Структура проекту, консольні проекти.

Осовні типи даних

Робота с qmake

Жоден програміст для компіляції своєї програми не буде кожен раз задавати параметри для компонування і передавати шлях до бібліотек "вручну". Набагато зручніше створити make-файл (makefile), який візьме на себе всю роботу по настройці компілятора і компоновщика.

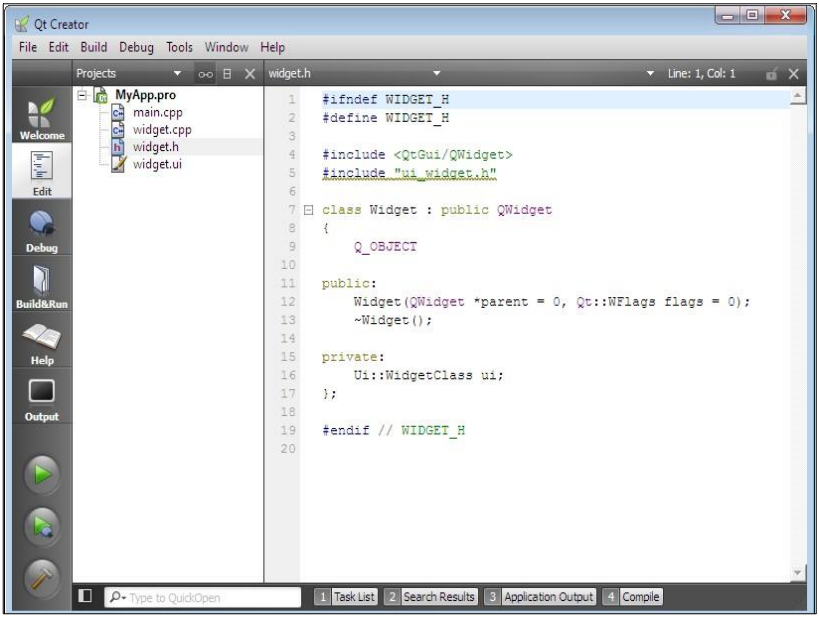


Рис. 3.1. Окно интегрированной среды разработки Qt Creator

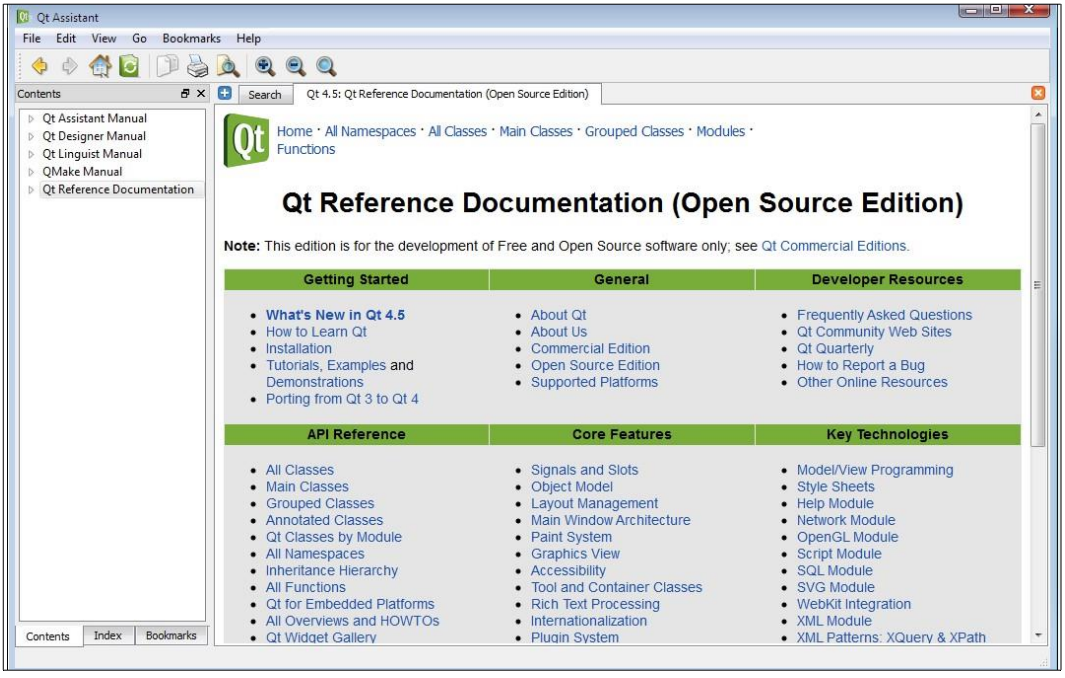


Рис. 3.2. Вікно програми Qt Assistant

Створення make-файлів вручну вимагає від їх автора досвіду і розуміння процесів компонування додатків, причому в залежності від платформи вид цих файлів буде відмінюватися. Раніше техніка створення подібних файлів була невід'ємною частиною про- грамування, але тепер багато що змінилося. І справа зовсім не в тому, що структура make- файлів стала простішою, швидше навпаки - вона стала ще складнішою. Просто з'явилися спеціальні утиліти - генератори, які виконають цю роботу за вас.

Утиліта qmake увійшла в поставку Qt, починаючи з версії 3.0.Слід відзначити, що ця утиліта так само добре переносима, як і сам Qt. Програма qmake інтерпретує файли проектів, які мають розширення pro і містять різні параметри, для створення make- файлів. Найдивовижніше, що утиліта qmake здатна не тільки створювати make-файли, але і самі pro-файли.

Припустимо, ви вказали, що в каталозі є вихідні файли C ++, виконавши наступну команду:

```
qmake -project
```

В результаті буде реалізована спроба автоматичного створення pro-файлу. Це зручно, т. К. На перших порах вам не знадобиться вникати в усі тонкощі створення pro-файлів. Також це може виявитися корисним в тому випадку, якщо ви володієте великим кількісно вом файлів, до яких потрібно створити pro-файл - тоді у вас відпаде необхідність вносити їх імена вручну. Створити з pro-файлу make-файл зовсім неважко, для цього потрібно просто виконати команду:`qmake file.pro -o Makefile`

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

створюваного платформозалежного make-файлу.

Примітка

На Mac OS X є дві основні можливості створення файлів за допомогою qmake. Перша - це створення make-файлу для GNU C ++: qmake -spec macx-g ++ -o Makefile, друга - створення проектних файлів для XCode: qmake -spec macx-xcode -o Makefile. Остання опція є опцією за замовчуванням.

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

```
qmake –project qmake
make
```

Звичайно, для більш серйозної роботи нам знадобиться змінювати вміст pro-файлів, що дозволить нам здійснювати більш тонке налаштування для наших проектів. Таблиця 3.1 містить деякі опції pro-файлу, повний список яких можна отримати в офіційній документації Qt, що постачається разом з самою бібліотекою (для цього ви можете просто запустити програму Qt Assistant).

Таблиця 3.1. *Деякі опції для файла проекта*

Опція	**Призначення**	**Приклад**
HEADERS	Список створених заголовних файлів	`HEADERS + = mainwindow.h`
SOURCES	Список створених файлів реалізації (з розширенням cpp)	`SOURCES + = main.cpp \ mainwindow.cpp`
FORMS	Список файлів з розширенням ui. Ці файли створюються програмою Qt Designer і містять опис інтерфейсу користувача в форматі XML (див. глави 41 і 45)	`FORMS + = mainwindow.ui`
TARGET	Ім'я програми. Якщо дане поле не заповнено, то назва програми буде відповідати імені проектного файлу	`TARGET = MyFirstProject`
LIBS	Задає список бібліотек, які повинні бути підключені для створення виконуваного модуля	`LIBS + = -L./libs \ -L./my_libs \ -lmycustomlib`
CONFIG	Задає опції, які повинен використовувати компілятор (наприклад: режим налагодження, вивід попереджень, компіляція динамічної бібліотеки і т.п.)	`CONFIG + = dll plugin \ warn_on release`
DESTDIR	Задає шлях, куди буде поміщений готовий виконуваний модуль	`DESTDIR = ./bin`
DEFINES	Тут можна передати опції для компілятора. Наприклад, це може бути опція приміщення налагоджувальної інформації для відладчика debuger в виконуваний модуль	`DEFINES + = DEBUG_OUTPUT \ CUSTOM_DEFINE`
INCLUDEPATH	Шлях до каталогу, де містяться заголовки. Цією опцією можна скористатися в разі, якщо вже є готові заголовки та ви хочете використати їх (підключити) в поточному проекті	`INCLUDEPATH + = ./includes \ ./my_header_fi`

		les
DEPENDPATH	В даному розділі зазначаються залежності, необхідні для компіляції	
SUBDIRS	Задає імена підкаталогів, які містять pro-файли	
TEMPLATE	Задає різновид проекту. Наприклад: app - додаток, lib - бібліотека, subdirs - підкаталоги	TEMPLATE = lib
TRANSLATIONS	Задає файли перекладів, використовувані в проекті	

Давайте трохи розберемо "анатомію" проектних файлів. Отже, pro-файл може виглядати наступним чином:

```
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 додаток, тому TEMPLATE = app (якби нам потрібно було створити бібліотеку, то TEMPLATE варто було б присвоїти значення lib). У другому рядку, в HEADERS, перераховуються всі заголовки, що належать проекту. В опції SOURCES перераховуються всі файли реалізації проекту. TARGET визначає ім'я програми, CONFIG - опції, які повинен використати компілятор відповідно до під'єднюваних бібліотек. Наприклад, в нашому випадку:

- `qt` вказує, що це Qt-додаток і використовується бібліотека Qt;
- `warn_on` означає, що компілятор повинен видавати якомога більше попереджувальних повідомлень;
- `release` вказує, що програма має бути відкомпільована в остаточному варіанті, без налагоджувальної інформації.

> ***Примітка***
>
> У Mac OS X в опції CONFIG також вказуються архітектури, для яких повинно бути створено програму. Наприклад, для 32-бітової архітектури Intel-процесорів потрібно поставити x86, для 64-бітової архітектури Intel-процесорів - x86_64. Для 32-бітової архітектури PowerPC має стояти ppc, а для її 64-бітової архітектури - ppc64. Крім того, ці значення можна комбінувати один з одним, для того щоб збільшити кількість платформ, на яких ваш додаток буде здатнний запускатися. Ось так буде виглядати секція CONFIG, якщо ваша програма призначена для експлуатації на всіх бітових архітектурах Intel і PowerPC: CONFIG + = debug x86 ppc x86_64 ppc64. Таким чином ми отримаємо універсальні файли - Universal Binaries, які добре знайомі всім користувачам Mac OS X.

Як видно з прикладу, програмою qmake не потрібно багато інформації, т. К. Вона спирається на файл локальної конфігурації, який визначений системної конфігурацією. Такий файл дуже важливий ще й тому, що один і той же виклик утиліти qmake призведе до созда- нию різних make-файлів в залежності від того, на якій платформі вона була викликана. Це один із дуже важливих кроків у бік платформонезалежності самих проектних файлів.

Проектний файл може бути використаний для компіляції проектів, розташованих в раз- них каталогах. Наочним прикладом може служити pro-файл каталогу прикладів, які доступні на FTP-сервері по посиланню ftp://85.249.45.166/9785977507363.zip. Цей файл виглядає приблизно так:

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

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

Рекомендації для проекта з Qt

При реалізації файли класів найкраще розбивати на дві окремі частини. Частина визначення класу поміщається в заголовки з розширенням h, а реалізація класу - в файл з розширенням cpp. Важливо пам'ятати, що в заголовки з визначенням класу повинна міститися директива препроцесора #ifndef. Сенс цієї директиви полягає в тому, щоб уникнути конфліктів в тому випадку, коли один і той же заголовок буде включатися у вихідні файли більш ніж один раз.

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

За традицією заголовний файл, як правило, носить ім'я класу, який міститься в ньому. У заголовних файлах, з метою більш швидкої компіляції, для покажчиків 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 потреби в метаінформації, цим макросом можна знехтувати. Але потрібно враховувати ту обставину, що через відсутність метаінформації не можна буде використовувати приведення типу `qobject_cast<T>(obj)`.

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

Основна програма повинна бути реалізована в окремому файлі, який є "стартовим майданчиком" додатка. Цьому файлу прийнято давати ім'я main.cpp. Це зручно ще й тому, що проект може складатися із сотень файлів, і якщо слідувати такому правилу, то знайти відправну точку всього проекту не складе труднощів.

Дотримання цих рекомендацій може послужити хорошу службу. Проектам властиво з часом розширюватися, тому непогано з самого початку дотримуватися

певної структури, щоб потім відчувати себе в своїх і чужих проектах, які дотримуються прийнятої в Qt структури, "як риба у воді".

Метаоб´єктний компілятор MOC

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

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

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

> Краще, якщо вони будуть окремо відкомпільовані і приєднані компоновщиком до основної програми. Хоча при написанні демонстраційних програм цим правилом можна знехтувати, щоб розмістити весь код в одному файлі main.cpp.

Якщо ви працюєте з файлами проекту, то про існування MOC можете і не здогадуватися, адже в цьому випадку управління MOC автоматизовано. 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'єкта, наприклад, сигнально-слотові з'єднання, ім'я класу і структуру успадкування.

Компілятор ресурсів RCC

Майже будь-яка програма так чи інакше звертається до сторонніх ресурсів, таких як растрові зображення, файли перекладу і т. Д. Це не є достатньо надійним і ефективним способом, т. як. ці ресурси можуть бути видалені або недоступні з яких-небудь інших причин. Це, безсумнівно, може відбитися на правильній роботі програми, її зовнішньому вигляді і працездатності. Компілятор ресурсів надає можливість впровадження таких файлів в виконувані модулі, для того щоб додаток отримував доступ до необхідних ресурсів в процесі його виконання. Існують

спеціальні угоди про іменування, завдяки яким можна однозначно звертатися до таких ресурсів. Всі необхідні для використання файли (ресурси) повинні бути описані в спеціальному файлі з розширенням qrc (Qt Resource Collection, колекція ресурсів Qt) разом з їхніми дорогами. 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 нашого прикладу буде підданий аналізу компілятором ресурсів - утилітою rcc, для створення з файлів open.png і quit.png одного вихідного файлу C ++, що містить всі їх дані, які будуть компілюватися і компонуватись разом з іншими файлами проекту. Всі дані ресурсу зберігаються в файлі C ++ у вигляді одного великого масиву.

Такий підхід дає впевненість в тому, що необхідні ресурси завжди доступні, що поможе уникнути проблем неправильної установки необхідних для виконуваної програми файлів. Сам же qrc-файл повинен бути вказаний в pro-файлі в секції RESOURCES, для того щоб утиліта qmake врахувала інформацію з файлу ресурсу. Наприклад:

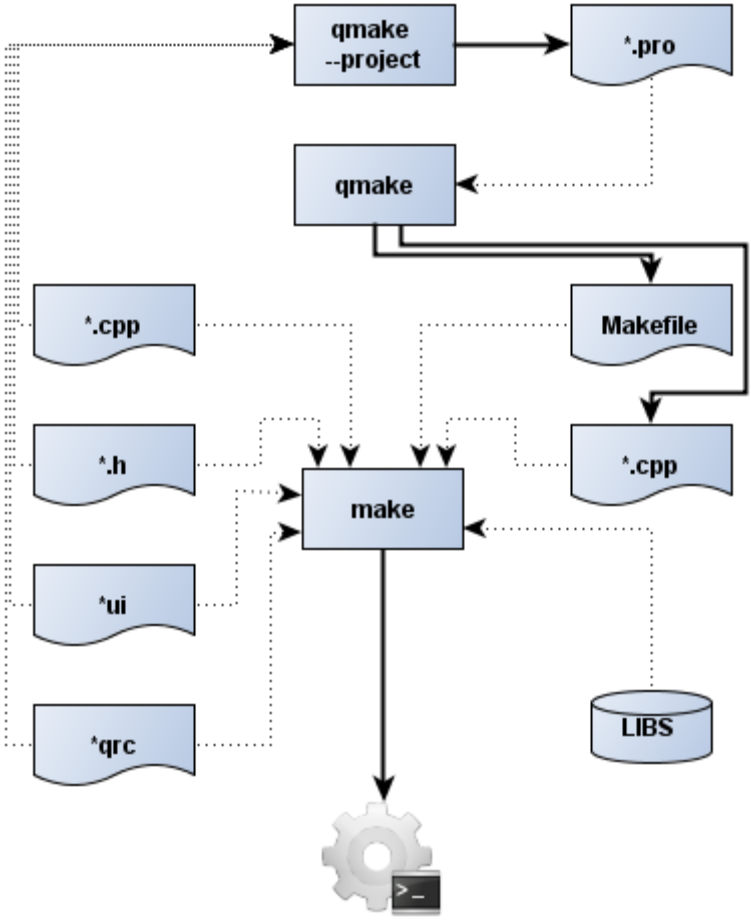
```
RESOURCES = images.qrc
```

Для того щоб скористатися файлом open.png, а точніше, представленим в ньому растровим зображенням, можна поступити наступним чином:

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

12.2 Компіляція проекту

Компіляція проекту проходить в два етапи. Спочатку виконується попередня обробка проекту за допомогою програми qmake. Цей інструмент Qt несе відповідальність за весь процес компіляції проекту. Він читає зміст проектного файлу і генерує необхідні проміжні файли (додаткові файли з вихідним кодом і make-файлидля компіляції). Це необхідно для того, щоб перетворити все особливі розширення Qt, які були використані в програмі, в код на мові C ++ і використовувати додаткові настройки для проекту, описані в pro-файлі. Після цього проект готовий до обробки компілятором. Другим етапом є безпосередньо процес компіляції. Всі ці дії виконуються автоматично в середовищі QtCreator.



Мал. 12.1. Процес компіляції проекту

Після успішної компіляції ми отримуємо виконуваний файл програми. Відкрийте папку проекту. Вона буде містити виконуваний файл і всі проміжні файли, згенеровані в процесі.

Таким чином, процесом побудови проекту керує .pro-файл.При наявності вихідних текстів програми і при відсутності .pro-файлу,його можна згенерувати. Для цього з командного рядка необхідно перейти в папку, яка містить вихідні тексти програми і викликати qmake з параметром --project.Цим прийомом зручно скористатися, щоб згенерувати файл проекту і використовувати оболонку QtCreator для роботи над програмою (навіть для звичайних програм на C ++ без Qt).

Розділ"Projects"(Проекти) містить набір необхідних налаштувань для процесу компіляції і для настройки середовища запуску проекту. Однією з таких налаштувань є опція ShadowBuild,яка дозволяє включити режим при якому для проміжних файлів, make-файліві продуктів компіляції створюється окрема папка поза папки з вихідним кодом проекту (настройки розміщення для неї - в

полеBuilddirectory).Це дозволяє побудувати і зберігати одночасно кілька варіантів побудованого проекту для різних інструментаріїв. Також це зберігає папку з вихідним кодом від засмічення файлами, створеними в процесі побудови проекту. При вимкненому Shadow build проміжні файли і папки з побудованої програмою зберігаються в папці, яка містить файлпроекту.

Звичайно, створені проміжні файли не є безпосередньою частиною проекту. Вони були згенеровані, і перезаписуватимуть при необхідності під часкомпіляції.Тому не варто додавати їх до pro-файлу або робити будь-які зміни в них. Також не варто їх додавати в систему контролю версій, якщо її використовують при розробці.

Іноді згенеровані файли разом з об'єктними та make-файлбуває необхідно видалити. Це необхідно робити перед тим як помістити його проект для збереження, оскільки згенеровані файли займають досить багато місця на диску в порівнянні з обсягом вихідного коду. Часом можуть виникати проблеми з компіляцією, коли після значних змін в структурі програми проміжні файли не були достатньо добре заново згенеровані. У таких випадках виникає необхідність очистити проект. Для цього виберіть в головному меню Build-> Clean Project (Сборка-> Очистити проект). Це дозволить видалити згенеровані файли, крім скомпільованої виконуваного файлу і make-файлів.

Для того, щоб очистити проект повністю, необхідно змінити деякі настройки. Відкрийте розділ Projects (Проекти) і в розділі Clean Steps (Етапи очищення) натисніть кнопку Details (Докладніше) і змініть параметр Make arguments (Аргументи make) з clean на distclean(рис.12.2).

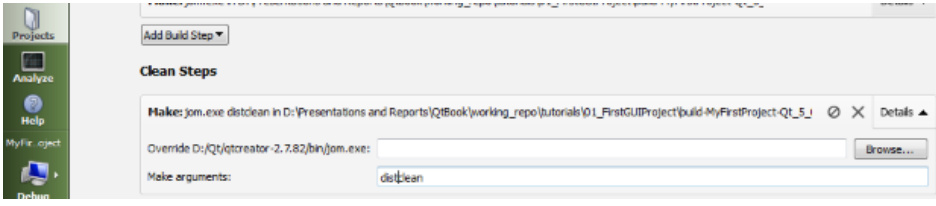


Рис. 12.2. Налаштування очищення проекту

Знову очистіть проект - все сгенерованние файли, включно з виконуваним файлом і make-файл,будуть видалені.

Методи налагодження

Немає програм без помилок і дефектів (bug, помилка чи інакше баг). У процесі розробки програм часто виникають проблеми з їх виявленням. Розробникам доводиться витрачати чималу частину робочого часу на те, щоб знайти і усунути їх. До засобів, що допомагають знизити їх кількість, можна віднести:

- надання вихідного коду для перегляду іншими розробниками (code review);
- створення класів для автоматизованих тестів, докладно описаних в *главі 46*.

Помилки можна мінімізувати, але, в будь-якому випадку, повністю їх уникнути не вдасться, і якщо в вашу програму раптом закрався підступний баг, то найпершим засобом, що допомагає в нелегкій праці його пошуку, буде відладчик. Роль

відладчика полягає в наданні оболонки, в якій можна відслідковувати зміну даних під час виконання програми, завдяки чому можна дізнатися, чому створена вами програма веде себе не так, як ви це задумували. Завдяки платформонезавісимость Qt розробник може для налагодження своїх програм за допомогою одного з вподобаних отладчиков, наприклад GDB або відладчик, вбудований в Microsoft Visual Studio. Якщо ви ще серйозно не стикалися з процесом налагодження програм, то рекомендую почати з відладчика без графічного інтерфейсу, оскільки подача команд в діалоговому режимі допоможе вам зрозуміти роботу відладчика як такого і в майбутньому гідно оцінити отладчики, що володіють графічним інтерфейсом. Тому ми докладніше розберемо відладчик GDB, доступний як для Windows, так для Linux і Mac OS X.

ПРИМІТКА

Інсталяційний файл GDB для MinGW (Windows) додається на компакт-диску, що йде разом з книгою, див. *додаток 3*.

Налагоджувач GDB (GNU Debugger)

GDB - це саме звичне засіб для налагодження програм в ОС Unix. Робота з цим отладчиком зазвичай здійснюється з командного рядка, хоча можна скористатися і оболонками, які надають можливість роботи з цим отладчиком в інтерактивному режимі, ось деякі з них: XXGDB, DDD і KDBG. Мабуть, найідеальнішим середовищем для роботи з налагоджувачем в інтерактивному режимі є IDE, в цьому випадку все необхідне знаходиться "під рукою". Якщо ви не збираєтеся працювати з налагоджувачем безпосередньо і надасте перевагу використовувати IDE, то подальше опис GDB можете пропустити. Але якщо ви хочете розібратися в процесі налагодження, то давайте створимо програму, яка свідомо містить проблемний код.

Лістинг 3.1. Проблемний код

```
void bug()
{
int n = 3; int* pn = &n;
// Помилка! delete pn;
}

int main()
{
bug(); return 0;
}
```

В лістингу 3.1 з функції main () здійснюється виклик функції bug (), в якій створюється і ініціалізується змінна n, а після присвоєння вказівником pn її адреси викликом оператора delete виконується спроба звільнення пам'яті, використовуваної змінної n. Оскільки пам'ять не була виділена динамічно, ця операція неминуче призведе до помилки.

Відкомпілюємо програму з параметром -g, щоб у виконуваний файл була включена інформація, необхідна для налагоджувача:

