxLayout new QVBoxLayout;
  pvbxLayout->addWidget(pcmdNormal);
  pvbxLayout->addWidget (pcmdToggle);
  pvbxLayout->addWidget (pcmdFlat);
  pvbxLayout->addWidget (pcmdPix);
  wgt.setLayout(pvbxLayout);

  wgt.show();

  return app.exec();
}
```

Існує можливість використання в натискних кнопках виринаючого меню (рис. 2). Подібні кнопки можна зустріти, наприклад, в переглядачі Microsof lntemet Explorer. Кнопка Start (Пуск) панелі завдань ОС Windows 7 також є такою кнопку. Додати меню можна, викликавши метод `setmenu()` і передавши покажчик на об'єкт виринаючого меню. Подібні кнопки можуть використовуватися як альтернатива для списку.

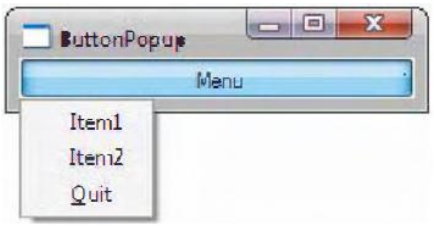


Рисунок 2 – Натискна кнопка із виринаючим меню

У роздруку 2 створюються віджети кнопки `cmd` і виринаючого меню `pmnu`. Викликом методу `addAction()` додається елемент меню. Остання команда Quit (Вихід) з'єднується зі слотом `quit()` об'єкта застосунку, що дозволяє користувачеві завершити застосунок, вибравши цю команду з меню. Встановлення створеного меню в натискній кнопці здійснюється викликом методу `setmenu()`.

Роздрук 2. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QPushButton cmd("Menu");
    QMenu* pmnu = new QMenu(&cmd);
    pmnu->addAction("Iteml");
    pmnu->addAction("Item2");
    pmnu->addAction("&Quit", &app, SLOT(quit()));

    cmd.setMenu(pmnu);
    cmd.show();

    return app.exec();
}
```

Прапорці

Більшість програм надають набори налаштувань, що дають можливість змінювати поведінку програми. Для цієї мети може стати в нагоді віджет прапорця, який дозволяє користувачеві вибирати відразу декілька опцій. Клас `QCheckBox` віджета кнопки прапорця визначено в заголовки `QCheckBox`.

Примітка. Якщо ж опцій більше п'яти, то краще використовувати віджет списку `QListWidget`.

Прапорець є маленьким прямокутником і може містити пояснюючий текст або картинку. При клацанні на віджет в прямокутнику з'явиться позначка. Цього ж можна досягти натисканням клавіші <Пропуск>, коли віджет знаходиться у фокусі. Віджет прапорця встановлюється в положення «включено» або «вимкнено» і є, за логікою дії, кнопкою-вимикачем (toggle button). Але, на відміну від останньої, прапорець може мати ще й третій стан – невизначений (рис. 3). Приклад використання такого стану можна побачити в діалоговому вікні Properties (Властивості) Провідника в ОС Windows при виборі декількох файлів, що мають різні атрибути.

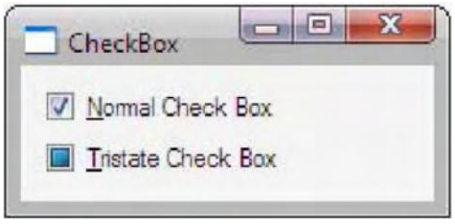


Рисунок 3 – Приклади прапорців

У роздруку 3 створюються два прапорці: покажчики `pchkNormal` і `pchkTristate`. Прапорець Normal Check Box (Звичайний прапорець) позначається викликом методу `setChecked()` з параметром `true`. Прапорець Tristate Check Box (прапорець з невизначеним станом) переводиться в режим підтримки третього, невизначеного стану, передачею значення `true` в метод `setTristate()`. Потім викликом методу `setCheckState()` і передачею в нього значення `Qt::PartiallyChecked` встановлюється третій стан.