```
g++ -g bug.cpp –o bug
```

Примітка

В даному випадку я не став створювати традиційного для Qt pro-файлу і обмежився визо- вом компілятора безпосередньо. Тут це простіше. У випадках же з pro-файлами Qt, для того щоб отримати виконуваний файл з включеною налагоджування, змінна pro-файлу CONFIG повинна містити значення `debug` або `debug_and_release`.

Налагоджувач можна запустити наступним чином, вказавши ім'я програми, призначеної для налагодження:

```
gdb bug.exe
```

Примітка

Крім імені програми, в GDB можна додатково передавати і core-файл, сгенеріро-ний операційною системою після аварійного завершення програми. Це дуже зручно, т. К. Можна не завантажувати програму на виконання в відладчик, а знайти проблемне місце за допомогою core-файлу. На жаль, ОС Windows не генерує подібних файлів, тому надалі будуть описані прийоми роботи з відладчиком GDB, застосовні на обох ОС (Windows I Linux).

Після цього налагоджувач відобразить рядок запрошення такого вигляду:

```
(gdb)
```

Отже, давайте запустимо саму програму під налагоджувач. Для цього потрібно ввести команду
`run`:

```
(gdb) run
```

В результаті ми отримаємо повідомлення, що сигналізують про ненормальний завершення програми. Для того щоб розібратися в проблемі, потрібно переглянути стек програми. Це робиться за допомогою команди `where`:

```
(gdb) where
```

Висновок налагоджувача буде приблизно таким:

```
#0 0x77f767ce in _libmsvcrt_a_iname ()
...
#6 0x00401305 in operator
delete(void*) () #7 0x004012ae in
bug() () at bug.cpp:5
#8  0x004012df in main () at bug.cpp:10
```

Зауважте, що функція main () викликала в десятому рядку функцію bug (), а виклик п'ятого рядка цієї функції створив проблему.

За допомогою команди up можна піднятися по стеку програми на певну кількість рівнів. Давайте піднімемося на один рівень, це буде відповідати функції `bug()`:

```
(gdb) up 1
```

Налагоджувач покаже наступне:

```
#7 0x004012ae in bug() () at bug.cpp:5
4           delete pn;
```

Для того щоб дізнатися значення будь-якої локальної змінної функції, потрібно подати команду `print`. Давайте виконаємо це для змінної `n`:

```
(gdb) print n
```

У відповідь налагоджувач покаже її значення:

```
$1 = 3
```

Установка контрольних точок (break points) здійснюється в налагоджувачі за допомогою команди break. Встановимо нашу точку в функції bug () і перезапустити нашу програму:

```
(gdb) break bug (gdb) run
```

Налагоджувач зупиниться на заданій нами контрольній точці і покаже наступне:

```
Breakpoint 1, bug() () at bug.cpp:3
3           int n = 3;
```

Для того щоб перейти на наступний рядок, скористаємося командою next. Ця команда виконує код через підрядник без переходу всередину тіла функції:

```
(gdb) next
4           int* pn = &n;
```

З цього видно, що відладчик перейшов з третього рядка на четверту. Якщо знадобиться виконати рядки коду, включаючи рядок всередині тіла функції, то для цього потрібно було б скористатися командою `step`. Наприклад, ми могли б встановити контрольну точку в функції `main` () і за допомогою команди step увійти всередину функції `bug` (). У табл. 3.2 зведені найбільш часто використовувані команди відладчика.

Таблиця 3.2. *Деякі команди налагоджувача GDB*

Команда	**Опис**
`quit`	Вихід з нлагоджувача
`help`	Висновок довідкової інформації. Якщо додатковим параметром вказана яка-небудь команда, то виводиться повна довідкова інформація по цій команді
`run`	Запуск програми
`attach`	Приєднання відладчика до запущеного процесу з зазначеним ідентифікатором
`detach`	Процедура відключення відладчика від приєднаного процесу
`break`	Установка контрольної точки. Виклик команди без параметра встановить точку на на- дме виконуваної інструкції. Як параметр можна передавати ім'я функ- ції, номер рядка і зміщення. Якщо потрібно, можна вказати ім'я конкретного результат- ного файлу в вигляді <ім'я файлу>: <номер рядка> або <ім'я файлу>: <ім'я функції>
`tbreak`	Аналогічна команді `break` з тією лише різницею, що контрольна точка буде видалена після її досягнення

clear	Видалення контрольної точки. Виклик команди без параметра видалить контрольну точку наступного виконуваної інструкції. Як параметр можна передавати ім'я функції, номер рядка і зміщення. Якщо потрібно, то можна вказати ім'я конкретного вихідного файлу в вигляді: <ім'я файлу>: <номер рядка> або <Ім'я файлу>: <ім'я функції>
delete	Видалення всіх контрольних точок
disable	Відключення всіх контрольних точок
enable	Включення всіх контрольних точок
continue	Продовження виконання програми. Додатковим параметром можна вказати кількість ігнорувань контрольної точки
next	Виконання наступного рядка вихідного коду програми. Додатковим пара- метром можна задати кількість виконуваних рядків
step	Виконання наступного рядка вихідного коду програми. Додатковим пара- метром можна задати кількість виконуваних рядків. На відміну від next, при виконанні функції відбувається вхід в неї і зупинка
until	Продовження виконання програми до виходу з функції

Інші методи налагодження

Одним із стандартних прийомів налагодження є вставка в вихідний код операторів виведення, що дозволяє побачити значення змінних і порівняти їх з очікуваними значеннями. Такий спосіб налагодження часто використовується розробниками, оскільки дуже просто помістити ці оператори або оформити їх у вигляді окремого дамп-методу. В Qt прикладом такого підходу є метод QObject :: dumpObjectInfo (), який виводить на екран метаінформацію об'єкта.

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

В заголовному файлі QtGlobal містяться визначення двох макросів `Q_ASSERT()` і `Q_CHECK_PTR()`:

- `Q_ASSERT()` приймає в якості аргументу значення булевого типу і виводить попередження, якщо це значення не дорівнює `true`;
- `Q_CHECK_PTR()` приймає покажчик і виводить попередження, якщо переданий покажчик дорівнює 0, а це означає, що або покажчик ні ініціалізований, або операція по виділенню пам'яті пройшла невдало.

Qt надає глобальні функції `qDebug()`, `qWarning()` и `qFatal()`, Які також визначені в заголовному файлі QtGlobal. Їх застосування схоже на функцію `printf()`. Як і в `printf()`,в ці функції передаються Форматована рядок і різні параметри. У Microsoft Visual Studio висновок цих функцій виконується в вікно відладчика, а в ОС Linux - в стандартний потік виведення помилок.

> ***Примітка***
>
> Виклик функції `qFatal()` після виведення повідомлення відразу завершує роботу всієї програми.

Якщо буде потрібно перенаправити потік виведення повідомлення, потрібно створити і встановити свою власну функцію для управління виводу. Встановлюється вона за допомогою функції `qInstallMsgHandler()`.Цій функції в якості аргументу передається адреса на функцію, управляючу повідомленнями і має наступний прототип:

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

На місці `fct` повинно стояти ім'я функції. Перший аргумент являє собою тип повідомлення, що приймає одне із значень перерахування `QtMsgType`: `QtDebugMsg`, `QtWarningMsg` чи `QtFatalMsg`. Другий аргумент - це вказівник на саме повідомлення.

Для полегшення процесу налагодження рекомендується привласнювати всім 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'єкта.

Також для налагодження можна скористатися установкою фільтра подій для об'єкта класу `QCoreApplication`, в в цьому випадку даний фільтр буде найпершим об'єктом, якийй отримує і обробляє події всіх об'єктів додатки (див. *главу 15)*.

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

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

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

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

Але, як ви бачите, попередній рядок виглядає більш компактно, тому рекомендую користуватися саме нею.

Важливо розуміти, що висновок інформації за допомогою функції `qDebug()`відбувається при налагоджувальних і релізних схем. Якщо висновок інформації повинен бути присутнім тільки в налагоджувальній версії програми, а в релізній версії він повинен бути відсутнім, то можна реалізувати макрос на подобі цього:

```
#if defined(QT_DEBUG)
    #define QDEBUG(X)
qDebug() << X; #else
    #define
QDEBUG(X) ;
#endif
```

І використати замість функції `qDebug()` наступний запис:

```
QDEBUG("Test1" << 123 << "Test2" << 456);
```

Тепер, роблячи реліз вашої програми, ви можете бути абсолютно впевнені в тому, що всі висновки інформації, призначені для налагодження, що проходять через цей макрос, ніхто не побачить.

Глобальні визначення Qt

Qt містить в заголовному 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 передане 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 максимум.

А ось ще одна цікава функція. Справа в тому, що порівняння на точне рівність двох значень з плаваючою точкою є однією з частих помилок в програмуванні. функція `qFuzzyCompare()` бере всю відповідальність за правильне порівняння на себе. Вона приймає два значення типу `double` чи `float` і повертає логічне значення `true`, якщо змінні вважаються рівними, в іншому випадку 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 представляє складність для цієї функції, - це нульове значення. Але є рішення для цього завдання. Потрібно 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, описаних в *главі 46*.

В табл. 3.3 наведено список типів Qt, які можна використовувати 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 біта
quint32	unsigned int	32 біта
qint64	__int64 или long long	64 біта
quint64	unsigned_int64 или unsigned long long	64 біта
qlonglong	Те ж саме, що і qint64	64 біта
qulonglong	Те ж саме, що і quint64	64 біта

Як видно з табл. 3.3, найсуперечливішими є типи qint64 и quint64, давайте перевіримо правильність зазначених для них у таблиці значень біт, а заодно 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 знати інформацію про самій бібліотеці, яка знаходиться у використанні на вашому комп'ютері зараз. Наприклад, вам захотілося дізнатися в якому каталозі Qt зберігає свої файли розширень (plug-ins) або вам необхідно дізнатися поточну версію Qt і т. д. За це відповідає клас QLibraryInfo і надає для цього цілий ряд статичних методів. Продемонструємо їх застосування на невеликому прикладі (лістинг 3.2).

Лістинг 3.2. Використання статичних функцій класу `QLibraryInfo`

```
#include <QtCore>
```

```
int main(int argc, char** argv)
{
    qDebug() << "Build date:"<< QLibraryInfo::buildDate().toString("yyyy-MM-
    dd"); qDebug() << "Build key:"<< QLibraryInfo::buildKey(); qDebug() <<
    "License Products:"<< QLibraryInfo::licensedProducts(); qDebug() <<
    "Licensee:"<< QLibraryInfo::licensee();

    qDebug() << "Locations"; qDebug() << "  Headers:" <<
    QLibraryInfo::location(QLibraryInfo::HeadersPath); qDebug() << "
    Libraries:"<< QLibraryInfo::location(QLibraryInfo::LibrariesPath);
    qDebug() << "  Binaries:"<<
    QLibraryInfo::location(QLibraryInfo::BinariesPath);

    qDebug() << "  Prefix"<< QLibraryInfo::location(QLibraryInfo::PrefixPath);
    qDebug() << " Documentation: "<<
    QLibraryInfo::location(QLibraryInfo::DocumentationPath); qDebug() << "
    Plugins:"<< QLibraryInfo::location(QLibraryInfo::PluginsPath); qDebug() <<
    "  Data:"<< QLibraryInfo::location(QLibraryInfo::DataPath); qDebug() << "
    Settings:"<< QLibraryInfo::location(QLibraryInfo::SettingsPath); qDebug()
    << "  Demos:"<< QLibraryInfo::location(QLibraryInfo::DemosPath); qDebug()
    << "  Examples:"<<
    QLibraryInfo::location(QLibraryInfo::ExamplesPath);
}
```

Ось вивід цієї програми для версії Qt4.7.1, встановленої на комп'ютер з операційною системою Windows.

```
Build date: "2010-11-22"
Build key: "Windows mingw debug
full-config" License Products:
"OpenSource"
Licensee:
"Open Source"
Locations
  Headers:
  "C:/Qt/4.7.1/include"
  Libraries:
  "C:/Qt/4.7.1/lib"
  Binaries:
  "C:/Qt/4.7.1/bin" Prefix
  "C:\Qt\4.7.1"
  Documentation:
  "C:/Qt/4.7.1/doc"
  Plugins:
  "C:/Qt/4.7.1/plugins"
  Data: "C:/Qt/4.7.1"
  Settings:
  "C:/Qt/4.7.1"
  Demos:
  "C:/Qt/4.7.1/demos"
  Examples: "C:/Qt/4.7.1/examples"
```

Консольний проект Qt. Вивід повідомлень.

Незважаючи на те, що Qt майже завжди розглядають як інструментарій для створення програм з графічним інтерфейсом, його також можна використовувати і в таких програмах, які працюють як фонові процеси, а також в консольних проектах. Для кількох наступних прикладів ми будемо використовувати останній створений нами в попередньому розділі проект.

В такому консольному проекті можна використовувати майже всі звичні для Qt кошти і класи. У наступних декількох прикладах ми розглянемо роботу з деякими важливими типами Qt саме на прикладі консольного проекту. А поки що обмежимося тільки оглядом кошти, яке дозволяє виводити в консоль повідомлення і різноманітну інформацію для налагодження в процесі роботи програми.

Для виведення інформації в консольному проекті можна використовувати всі звичні засоби стандартної бібліотеки C++.Але в Qt для цього є зручний інструмент - функція qDebug().Розглянемо приклад її використання:

```
#include <QDebug>
// Власний тип даних – структура для комплексних чисел
struct complex
{
  double re;
  double im;
};
// Визначення потокового оператора для підтримки виведення власного типу
// complex за допомогою qDebug ()
QDebug operator << (QDebug dbg, const complex & c)
{
dbg.nospace () << "(" << c.re << "+ i *" << c.im << ")";
  return dbg.space ();
}
Int main (int lArg c, char * lArgv [])
{
  // вивід різноманітних типів даних
  qDebug () << "Hello," << "this is debug output";
  qDebug () << "Integer values:" << 1 << 10 << 100;
  qDebug () << "Doubles and floats:" <<. 1 << .123 << 0.112345;
  qDebug () << "Characters:" << "c" << "\ t" << "$" << "\ n" << "newline";
  qDebug () << "Booleans:" << true << false;
  qDebug () << "Pointers:" << lArgv;
  qDebug () << "and much more ...";
  // вивід власного типу даних
  complex c;
  c.re = 0.2;
  c.im = 1.5;
  qDebug () << "including custom types:" << c;
  return 0;
}
```

Після виконання програми в консолі побачимо текст:

```
Hello, this is debug output
Integer values 1 10 100
Doubles and floats: 0.1 0.123 0.112345
Characters: c $
newline
Booleans: true false
Pointers: 0x3278fc8
and much more ...
including custom types: (0.2 + i * 1.5)
```

Крім qDebug () існують інші функції для виведення повідомлень різного рівня. Опис і приклади цих функцій розглянемо в табл.12.2.

Таблиця 12.2. Функції для виведення повідомлень

Функція	Опис	Особливості	Приклад

qDebug ()	Виводить повідомлень для налагодження, різноманітної інформації при роботі програми.	Повідомлення можуть бути вимкнені за допомогою спеціального макроозначення QT_NO_DEBUG_OUTPUT наприклад, у файлі проекту:
		`DEFINES + =`
		`QT_NO_DEBUG_OUTPUT`
			`int error_num = 59;`
			`std :: string error_string ( "uknown error");`
			`qDebug ( "result:% d, description:% s ", error_num, error_string . c_str ());`
			`#include <QDebug>`
			`...`
			`qDebug () << "result:" "<< error_num << ", description: "<< error_string.c_str ();`

qWarning ()	Виводить повідомлень при роботі програми.	Повідомлення можуть бути вимкнені за допомогою спеціального макроозначення QT_NO_WARNING_OUTPUT наприклад, у файлі проекту:
		`DEFINES + =`
		`QT_NO_WARNING_OUTPUT`
			`qWarning ( "warning:% d, description:% s ", error_num, error_string .c_str ());`
			`#include <QDebug>`
			`...`
			`qWarning () << "warning:" << error_num << ", description: "<< error_string.c_str ();`

qCritical ()	Виводить повідомлень про критичні помилки.		qCritical ("critical error :% D, description:% s ", error_num, error_string .c_str ()); _____ #include <QDebug> ... qCritical () << "critical error: "<< error_num << ", description:" <<error_string.c_str ();
qFatal ()	Виводить повідомлень про фатальні для програми помилках	Після виведення повідомлення відбувається аварійне завершення роботи програми	qFatal ("fatal error:% d, description:% s ", error_num, error_string .c_str ());

lec1 Вступ. Основні поняття.

1. **Qt – історія.**
2. **Метаоб'єктний компілятор MOC**
3. **Основні складові Qt 5.**
4. **Механізм сигналів і слотів**
5. **Реурси**

Qt – історія.

Багатоплатформовий інструментарій розробки Qt з'явився вперше в 1995 році завдяки своїм розробникам Хаарварду Норду і Айріка Чеймб- Інгу. З самого початку створювався як програмний каркас, що дозволяє створювати Кросплатформені програми з графічним інтерфейсом. Перша версія Qt вийшла 24 вересня 1995. Програми, розроблені з Qt, працювали як під управлінням операційних систем сімейства Microsoft Windows ™ так і під управлінням Unix-подібних систем.

За роки розробки можливості Qt значно зросли. Робота з мережею, базами даних, графікою, мультимедіа, Інтернет і інші розширення перетворили його в універсальний інструментарій для створення програм. Qt перетворився в повноцінний і потужний інструмент розробки, який значно перевершив свої початкові можливості.

У червні 1999 року вийшла друга версія - Qt 2.0. А у 2000 році відбувся випуск версії для вбудованих систем, який називався Qt Embedded. Версія Qt 3.0 - 2001 рік - працювала в ОС сімейства Windows ™ і багатьох Unix-подібних ОС, таких як MacOS, xBSD, в різних варіантах Linux для персональних комп'ютерів і вбудованих систем. Він мав 42 додаткових класи, обсяг виріс до більш ніж 500 000 рядків коду. Влітку 2005 року відбувся випуск Qt 4.0, який включав в сукупності близько 500 класів і мав величезну кількість істотних поліпшень. Разом з випуском Qt 4.5 вийшло і спеціалізована інтегроване середовище розробки QtCreator.

У грудні 2012 року відбувся офіційний випуск Qt5. Ця версія кроссплатформенного додатку розробки сумісна з Qt4. Перенесення коду з Qt4 на Qt5 не вимагає багато зусиль. У той же час, Qt5 відрізняється рядом особливостей, поліпшень і великою кількістю нових можливостей.

Сучасне програмне забезпечення досить складне і має відповідати багатьом вимогам. Крім користувальницьких вимог, що накладаються на зручність і можливості програмного продукту, є й інші вимоги, що стосуються розробки програмного забезпечення. Велику роль тут відіграють засоби, якими програміст користується в процесі своєї роботи. У багатьох випадках буває зручно володіти інструментарієм, який має досить широку область застосування і може служити для вирішення великої кількості завдань різного масштабу: від побудови невеликих програм для створення потужних програмних пакетів. Також часто виникає питання про підтримку декількох програмних платформ, адже, орієнтуючись тільки на одну платформу, можна втратити велику кількість потенційних користувачів.

Інструментарій розробки Qt використовують для створення кроссплатформенних програм. Тут під цим твердженням ми маємо на увазі програми, вихідний текст яких можна скомпілювати на різних програмних платформах (різні різновиди Linux, Windows, MacOS і т.д.) практично без змін або з незначними змінами. Крім того Qt використовують і для розробки програм, що мають характерний ("рідний", native) для програмного оточення або навіть власний стилізований інтерфейс. Все це завдяки відкритості вільного програмного коду, зручному і логічного API і широким можливостям застосування.

Qt розширює можливості програміста за допомогою набору макросів, метаінформації і сигнально-слотових з'єднань, але використовує при цьому лише засоби мови C ++ і є сумісним з усіма поширеними сучасними його компіляторами.

Поряд з традиційним для попередніх версій Qt способом створення користувацьких інтерфейсів, заснований на віджетах - візуальних елементах інтерфейсу (кнопки, прапорці, списки, що випадають, поля введення, слайдери і т.д.), Qt5 ставить великий акцент на використанні технології QtQuick. У Qt5 деякі нововведення торкнулися і синтаксису для створення сигнально-слотових з'єднань.

Програмний код, що залежить від віконної системи в Qt5, був відділений і реорганізований в окремі бібліотеки розширення, що дозволило спростити перенесення Qt на нові платформи і

адаптації для підтримки інших віконних систем. Завдяки QPA (Qt Platform Abstraction) в Qt5 реалізована підтримка багатьох платформ для мобільних пристроїв.

Підтримка безлічі мов програмування, таких як:

1. C++
2. PySide
3. Python– Py Qt
4. Ruby – Qt Ruby
5. Java – Qt Jambi
6. PHP –PHP- Qt

Добре продуманий, логічний і стрункий набір класів, що надає програмісту дуже високий рівень абстракції. Завдяки цьому програмістам, які використовують Qt, доводиться писати значно менше коду. Сам же код виглядає простіше, логічніше і зрозуміліше, ніж аналогічний за функціональністю код MFC. Його легше підтримувати і розвивати.

Метаоб'єктний компілятор MOC

Проблема розширення мови C ++ вирішена в Qt за допомогою спеціального препроцесора **MOC (Meta Object Compiler, метаоб'єктний компілятор)**. Він аналізує класи на наявність в їх визначенні спеціального макросу Q_OBJECT і впроваджує в окремий файл всю необхідну додаткову інформацію. Це відбувається автоматично, без безпосередньої участі розробника. Подібна операція автоматичного створення коду який суперечить звичному процесу програмування на C ++ - адже стандартний препроцесор перед компіляцією самої програми теж створює проміжний код, що містить виконані команди препроцесора. Подібним чином діє і MOC, записуючи всю необхідну додаткову інформацію в окремий файл, вміст якого не вимагає уваги розробника. Макрос Q_OBJECT повинен розташовуватися відразу 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 в визначення класу має сенс в тих випадках, коли створений клас використовує механізм сигналів та слотів або якщо йому необхідна інформація про властивості.

Метаоб'єктний компілятор (MOC, Meta Object Compiler), по суті справи, є не компілятором, а препроцесором, який виконується в ході компіляції додатка, створюючи, відповідно до визначення класу, додатковий код на мові C ++. Це відбувається через те, що визначення сигналів і слотів в вихідному коді програми недостатньо для компіляції. Сигнально-слотовий код повинен бути перетворений в код, зрозумілий для компілятора C ++. Код зберігається в файлі з прототипом імені: MOC_ <filename> .cpp.
Якщо ви працюєте з файлами проекту, то про існування MOC можете і не здогадуватися, адже в цьому випадку управління MOC автоматизовано. 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'єкта - наприклад, сигнально-слотові з'єднання, ім'я класу і структуру успадкування.

Основні складові Qt 5.

Розглянемо основні складові кроссплатформенного засобу розробки Qt: модулі та інструменти.

На рис. 1 зображені основні складові Qt. Модулі та інструменти доступні для розробки під цільові (Reference) та інші (Other) платформи. Засоби Qt розділені за призначенням на окремі частини - модулі. Кожен з модулів виконаний у вигляді окремої бібліотеки. Розробник має можливість вибрати модулі, які він використовує в програмі. Модулі мають взаємозалежності: одні модулі використовують можливості, які надають інші. Основу складають основні (Essentials) модулі:

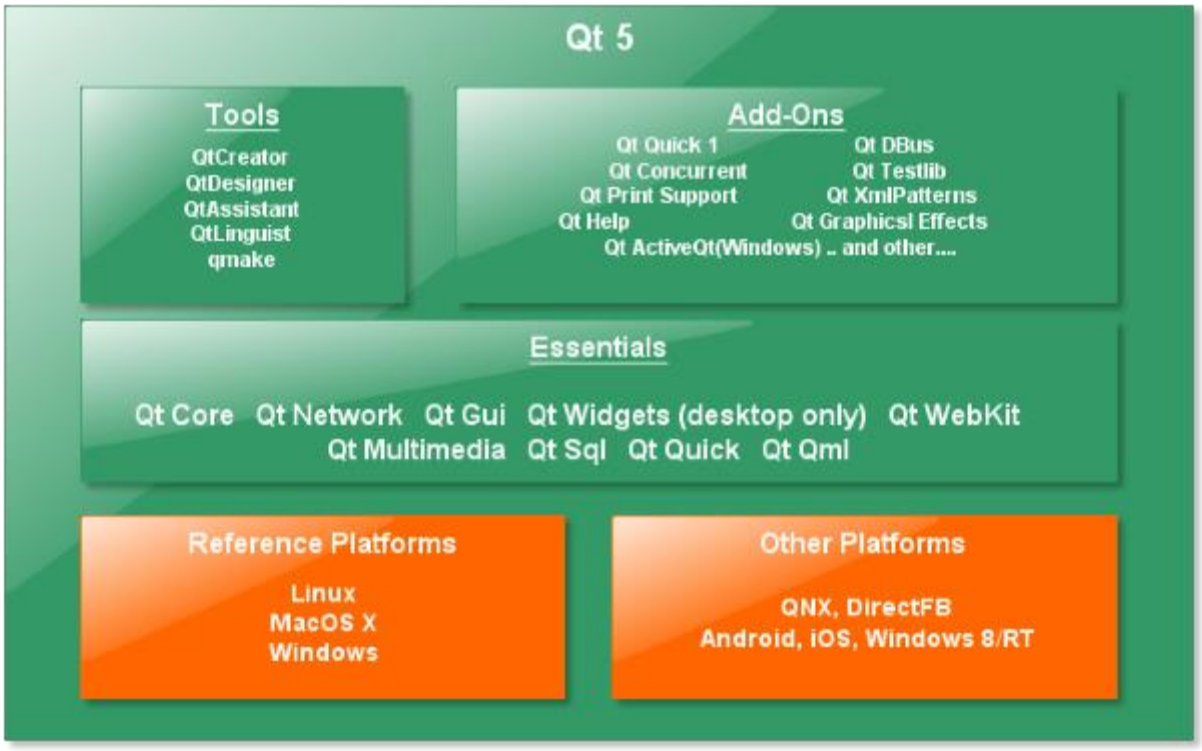


Рис. 1 Основні складові Qt5

Бібліотека розділена на декілька модулів, для п’ятої версії це:

1. **QtCore** — класи ядра бібліотеки використовувані іншими модулями;
2. **QtGui** — компоненти графічного інтерфейсу;
3. **QtNetwork** — набір класів для мережевого програмування. Підтримка різних високорівневих протоколів може мінятися від версії до версії.
4. **QtOpenGL** — набір класів для роботи з OpenGL;
5. **QtSql** — набір класів для роботи з базами даних мовою структурованих запитів SQL.
6. **QtScript** — класи для роботи з Qt Scripts;
7. **QtSvg** — класи для відображення і роботи зі Scalable Vector Graphics (SVG) даними;
8. **QtXml** — модуль для роботи з XML, підтримується SAX і DOM моделі роботи;
9. **QtDesigner** — класи створення розширень QtDesigner'a для своїх власних віджетів;
10. **QtUiTools** — класи для обробки в застосунку форм Qt Designer;
11. **QtAssistant** — довідкова система;
12. **QtWebKit** — модуль WebKit інтегрований в Qt і доступний через її класи;
13. **QtMultimedia** — модуль для підтримки відтворення і запису відео і аудіо, як локально, так і з пристроїв і з мережі;
14. **QtCLucene** — модуль для підтримки повнотекстового пошуку, застосовується в новій версії Assistant в Qt 4.4;
15. **QtActiveQt** — модуль для роботи з ActiveX і COM технологіями для Qt-розробників під Windows. Модуль доступний тільки в комерційній редакції Qt.
16. **QtGraphical** Effects — забезпечує набір типів QML для додавання візуально вражаючих і настроюваних ефектів для користувацьких інтерфейсів.
17. **QtQuick** — стандартна бібліотека для написання додатків QML.

Існує також багато додаткових (Add-On) модулів. Варто зауважити, що поділ на основні та додаткові модулі характерно Qt5 на відміну від попередніх версій. Назви деяких модулів в Qt5 в порівнянні з Qt4 були змінені, а деякі засоби були винесені в окремі або перенесені в інші модулі. Ці зміни необхідно враховувати при перенесенні програм, які були розроблені з використанням Qt4. Майже всі приклади, які ми будемо розглядати, працюють як з Qt4 так і Qt5. У випадках, коли це суттєво, ми будемо вказувати на відмінності.

Крім модулів, до складу інструментарію входять інструменти розробки, вихідні тексти Qt, приклади програм і документація.

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

class MyClass : public QObject, public AnotherClass {

Клас QObject містить в собі підтримку:

- сигнали і слоти (signal / slot);
- таймера;
- механізму об'єднання об'єктів в ієрархії;
- подій і механізми їх фільтрації;
- організації об'єктних ієрархій;
- метаоб'єктну інформацію;
- перетворення типів
- властивості

Сигнали і слоти - це засоби, що дозволяють ефективно проводити обмін інформаціею про події, що виробляються об'єктами.

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

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

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

Метаоб'єктна інформація включає в себе інформацію про успадкування класів, що дозволяє визначати, чи є класи безпосередніми спадкоємцями, а також дізнатися ім'я класу.

Для **перетвореня типів** Qt надає шаблонну функцію qobject_cast<T>(), основана на метаінформації, створюваної метаоб'єктним компілятором MOC для класів, успадкованих від QObject.

Властивості - це поля, для яких обов'язково повинні існувати методи читання. З їх допомогою можна отримувати доступ до атрибутів об'єктів ззовні - наприклад, з мови сцена¬ріев Qt Script. Властивості також широко задіяні в візуальному середовищі розробки призначеного для користувача інтерфейсу Qt Designer.

Механізм сигналів і слотів

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

Стара концепція функцій зворотного виклику (callback functions), що лежить в основі X Window System, заснована на використанні звичайних функцій, які повинні викликатися в результаті дій користувача. Застосування такої концепції значно ускладнює вихідний код програми, роблячи його менш зрозумілим. Крім того, тут відсутня можливість проводити перевірку типів значень, що повертаються, тому що у всіх випадках повертається покажчик на порожній тип void. Наприклад, для того щоб зіставити код з кнопкою, необхідно передати в функцію покажчик на кнопку. Якщо користувач натискає на кнопку, функція буде викликатися. Самі бібліотеки не перевіряють, чи були аргументи, передані в 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, в 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 у них напевно з'являться труднощі.

Існують, втім, спеціальні бібліотеки класів мови C ++, що полегшують програмування для ОС Windows. Однією з найперших таких бібліотек (і до сих пір на подив знаходяться в застосуванні у цілого ряду індивідуальних розробників і компаній) є Microsoft Foundation Classes (MFC). Назвати її об'єктно-орієнтованої можна лише з великою натяжкою, оскільки вона створювалася людьми, які не підозрювали про існування найелементарніших принципів об'єктно-орієнтованого підходу. Одна з головних фундаментальних заповідей 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 така вимога не дотримано . Сама бібліотека MFC є надбудовою, що надає доступ до функцій Windows, реалізованим на мові C, що змушує розробників час від часу використовувати застарілі структури, які не вписуються в рамки концепції об'єктно-орієнтованого підходу. Цікаво також відзначити, 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 задіюються спеціальні макроси - так звані карти повідомлень (лістинг 1). Вони дуже сильно захаращують вихідний код програми, помітно знижуючи її читаність.

Лістинг 1. Фрагмент гтрограмми, 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 показаної в лістингу 2.3, дуже незручні для людського сприйняття і бентежать при проведенні аналізу коду програми. Нехай багато розповідають про зручність використання засобів для автоматичної генерації подібного коду, але створені вони були не від хорошого життя. Так, непродуманість самої бібліотеки змушує розробника при внесенні незначних змін модифікувати код самої програми відразу в декількох місцях. Наприклад, для того щоб додати в діалогове вікно текстове поле, необхідно провести цілий ряд операцій. По-перше, потрібно створити в класі діалогу атрибут, призначений для зберігання значень, що вводяться в текстовому полі. По-друге, треба задати ідентифікатор ресурсу текстового поля. По-третє, поставити ідентифікатор ресурсу і атрибут в методі DoDataExchange () у відповідність один з одним за допомогою методу DDX_Text (), після чого може здійснюватися обмін даними між текстовим полем і атрибутом. По-четверте, цим обміном необхідно управляти, передаючи в методі UpdateData () значення булевого типу true або false. І лише за допомогою засобів автоматичного створення коду можна частково позбутися від такої проблеми, змусивши виконати ці зміни за вас і отримавши натомість інші недоліки, - наприклад, додаткове засмічення коду програми непотрібною інформацією і можлива розбіжність створеного коду до затверджених для проекту вимогами щодо форматування і нотації (якщо не використовується угорська нотація

У цій ситуації частина провини прихована в самій мові C ++. Справа в тому, що C ++ не створювалася як засіб для написання користувальницького інтерфейсу, і тому він не надає належної підтримки, що б зробило програмування в цій області більш зручним. Наприклад, якби робота по передачі подій реалізовувалася засобами самої мови, то відпадала б необхідність у використанні подібного роду макросів. До теперішнього часу не вдавалося зробити нічого подібного, саме тому бібліотека Qt з'явилася «як грім серед ясного неба». Тому що, на відміну від більшості інших бібліотек програмування, Qt розширює мову C ++ додатковими ключовими словами для виконання цього завдання.

Механізм сигналів і слотів повністю заміщає стару модель функцій зворотного виклику, він дуже гнучкий і повністю об'єктно-орієнтований. Сигнали і слоти- це наріжний концепт 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. Мовою сигналів і слотів подібну ситуацію можна описати таким чином: об'єкт «телефон» вислав сигнал «дзвінок», на який об'єкт «людина» відреагував слотом «зняття трубки».

Використання механізму сигналів і слотів дає програмісту наступні переваги:

- кожен 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 з'єднувати з різною кількістю слотів. Відправлений сигнал надійде до всіх приєднаних слотів;
- слот може приймати повідомлення від багатьох сигналів, що належать різним об'єктам;
- з'єднання сигналів і слотів можна здійснювати в будь-якій точці додатка;
- сигнали і слоти є механізмами, що забезпечують зв'язок між об'єктами. Більш того, цей зв'язок може виконуватися між об'єктами, які знаходяться в різних потоках;
- при знищенні об'єкта відбувається автоматичне роз'єднання всіх сигнально слотових зв'язків. Це гарантує, що сигнали не будуть відправлятися до неіснуючих об'єктів.

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

- сигнали і слоти не є частиною мови C ++, тому перед компіляцією програми потрібно запуск додаткового препроцесора;
- відправка сигналів відбувається трохи повільніше, ніж звичайний виклик функції, який здійснюється при використанні механізму функцій зворотного виклику;
- існує необхідність у спадкуванні класу QObject;
- в процесі компіляції не проводиться жодних перевірок: чи є сигнал або слот у відповідних класах чи ні, чи сумісні сигнал і слот один з одним і чи можуть вони бути пов'язані один з одним. Про помилку стане відомо лише тоді, коли додаток буде запущено в відлатчику або на консолі. Вся ця інформація виводиться на консоль, тому, щоб побачити її в Windows, в проектному файлі необхідно в секції coNFiG додати опцію console (для Mac OS X і Linux ніяких додаткових змін проектного файлу не потрібно).

> ***АЛЬТЕРНАТИВНА ФОРМА сигнально-слотових з'єднань***
> *Використання альтернативної форми сигнально-слотових з'єднань усуває цей недолік, дозволяючи виявляти помилки з'єднань сигналів зі слотами на етапі компіляції програми.*

Серед відомих проектів особливо треба відзначити:

- програма для IP-телефонії Skype;
- програма для обробки зображень Adobe Photoshop Album;
- мережева карта світу Google Earth.

Встановлення (станом на серпень 2019**).**

Завантажуємо **qt-opensource-windows-x86-5.13.0-rc3**

Важливо при встановленні не поміщати в папки з кирилицею. В цьому релізі буде вже встановлений і qt-creator 4.9

Ресурси.

https://www.qt.io/
https://doc.qt.io/
Qt5 C++ GUI Programming Cookbook
Шлее М. - Qt 5.10. Профессиональное программирование на C++

Кнопки, прапорці та перемикачі

З чого починаються кнопки. Клас *QAbstractButton*

Клас `QAbstractButton` — базовий для всіх кнопок. У додатках застосовується три основних види кнопок: які натискаються (`QPushButton`), які зазвичай називають просто кнопками, прапорці (`QCheckBox`) і перемикачі (`QRadioButton`). В класі `QAbstractButton` реалізовані методи і можливості, властиві всім кнопкам. Спочатку ми обговоримо основні з цих можливостей, а потім поговоримо про кожен вид окремо.

Установка тексту і зображення

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

Растрове зображення встановлюється на кнопці за допомогою методу `setIcon()`. Після установки 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 зображене, потрібно викликати метод `icon()`.

Взаємодія з користувачем

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

- `clicked()` — відправляється при клацанні 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, так як реагує на дії користувача, якщо `isEnabled()` повертає значення `true`. Змінити поточний стан можна викликом методу `setEnabled()`.

Кнопки

Віджет кнопки можна зустріти в будь-якому додатку, наприклад, майже завжди є кнопки **Ok** або **Cancel** (Відміна) — без них не обходиться жодне діалогове вікно. Іноді такий віджет називають "командною кнопкою". Він являє собою прямокутний елемент управління і використовується, як правило, для виконання певної операції при натисканні на неї. Клас `QPushButton` віджета натискати кнопки визначено в файлі заголовку QPushButton.

Створити кнопку яка натискаються можна наступним чином:

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

Перший параметр (типу рядок) задає напис кнопки; як зазвичай, в конструктор можна передавати також віджет предка, але це робити не варто, так як менеджер зробить це за нас.

Як ми вже знаємо, при натисканні на кнопці відправляється сигнал `clicked()`. Кнопка вважається натиснутою, якщо користувач клацнув на неї кнопкою миші або натиснув на клавішу <Enter>, за умови, що кнопка була обрана (була в фокусі). Для того щоб кнопку зробити такою, можна скористатися методом `setDefault()`.

Приклад, показаний на мал. 8.1, демонструє різні варіанти натискати кнопок:

- **Normal Button** (Звичайна кнопка) відповідає звичайній кнопці, яку ми звикли бачити в більшості випадків. Після відпускання кнопка завжди повертається в своє вихідне положення;
- **Toggle Button** (Вимикач) може перебувати в двох станах: натиснутому або не натиснутому, які відповідають положенням "включено" або "вимкнено". Логіка дії цієї кнопки ідентична, наприклад, логіці звичайного кімнатного вимикача;
- **Flat Button** (Плоска кнопка) за своїми функціональними особливостями ідентична звичайній 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 з зображенням) являє собою кнопку, що містить растрове зображення.

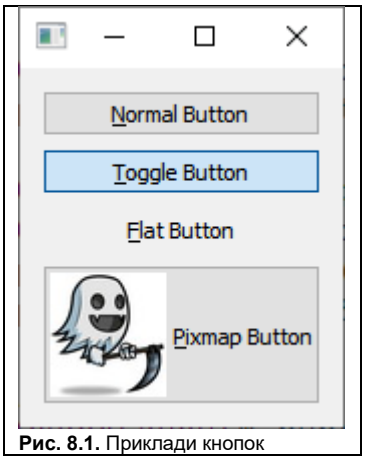


Рис. 8.1. Приклади кнопок

Лістинг 8.1. Файл main.cpp

```
#include <QApplication>
#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->setCheckable(true);
pcmdToggle->setChecked(true);

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

QPixmap pix("1.jpg");
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();
}
```

У лістингу 8.1 наводиться текст програми, вікно якого показано на мал. 8.1. Спочатку створюється віджет звичайної кнопки (об'єкт pcmdNormal). Кнопка-вимикач (об'єкт `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.

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

Для створення кнопки з растровим зображенням спочатку створюється об'єкт растрового зображення `pix`, в який завантажується файл 1.jpg, вказаний в конструкторі. Потім створюється кнопка (об'єкт `pcmdPix`), після чого об'єкт растрового зображення передається в метод `setIcon()` для установки на кнопці. Слот `setIconSize()` задає розміри растрового зображення, в даному випадку вони відповідають оригінальним розмірами зображення, які повертає метод `size()` об'єкту `QPixmap`.

CheckBox

Більшість програм надають цілий ряд налаштувань, що дозволяють змінювати поведінку програми. Для цих цілей може бути корисним віджет прапорця, який дозволяє користувачеві вибирати відразу кілька опцій. Клас `QCheckBox` віджета кнопки прапорця визначено в заголовному файлі QCheckBox.

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

Прапорець складається з маленького прямокутника і може містити пояснювальний текст або картинку. При натисканні на віджет в прямокутнику з'явиться відмітка. Цього ж можна домогтися натисканням клавіші <Пробіл>, коли віджет знаходиться у фокусі. Цей віджет установлюють 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 (мал. 8.3). Приклад використання такого стану можна побачити в діалоговому вікні **Properties** (Властивості) Провідника в ОС Windows при виборі декількох файлів, що мають різні атрибути.

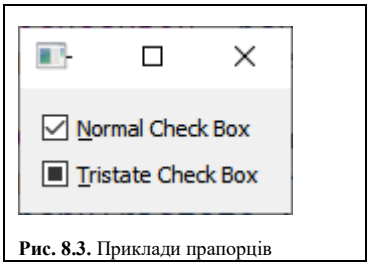


Рис. 8.3. Приклади прапорців

Лістинг 8.3. Файл main.cpp

```
#include <QtGui>

#include <QApplication>
#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* pchkTristate = 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();
}
```

У лістингу 8.3 створюються два прапорці - вказівники `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.

Перемикачі

Свою англійську назва — radio button — віджет перемикача отримав завдяки своїй схожості з кнопками радіоприймача, на панелі якого може бути натиснута тільки одна з кнопок. Натискання на іншу кнопку радіоприймача призводить до того, що в групі автоматично відключається кнопка, натиснута до цього.

Перемикач являє собою віджет (мал. 8.4), який може знаходитися в одному з двох станів: включено (on) або вимкнено (off). Ці стани користувач може встановлювати за допомогою миші або клавіші <Пробіл>, коли кнопка перебуває у фокусі. Клас `QRadioButton` віджета перемикача визначено в заголовки QRadioButton.

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

Пояснюючі написи повинні бути визначені для кожного використовуваного в групі вимикачів, а також бажано задати і поєднання клавіш для швидкого доступу до кожного з перемикачів. Це досягається включенням в напис символу `&` перед потрібною 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 QComboBox (див. *главу 11*).

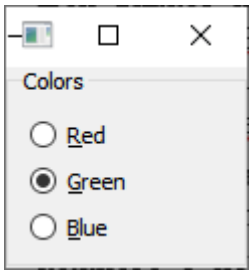


Рис. 8.4. Перемикачі

Лістинг 8.4. Файл main.cpp

```
#include <QApplication>
#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();
}
```

У лістингу 8.4 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. мал. 8.4).

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 групування і являє собою контейнер, що містить в собі різні елементи управління. Він може мати пояснювальний напис у верхній області, і цей напис може містити клавішу швидкого доступу, при натисканні на яку фокус перекладається на саму групу, а також можна забезпечити його додатковим прапорцем, який буде управляти доступністю згрупованих елементів (мал. 8.5). Цей клас визначено в заголовному файлі QGroupBox.

Додаток, показанbq на мал. 8.5, являє собою групу перемикачів, відповідаючих за колір фону. Наприклад, вибір перемикача **Red** (Червоний) призведе до того, що фон віджету верхнього рівня стане червоним. Прапорець **Light** (Яскравість) управляє яскравістю кольору, а кнопка **Exit** (Вихід) здійснює вихід з програми. Текст відповідного файлу main.cpp приведений в лістингу 8.5.

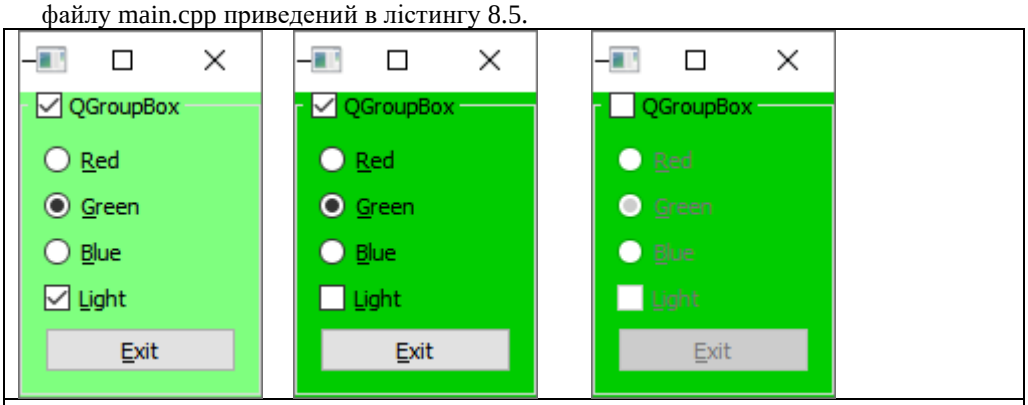


Рис. 8.5. Приклад угруповання кнопок

Лістинг 8.5. Файл main.cpp

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

Buttons buttons;
buttons.show();

return app.exec();
}
```

Лістинг 8.6. Файл widget.h

```
#ifndef WIDGET_H
```

```
#define WIDGET_H

//#include <QWidget>
#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();
};


#endif // WIDGET_H
```

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

Лістинг 8.7. Файл widget.cpp. Конструктор класу `Buttons`

```
Buttons::Buttons(QWidget* pwgt/*= 0*/) : QGroupBox("QGroupBox", pwgt)
{
resize(100, 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 конструкторі класу `Buttons` (лістинг 8.7) 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 `slotButtonClicked()`.

Прапорець **Light** (Яскравість) (курсор `m_pchk`) включається відразу після свого створення за допомогою методу `setChecked()`.

Останньою з кнопок створюється кнопка **Exit** (Вихід) (курсор `pcmd`). Для виходу з додатку при натисканні на цю кнопку сигнал `clicked()` з'єднується зі слотом `quit()` об'єкта додатка.

Потім створені віджети розміщуються у вертикальному порядку за допомогою об'єкта класу `QVBoxLayout`.

В останньому рядку викликається слот `slotButtonClicked()` для ініціалізації.

Лістинг 8.8. Файл widget.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);
}
```

У лістингу 8.8 наводиться реалізація методу `slotButtonClicked()`, в якому при кожному виклику створюється об'єкт палітри і перевіряється стан прапорця **Light** (Яскравість) для установки змінної `nLight`, керуючої яскравістю кольору. В операторах `if` виконується аналіз стану кнопок-перемикачів за допомогою методу `isChecked()`. Залежно від поміченої кнопки перемикача, колір фону палітри `QWidget::backgroundRole()` змінюється викликом методу `QPalette::setColor()` і встановлюється в віджеті за допомогою методу `QWidget::setPalette()`.

Резюме

Існує 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 використовуються в діалогових вікнах, що містять опції. Група прапорців використовується для одночасного вибору декількох опцій. Можливий варіант, коли не буде обраний жоден 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.

Елементи відображення

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).

Написи

Віджет напису служить для показу стану додатка або пояснюючого тексту і являє собою текстове поле, текст якого не підлягає зміні з боку користувача. Інформація, яка відображається цим віджетом, може змінюватися тільки самим додатком. Таким чином, програма може повідомити користувачеві про свої зміни стану, але користувач не може змінити цю інформацію в самому віджеті. Клас віджета напису QLabel визначено в файлі заголовку QLabel.