Роздрук 3. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget wgt;
    QCheckBox* pchkNormal = new QCheckBox("&Normal Check Box");
    pchkNormal->setChecked(true);
```

```
    QCheckBox* pchkTri state = new QCheckBox("&Tristate Check Box");
    pchkTristate->setTristate(true);
    pchkTristate->setCheckState(Qt::PartiallyChecked);

    //Layout  setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pchkNormal);
    pvbxLayout->addWidget(pchkTristate);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

Перемикачі

Свою англійську назва (radiobutton) – віджет перемикач отримав завдяки подібності з кнопками перемикання діапазонів радіоприймача, на панелі якого може бути натиснута тільки одна з таких кнопок. Натискання на кнопку іншого діапазону призводить до того, що автоматично відключається натиснута до цього кнопка діапазону.

Перемикач є віджетом (рис. 4), який повинен знаходитися в одному з двох станів: включено (on) або вимкнено (off). Ці стани користувач може встановлювати за допомогою миші або клавіші <Пропуск>, коли кнопка перебуває у фокусі. Клас `QRadioButton` віджета перемикача визначено в заголовковому файлі `QRadioButton`.

Рисунок 4 – Перемикачі

Віджети перемикачів повинні надавати користувачеві, щонайменше, вибір однієї з двох альтернатив, не можуть використовуватися окремо і повинні бути згруповані разом. Їх групування можна виконати, наприклад, за допомогою класу `QGroupBox`.

Пояснюючі написи повинні бути визначені для кожного використовуваного в групі перемикача, бажано також задати і поєднання клавіші для швидкого доступу до кожного з перемикачів. Це досягається включенням в напис символу & перед потрібної буквою.

Примітка. Перевага перемикачів полягає в тому, що всі опції видно відразу, але вони займають багато місця. Тому якщо кількість перемикачів більше п'яти, то краще скористатися віджетом списку `QComboвох`.

У роздруку 4 створюється віджет для групи перемикачів `gbx`. Після створення перемикачів – `pradRed`, `pradGreen` і `pradBlue` – один з них (`pradGreen`) встановлюється у включений стан викликом методу `setChecked()` з параметром `true`. Перемикачі розміщуються на поверхні віджета групи `gbx`, а об'єкт класу `QVBoxLayout` автоматично вибудовує їх у вертикальному порядку (див. рис. 4).

Роздрук 4. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    QGroupBox gbx("&Colors");

    QRadioButton* pradRed = new QRadioButton("&Red");
    QRadioButton* pradGreen = new QRadioButton("&Green");
    QRadioButton* pradBlue = new QRadioButton("&Blue");
    pradGreen->setChecked(true);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pradRed);
    pvbxLayout->addWidget(pradGreen);
    pvbxLayout->addWidget(pradBlue);
    gbx.setLayout(pvbxLayout);

    gbx.show();

    return app.exec();
}
```

Групування кнопок

Віджети групування кнопок, в основному, не призначені для взаємодії з користувачем. Їх основне завдання – зробити його зручнішим для використання і спростити розуміння програми. Для цього важливо, щоб елементи інтерфейсу були об'єднані в окремі логічні групи. Крім того, потрібно пам'ятати, що кнопки перемикачів `QRadioButton` не можуть використовуватися окремо один від одного і повинні бути об'єднані разом.

Клас `QGroupBox` є класом для такого угрупування і являє собою контейнер, що містить в собі різні елементи керування. Він може мати пояснюючий напис в верхній частині вікна і цей напис може містити клавішу швидкого доступу, при натисканні на яку фокус переводиться на саму групу. Цей контейнер також можна забезпечити додатковим прапорцем, який буде керувати доступністю згрупованих елементів. Клас `QGroupBox` визначено в заголовковому файлі `QGroupBox`.

Наступний приклад створює групу перемикачів, що відповідають за колір фону (рис. 5). Наприклад, вибір перемикача Red (Червоний) призведе до того, що фон віджету верхнього рівня стане червоним. Прапорець Light (Яскравість) керує яскравістю кольору, а кнопка Exit (Вихід) здійснює вихід з програми. Текст відповідного файлу `main.cpp` показаний у роздруку 5.

Рисунок 5 – Приклади групування кнопок