Віджет напису успадкований від класу QFrame і може мати рамку. Інформація що відображається може бути текстового, графічного або анімаційного характеру, для передачі її використовуються слоти setText (), setPimap () і setMovie ().
Розташуванням тексту можна керувати за допомогою методу setAlignnent (). Метод використовує велику кількість прапорів, деякі з них наведені в табл. 7.1. 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 A1ignHCenter.

Таблиця 7.1. Значення прапорів AlignnentF1ag простору імен Qt

Константа	Значення	Опис
A1ignLeft	0x0001	Розташування тексту зліва
A1ignRight	0x0002	Розташування тексту справа
A1ignHCenter	0x0004	Центрування тексту по горизонталі
A1ignJustify	0x0008	Розтягування тексту по всій ширині
A1ignTop	0x0010	Розташування тексту вгорі
A1ignBottom	0x0020	Розташування тексту внизу
A1ignVCenter	0x0040	Центрування тексту по вертикалі
A1ignCenter	A1ignVCenter | A1ignHCenter	Центрування тексту по вертикалі і горизонталі

Мал. 7.1. Відображення виджетом написи інформації в форматі HTML

Як видно з рис. 7.1, віджет напису може відображати не тільки звичайний текст, а й текстову інформацшо в форматі HTML (HyperText Markup Language, мова гіпертекстової розмітки). У цьому прикладі (лістинг 7.1) використовувався HTML, для виведення тексту, таблиці і растрового зображення.

Лістинг 7.1. Створення віджета написи з використанням формату HTML (файл main.cpp)

```
#include <QApplication>
#include <QtWidgets>
int main(int argc, char** argv)
{
QApplication app(argc, argv);
QLabel lbl("<H1><CENTER>QLabel - HTML Demo</CENTER></H1>"
" <H2 ><CENTER> Image< / CENTER></H2 > "
"<CENTER><IMG BORDER=\"1\" SRC=\"1.jpg\"> </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>"
" < / FONTX / CENTER> "
"<H2><CENTER>Table</CENTERX></H2>"
           "<CENTER> <TABLE>"
"       <TR BGCOLOR=#ff00ff>"
"       <TD>1,1</TDXTD>1,2</TDXTD>1, 3</TDXTD>1,      4</TD>"
"       </TR>"
"       <TR BGCOLOR=YELLOW>"
"       <TD>2,1</TDXTD>2,2</TDXTD>2,3</TDXTD>2,4</TD>"
```

```
"       </TR>"
"       <TR BGCOLOR=#OOfOOO>"
"       <TD>3,1</TDXTD>3, 2</TDXTD>3, 3</TDXTD>3,     4</TD>"
"       </TR>"
"</TABLE> </CENTER>"
);
lbl. show (); return app. exec ();
}
```

У лістингу 7.1 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 setText (). Другий параметр конструктора опущений, а так як по замовчуванню він дорівнює 0, то це робить його віджетом верхнього рівня.
Наступний приклад (лістинг 7.2), показаний на рис. 7.2, демонструє можливість відображення інформації графічного характеру в віджеті написи без використання формату HTML.

Мал. 7.2. Відображення виджетом написи графічної інформації

Лістинг 7.2. Створення віджета написи без використання формату HTML (файл main.cpp)

```
#include <QApplication>
# include <QtWidgets>
int main (int argc, char ** argv)
{
QApplication app(argc, argv);
QPixmap pix;
pix.load ( "1.jpg");
QLabel lbl;
lbl.resize (pix.size());
lbl.setPixmap (pix);
lbl.show ();
return app. exec ();
}
```

Як видно з лістингу 7.2 , спочатку створюється об'єкт растрового зображення QPimap. Після цього викликом методу load() в нього завантажується з ресурсу файл mira.jpg. Наступним кроком є створення самого віджета написи - об'єкта lbl класу QLabel. Потім викликом методу resize () його розміри приводяться у відповідність з розмірами растрового зображення. І, нарешті, виклик методу setPimap () встановлює в віджеті саме растрове зображення.

За допомогою методу setBuddy () віджет напису може асоційовуватись з будь-яким іншим віджетом. Якщо текст напису містить знак (амперсанд), то символ, перед яким він стоїть, буде підкресленим. При натисканні кнопки цього символу спільно з клавішею <Alt> фокус перейде до віджету, встановленому методом setBuddy ().

> ОСОБЛИВОСТІ ДЛЯ MAC 0S X
> За замовчуванням в Mac OS X управління фокусом за допомогою амперсанда неактивно, і для його активації необхідно викликати функцію qt_set_sequence_auto_mnemonic () зі значенням true.

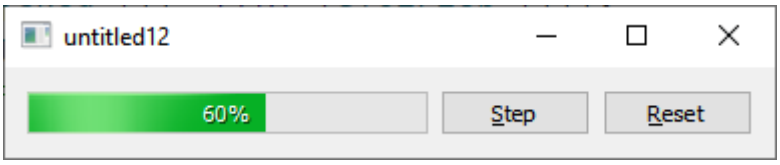
У всіх віджетах є можливість обробки подій клавіатури і миші. Цим можна скористатися, наприклад, для створення гіпертекстового посилання, яка при натисканні викличе певну HTML сторінку. Однак можна вчинити і простіше. Справа в тому, що клас QLabel надає підтримку для гіпертекстових посилань і при натисканні 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 веб-браузері (що буде досягатися неявним викликом статичного методу QDesktopServices:: openUrl (). Для цього потрібно просто викликати метод setOpenExternalLinks () з параметром true. Наприклад:

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

QVBoxLayout * pvbxLayout = new QVBoxLayout;
QLabel * plbl = new QLabel (
 "<H1><A href=\"https://comp-sc.pnu.edu.ua/ \"> comp-sc.if.ua </A></H1>");
plbl->setOpenExternalLinks(true);
 pvbxLayout-> addWidget (plbl);


wgt.setLayout (pvbxLayout);
wgt. show ();

return app. exec ();
}
```

Індикатор виконання

Індикатор виконання (progress bar) - це віджет, який демонструє хід процесу 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

QProgessBar віджета індикатора виконання визначено у класі QProgressBar. Зазвичай індикатори виконання розташовуються в горизонтальному положенні, але це можна змінити, передавши в слот setOrientation () значення Qt::Vertical, - після цього він буде розташований вертикально.
Наступний приклад демонструє використання індикатора виконання. При натисканні кнопки Step (Крок) виконується збільшення значення індикатора на один крок. Натискання кнопки Reset (Скинути) скидає значення індикатора. В основній програмі, наведеної в лістингу 7.4, створюється віджет, показаний на рис. 7.4.



Мал. 7.4. Індикатор виконання

Лістинг 7.4. Створення індикатора виконання (файл main.cpp)

```
#include <QApplication>
# include <QtWidgets>
# include "calculator.h"
int main (int argc, char ** argv)
{
QApplication app(argc, argv);
Progress progress;
progress.show ();


return app. exec ();
}
```

У лістингу 7.5 приведений файл Progress.h, який містить визначення класу Progress, успадкованого від QWidget. Клас містить два атрибути: покажчик на віджет індикатора виконання і ціле значення, що представляє номер кроку. У класі визначено два слота: slotStep () і slotReset (). Перший призначений для нарощування кроку на одиницю, а другий - для установки індикатора виконання в нульове положення.

Лістинг 7.5. Визначення класу Proqress (файл Progress.h)

```
#pragma once
# include <QWidget>

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

У конструкторі класу (лістинг 7.6) атрибуту m_nStep присвоюється значення 0. Після створення об'єкта індикатора m_pprb викликом методу setRange () задається кількість кроків, рівне 5, а метод setMinimumWidth () встановлює мінімальну довжину віджета індикації виконання, - в нашому випадку ми забороняємо йому мати довжину менше

двохсот пікселів. Виклик методу setAlignment () з параметром Qt :: AlignCenter переводить індикатор в режим відображення відсотків в центрі (див. Табл. 7.1). Потім створюються дві кнопки: Step (Крок) і Reset (Скинути), які з'єднуються зі слотами slotStep () і slotReset (). У слоті slotStep () значення атрибута m_nStep збільшується на 1 і передається в слот QProgressBat:: setValue () об'єкта індикатора виконання. Слот slotReset () встановлює значення атрибуту m_nStep рівним 0 і, викликавши слот QProgressBar:: 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.

Лістинг 7.6 Конструктор класу Progress (файл Progress.cpp)

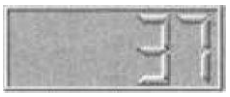
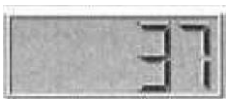
```
#include "calculator.h"
#include <QtWidgets>
Progress:: Progress (QWidget * pwgt /* = 0*/) :QWidget (pwgt) , m_nStep (0)
{
m_pprb = new QProgressBar;
m_pprb-> setRange (0, 5);
m_pprb->setMinimumWidth(200);
m_pprb-> setAlignment (Qt:: AlignCenter);
QPushButton * pcmdStep = new QPushButton ( "&Step");
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);
phbxLayout-> addWidget (pcmdStep);
phbxLayout-> addWidget (pcmdReset);
setLayout (phbxLayout);}
//=============================================

void Progress::slotStep ()
{
m_pprb-> setValue (++ m_nStep);}
//=============================================
void Progress:: slotReset ()
{
m_nStep = 0;
m_pprb->reset ();
}
```

Електронний індикатор

Клас QLCDNunber віджета електронного індикатора визначено в класі 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 точки, яку можна відображати між позиціями сегментів або як окремий ствол, викликаючи метод setSmallDecimalPoint() і передаючи в нього true або false відповідно. Кількість відображуваних сегментів можна задати в конструкторі або за допомогою методу setDigitCount (). У тому випадку, коли для відображення числа не вистачає сегментів індикатора, відсилається сигнал overflow ().

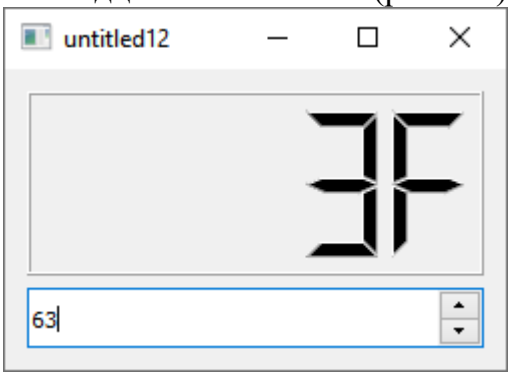
За замовчуванням стиль електронного індикатора відповідає стилю outline, але його можна змінити, передавши методу setSegmentSty1e () одне з наступних значень: QLCDNumber::Outline, QLCDNumber::Filled АБО QLCDNunber::Flat. У табл. 7.2 показаний зовнішній вигляд віджету для кожного з перерахованих стилів.

Таблиця 7.2. Стилі електронного індикатора

Константа	Зовнішній вигляд
0utline	
Flat	
Filled	

Електронний індикатор можна включати в режимі відображення двійкової, вісімкової, десяткової або шістнадцяткової систем числення. Режим відображення змінюється з 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 (лістинг 7.7) демонструє електронний індикатор, що відображає шістнадцяткові значення (рис. 7.5).



Мал. 7.5. Електронний індикатор

В лістингу 7.7 створюються віджети електронного індикатора (покажчик plcd) і лічильника (покажчик pspb). Потім викликом методу setSegmentStyle () змінюється стиль сегментних покажчиків. Електронний індикатор викликом методу setMode () з параметром QLCDNumber::Hex перемикається в режим шістнадцятирічного відображення. У елемента лічильника методом setFixedHeight () встановлюється незмінна висота, рівна 30. Після цього сигнал va1ueChanged () віджета лічильника з'єднується зі слотом display () електронного індикатора

Лістинг 7.7. Створення електронного індикатора (файл main.cpp)

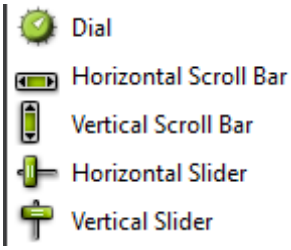
```
#include <QApplication>
# include <QtWidgets>
# include "calculator.h"
int main (int argc, char ** argv)
{
QApplication app(argc, argv);
QWidget  wgt;
QLCDNumber * plcd=new QLCDNumber;
QSpinBox * pspb = new QSpinBox;
plcd-> setSegmentStyle (QLCDNumber:: Filled);
plcd-> setMode (QLCDNumber::Hex);
```

```
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 ();
}
```

Елементи налаштування

Група віджетів, що відносяться до елементів налаштування, використовується, як правило, для установки значень, що не вимагають великої точності, наприклад - щоб відрегулювати гучність звуку, швидкості руху курсору (покажчика) миші, скролінг вмісту вікна та інші схожі дії.

Клас *QAbstractSlider*



Цей клас є базовим для всіх віджетів настройки: повзунка (`QSlider`), смуги прокрутки (`QScrollBar`) і установника (`QDial`). Всі перераховані далі можливості також доступні і у всіх успадкованих від нього класах. Його визначення зберігається в заголовному файлі в 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 `QAbstractSlider`, можуть бути як горизонтальними, так і вертикальними. Для зміни розташування використовується слот `setOrientation()`, в який для завдання горизонтального розташування передається значення `Qt::Horizontal`, а для вертикального — `Qt::Vertical`.

Установка діапазону

Для установки 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

```
psld->setMinimum(1);

psld->setMaximum(10);
```

Установка кроку

За допомогою методу `setSingleStep()` можна задати крок, значення, на яке, наприклад, повзунок зрушиться при натисканні на стрілки смуги прокрутки або на клавіші курсора 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 актуальне 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()` відправлятиметься тільки при відпусканні курсора 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 настройки деяких параметрів . Клас `QSlider` повзунок визначено в заголовному 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 місцезнаходження і показують крок. Можливі значення, які можна передати в метод `setTickPosition()`, наведені в табл. 9.1.

***Таблиця 9.1.** Значення перерахування `TickPosition` класу `QSlider`*

Константа	Опис	Вид
`NoTicks`	Повзунок без рисок	
`TicksAbove`	Відображення рисок на верхній стороні повзунка	
`TicksBelow`	Відображення рисок на нижній стороні повзунка	
`TicksBothSides`	Відображення рисок на верхній і нижній сторонах повзунка	

Метод `setTickInterval()` задає крок малювання рисок. Не слід задавати велику кількість рисок, так як це призведе до появи суцільної сірої лінії і не принесе ніякої користі.

Вікно додатка, зображене на мал. 9.1, містить віджети повзунка і написи, притому текст напису змінюється в залежності від положення повзунка.

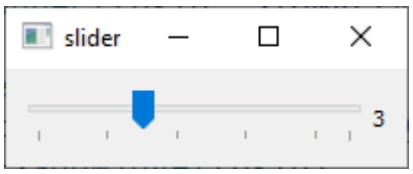


Рис. 9.1. Вікно додатка, яке демонструє роботу повзунка

Лістинг 9.1. Файл main.cpp

```
#include <QApplication>
#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.setWindowTitle("slider");
wgt.show ();

return app.exec();
```

}

У лістингу 9.1 створюються віджети повзунка (вказівник `psld`) і написи (вказівник `plbl`). Після цього викликом методу `setRange()` здійснюється установка діапазону значень повзунок 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 управління, які працюють разом з повзунком. З його допомогою можна зробити так, щоб величини віджетів збігалися один з одним, а також він може служити просто для завдання початкового значення при першому показі елемента. Таким чином, в метод повзунка `setValue()` передається значення 3, синхронізуючи його зі значенням, що відображається при створенні напису.

Крок для малювання рисок встановлюється рівний двом, для чого в метод повзунка `setTickInterval()` передається значення 2. Виклик методу `setTickmarks()` здійснює установку рисок знизу. Методом `connect()` сигнал повзунка `valueChanged(int)` з'єднується зі слотом напису `setNum(int)`.

На завершення виконується розміщення елементів на поверхні віджета `wgt` за допомогою горизонтальної компоновки.

Полоса прокрутки

Полоса прокрутки - це важлива складова практично будь-якого користувацького інтерфейсу. Вона інтуїтивно сприймається користувачем, і з її допомогою відображаються текстові або графічні дані, за розмірами перевищують відведену для них область. Використовуючи покажчик поточного становища полоси прокрутки, можна переміщати дані в видиму область. Він показує відносну позицію видимої частини об'єкта, завдяки якій можна отримати уявлення про розмір самих даних. Клас `QScrollBar` є реалізацією віджета полоси прокрутки. Він визначений в заголовному файлі QscrollBar і не містить ніяких додаткових методів і сигналів, розширюючих визначення класу `QAbstractSlider`.

Окремо полоси прокрутки використовуються дуже рідко. Вони вбудовані в віджет `QAbstractScrollArea`. Тому якщо ви маєте намір скористатися класом полоси прокрутки `QScrollBar`, то не виключено, що найкращим варіантом може виявитися використання одного з віджетів, які успадковують базовий клас для видової прокрутки `QAbstractScrollArea`.

У об'єктів, успадкованих від класу `QScrollBar`, можна викликати контекстне меню з параметром навігації за замовчуванням (мал. 9.2).

Віджет смуги прокрутки має мінімальне і максимальне значення, поточне значення і орієнтацію. Переміщення покажчика поточного положення здійснюється за допомогою лівої кнопки миші. В якості альтернативи можна просто натиснути на кнопки стрілок, розташованих на кінцях смуги прокрутки.

Вікно додатка, показане на мал. 9.3, складається з електронного індикатора і смуги прокрутки. Значення, що відображається електронним індикатором, змінюється в залежності від положення покажчика поточного положення.

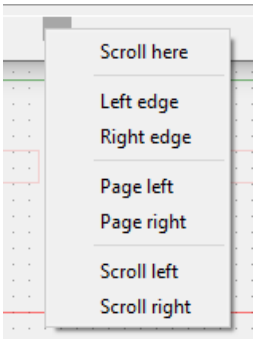


Рис. 9.2. Контекстне меню смуги прокрутки

Рис. 9.3. Вікно додатка, яке демонструє роботу смуги прокрутки

Лістинг 9.2. Файл main.cpp

```
#include <QApplication>
#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.setWindowTitle("ScrollBar");
wgt.resize(250, 150); wgt.show();

return app.exec();
}
```

У лістингу 9.2 створюються віджети електронного індикатора (вказівник `plcd`) і смуги прокрутки (вказівник `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`.

Кругова шкала

Клас `QDial` віджета установника визначено в заголовному файлі QDial. Цей віджет дуже схожий на регулятор гучності радіоприймача, яким можна маніпулювати за допомогою миші або клавіш управління курсором. За своїми функціональними можливостями він схожий 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 додатка, представлене на мал. 9.4, містить віджети установника і індикатора прогресу. Стан останнього залежить від місця розташування стрілки установника.

Лістинг 9.3. Файл main.cpp

```
#include <QApplication>
#include <QtWidgets>
```

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

QWidget wgt;
QDial*      pdia = new QDial;
 QProgressBar* pprb = new QProgressBar;

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.setWindowTitle("Dial");
wgt.resize(180, 200); wgt.show();

return app.exec();
}
```

У лістингу 9.3 створюються віджети установника (курсор `pdia`) і індикатора прогресу (курсор `pprb`). Викликом методу `setRange()` віджета установника задається діапазон значень від 0 до 100. Метод `setNotchTarget()` встановлює крок малювання рисок рівний 5, а слот `setNotchesVisible()` робить їх видимими, отримавши значення `true`.

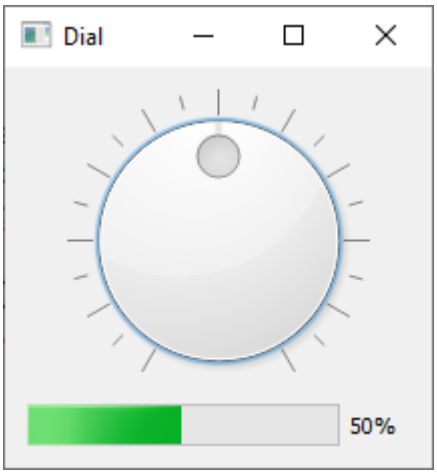


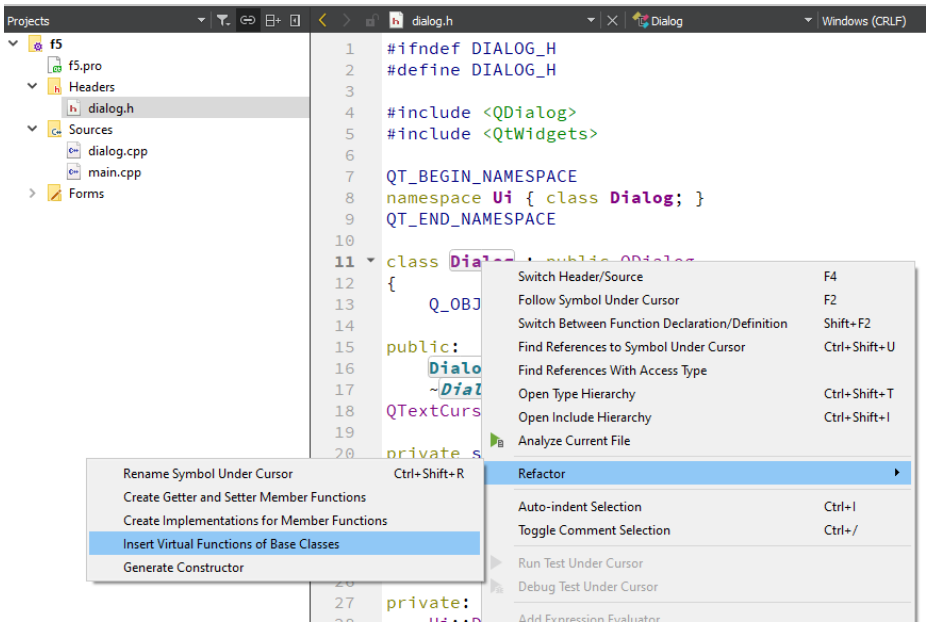
Рис. 9.4. Вікно додатка, яке демонструє роботу установника

Сигнал віджета установника `valueChanged(int)` з'єднується за допомогою методу `connect()` зі слотом індикатора прогресу `setProgress(int)`.

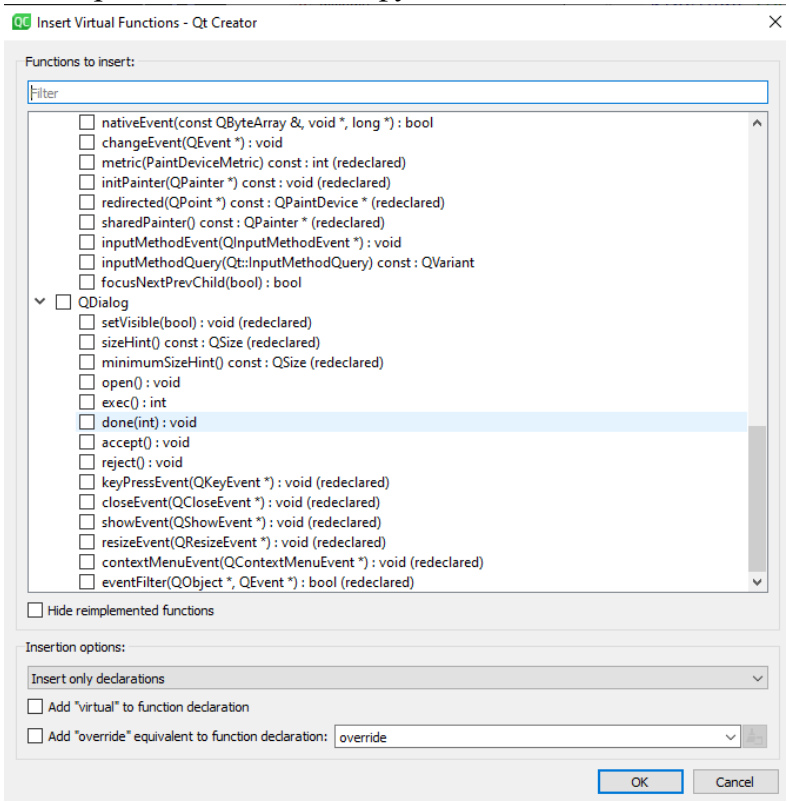
На завершення віджети установника і індикатора прогресу за допомогою об'єкта класу `QVBoxLayout` розміщуються вертикально.

Подія закриття.

Можна добавити в клас функцію, яка перехоплює подію закриття програми. Тобто перед тим як запустити деструктор ми можемо запустити якийсь код. (щось таке подібне, як запитується перед закриттям програми чи потрібно зберегти дані)



і вибираємо з можливих функцій



Нас цікавить функція

void ***closeEvent***(QCloseEvent *event); - ця функція викликається коли ви нажимаєте на хрестик вікна програми.

```
void Dialog::closeEvent(QCloseEvent *event)
{
    qDebug()<<"byby";
    event->accept();
}
```

Елементи введення

Група віджетів елементів введення являє собою основу для введення і редагування даних - тексту і чисел - користувачем. Велика частина елементів введення може працювати з буфером обміну і підтримує технологію *перетягування* (drag & drop), що позбавляє розробника від додаткової реалізації. Текст можна виділяти за допомогою миші, клавіатури і контекстного меню.

Однорядкове текстове поле

Цей віджет є найпростішим елементом введення. Клас `QLineEdit` однорядкового текстового поля визначено в заголовному файлі QLineEdit.

Текстове поле складається з прямокутної області для введення рядка тексту, тому не слід використовувати цей віджет в тих випадках, коли потрібно вводити більше одного рядка тексту. Для введення багаторядкового тексту є клас `QTextEdit`.

Текст, що знаходиться в віджеті, повертає метод `text()`. Якщо вміст віджета змінився, то відправляється сигнал `textChanged()`. Сигнал `returnPressed()` повідомляє про натискання користувачем клавіші <Enter>. Виклик метода `setReadOnly()` з параметром `true` установлюють режим "тільки для читання", в якому користувач може тільки переглядати текст, але не редагувати. Текст для ініціалізації віджета можна передати в слот `setText()`.

Для однорядкового текстового поля можна включити режим введення пароля, який встановлюється викликом методу `setEchoMode()` з прапором `Password`. В результаті цього вводимі символи не відображаються, а замінюються символом *.

Вікно програми, показане на мал. 10.1, має два однорядкових поля, одне з яких встановлено в режим введення пароля. Введений в це поле текст відображається в віджеті написи.

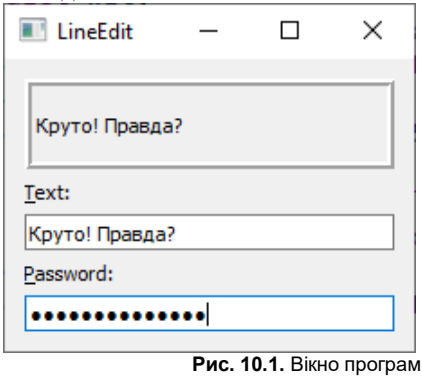


Рис. 10.1. Вікно програми з однорядковими полями введення

Лістинг 10.1. Файл main.cpp

```
#include <QApplication>
```

```
#include <QtWidgets>

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

QWidget wgt;
QLabel* plblDisplay = new QLabel;
plblDisplay->setFrameStyle(QFrame::Box | QFrame::Raised);
plblDisplay->setLineWidth(2);
plblDisplay->setFixedHeight(50);

QLabel*      plblText = new QLabel("&Text:");
QLineEdit* ptxt      = new QLineEdit;
plblText->setBuddy(ptxt);
QObject::connect(ptxt, SIGNAL(textChanged(const QString&)),
plblDisplay, SLOT(setText(const QString&)));

QLabel*      plblPassword = new QLabel("&Password:");
QLineEdit* ptxtPassword = new QLineEdit;
plblPassword->setBuddy(ptxtPassword);
ptxtPassword->setEchoMode(QLineEdit::Password);
QObject::connect(ptxtPassword, SIGNAL(textChanged(const
QString&)),plblDisplay, SLOT(setText(const QString&)));

//Layout setup
QVBoxLayout* pvbxLayout = new QVBoxLayout;
 pvbxLayout->addWidget(plblDisplay);
pvbxLayout->addWidget(plblText);
pvbxLayout->addWidget(ptxt);
pvbxLayout->addWidget(plblPassword);
pvbxLayout->addWidget(ptxtPassword);
wgt.setLayout(pvbxLayout);


wgt.setWindowTitle("LineEdit");
//wgt.resize(180, 200);
wgt.show();

return app.exec();
}
```

У лістингу 10.1 створюється віджет напису (вказівник `plblDisplay`). Метод `setFrame()` встановлює стиль рамки, а `setLineWidth()` — її товщину. Висота віджета написи фіксується за допомогою методу `setFixedHeight()`. Ще два написи, `plblText` і `plblPassword`, зв'язуються з віджетами однорядкового текстового поля методом `setBuddy()`. Потім сигнали `textChanged()` з'єднуються зі слотами `setText()` віджета написи `plblDisplay` для відображення тексту, що вводиться. І на завершення віджети розміщуються вертикально за допомогою об'єкта класу `QVBoxLayout`.

Кількість символів, що вводять можна обмежити методом `setMaxLength()`, передавши в

нього ціле значення, що обмежує максимальну довжину рядка. Для отримання поточного максимального значення довжини існує метод `maxLength()`.

Клас `QLineEdit` надає наступні слоти для роботи з буфером обміну:

- `copy()` — копіює виділений текст;
- `cut()` — копіює виділений текст і видаляє його з поля введення;
- `paste()` — вставляє текст (починаючи з позиції курсора), стираючи виділений текст.

Метод `undo()` скасовує останню виконану операцію, а метод `redo()` повторює останню скасовану. Дізнатися, чи можливо скористатися операціями відміни і затримки, можна за допомогою методів `isUndoAvailable()` і `isRedoAvailable()`, повертають булеві значення.

> ***ПРИМІТКА***
>
> Інформація в LineEdit має тип Qstring. Для розгляду ведених даних як числовий тип, можна скористатись методами класу QString – toInt(), number().

```
QString x= QLineEdit->text();
     QString y= QLineEdit->text();
     int S1=x.toInt();  int S2=y.toInt();
     int z=S1+S2;
QLineEdit->setText(QString::number(z));
```

Редактор тексту

Клас `QTextEdit` дозволяє здійснювати перегляд і редагування як простого тексту, так і тексту в форматі HTML. Він успадкований від класу `QAbstractScrollArea`, що дає можливість автоматично відображати смуги прокрутки, якщо текст не може бути повністю відображений у відведеній для нього області.

> ***ПРИМІТКА***
>
> Якщо вам потрібен редактор для звичайного тексту, то доцільніше буде скористатися замість класу `QTextEdit` класом `QPlaintTextEdit`. Клас `QPlaintTextEdit` не підтримує RTF (Rich Text Format, формат збагаченого тексту), в силу чого є більш легким, простим і ефективним.

Клас `QTextEdit` містить наступні методи:

- `setReadOnly()` встановлює або знімає режим блокування зміни тексту;
- `text()` повертає поточний текст.

А ось і деякі його слоти:

- `setPlainText()` — установка звичайного тексту;
- `setHtml()` — установка тексту в форматі HTML;
- `copy()`, `cut()` і `paste()` — робота з буфером обміну (копіювати, вирізати і вставити відповідно);
- `selectAll()` або `deselect()` — виділення або зняття виділення всього тексту;

- `clear()` — очищення поля ведення.

І сигнали:

- `textChanged()` — відправляється при зміні тексту;
- `selectionChanged()` — відправляється при змінах виділення тексту.

Для роботи з виділеним текстом служить клас `QTextCursor`, і об'єкт цього класу міститься в класі `QTextEdit`. Клас `QTextCursor` надає методи для створення ділянок виділення тексту, отримання вмісту виділеного тексту і його видалення. Покажчик на об'єкт класу `QTextCursor` можна отримати викликом методу `QTextEdit::textCursor()`.

Віджети класу `QTextEdit` також містять в собі об'єкт `QTextDocument`, покажчик на який можна отримати за допомогою методу `QTextEdit::document()`. Можна також присвоїти інший документ за допомогою методу `QTextEdit::setDocument()`. Клас `QTextDocument` надає слот `undo()` (для відміни) або `redo()` (для повтору дій). При виклику слотів `undo()` і `redo()` надсилаються сигнали `undoAvailable(bool)` і `redoAvailable(bool)`, повідомляють про успішне (або безуспішне) проведення операції. Ці сигнали відправляються як з класу `QtextDocument`, так і з `QTextEdit`. У більшості випадків зручніше буде використовувати сигнали класу `QTextEdit`.

Більшість методів класу `QTextEdit` являються делегуючими для класу `QTextDocument`. Наприклад, як вже було сказано раніше, клас `QTextEdit` здатний відображати файли з кодом на мові HTML, містять таблиці та растрові зображення. Для його розміщення і показу можна скористатися методом `setHtml()`, в який передається рядок, що містить в собі текст в форматі HTML, або скористатися слотом `insertHtml()`. Ці методи визначені в обох класах, і їх виклик з об'єкта класу `QTextEdit` призведе до того, що буде викликаний аналогічний метод з об'єкта класу `QTextDocument`.

Для переміщення звичайного тексту в область віджета можна скористатися методом `setPlainText()` або слотом `insertPlainText()`. За допомогою слота `append()` здійснюється додавання тексту, причому доданий текст не вноситься в список операцій, дія яких можна повернути за допомогою слота `undo()`, що робить цей слот швидким і не вимагає додаткових витрат пам'яті. Метод `find()` може бути використаний для пошуку і виділення заданого рядка в тексті.

> ***ПРИМІТКА***
>
> Клас `QTextEditor` можна використовувати спільно з класом `QSyntaxHighlighter` для підсвічування синтаксису.

Слоти `zoomIn()` і `zoomOut()` призначені для збільшення або зменшення розміру шрифту, і їх дія не поширюється на растрові зображення.

> ***ПРИМІТКА***
>
> Якщо вам потрібно тільки відобразити текст у форматі HTML, то, можливо, краще користуватися класом `QLabel` (див. *главу 7*).

Наведений на мал. 10.2 приклад відображає HTML-документ. Текст документа можна редагувати.

Лістинг 10.2. Файл main.cpp

```
#include <QApplication>
#include <QtWidgets>
```

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

QTextEdit       txt;

txt.setHtml("<HTML>"
"<BODY BGCOLOR=MAGENTA>"
"<CENTER><IMG SRC=\"1.jpg\"></CENTER>"


"<H2><CENTER>THE WINNER TAKES IT ALL</CENTER></H2>"
"<H3><CENTER>(B.Andersson/B.Ulvaeus)</CENTER><H3>" "<FONT
COLOR=BLUE>"
"<P ALIGN=\"center\">" "<I>"
"I don't wanna talk<BR>"
"About the things we've gone through<BR>" "Though it's hurting
me<BR>"
"Now it's history<BR>"
"I've played all my cards<BR>"
"And that's what you've done too<BR>" "Nothing more to say<BR>"
"No more ace to play<BR><BR>" "The winner takes it all<BR>" "The
loser standing small<BR>" "Beside the victory<BR>" "That's her
destiny<BR>" "..."
"</I>"
"</P>" "</FONT>" "</BODY>" "</HTML>"
);
txt.resize(320, 250); txt.show();




return app.exec();
  }
```

У лістингу 10.2 створюється віджет верхнього рівня `txt`. Метод `setHtml()` встановлює в віджеті `QTextEdit` текст в форматі HTML.

Рис. 10.2. Вікно програми, що відображає HTML-документ

З чого починаються віджети лічильників

Віджет абстрактного лічильника - клас QAbstractSpinBox надає всім унаследован- ним від нього класах невелике текстове поле і дві стрілки для зменшення або збіль- чення числових значень. Від цього віджета успадковані такі класи (див. Рис. 5.1):

- `QSpinBox` — лічильник;
- `QDateTimeEdit` — елемент для введення дати і часу;
- `QDoubleSpinBox` — елемент для введення значень, що мають тип double

Можна встановити циклічний режим, коли за максимально можливим значенням буде слідувати мінімально можливе, і навпаки. Цей режим встановлюється викликом методу setWrapping() з параметром true. Реалізовані два метода покрокової зміни значення `stepUp()` и `stepDown()`, які симулюють нажаття на кнопки стрілок.

С допомогою методу setSpecialValueText () встановлюється текст, незалежно від числового значення, наприклад:

```
pspb->setSpecialValueText("default");
```

Лічильник

Віджет QSpinBox надає користувачеві доступ до обмеженого діапазону чисел. Все що вводяться значення перевіряються, для запобігання виходу за межі установленого діапазону, який встановлюється методом setRange (). Значення можна встановлювати за допомогою методу setValue (), а отримувати - методом value (). При зміні значення надсилаються відразу два сигнали valueChanged (): один з параметром типу int, а інший - з const QString &. Можна змінити спосіб відображення за допомогою методів setPrefix() і setSuffix (). Наприклад, виклик наступних методів призведе до того, що число буде відображатися в дужках:

```
pspb->setPrefix("(");
pspb->setSuffix(")");
```

Можна змінити зображення стрілок на символи + (плюс) або - (мінус), передавши методу

setButtonSymbols () прапор PlusMinus.

На рис. 10.4 зображено приклад, що дозволяє вибирати і встановлювати числа в діапазоні від 1 до 100.

Рис. 10.4. Приклад лічильника

Листинг 10.10. Файл main.cpp

```
#include <QApplication>
#include<QSpinBox>
```

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QSpinBox   spb;

    spb.setRange(1, 100); spb.setSuffix(" MB");
    spb.setButtonSymbols(QSpinBox::PlusMinus);
    spb.setWrapping(true);
    spb.show(); spb.resize(50, 30);

    return a.exec();
}
```

У лістингу 10.10 створюється віджет лічильника spb. Після його створення виконується установка діапазону за допомогою методу setRange (). Виклик методу setSuffix () додає рядок "MB" після відображається лічильником величини, а метод setButtonSymbols (), якому пере- дається прапор PlusMinus, замінює стрілки лічильника знаками +/-. Метод setWrapping () установлюється циклічний режим.

Елемент введення дати і часу

При зміні дати або часу посилається сигнал dateTimeChanged (). для класу

QDateTimeEdit цей сигнал передає константну посилання на об'єкт типу QDateTime.

Приклад, зображений на рис. 10.5, відображає актуальну дату і час запуску програм-ми, які можна модифікувати

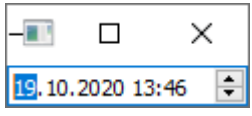


Рис. 10.5. Дата и час

Лістинг 10.11. Файл main.cpp

```
#include <QApplication>
#include<QDateTimeEdit>


int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QDateTimeEdit dateTimeEdit(QDateTime::currentDateTime());
    dateTimeEdit.show();
    return a.exec();
}
```

Як видно з лістинга 10.11, при створення віджету `QDataTimeEdit` в його конструктор передається поточна дата та час, які повертаються викликом статичного методу `QDateTime::currentDateTime()`. Віджет відображається на екрані після виконання методу `show()`.

Перевірка введення

Об'єкт класу QValidator (далі контролер) гарантує правильність введення користувачем. Для установки об'єкта класу QValidator необхідно передати його в метод setValidator (), який міститься в класах QComboBox і QLineEdit. Для перевірки введення чисел користуються готовими класами QIntValidator і QDoubleValidator. Створюючи свій клас перевірки введення, потрібно успадкувати клас від QValidator і перезаписати метод validate (), в який передається вводиться рядок і позиція курсора. Метод повинен віз обертати наступні значення:

- `QValidator::Invalid` — якщо рядок не можу бути прийнятий;
- `QValidator::Intermediate` — рядок не може бути прийнята в якості кінцевого результату. Наприклад, якщо рядок повинена представляти чисельне значення від 50 до 100, то введення числа 1 буде являти собою проміжне значення;
- `QValidator::Acceptable` — якщо рядок не може бути прийнятий без змін

У прикладі, зображеному на рис. 10.6, користувачеві пропонується ввести своє ім'я, і циф- ри в цьому випадку неприпустимі. Це відстежується класом контролера, яка не допускає того, щоб ім'я містило цифри.

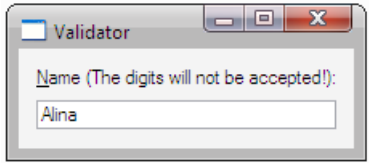


Рис. 10.6. Вікно програми, яке перевіряє правильність введення

Лістинг 10.12. Класс `NameValidator`. Файл main.cpp

```
class NameValidator : public QValidator { public:
NameValidator(QObject* parent) : QValidator(parent)
{
}

virtual State validate(QString& str, int& pos) const
{
QRegExp rxp = QRegExp("[0-9]"); if (str.contains(rxp)) {
return Invalid;
}
return Acceptable;
}
};
```

Як видно з лістингу 10.12, клас NameValidator успадкований від класу QValidator. В цьому класі виконується перезапис методу validate (), який отримує рядок і позицію курсору. Усередині методу створюється об'єкт регулярного виразу rxp, в конструктор якого передається шаблон, який представляє собою діапазон цифр від 0 до 9. У операторі if викликом методу QString ::contains () здійснюється перевірка рядка на вміст у ній заданого шаблону. У разі, якщо збіг знайдено, метод поверне значення Invalid, повідомляючи, що введення не задовольняє заданим критеріям, в протилежному

випадку повертається значення Acceptable і введення приймається.

Листинг 10.13. Основная программа. Файл main.cpp

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget     wgt;

    QLabel* plblText =   new QLabel("&Name (The digits will not be
accepted!):");
    QLineEdit* ptxt = new QLineEdit;
    NameValidator* pnameValidator = new NameValidator(ptxt);
    ptxt->setValidator(pnameValidator);
    plblText->setBuddy(ptxt);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plblText);
    pvbxLayout->addWidget(ptxt);
    wgt.setLayout(pvbxLayout);
    wgt.show();


    return a.exec();
}
```

У лістингу 10.13 створюється віджет напису (вказівник plblText), віджет однорядкового текстового поля (вказівник ptxt) і об'єкт контролера (вказівник pnameValidator). Після створення викликом методу setValidator () контролер встановлюється в віджеті одно-сатиричного текстового поля (вказівник ptxt), а останній зв'язується з написом викликом методу setBuddy ().

На рис. 7.3 показані звязані віджети setBuddy () методом , а в лістингу 7.3 приведений текст відповідної програми.

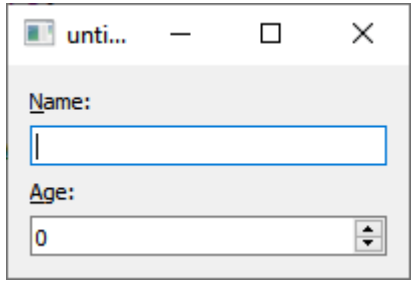


Рис. 7.3. Результат використання знака & символи N і A- підкреслені

Лістинг 7.3. Визначення в тексті віджета символів швидкого доступу за допомогою знака & (файл main.cpp)

```
#include <QApplication>
# 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 (pspbAge);
wgt.setLayout (pvbxLayout);
wgt. show ();

return app. exec ();
}
```

У лістингу 7.3 віджет wgt класу QWidget є віджетом верхнього рівня, так як за замовчуванням його конструктор привласнює 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. в віджет напису Name (Ім'я) в тексті символ N визначено як символ для швидкого доступу. Потім створюється віджет однорядкового текстового поля. Далі, виклик методу setBuddy() пов'язує віджет напису з створеним текстовим полем, використовуючи покажчик на поле в якості аргументу. Аналогічно відбувається створення віджета напису Age (Вік), поля для введення віку (класу QSpinBox) і їх зв'язування.

Додатково про регулярні вирази можна прочитати в [Шлее Макс - Qt 5.10. Профессиональное программирование на C++. , ст. 97]

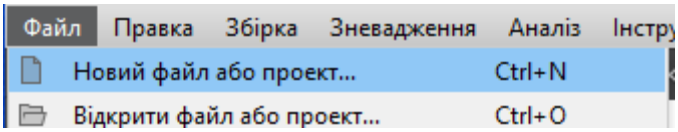
QTextCursor Class QTextCursor cursor; QTextEdit *spb= new QTextEdit ();		
textCursor()	Повертає вказівник накурсор	QTextCursor cursor = spb->textCursor();
	взаємодія з курсором щоб змінити виділений текст	QTextCursor cursor = spb->textCursor(); QString text=cursor.selectedText(); cursor.insertHtml(""+text+"");
selectedText()	виокремлення виділеного тексту	QString text=cursor.selectedText();
cursor.insertHtml() cursor.insertText()	вставити текст в позицію курсора (або замість виділеного тексту) можна як звичйний текст, а можна як HTML	cursor.insertHtml(""+text+""); cursor.insertText("some text"+text+" some text ");

Робота з вікнами

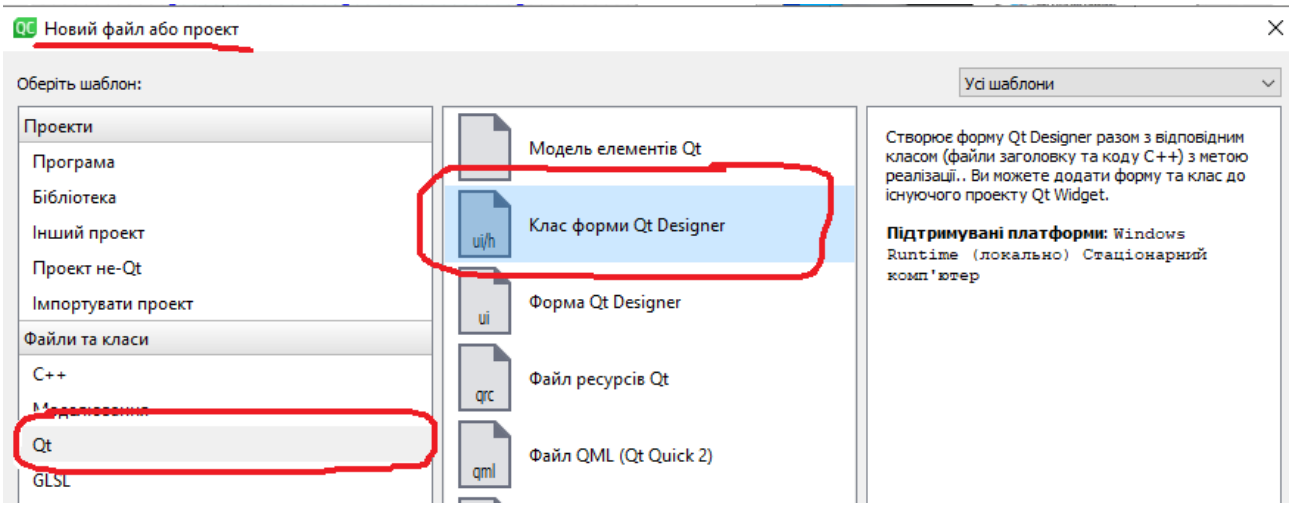
Для збільшення функціональності додатку, програми що розробляються насичують багатовіконним інтерфейсом.

Нове вікно це та сама форма з можливістю добавляти в неї всі віджети що і на головній формі програми.