```
#include <QApplication>
#include "Buttons.h"

int  main(int argc, char** argv)
{
    QApplication app(argc, argv);
    Buttons buttons;
    buttons.show();
```

```
    return app.exec();
}
```

Як видно з роздруку 8.6, в якому визначається клас `Buttons`, він успадковується від классу `QGroupBox`. У класі визначаються атрибути `m_pchk` для прапорця і `m_pradRed`, `m_pradGreen`, `m_pradBlue` для перемикачів. Це необхідно, щоб перераховані атрибути були доступні із слоту `slotButtonClicked()`.

Роздрук 5. Файл `Buttons.h`

```
#pragma once
#include <QGroupBox>
class QCheckBox;
class QRadioButton;
// ================================
class Buttons : public QGroupBox {
   Q_OBJECT
private:
   QCheckBox* m_pchk;
   QRadioButton* m_pradRed;
   QRadioButton* m_pradGreen;
   QRadioButton* m_pradBlue;
public:
    Buttons (QWidget*  pwgt = 0) ;
public slots:
   void slotButtonClicked();
};
```

У конструкторі класу `Buttons` (роздрук 7) викликом методу `setCheckable()` з параметром `true` встановлюється прапорець, який включається методом `setChecked()`.

Після створення трьох перемикачів (вказівники `pradRed`, `pradGreen` і `pradBlue`) перший з них виділяється викликом методу `setChecked()` з параметром `true`. Сигнал `clicked()` для кожного перемикача з'єднується зі слотом `slotвuttonClicked()`.

Прапорець Light (Яскравість) (покажчик `m_pchk`) включається відразу після свого створення при допомогою методу `setChecked()`.

Останньою з кнопок створюється кнопка Exit (Вихід) (покажчик `pcmd`). Для виходу із застосунку при натисненні на цю кнопку сигнал `clicked()` з'єднується із слотом `quit()` об'єкта звстосунку. Потім створені віджети розміщуються у вертикальному порядку за допомогою об'єкта класу `QVBoxLayout`. В останньому рядку викликається слот `slotButtonClicked()` для ініціалізації.

Роздрук 7. Файл Buttons.cpp. Конструктор класу Buttons.

```
Buttons::Buttons(QWidget* pwgt /*=0*/) : QGroupBox("QGroupBox",pwgt)
{
    resize(l00, 150);
    setCheckable(true);
    setChecked(true);
    m_pradRed   = new QRadioButton("&Red");
    m_pradGreen = new QRadioButton("&Green");
    m_pradBlue  = new QRadioButton("&Blue");
    m_pradGreen->setChecked (true) ;
    connect(m_pradRed, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    connect(m_pradGreen, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    connect(m_pradBlue, SIGNAL(clicked()), SLOT(slotButtonClicked()));

    m_pchk = new QCheckBox("&Light");
    m_pchk->setChecked(true);
    connect(m_pchk, SIGNAL(clicked()), SLOT(slotButtonClicked()));

    QPushButton* pcmd = new QPushButton("&Exit");
    connect(pcmd, SIGNAL(clicked()), qApp, SLOT(quit()));
```

```
    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(m_pradRed);
    pvbxLayout->addWidget(m_pradGreen);
    pvbxLayout->addWidget(m_pradBlue);
    pvbxLayout->addWidget(m_pchk);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);

    slotButtonClicked();
}
```

У роздруку 8 показана реалізація методу `slotвuttonClicked()`, в якому при кожному виклику створюється об'єкт палітри і перевіряється стан прапорця Light (Яскравість) для встановлення змінної `nLight`, яка керує яскравістю кольору. В операторах `if` виконується аналіз стану кнопок-перемикачів за допомогою методу `isChecked()`. В залежності від позначеної кнопки перемикача, колір фону палітри `QWidget::backgroundRole()` змінюється викликом методу `QPalette::setColor()` і встановлюється у віджеті за допомогою методу `QWidget::setPalette()`.

Роздрук 8. Файл Buttons.cpp. Метод slotButtonClicked()