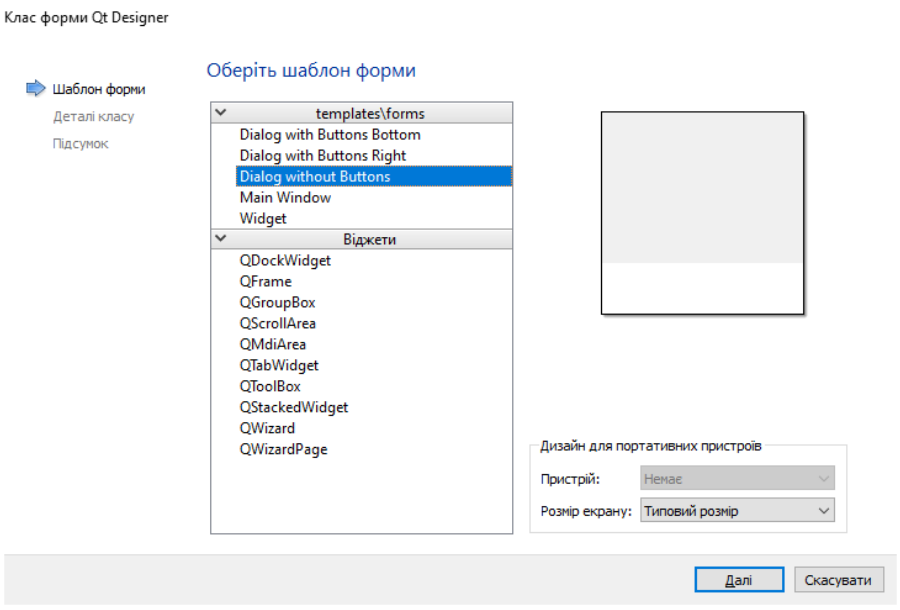
Для розробки багатовіконного інтерфейсу потрібно спочатку добавити нову форму в проект.



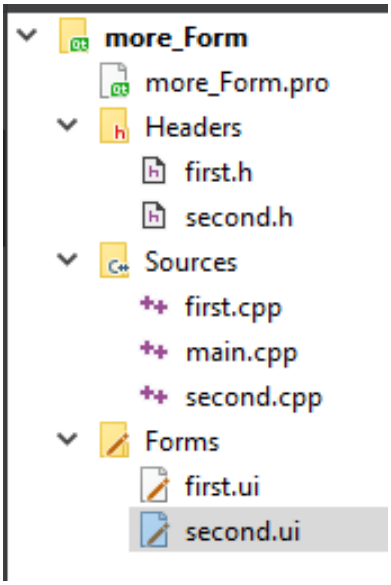
В вікні Новий файл або проект вибираємо в категорії Файли та класи шаблон Qt і в центральному вікні тип Клас форми Qt Designer



Наступним кроком можна вибрати шаблон форми, що буде добавлятись в проект



Наступні два кроки передбачають заповнення назви файлів та класів необхідних для роботи форми що добавляється. Після підтверження всіх необхідних дій в проекті появиться нова форма та всі необхідні для її роботи файли (***.h, ***.cpp, ***.ui).



Для можливості переходити з головного вікна в додаткове необхідно в файлі *.cpp головної форми добавити заголовочний файл додаткової форми.

В нашому випадку в mainwindow.cpp прописати #include "second.h"

Для запуску вікна необхідно прописати наступний код:

```
second f2;
f2.setModal(true);
f2.exec();
```

`second f2` - створюється об'єкт класу, представник другої форми в програмі (`second` – ім'я класу в другій формі); `f2.setModal(true)` - встановлюється тип вікна, модальне вікно, тобто вікно, яке буде відкриватись (в такому режимі вікно з якого відкриватиметься інша форма стане неактивним, аж поки друга форма не буде закрита); f2.exec() – команда виконання (команда, яка безпосередньо запустить вікно).

Клас QDialog є базовим для всіх діалогових вікон, представлених в класовій ієрархії Qt. Хоча діалогове вікно можна створювати за допомогою будь-якого віджета, зробивши його виджетом верхнього рівня, тим не менш, зручніше скористатися класом QDialog, який надає ряд можливостей, необхідних всім діалогових вікон. Діалогові вікна поділяються на дві групи:

- модальні;
- немодальні.

Режим модальності і немодального можна встановити і дізнатися за допомогою методів QDialog::setModal () і QDialog::isModal () відповідно.

При установці true відповідає модальному режиму, а false - немодальному.

Модальні діалогові вікна

Такі вікна зазвичай використовуються для виведення важливих повідомлень. Наприклад, є помилки, на які користувач повинен відреагувати, перш ніж продовжити працювати з додатком. Модальні вікна переривають роботу програми і для продовження його роботи таке вікно має бути закрите.

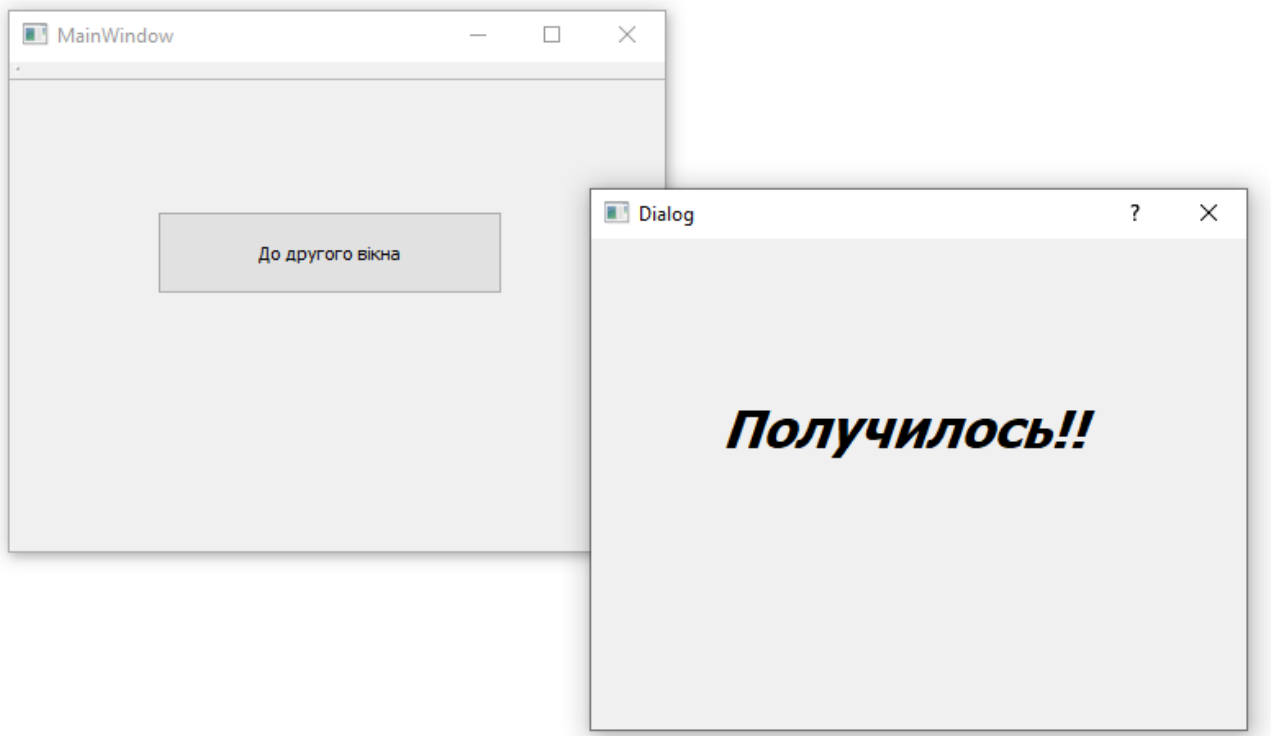
У цих випадках модальний діалог - ідеальний засіб для звернення уваги користувача на себе.

Немодальні діалогові вікна

Немодальні діалогові вікна поводяться як нормальні віджети, не перериваючи, зі своєю появою, роботу додатка. На перший погляд може здатися, що застосування

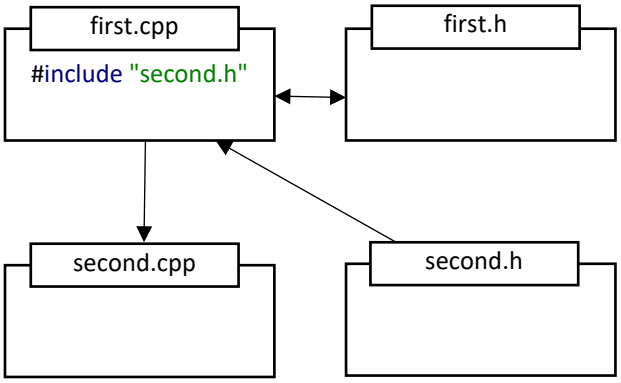
немодального діалогових вікон має більше сенсу, так як в цьому випадку користувач має більше свободи в своїх діях. Але, насправді, більшість додатків потребує зупинці і очікуванні рішення користувача для відновлення своїх подальших дій.

Немодального вікно може бути відображено за допомогою методу show (), також як і для звичайного віджета. Метод show () не повертає ніяких значень і не зупиняє виконання всієї програми. Метод hide () дозволяє зробити вікно невидимим. Цим властивістю можна скористатися, і в цьому випадку не потрібно створювати кожен раз об'єкт діалогового вікна, а при закритті видаляти його з пам'яті. Можна обмежитися викликом методів show () і hide (), що дасть можливість відобразити діалогове вікно, на тому ж місці, на якому воно було згорнуто. Немодальні діалогові вікна необхідно постачати кнопкою Close (Закрити) для того, щоб дати можливість користувачу закрити його.



Зауваження. Весь код виклику прописується в методі, який запускається при нажатті на кнопку «До другого вікна».

Таким чином сехма звязків між класами виглядатиме наступним чином:



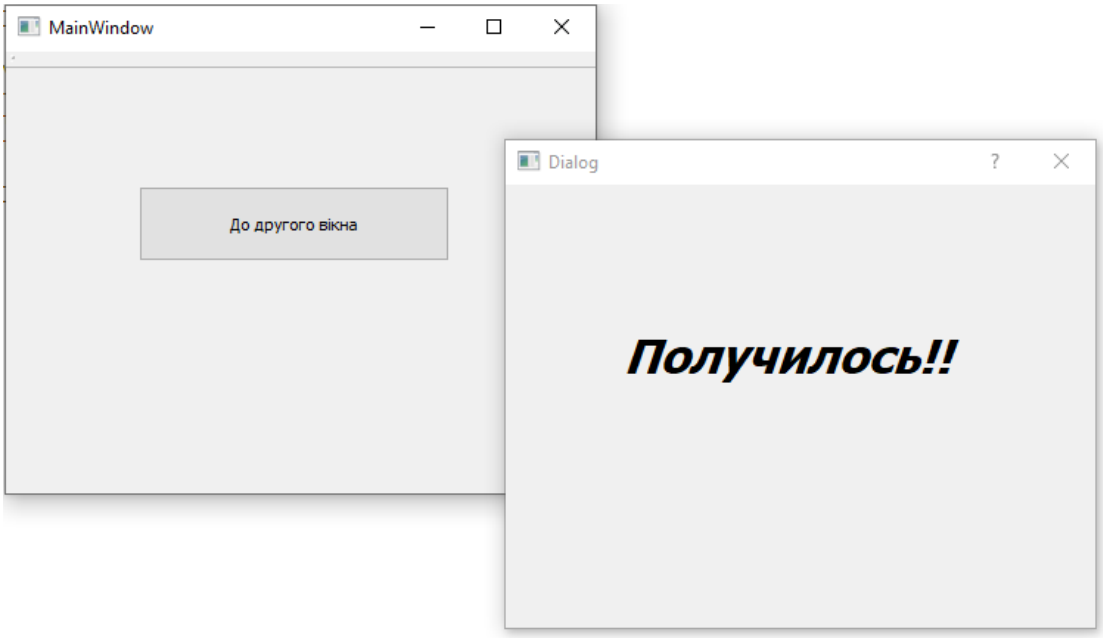
Для того, щоб обидва вікна були активними, необхідно заголовочний файл підключити до *.h та сворити вказівник на відповідний клас другої форми в класі **MainWindow. -** Dialog *f2;

І тоді код запуску форми виглядатиме наступним чином:

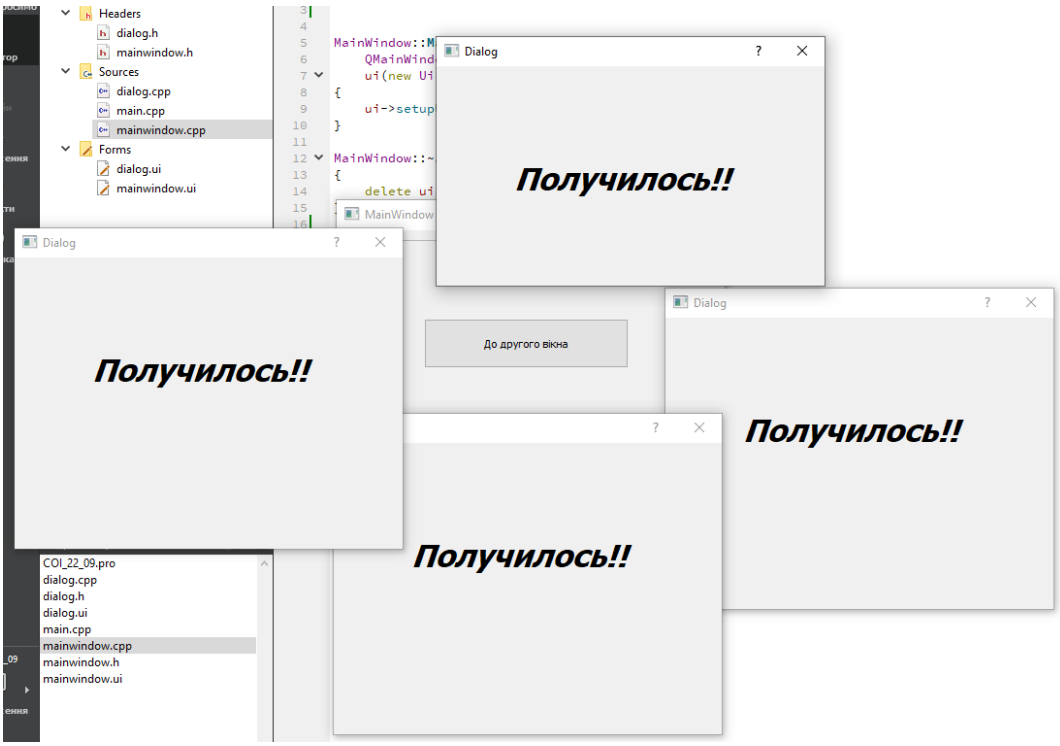
```
f2 = new Dialog(this);
f2->show();
```

`f2 = new Dialog(this)` – виділяється пам'ять під об'єкт f2;

`f2->show()` – запускається показ вікна.



Активність першого вікна дозволяє повторно виконувати нам дії з ним.



Функція hide() – дозволить сховати поточне вікно при відкритті нового.

Підсумовуючи.

//можна відкривати скільки хочеш вікон

```
void Dialog::on_btnWithParent_clicked()
{
    //With Parent
    Dialog2 *dialog = new Dialog2(this);

    dialog->show();
}

void Dialog::on_btnWithoutParent_clicked()
{
    //Without Parent
    Dialog2 *dialog = new Dialog2(nullptr);
    dialog->show();
        }
```

//With Parent – якщо закрити головну форму то закриється і дочірне вікно

```
//Without Parent - якщо закрити головну форму то дочірне вікно залишиться
активне
```

Модальний режим

//відкриття в модальному режимі, поки відкрите дочірне то батьківське не активне

```
void Dialog::on_btnWithParent_clicked()
{
    //With Parent
    Dialog2 *dialog = new Dialog2(this);

    dialog->exec();
}

void Dialog::on_btnWithoutParent_clicked()
{
    //Without Parent
    Dialog2 *dialog = new Dialog2(nullptr);
    dialog-> exec();
        }
```

```
//Without Parent – при поверненні з дочірного вікна, вказівник на саму форму
залишається в пам’яті, аж доки ОС не подумає що його треба знищити получиться
висячий вказівник
```

Щоб уникнути висячих вказівників можна використовувати клас `QScopedPointer`.

Клас `QScopedPointer` зберігає вказівник на динамічно виділений об’єкт і видаляє його після знищення.

Управління вручну об’єктами, пам’ять для якихвиділялась з купи, є складним і схильним до помилок процесом. `QScopedPointer` — це невеликий службовий клас, який значно спрощує це, призначаючи право власності на пам’ять на основі стека розподілу купи, що загальніше називається отриманням ресурсу — ініціалізацією (RAII).

`QScopedPointer` гарантує, що об’єкт, на який вказано, буде видалено, коли поточна область зникне.

Для оголошення об’єкту потрібнопрописати `QScopedPointer<Dialog> dlg(new Dialog());`

Для звернення до методів з дочірної форми викоистовується метод data() (хоча можна і без нього, проте відсутність метода в деяких ОС може працювати некоректно)

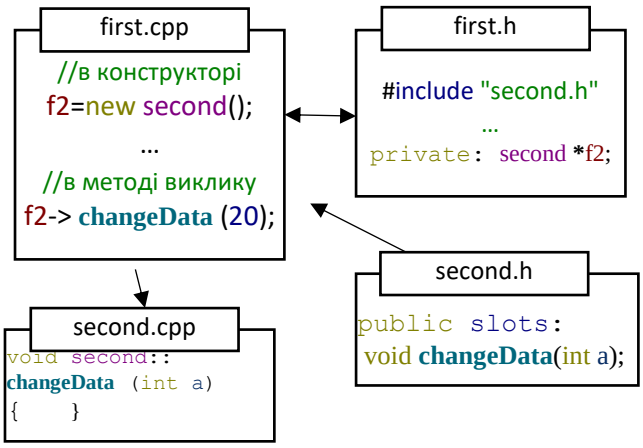
```
dlg.data()->setList(options); //передача даних в метод setList
    dlg.data()->exec(); // відкриття вікна в модальному режимі
```

ui->plainTextEdit->insertPlainText(dlg.data()->selected()); // отримання значень з метода дочірної форми після закриття форми.

ПЕРЕДАЧА ДАНИМИ МІЖ ФОРМАМИ.

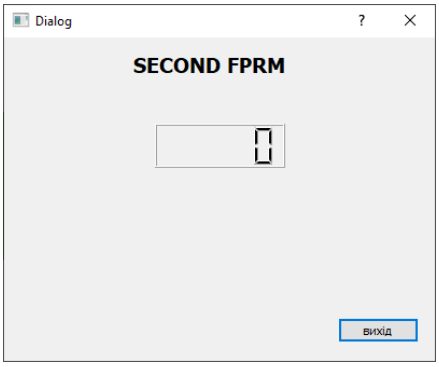
Передача даних в дочірну форму.

Загальна структура проекту матиме наступний вигляд



ГОЛОВНЕ. Концепція полягає в використанні сигнально-слотової архітектури. В головній формі викликається слот дочірної форми, якому передається значення. І вже в дочірній формі в слоті прописується, що із переданим значенням робити.

1. Створюємо проект та добавляємо клас для другої форми (див вище). Зробимо так, щоб при нажатті на кнопку в першій формі в другій формі змінювалось значення LCDNumber віджета. Для цього створимо відповідний дизайн наших форм.



2. в заголовочний файлі першої форми (first.h) підключаємо другу форму і створюємо вказівник на неї.

```
#ifndef FIRST_H
#define FIRST_H

#include <QMainWindow>
```

```
#include "second.h"

namespace Ui {
class First;
}

class First : public QMainWindow
{
    Q_OBJECT

public:
    explicit First(QWidget *parent = nullptr);
    ~First();

private:
    Ui::First *ui;
      Second *f2;
};

#endif // FIRST_H
```

3. Створємо бузпосередньо об'єкт нашої форми (f2=new Second();) . Робиться це в конструкторі першої форми (first.cpp).

```
First::First(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::First)
{
f2=new Second();
    ui->setupUi(this);
}
```

4. Для виклику другої форми потрібно виклакати слот show() для немодального способу та exec() для модального виклику. В нашому випадку ми пропишемо це в слоті нажаття на кнопку (first.cpp).

```
void First::on_pushButton_clicked()
{
f2->exec();
}
```

5. Всі дії, які нам потрібно здійснити в другій формі повині запускатись відповідним слотом (методом), який викликатиметься з першої форми. Створимо цей слот в класі форми:

```
#ifndef SECOND_H
#define SECOND_H

#include <QDialog>

namespace Ui {
class Second;
}

class Second : public QDialog
{
    Q_OBJECT

public:
    explicit Second(QWidget *parent = nullptr);
    ~Second();

private:
    Ui::Second *ui;

public slots:
    void changeData(int f);
};
```

```
#endif // SECOND_H
```

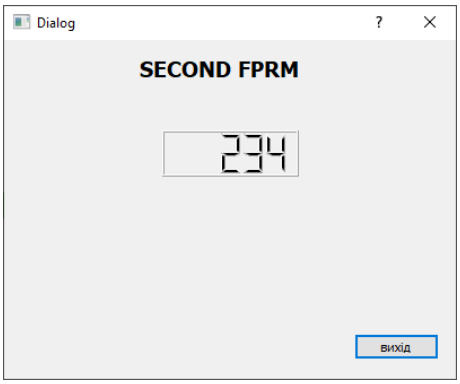
Якщо на даному етапі скомпілювати задачу, то може виникнути помилка unresolved external symbol. Говорить про те, що не визначений пропиманий слот. Потрібно відразу задати дію даного слоту.

6. Далі в second.cpp прописуємо дію даного слоту.

```
void Second::changeData(int a)
{
ui->lcdNumber->display(a);

}
```

7. І підкінець потрібно у341викликати даний метод changeData(int a) у відповідному сигналі чи слоту , який виконується. В нашому прикладі його виклик відбуватиметься при натисканні кнопки в першій формі

```
void First::on_pushButton_clicked()
{
f2->changeData(234);
f2->exec();
}
```



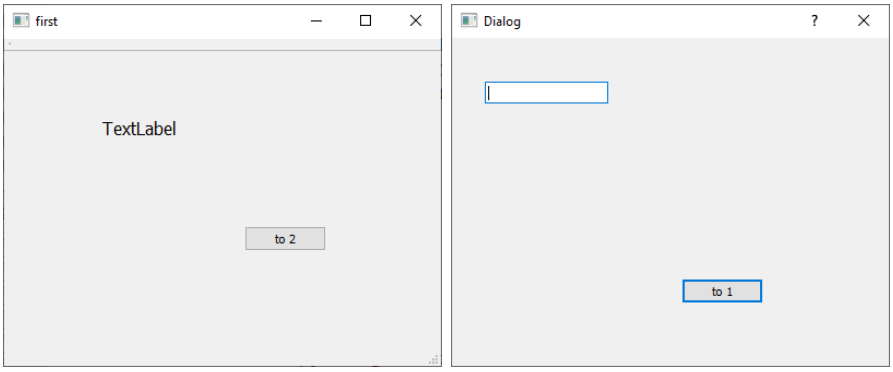
Якщо потрібно заховати головне вікно, коли відображається інша, формма, під час виклику другої форми потрібно застосувати методо hide().

```
void First::on_pushButton_clicked()
{
hide();
f2->changeData(234);
f2->exec();
}
```

Зворотна передача даних.

Принцип зворотної передачі даних відбувається за тим самим принципом. В першій формі потрібен метод, який отримає та опрацює значення від другої форми. Але в другій формі потрібен сигнал, який передаватиме це значення, а також потрібна подія, яка згенерує цей сигнал. Насправді сигнал може запустити і звичайний слот (метод) викликаний в потрібний момент. В попередньому варіанті таким сигналом слугував сигнал нажаття на кнопку та відповідний метод, який зв'язаний був з цим сигналом.

Отож, маємо дві форми:



При нажатті на кнопку другої форми в TextLabel має записатись значення, яке пропишеться в LineEdit другої форми.