```
void Buttons::slotButtonClicked()
{
    QPalette pal = palette();
    int  nLight = m_pchk->isChecked() ? 150 : 80;
    if (m_pradRed->isChecked())
    {

       pal.setColor(backgroundRole(), QColor(Qt::red).light(nLight));
    }
    else  if (m_pradGreen->isChecked () )
    {
       pal.setColor(backgroundRole(), QColor(Qt::green).light(nLight));
    }
    else
    {
       pal.setColor(backgroundRole(), QColor(Qt::blue).light(nLight));
    }
     setPalette(pal);
}
```

Висновки

• Існують три основних віджета, успадкованих від класу `QAbstractButton`: кнопки, прапорці та перемикачі.

• Кнопки використовуються для виконання певних дій. При натисканні на кнопку відправляється сигнал `pressed()`, після відпускання – сигнал `released()`. Найчастіше використовується сигнал `clicked()`, який відправляється, якщо користувач натиснув і відпустив кнопку.

• Прапорці часто застосовуються в діалогових вікнах, що містять опції. Група прапорців служить для одночасного вибору декількох опцій. Можливий варіант, коли не буде обраний ны один з них.

• Перемикачі використовуються тільки в групі, в якій одночасно можна вибрати тільки один з перемикачів. Тим самим за допомогою цієї групи моделюється відношення «один-до-багатьох».

• Основне завдання віджета групування – полегшити сприйняття і роботу з програмою. З його допомогою елементи інтерфейсу об'єднуються за належністю в окремі логічні групи.

Запитання.

1. Базовий класс `QAbstractButton` для створення кнопок.
2. Прапорці, перемикачі, радіокнопки і класи для їх створення.
3. Групування кнопок.

6. ЕЛЕМЕНТИ НАЛАШТУВАННЯ

Мета. Вивчення можливостей елементів налаштування.

Вступ.

Група віджетів, що відносяться до елементів налаштувань, використовується, як правило, для встановлення значень, що не вимагають великої точності, – наприклад, налаштування гучності звуку, швидкості руху курсору (покажчика) миші, прокручування вмісту вікна і інших подібних дій.

План.

1. Класс QAbstractSlider
2. Індикатор процессу
3. Електронний індикатор

1. Класс QAbstractSlider

Цей клас є базовим для всіх віджетів налаштування: повзунка (`QSlider`), смуги прокрутки (`QScrollBar`) і встановлювача (`QDial`). Всі перераховані далі можливості також доступні і у всіх успадкованих від нього класах. Визначення класу міститься в заголовковому файлі `QAbstractSlider`.

Якщо потрібно створити свій власний віджет, то можна успадкувати цей клас і реалізувати метод `sliderChange()`, який викликається щоразу при зміні значення.

Зміна положення

Класи віджетів, успадковані від класу `QAstractSlider`, можуть бути як горизонтальними, так і вертикальними. Для зміни розташування використовується слот `setOrientation()`, в який для задання горизонтального розташування передається значення `Qt::Horizontal`, а для вертикального – `Qt::Vertical`.

Встановлення діапазону

Для встановлення діапазону значень використовується метод `setRange()`. У цей метод першим параметром передається мінімально можливе значення (його нижня межа), а другим задається його максимально можливе значення (верхня межа). Також можна скористатися методами `setMinimum()` і `setMaximum()` відповідно. Наприклад, для того щоб задати діапазон від 1 до 10, можна поступити наступним чином:

```
psld->setRange(1, 10);
```

або так:

```
psld->setMinimum(1);
psld->setMaximum(10);
```

Встановлення кроку

За допомогою методу `setSingleStep()` можна задати крок, тобто значення, на яке, наприклад, повзунок зрушиться при натисканні на стрілки смуги прокрутки або на клавіші курсору клавіатури.

Метод `setPageStep()` задає крок для сторінки. Переміщення сторінок виконується, наприклад, для елемента повзунка при натисканні на область, що знаходиться між стрілками і головкою повзунка або клавішами <Page Up>, <Page Down>.

Установка і отримання значень

Для того щоб встановити будь-яке значення, необхідно скористатися слотом `setValue()`. Для отримання поточного значення можна викликати метод `value()`.

Сигнал `sliderMoved(int)` передає актуального значення положення і відправляється при зміні користувачем покажчика поточного положення.

Сигнал `valueChanged()` надсилається одночасно з сигналом `sliderMoved(int)` відразу після зміни положення повзунка і також передає змінене значення смуги прокрутки.

Поведінка сигналу змінюється викликом методу `setTracking()`. Якщо передати йому значення `false`, це призведе до того, що сигнал `valueChanged()` буде відправлятися тільки при відпуску покажчика поточного положення.

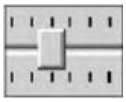
Щоб дізнатися, відпустив користувач покажчик поточного становища повзунка або все ще утримує його, можна приєднатися до сигналів `sliderPressed()` або `sliderReleased()`.

Повзунок

Повзунок дозволяє досить комфортно виконувати налаштування Встановлення програми. Клас `QSlider` повзунка визначено в заголовковому файлі `QSlider`.

Клас `QSlider` містить метод, керуючий розміщенням рисок (шкали) повзунка. Риски дуже важливі при відображенні повзунка. Вони дають користувачеві візуально більш чітке уявлення про його місцезнаходження і показують крок. Можливі значення, які можна передати в метод `setTickPosition()`, показані в табл. 1.

Таблиця 1 – Значення перечислення TickPosition классу QSlider

Константа	Описання	Вид
`NoTicks`	`Повзунок без рисок`	
`TicksAbove`	`Риски на верхній стороні`	
`TicksBelow`	`Риски на нижній стороні`	
`TicksBothSides`	`Риски на верхній і нижній стороні`	

Метод `setTickInterval()` задає крок малювання рисок. Не слід задавати велику кількість рисок, так як це призведе до появи суцільною сірою лінії і не принесе ніякої користі.

Наступний приклад (роздрук 1) демонструє створення повзунка. Вікно застосунку, зображене на рис. 1, містить віджети повзунка і написи, причому текст напису змінюється в залежності від положення повзунка.

Рисунок 1 – Вікно з повзунком

Роздрук 1. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
```

```
{
    QApplication app(argc, argv);

    QWidget wgt;
    QSlider* psld = new QSlider(Qt::Horizontal);
    QLabel*  plbl = new QLabel("3");

    psld->setRange(0, 9);
    psld->setPageStep(2);
    psld->setValue(3);
    psld->setTickinterval(2);
    psld->setTickPosition(QSlider::TicksBelow);
    QObject::connect(psld, SIGNAL(valueChanged(int)), plbl,SLOT(setNum(int)));

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(psld);
    phbxLayout->addWidget(plbl);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

У роздруку 1 створюються віджети повзунка (покажчик `psld`) і напису (покажчик `psbl`).

Після цього викликом методу `setRange()` здійснюється встановлення діапазону значень повзунка від 0 до 9. Крок сторінки встановлюється рівним 2 методом `setPageStep()`.