1. В second.h як атрибути класу пропишемо сигнал і слот, які і забезпечать передачу значеня:

```
signals:
    void sendData(int&);
private slots:
  void onButtonSend();
```

2. В second.cpp пропишемо як повиннні діяти наші атрибути класу, як діє сигнал не потрібно прописувати, а от дію слоту обов'язково, плюс до всього ще запрограмуємо слот нажаття відповідної кнопки

```
void Second::onButtonSend()
{int a = ui->lineEdit->text().toInt();
    emit sendData(a); //emit - ключове слово, яке запускає сигнал
}

void Second::on_pushButton_clicked()
{
onButtonSend(); //викликаємо відповідний метод
    hide(); // ховаємо другу форму
        }
```

3. Отож, після прописання другого пункту, форма second запускатиме сигнал sendData. Даний сигнал повинен «зловитись» першою формою і запустити в ній відповідний слот. Пропишемо слот, який запускатиметься при генерації сигналу sendData.

first.h

```
public slots: void from2(int&);
```

first.cpp

```
void first::from2(int &a)
{
    ui->label->setText(QString::number(a));
        }
```

4. Залишається зв'язати сигнал і слот. Прописується це в конструкторі головної (першої) форми.

```
first::first(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::first)
{
    f2 = new Second(this);
    ui->setupUi(this);
```

```
    connect(f2,SIGNAL(sendData(int&)),this,SLOT(from2(int&)));
}
```

> Для меншої кількості коду, в конструкторі головної форми можна відразу прописати і відкриття другої форми при нажатті на кнопку першої форми.
>
> connect(ui->pushButton,SIGNAL(clicked()),f2,SLOT(show()));

Інший спосіб обміну даними, можна постворювати в дочірній формі методи, які повертатимуть та отримуватимуть дані.

Метод виклику дочірної форми

```
void Dialog::on_pushButton_clicked()
{
   Selections *dlg = new Selections(this); // - інша форма.
   // передаємо дані в форму
   dlg->setSelected(ui->lineEdit->text());
   //викликаємо форму
   dlg->exec();
   // отримуємо дані з форми
   ui->lineEdit->setText(dlg->selected());
}
```

клас дочірної форми

```
class Selections : public QDialog
{
   Q_OBJECT
public:
   explicit Selections(QWidget *parent = nullptr);
   ~Selections();
   QString selected();
   void setSelected(QString value);
}
```

QSettings

Зазвичай користувачі очікують, що програма запам’ятає свої налаштування (розміри та положення вікон, параметри тощо) протягом сеансів. Ця інформація часто зберігається в системному реєстрі Windows і у файлах списку властивостей у macOS та iOS. У системах Unix за відсутності стандарту багато програм (включаючи програми KDE) використовують текстові файли INI.

QSettings — це абстракція, пов’язана з цими технологіями, яка дає змогу зберігати та відновлювати параметри програми портативним способом. Він також підтримує спеціальні формати зберігання.

QSettings зберігає налаштування. Кожне налаштування складається з QString, що вказує назву налаштування (ключ), і QVariant, який зберігає дані, пов’язані з ключем. Щоб написати параметр, використовуйте setValue().

Наприклад: settings.setValue("editor/wrapMargin", 68);

Якщо вже існує налаштування з таким самим ключем, наявне значення буде перезаписано новим значенням. Для ефективності зміни не можуть бути збережені в постійному сховищі відразу. (Ви завжди можете викликати sync(), щоб зафіксувати свої зміни.)

Ви можете повернути значення налаштування за допомогою value():

```
int margin = settings.value("editor/wrapMargin").toInt();
```

Якщо параметр із вказаною назвою відсутній, QSettings повертає нульовий QVariant (який можна перетворити на ціле число 0). Ви можете вказати інше значення за замовчуванням, передавши другий аргумент у value():

```
int margin = settings.value("editor/wrapMargin", 80).toInt();
```

```
QSettings settings;
settings.setValue("animal/snake", 58);
settings.value("animal/snake", 1024).toInt();   // returns 58
settings.value("animal/zebra", 1024).toInt();   // returns 1024
settings.value("animal/zebra").toInt();         // returns 0
```

Щоб перевірити, чи існує даний ключ, викличте contains(). Щоб видалити налаштування, пов’язані з ключем, викличте remove(). Щоб отримати список усіх ключів, викличте allKeys(). Щоб видалити всі ключі, викличте clear().

Ключі налаштування можуть містити будь-які символи Unicode. Реєстр Windows і файли INI використовують ключі без урахування регістру, тоді як API CFPreferences на macOS і iOS використовує ключі з урахуванням регістру. Щоб уникнути проблем з переносимістю, дотримуйтеся простих правил:

Завжди посилайтеся на той самий ключ, використовуючи той самий регістр. Наприклад, якщо ви називаєте ключ «текстовими шрифтами» в одному місці свого коду, не називайте його «текстовими шрифтами» в іншому місці.

Уникайте ідентичних імен ключів, за винятком регістру. Наприклад, якщо у вас є ключ під назвою "MainWindow", не намагайтеся зберегти інший ключ як "mainwindow".

Не використовуйте косі риски ('/' і '\') у назвах розділів або ключів; символ зворотної косої риски використовується для розділення підключів (див. нижче). У вікнах '\' перетворюються QSettings на '/', що робить їх ідентичними.

Ви можете формувати ієрархічні ключі, використовуючи символ «/» як роздільник, подібно до шляхів до файлів Unix. Наприклад:

```
settings.setValue("mainwindow/size", win->size());
settings.setValue("mainwindow/fullScreen", win->isFullScreen());
settings.setValue("outputpanel/visible", panel->isVisible());
```

Якщо ви хочете зберегти або відновити багато налаштувань з однаковим префіксом, ви можете вказати префікс за допомогою beginGroup() і викликати endGroup() наприкінці. Ось знову той самий приклад, але цього разу з використанням механізму групування:

```
settings.beginGroup("mainwindow ");
settings.setValue("size", win->size());
settings.setValue("fullScreen", win->isFullScreen());
settings.endGroup();

settings.beginGroup("outputpanel ");
settings.setValue("visible", panel->isVisible());
settings.endGroup();
```

Якщо групу встановлено за допомогою beginGroup(), поведінка більшості функцій відповідно змінюється. Групи можна встановлювати рекурсивно.

Окрім груп, QSettings також підтримує концепцію «масиву». Див. beginReadArray() і beginWriteArray() для отримання додаткової інформації.

Приклад використання (проект з UI Forms)

*.h

```
#ifndef DIALOG_H
#define DIALOG_H

#include <QDialog>

QT_BEGIN_NAMESPACE
namespace Ui { class Dialog; }
```

Створюємо `void load();` яка і буде завантажувати із системи збережені дані

```
QT_END_NAMESPACE

class Dialog : public QDialog
{
    Q_OBJECT

public:
    Dialog(QWidget *parent = nullptr);
    ~Dialog();



private slots:
    void on_pushButton_clicked();

private:
    Ui::Dialog *ui;

    void load();
};
#endif // DIALOG_H
```

*.cpp

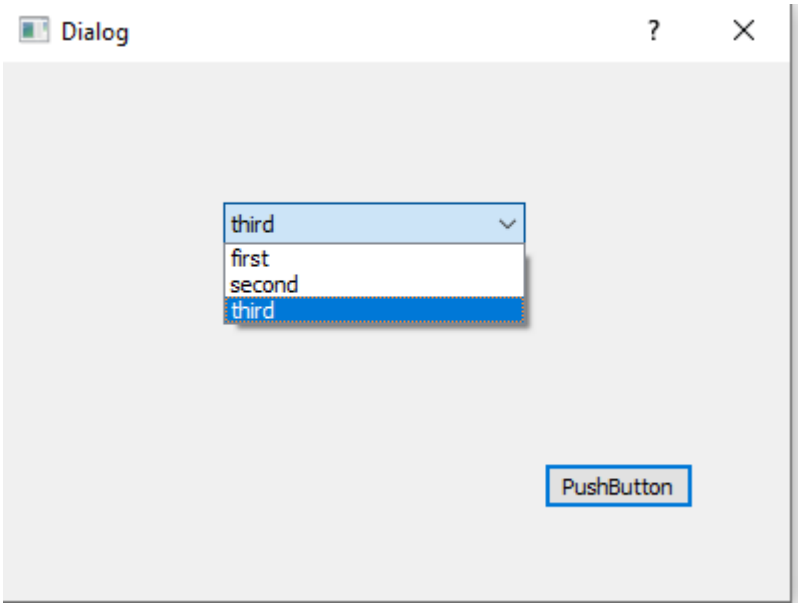
```
#include "dialog.h"
#include "ui_dialog.h"
#include <QSettings>
#include <QVariant>

Dialog::Dialog(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::Dialog)
{
    ui->setupUi(this);
    load();
}

Dialog::~Dialog()
{
    delete ui;
}




void Dialog::on_pushButton_clicked()
{
    QSettings settings("MySoft", "MyApp");
       settings.setValue("settings", ui->comboBox-
>currentIndex());
       accept();
}

void Dialog::load()
{
    QSettings settings("MySoft", "MyApp");
        int index =
settings.value("settings",0).toInt();
    ui->comboBox->setCurrentIndex(index);
}
```

`void load();` прописана в конструкторі

Наша кнопка буде закривати нашу програму, але перед тим створиться системний запис, і запишеться параметр за ключем "settings", а саме індекс значення comboBox при якому закривається програма.

Функція `void load();` створює об'єкт settings з тими самими параметрами.

І витягує значення за ключем "settings" записуючи його в змінну index перевівши до інтового значення.



Для кросплатформності, ідеальним буде наступний запис функції `void load();`

```
void Dialog::load()
{
    QSettings settings("MySoft", "MyApp");
    QVariant value = settings.value("settings",0);

    bool ok;
    int index = value.toInt(&ok);//ok=0? коли все гаразд
    if(!ok)
    {
        QMessageBox::critical(this,"Loading Error","Error loading selection!");
        return;
    }

    if(index < ui->comboBox->count())
    {
        ui->comboBox->setCurrentIndex(index);
    }
    else
    {
        ui->comboBox->setCurrentIndex(0);
    }
}
```

Можуть бути випадки коли значення ключа може перевищити індекси значень comboBox щоб це не виникало збоїв, потрібно здійснити додаткову перевірку (це більше потрібно для Mac and IOS систем .

Запис та встановлення параметрів(розмірність та положення) головного віджету

```
settings.setValue("geometry", saveGeometry());
```

.....

```
QVariant geometry = settings.value("geometry", QByteArray());

  restoreGeometry(geometry.toByteArray());
```

Встановлення параметрів вікна, так само як у випадку comboBox в ідеалі, також потребує перевірок:

```
const auto geometry = settings.value("geometry", QByteArray()).toByteArray();
    if (geometry.isEmpty())
        setGeometry(200, 200, 400, 400);
    else
    restoreGeometry(geometry);
```

Запис і зчитування масивів значень (на прикладі comboBox і listWidget)

Створюється поточний масив `"combo"` методом `beginWriteArray`, по замовчуванню розмірність -1, тобто вона визначатиметься автоматично при запису записів, закінчення праці з цим масивом ознаменовано методом `endArray()`. Встановлюється поточний індекс методом `setArrayIndex(i)`, і тоді до даного елемена можна застосовувати методи `setValue()`, `value()`, `remove()`, and `contains()`

Аналогічно і по масиву `"list"`

```
void Dialog::save()
{
    QSettings settings("MyCompany","MyApp");
    settings.clear();

    settings.setValue("text",ui->comboBox->currentText());

    settings.beginWriteArray("combo");
    for(int i = 0; i < ui->comboBox->count(); i++)
    {
        settings.setArrayIndex(i);
        settings.setValue("item", ui->comboBox->itemText(i));
    }
    settings.endArray();

    settings.beginWriteArray("list");
    for(int i = 0; i < ui->listWidget->count(); i++)
    {
        settings.setArrayIndex(i);
        settings.setValue("item", ui->listWidget->item(i)->text());
    }
    settings.endArray();
}
```

```
void Dialog::load()
{


        QSettings settings("MyCompany","MyApp");


            int combosize = settings.beginReadArray("combo");
            for (int i = 0;i < combosize; i++)
            {
                settings.setArrayIndex(i);
                ui->comboBox->addItem(settings.value("item","").toString());
            }
            settings.endArray();

            int listsize = settings.beginReadArray("list");
```

```
            for (int i = 0;i < listsize; i++)
            {
                settings.setArrayIndex(i);
                ui->listWidget->addItem(settings.value("item","").toString());
            }
            settings.endArray();
            ui->comboBox->setCurrentText(settings.value("text","").toString());

}
```

Робота з файлами.

Загальна концепція використання файлів.

Всі пристрої введення / виводу в Qt успадковують від абстрактного класу QIODevice.Серед його нащадків: *буфер* для даних(QBuffer), процес - *програма* яка виконується в системі(QProcess), мережевий *сокет* (QAbstractSocket)та інші.

Для роботи з файлом необхідно створити *об'єкт* класу QFile і задати для нього *шлях* до файлу (абсолютний або відносний), з яким ви будете працювати. *Шлях* і ім'я передають як *параметр* конструктора або за допомогою методу setFileName().

Далі *файл* необхідно відкрити і поставити *режим доступу* до нього. Метод open() приймає прапори доступу і повертає true, якщо *файл* вдалося відкрити. Доступні прапори доступу:

- QIODevice::ReadOnly - відкрити для читання;
- QIODevice::WriteOnly - відкрити для запису;
- QIODevice::ReadWrite - відкрити для читання і запису;
- QIODevice::Append - всі дані будуть додаватися в кінець файлу (після вже існуючих даних);
- QIODevice::Truncate - якщо можливо, стерти вміст файлу перед відкриттям;
- QIODevice::Text - режим роботи з текстовим файлом (важливо для текстових файлів для коректної обробки символів завершення рядка в Windows і Linux).

Прапори*(клас* QFlags)часто використовують в Qt для завдання комбінації налаштувань. Для комбінації декількох налаштувань, так само як і бінарної арифметики, використовують операцію побітового OR.

Для запису і читання використовують методи read () і write(), які перевантажені в декількох варіантах. Для читання одного рядка текстового файлу використовують метод readLine(). Для читання всього вмісту можна скористатися методом readAll(). Поточну позицію при читанні з файлу визначають за допомогою методу pos(). Встановити позицію можна за допомогою методу seek(). Метод atEnd () дозволяє визначити чи досягли ми кінця файлу при читанні. Після завершення роботи з файлом його потрібно закрити викликом методу close().

```
QFile lFile; - створення об’єкту для роботи з файлом. (можна використати перевантажену форму задавши відразу ім’я файлу QFile lFile("file.txt");
lFile.setFileName (lFileName); - задання імені файлу на диску, можна вказати конкретно ім’я файлу без попередньої перевірки, як це показано в коді (lFile.setFileName ("file.txt");)
lFile.close (); - команда закриття файлу
```

Зверніть увагу, що дані відразу не записуються в файл на накопичувачі, вони записуються в буфер в оперативній пам'яті. Після закриття файлу дані з буфера записуються в файл на носії. Це зроблено для того, щоб не навантажувати жорсткий диск або будь-який інший тип накопичувача, на якому знаходиться файл. Інформацію з буфера в файл можна записати примусово без закриття файлу, викликавши метод QFile::flush ()

```
lFile.flush();
```

Для перевірки, чи файл відкритий можна використовувати конструкцію

```
QFile file("d:/temp/123.txt");
    file.open (QIODevice ::ReadOnly);
    if(file.isOpen())
    {
    qDebug() << "File is open";
    }
  else qDebug() << "File isn`t open";
```

Існує дуже корисний метод QFile :: exists (). Він приймає на вхід рядок з ім'ям файлу і повертає значення true, якщо такий файл існує. Існує статичний і нестатичних методи. Для роботи зі статичним методом необхідно вказати ім'я файлу.

```
if(QFile::exists("myfile.txt"))
    {
    qDebug() << " File %s does not exists ";
  }
```

Для нестатичного способу можна обійтись без імені файлу

```
if(QFile::exists())
    {
    qDebug() << " File %s does not exists ";
  }
```

Наступний приклад демонструє читання текстового файлу і *виведення* його в консоль.

```
const QString lFileName ( "d:/temp/123.txt");
            // Перевіряємо наявність файлу
            if (! QFile :: exists (lFileName))
            {
                qCritical ( "File %s does not exists.", qPrintable (lFileName));

            }
            QFile lFile;
            // Встановлюємо ім'я файлу
            lFile.setFileName (lFileName);
            // Відкриваємо файл - текстовий, тільки для читання
            if (! lFile.open (QIODevice :: ReadOnly | QIODevice :: Text))
            {
                // Якщо відкрити файл не вдалося - виводимо повідомлення про помилку
            qCritical ( "Error% d:% s.", lFile.error(), qPrintable(lFile.errorString()));

            }
            // Поки можна прочитати рядок
            while (! lFile.atEnd ())
            {
                // ... виводити її в консоль
                qDebug () << lFile.readLine ();
            }
            // Закінчуємо роботу з файлом
        lFile.close ();
```

В ще одному прикладі напишемо програму, яка зчитує з файлу блок з перших 10-ти символів, а потім вставляє в інший файл.

```
QFile fileIn ("file.txt");
         QFile fileOut ("fileout.txt");
         if (fileIn.open (QIODevice :: ReadOnly) && fileOut.open (QIODevice ::
WriteOnly))
         {// Якщо перший файл відкритий для читання, а другий для запису вдалий
             QByteArray block = fileIn.read (10); // Прочитуємо 10 байт в масив block з
filein.txt
             fileOut.write (block); // Записуємо 10 байт в файл fileout.txt
             fileIn.close (); // Закриваємо filein.txt
        fileOut.close (); // Закриваємо fileout.txt
```

Можна було зчитати всі байти, тоді весь вміст першого файлу копіювався б в другій.

Для повного зчитування рядок QByteArray block = fileIn.read (10);
Потрібно замінити на рядок QByteArray block = fileIn.readAll ();
В результаті програма зчитує всі байти в масив block, а після запише їх у другій файл.

Ми можемо записувати інформацію в файл рядками, для цього його потрібно відкрити в текстовому режимі.

```
fileOut.open (QIODevice :: WriteOnly | QIODevice :: Text)
```
Після передати адресу в конструктор нового об'єкта класу QTextStream.

```
QTextStream writeStream (&fileOut);
```
А далі за допомогою оператора << посилати рядки в потік записи.

Приклад програми, в яка записує в файл fileout.txt рядок «Text, text, text.»

```
QFile fileOut ( "fileout.txt"); // Зв'язуємо об'єкт з файлом fileout.txt
        if (fileOut.open (QIODevice :: WriteOnly | QIODevice :: Text))
        {// Якщо файл успішно відкритий для запису в текстовому режимі
            QTextStream writeStream (& fileOut); // Створюємо об'єкт класу QTextStream
    // і передаємо йому адресу об'єкта fileOut
            writeStream << "Text, text, text."; // Надсилаємо рядок в потік для запису
            fileOut.close (); // Закриваємо файл
    }
```

Нижче наведено приклад порядкового читання текстового файлу. (txtbr=new QTextBrowser(this);)

```
QFile fileOut ( "fileout.txt"); // Зв'язуємо об'єкт з файлом fileout.txt
        if (fileOut.open (QIODevice :: ReadOnly | QIODevice :: Text))
        {QString str = "";
            while (! fileOut.atEnd ())
            {
                str = str + fileOut.readLine ();
            }
            txtbr-> setText (str);
      fileOut.close ();
```

Такий код по замовчуванню, не буде читати кирилицю, оскільки в віндовс по замовчуванню стоїть UTF-8, виправити дану ситуацію зможе наступний код:

```
QTextCodec* defaultTextCodec = QTextCodec::codecForName("Windows-1251");
    QTextDecoder *decoder = new QTextDecoder(defaultTextCodec);

    QFile fileOut ( "fileout.txt"); // Зв'язуємо об'єкт з файлом fileout.txt
        if (fileOut.open (QIODevice :: ReadOnly | QIODevice :: Text))
        {QString str = "";
            while (! fileOut.atEnd ())
            {
                str = str + decoder->toUnicode(fileOut.readLine());
            }
            txtbr-> setText (str);
      fileOut.close ();
```

!!!Або зберігати файл в кодуванні UTF-8!!!

Якщо необхідно прочитати весь файл цілком, можна значно спростити реалізацію скориставшись методом readAll. (txtbr=new QTextBrowser(this);)

```
QFile fileOut ( "fileout.txt"); // Зв'язуємо об'єкт з файлом fileout.txt
        if (fileOut.open (QIODevice :: ReadOnly | QIODevice :: Text))
        {txtbr->setText(fileOut.readAll());
            fileOut.close (); // Закриваємо файл
    }
```

Розглянемо роботу з файлами в Qt на іншому прикладі запису і читання текстової інформації. У цьому прикладі використовується *клас* QTextStream для отримання введеної користувачем інформації в *стандартний потік* введення. К*онструктор* QTextStream може приймати в якості параметра *вказівник* на *нащадок* QIODevice, *вказівник* на QString або QByteArray, а також файлову змінну. У прикладі ми

перенаправляємо *потік* введення в QTextStream. Далі ми читаємо рядок з потоку введення і записуємо її в файл.

```
#include <QDebug>
#include <QIODevice>
#include <QFile>
#include <QTextStream>
int main (int argc, char *argv[])
{
        QTextStream in (stdin);
        QFile lFile ("in.txt");
        if (lFile.open (QIODevice :: WriteOnly | QIODevice :: Truncate))
        {
            QString lData = in.readLine ();
            lFile.write (qPrintable (lData));
            lFile.close ();
        }
        else
        {
            qDebug () << "Can not open file!";
        }
        return 0;
}
```

У наступному фрагменті демонструємо *зворотний* процес - читання рядки з файлу і *виведення* прочитаного рядка в *консоль*. Знову ж для читання з файлу ми використовуємо QTextStream.

```
QFile lFile ( "in.txt");
    if (lFile.open (QIODevice :: ReadOnly | QIODevice :: Truncate))
    {
            QTextStream in (&lFile);
            QString lData = in.readLine ();
            qDebug () << lData;
            lFile.close ();
    }
    else
    {
            qDebug () << "Can not open file!";
    }
```

Запис в кінець файлу

Попередній метод повністю перезаписував дані у файлі, тобто очищав весь його вміст і записував нові дані. Перезапису можна уникнути і записувати нові дані в кінець файлу.
Прапор QIODevice :: Append поміщає покажчик для запису (seek) в кінець файлу, в результаті вхідний потік записується відразу після наявною інформацією в файлі. Приклад фрагмента використання:

```
fileOut.open (QIODevice :: Append | QIODevice :: Text);
```

Читання і запис окремого рядка

Найскладнішими і водночас не оптимізованими операціями при роботі з текстовими файлами в Qt, є читання і запис однієї конкретної рядки. Справа в тому, що QFile не підтримує навігацію по рядках і отже звернутися безпосередньо до заданої рядку не можна.
Дане незручність в принципі можна вважати своєрідною платою за універсальність, але суті це не міняє.
Щоб отримати доступ до необхідної рядку для читання необхідно скористатися одним з двох способів:
Попередньо «прочитати» всі попередні рядки.

```
int i = 0;
    while ((i <3) && (! file.atEnd ()))
```

```
    {
        file.readLine ();
        i ++;
    }
txtbr -> setText (file.readLine ());
```

Прочитати вміст файлу в контейнер, який підтримує доступ за індексом і отримувати потрібні дані вже з нього.

```
QStringList strList;
    while (! file.atEnd ())
    {
        strList << file.readLine ();
    }
  txtbr -> setText (strList [3]);
```

При використанні контейнерів слід обов'язково читати текстовий файл через підрядник. У разі читання за допомогою методу readAll символи переходу рядка ігноруються і вміст файлу буде завантажено в контейнер цілком у вигляді одного єдиного елемента.

Запис окремого рядка найкраще робити за допомогою контейнера.

Для цього необхідний спочатку прочитати файл як було показано вище, а потім вставити рядок на потрібну позицію в контейнері і записати його вміст назад в файл.