За допомогою методу `setValue()` можна задавати стартове значення, яке використовується для синхронізації з іншими елементами керування, які працюють разом з повзунком. З його допомогою можна зробити так, щоб величини віджетів збігалися один з одним, він також може служити просто для задання початкового значення при першому показі елемента. Тут в метод повзунка `setValue()` передається значення 3, синхронізуюче його із значенням, відображуваним при створенні напису.

Крок для малювання рисок встановлюється рівний двом, для чого в метод повзунка `setTickinterval()` передається значення 2. Виклик методу `setTickarks()` здійснює встановлення рисок знизу. Методом `connect()` сигнал повзунка `valueChanged(int)` з'єднується із слотом напису `setNum(int)`.

На завершення за допомогою горизонтальної компоновки виконується розміщення елементів на поверхні віджета `wgt`.

Смуга прокрутки

Смуга прокрутки – це важлива складова практично будь-якого призначеного для користувача інтерфейсу. Вона інтуїтивно сприймається користувачем, і з її допомогою відображаються текстові або графічні дані, які за розмірами перевищують відведену для них область. Використовуючи покажчик поточного стану смуги прокрутки, можна перемішати дані у видиму область. Цей покажчик показує відносну позицію видимої частини об'єкта, завдяки якій можна отримати уявлення про розмір самих даних.

Клас `QScrollBar` є реалізацією віджета смуги прокрутки. Він визначений в заголовковому файлі `QscrollBar` і не містить ніяких додаткових методів і сигналів, які розширюють визначення класу `QAstractSlider`.

Окремо смуги прокрутки використовуються дуже рідко. Вони вбудовані у віджет `QAbstractScrollArea`. Тому, якщо потрібно скористатися класом смуги прокрутки `QScrollBar`, то не виключено, що найкращим варіантом може виявитися використання одного з віджетів, які успадковують базовий клас для видової прокрутки `QAstractScrollArea`.

У об'єктів, успадкованих від класу `QScrollBar`, можна викликати контекстне меню з параметрами навігації за замовчуванням (рис. 2).

Віджет смуги прокрутки має мінімальне і максимальне значення, поточне значення і орієнтацію. Переміщення покажчика поточного положення здійснюється за допомогою лівої кнопки миші. Як альтернативу, можна просто натиснути на кнопки стрілок, розташованих на кінцях смуги прокрутки.

Вікно програми (роздрук 2), показане на рис. 3, складається з електронного індикатора і смуги прокрутки. Значення, що відображається електронним індикатором, змінюється в залежності від положення покажчика поточного положення.

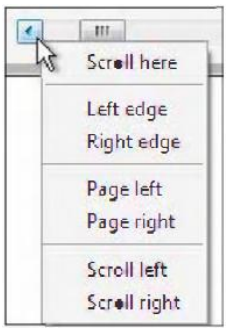


Рисунок 2 – Контекстне вікно прокрутки

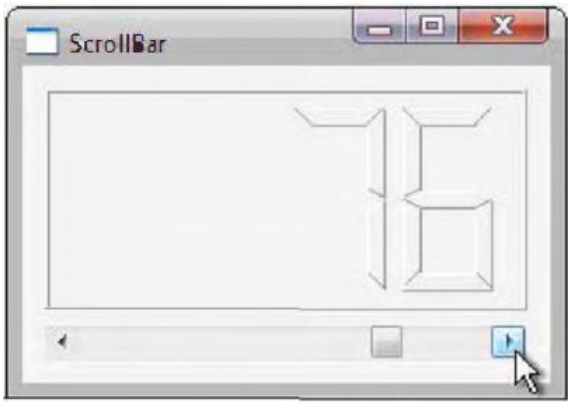


Рисунок 3 – Демонстрація роботи смуги прокрутки

Роздрук 2. Файл main,cpp

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget wgt;
    QLCDNumber* plcd = new QLCDNumber(4);
    QScrollBar* phsb = new QScrollBar(Qt::Horizontal);
    QObject::connect(phsb,SIGNAL(valueChanged(int)),plcd,SLOT(display(int)));

    //Layout  setup
    QVBoxLayout*  pvbxLayout  =  new  QVBoxLayout ;
    pvbxLayout->addWidget(plcd);
    pvbxLayout->addWidget(phsb);
    wgt.setLayout(pvbxLayout);

    wgt.resize(250,150);
    wgt.show();
```

```
    return app.exec();
}
```

У роздруку 2 створюються віджети електронного індикатора (покажчик `plcd`) і смуги прокрутки (покажчик `phsb`). Після цього сигнал `valueChanged()` смуги прокрутки з'єднується зі слотом `display()` електронного індикатора, що служить для відображення значень цілого типу, за допомогою методу `connect()`. На завершення віджети електронного індикатора і смуги прокрутки розміщуються вертикально на поверхні віджета `wgt` за допомогою об'єкта класу `QVBoxLayout`.

Встановлювач

Клас `QDial` віджета встановлювача визначено в заголовковому файлі `QDial`. Цей віджет подібний на регулятор гучності радіоприймача, який можна обертати за допомогою миші або клавіш керування курсором. За своїми функціональними можливостями він подібний на повзунок. Різниця в тому, що кругла форма цього віджета дозволяє користувачеві після досягнення максимального значення відразу перейти до мінімального, і навпаки. Щоб дозволити або заборонити прокручування служить слот `setWrapping()`.

За відображення рисок відповідають метод `setNotchTarget()`, який встановлює їх кількість, і слот `setNotchesVisible()`, який керує їх видимістю.

Вікно програми (роздрук 3), показане на рис. 4, містить віджети встановлювача і індикатора процесу. Стан останнього залежить від місця розташування стрілки встановлювача.

Роздрук 3. Файл `main.cpp`

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget wgt;
    QDial* pdia = new QDial;
    QProgressBar* pprb = new ProgressBar;

    pdia->setRange(0, 100);
    pdia->setNotchTarget(5);
    pdia->setNotchesVisible(true);
    QObject::connect(pdia,SIGNAL(valueChanged(int)),pprb,SLOT(setValue(int)));

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pdia);
    pvbxLayout->addWidget(pprb);
    wgt.setLayout(pvbxLayout);

    wgt.resize(180, 200);
    wgt.show();

    return app.exec();
}
```

У роздруку 3 створюються віджети встановлювача (покажчик `pdia`) і індикатора процесу (покажчик `pprb`). Викликом методу `setRange()` віджета встановлювача задається діапазон значень від 0 до 100. Метод `setNotchTarget()` встановлює крок рисок, рівний 5, а слот `setNotchesVisible()` робить їх видимими, отримавши значення `true`.

Рисунок 4 – Вікно застосунку, яке демонструє роботу встановлювача

Сигнал віджета встановлювача `valueChanged(int)` з'єднується за допомогою методу `connect()` зі слотом індикатора процесу `setProgress(int)`.

На завершення віджети встановлювача і індикатора процесу за допомогою об'єкта класу `QVBoxLayout` розміщуються вертикально.

Висновки

• Віджети налаштувань застосовуються у тих випадках, коли не потрібне точне встановлення значень. Базовим класом для всіх віджетів встановлювача є клас `QAbstractSlider`, який містить основні методи для визначення діапазону, кроку і поточного значення.

• Повзунок дозволяє здійснювати налаштування параметрів програми – наприклад, гучність звуку, швидкість руху покажчика миші і т. п. Цей віджет містить риски, що дають користувачеві візуально більш чітке уявлення про крок і місце знаходження повзунка.

• Смуга прокрутки дозволяє відображати текстові або графічні дані, що перевищують за розмірами відведену для них область. За допомогою покажчика поточного положення можна переміщати дані з невидимої області у видиму.

• Встановлювач дуже подібний на віджет повзунка, різниця полягає у зовнішньому вигляді, в префіксах відправляються сигналів і в можливості вирішення прокручування при переході від максимального значення до мінімального.

Запитання.

1. Клас QAbstractSlider і його методи.

СПИСОК ЛІТЕРАТУРИ

1. Шлее М. Qt 5.3. Профессиональное программирование на C++. – СПб.: БХВ-Петербург, 2015. – 928 с.

2. Lee Zhi Eng. Qt 5 GUI Progranning Book. Birmingham, UK: Pack Publishing, 1026. – 283.