Як приклад можна привести додавання нового рядка між другою і третьою рядками вихідного файлу.

```
  QStringList strList;
    QFile file ( "fileout.txt");
        /* Прочитуємо вихідний файл в контейнер */
        if ((file.exists ()) && (file.open (QIODevice :: ReadOnly)))
        {
            while (! file.atEnd ())
            {
            strList << file.readLine ();
            }
            file.close ();
        }
        /* Додаємо рядок і зберігаємо вміст контейнера в той же файл */
        if ((file.exists ()) && (file.open (QIODevice :: WriteOnly)))
        {
            strList.insert (2, "Inserted string \n");
            QTextStream stream (& file);
            for (QString s: strList) //foreach(QString s, strList)
            {
                stream << s;
            }
            file.close ();
    }
```

Аналогічним способом можна відредагувати будь-яку з уже наявних рядків:

```
strList [3] = "Test editing \ n";
```

Або видалити:

```
strList.removeAt (3);
```

Ідеальне рішення - правка вмісту текстового файлу в віджеті для редагування (наприклад, Plain Text Edit), але, як відомо далеко не завжди текстові файли служать для потреб користувача. Дуже часто їх доводиться використовувати для зберігання службової інформації пов'язаної з роботою програм.

Тому незважаючи на всю складність подібних маніпуляцій програмісти змушені іноді до них вдаватися.

Робота з UI-компонентами (Text Edit, Plain Text Edit, Text Browser)

Для роботи з файлами можна створити об’єкт файлу, як компонент класу, або створювати його в конструкторі. В конструкторі головного класу створимо об’єкт файлу, та перевіримо чи правильно прикріпився файл:

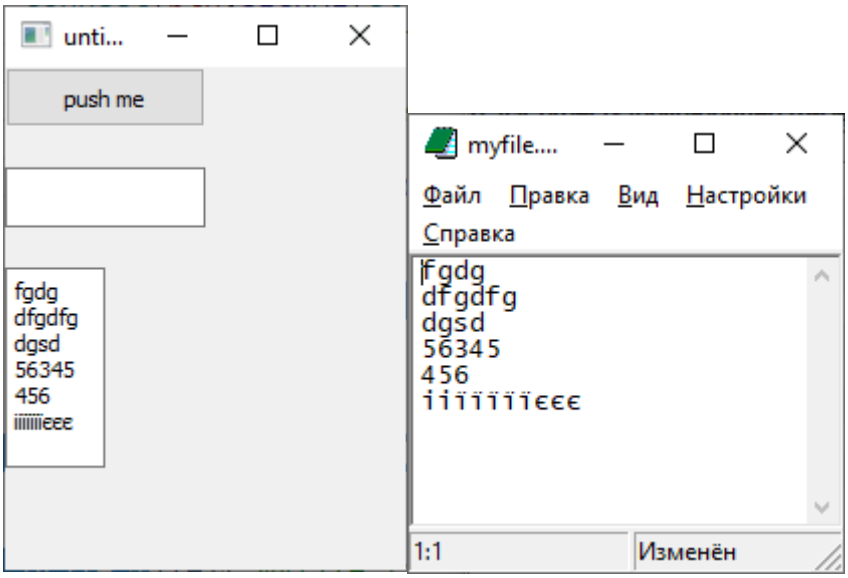
```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QFile file("myfile.txt");
    file.open (QIODevice :: ReadOnly| QIODevice :: Text);
    if(file.isOpen() )
    {
    qDebug() << "File is open";
    } else  qDebug() << "File is not open";
}
```

Весь вміст файлу можна побачити вивівши його в віджет **textBrowser**:
в mainwindow.h: QTextBrowser *txtbr;
в mainwindow.cpp: txtb->setText(file.readAll());
якщо тільки перший рядок, то можна використати наступну команду:
txtb ->setText(file.readLine());

Для запису інформації в файл можна скористатись елементом **textEdit:**

Код слоту нажаття на кнопку: (txtedit=new QTextEdit(this);)

```
void MainWindow::btn_click()
{

    QFile file1("myfile.txt");
        file1.open (QIODevice :: WriteOnly| QIODevice :: Text);
        if(file1.isOpen() )
        {
        qDebug() << "File1 is open";
        } else  qDebug() << "File1 is not open";
    QTextStream stream (&file1);
       stream<<txtedit->toPlainText();
       file1.close();}
```

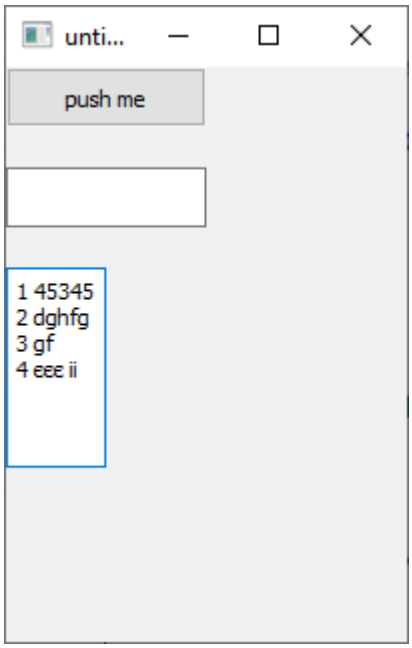


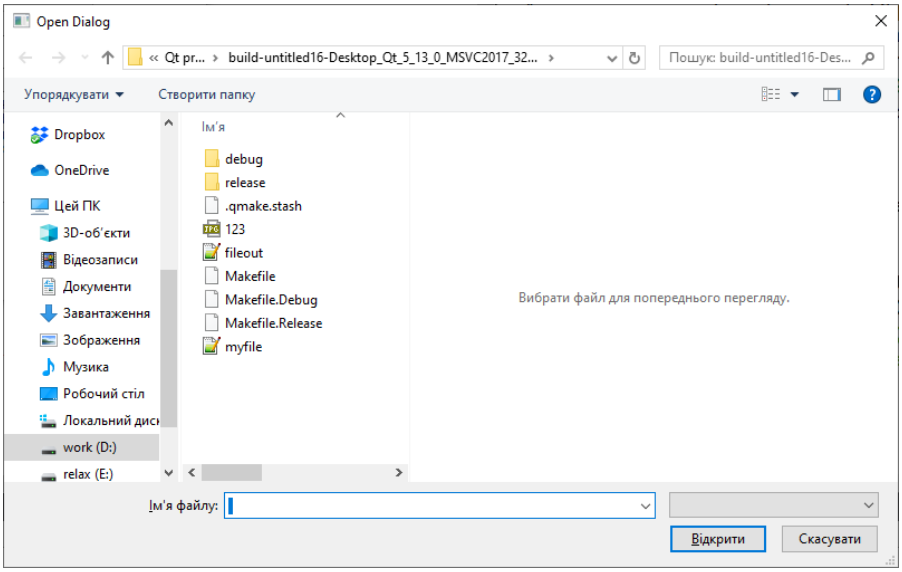
textBrowser - елемент з класу QtextBrowser, який дозволяє здійснювати навігацію по документах. Даний елемент може відображати документи в HTML коді. (QTextBrowser::setHtml(QString text);)

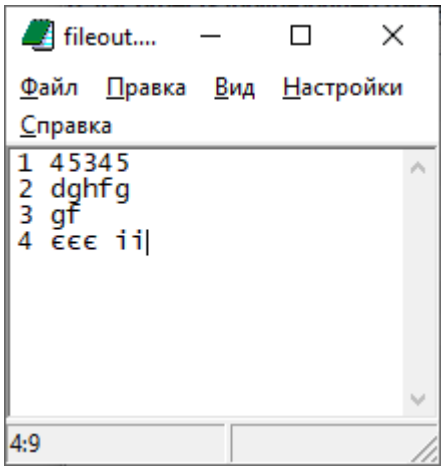
QPlainTextEdit оптимізований для обробки простого тексту та може використовуватися навіть із дуже великими текстовими файлами. Якщо планується обробка лише звичайних текстів, то QPlainTextEdit - найоптимальніший.

Можна скористатись і діалоговими вікнами для навігації по файловій системі

```
void MainWindow::btn_click()
{
QString str = QFileDialog::getOpenFileName (0, "Open Dialog"," ","   ");
    QFile file1(str);
        file1.open (QIODevice :: WriteOnly| QIODevice :: Text);
        if(file1.isOpen() )
        {
        qDebug() << "File1 is open";
        } else  qDebug() << "File1 is not open";
    QTextStream stream (&file1);
       stream<<txtedit->toPlainText();
       file1.close();
}
```







Запис в файл

Клас `QTextDocumentWriter` надає три формати для запису вмісту об'єкта класу `QTextDocument` в PlainText (Простий текст), HTML і ODF (OpenDocument Format, відкритий формат документів для офісних додатків). Останній формат використовується багатьма додатками, включаючи і OpenOffice.org. Для того щоб записати файл в потрібному форматі, необхідно передати рядок з форматом в метод `setFormat()`. Наприклад, для запису в ODF- формат програмний код може бути наступним (лістинг 10.3):

Лістинг 10.3. Запис в форматі ODF

```
QTextEdit* ptxt = new QTextEdit("This is a <b>TEST</b>");

QTextDocumentWriter writer;

writer.setFormat("odf");

writer.setFileName("output.odf");

writer.write(ptxt->document());
```

У лістингу 10.3 ми створили віджети редактора тексту (курсор `ptxt`) і об'єкт підтримки запису (`writer`). Потім ми встановили викликом методу `setFormat()` потрібний нам формат ODF і задали ім'я файлу викликом методу `setFileName()`. Виклик методу `write()` виконує запис в файл, цей метод приймає як параметр покажчик на об'єкт класу `QTextDocument`.

Запис в формат PDF класом `QTextDocumentWriter` не підтримується, але його легко здійснити шляхом малювання в контексті `QPrinter`. Ось невеликий приклад, як це можна зробити (лістинг 10.4)

Лістинг 10.4. Запис в форматі PDF

```
QTextEdit* ptxt = new QTextEdit("This is a <b>TEST</b>");

 QPrinter printer(QPrinter::HighResolution);

printer.setOutputFormat(QPrinter::PdfFormat);

printer.setOutputFileName("output.pdf");

ptxt->document()->print(&printer);
```

У лістингу 10.4 ми створюємо віджет текстового редактора (курсор `ptxt`) і об'єкт принтера (`printer`) і встановлюємо його при створенні в режим високого дозволу `HighResolution`. Далі ми встановлюємо формат для виводу, викликом методу `setOutputFormat()`, в який передаємо значення `PdfFormat`, можливі також варіанти `NativeFormat` і `PostScriptFormat` для друку в системний принтер або в запис у форматі PostScript відповідно. Потім за допомогою методу `setOutputFileName()` ми задаємо ім'я файлу для запису і викликаємо з об'єкта класу `QTextDocument` метод, в який передаємо в нього адресу на наш об'єкт принтера. Ця операція здійснює запис в призначений нами файл. Додатково потрібно в файлі `*.pro` прописати додаткову вказівку компілятору QT += printsupport, і за потреби підключити заголовочний файл #include <QPrinter>

Таблиці

Клас QTableWidget є таблицею. Об'єкт клітинок реалізований в класі QTableWidgetItem. Створитти осередок можна викликом методу QTableWidget :: setItem (). Першим параметром методу setItem () є номер рядка, а другим - номер стовпчика. Таким чином, ці параметри задають місце розташування осередку в таблиці. Встановити текст в самій комірці можна за допомогою методу QTableWidgetItem :: setText (), а для розміщення растрового зображення можна скориста тися методом QTableWidgetItem :: setIcon (). Якщо в осередку встановлені як текст, так і растрових зображень, то растрове зображення займе місце зліва від тексту. Клас осередку QTableWidgetItem надає конструктор копіювання, що дозволяє створювати копії елементів. Також для цієї мети можна скористатися методом clone (). У таблицю допускається, крім тексту і іконок, поміщати і віджети. Для цього використовується метод setCellWidget

().

При подвійному натисканні кнопкою миші на поле осередку вона переходить в режим редагування, при цьому використовуючи віджет QLineEdit.

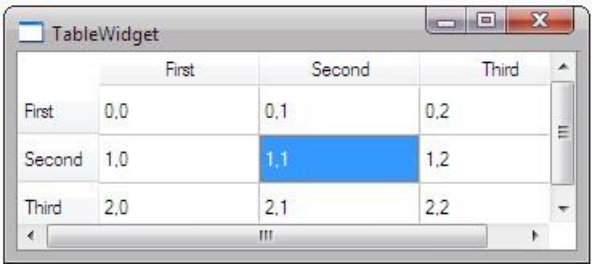


Рис. 11.5. Таблиця

Додаток, показаний на рис. 11.5, використовує таблицю, що складається з трьох стовпців і трьох рядків. Вміст осередків можна змінювати.

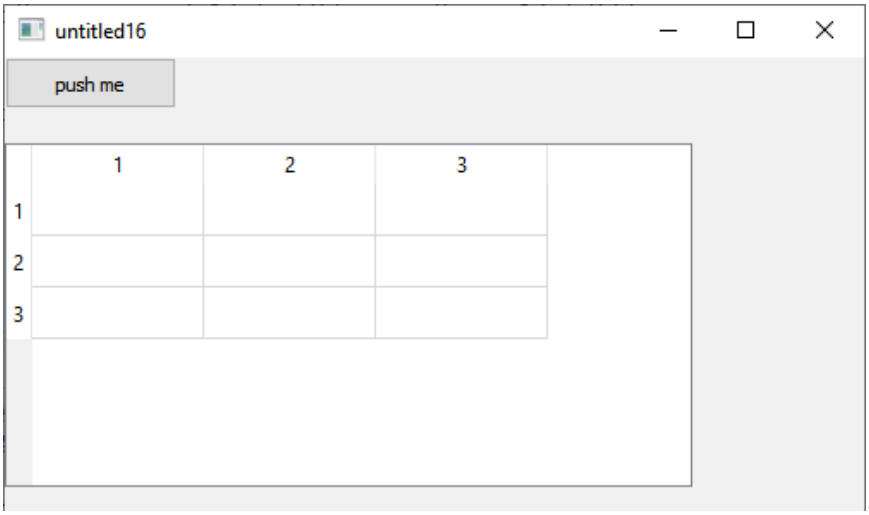
Листинг 11.6. Файл mainwindow.cpp

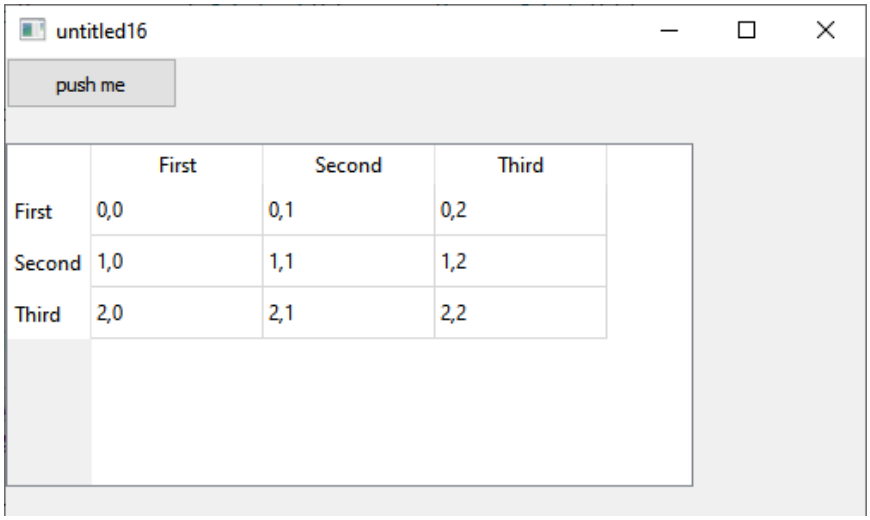
```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    btn=new QPushButton("push me",this);
    connect(btn,SIGNAL(clicked()),SLOT(btn_click()));

tblwdg=new QTableWidget(3,3,this);
tblwdg->setGeometry(0,50,400,200);
}

void MainWindow::btn_click()
{
    QTableWidgetItem* ptwi;
    QStringList        lst;
    lst << "First" << "Second" << "Third";
    tblwdg->setHorizontalHeaderLabels(lst);
    tblwdg->setVerticalHeaderLabels(lst);
    for (int i = 0; i < 3; ++i)
    {  for (int j = 0; j < 3; ++j) {
    ptwi = new QTableWidgetItem(QString("%1,%2").arg(i).arg(j));
    tblwdg->setItem(i, j, ptwi);
    } }

}
```

У лістингу 11.6 створюється віджет таблиці tblwdg. Перший і другий параметри, що передаються в конструктор, задають кількість рядків і стовпців. Наша таблиця буде мати розмірність 3 на 3. Щоб задати заголовки рядків і стовпців таблиці (які у нас збігаються), ми спочатку формуємо їх список, а потім передаємо його в метод

setHorizontalHeaderLabel () для горизонтальних і в setVerticalHeaderLabel () для вертикальних заголовків.

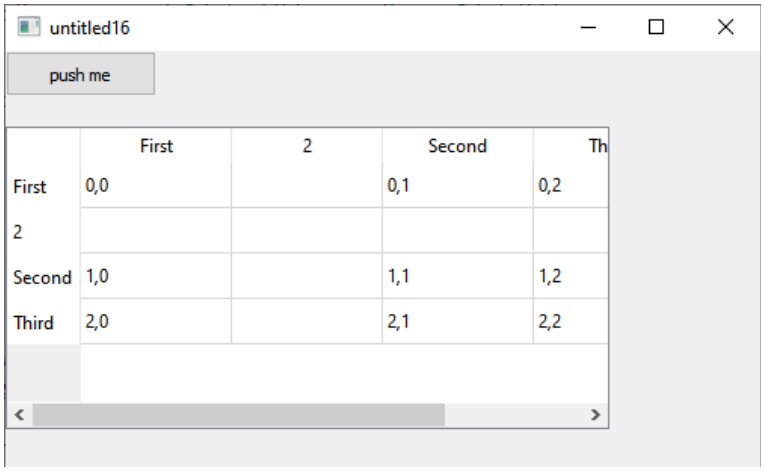
Створення об'єктів осередків виконується в циклі (вказівник ptwi). Як текст в конструктор передаються номера рядка і стовпчика комірки. Викликом методу setItem () створюється осередок таблиці встановлюється в позицію, зазначену в першому і другому параметрах.





Вставити рядок

`tblwdg->insertRow(int);` tblwdg->insertColumn(int); int – номер позиції куди буде вставлено рядок/стовпчик



Внесення даних в клітинку передбачає процедуру створення об’єкту QTableWidgetItem аргументом якого може бути рядок QString, і потім безпосередньо запис інформації в клітинку методом setItem

```
QTableWidgetItem *qq = new QTableWidgetItem("Привіт");
  tblwdg->setItem(0,0,qq);
```

або

```
QTableWidgetItem *qq = new QTableWidgetItem();
 qq->setText("f Привіт");
```

```
tblwdg->setItem(0,0,qq);
```

Для того щоб записати інформацію з конкретної клітинки, в текстовому форматі можна скористатись записом
`tblwdg ->item(row, col)->text();`

Для числового запису можна скористатись записом
```
tblwdg ->item(row, col)->data(Qt::UserRole).toInt();
```

rowCount()	кількість рядочків, int значення	`tblwdg ->rowCount();`
columnCount()	кількість стовпчиків, int значення	`tblwdg ->columnCount();`

Для запису з файлу в таблиці (кожен рядок в окрему клітинку
```
QList <QString>strList; int i=0;
        QString str = QFileDialog::getOpenFileName (0, "Open Dialog"," ","  ");
        QFile fileIn (str);
        fileIn.open (QIODevice :: ReadOnly| QFile :: Text);

        while (!fileIn.atEnd())
                {
                strList << fileIn.readLine (); //зчитуємо всі рядки з файлу в список
                }
        fileIn.close();


    for (QString s: strList) // записуємо кожен елемент списку(рядочок з файлу) в першу
клітинук рядочку таблиці
     {QTableWidgetItem *qq = new QTableWidgetItem(""); // мусить бути для кожного
елемента окремо виділена пам'ять
      qq->setText(s);
       tblwdg ->setItem(i,0,qq);i++;

    qDebug()<<s<<i-1;
  }
```

Модуль QtSql – модуль призначений для роботи з базами даних. В нього входять класи, які дають можливість маніпулювати базами даних.

Бази даних SQL

База даних є системою збереження записів, організованих в таблицю. База даних може містити від одного до кількох сотень таблиць, які бувають зв'язані між собою. Таблиця складається з набору рядків і стовбців (Рис. 1). Стовбці мають імена і за ним закріплено тип даних або область значень. Рядки таблиці баз даних називаються записами, а клітинки – поля. Початковий ключ – це унікальний ідентифікатор запису, який може бути як одним стовпцем так і комбінацією стовпців.

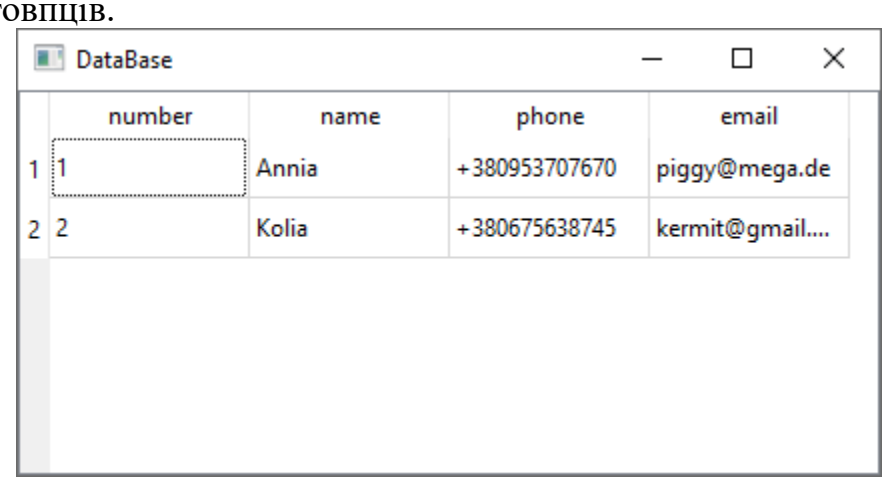


Рис. 1 – Приклад таблиці

Основними діями , які можна виконувати з базами даних є: створення нової таблиці, зчитування, вставка, змінення і видалення даних. Особливі можливості Qt SQL підтримують більш розгорнутий синтаксис для різних команд, не говорячи про конкретні СУБД (системи управління базами даних), виробники яких зазвичай представляють розширений синтаксис для найкращого використання їх особливостей. Одразу варто сказати, що мова SQL нечутлива до регістру ("SELECT", "select", "Select" і т. д. в мові SQL означають одне і те ж).

Створення таблиці

Для створення таблиці, показаної на рис. 1, використовується команда CREATE TABLE, в якій вказуються імена стовбців і їх тип, а також задається початковий ключ (при потребі):

```
QString str = "CREATE TABLE addressbook ( "
                  "number INTEGER PRIMARY KEY NOT NULL, "
                  "name VARCHAR(15), "
                  "phone VARCHAR(15), "
                  "email VARCHAR(15) "
                  ");";
```

Операція вставки

Після створення таблиці мажна добавляти дані. Для цього в SQL є команда вставки INSERT INTO. Зразу після назви таблиці потрібно вказати в лапках імена

стовбців, в які будуть вноситися дані. Самі ж дані вказуються після ключового слова VALUES:

```
query.prepare("INSERT INTO addressbook (number, name, phone, email)"
              "VALUES(?, ?, ?, ?);");
```

Якщо якась частина цих даних ще невідома на їх місце ставиться «?».

Зчитування даних

Зчитування складається з команд SELECT ... FROM ... WHERE виконує операцію вибірки і проекції. Вибірка відповідає вибору рядка, а проекція – вибору стовбців. Ця команда повертає таблицю з частиною початкових даних, створену згідно з заданими критеріями. В цій команді три основні частини:

1. Ключове слово SELECT є командою для проведення проекції, тобто після нього вказуються стовбці, які повинні стати відповіддю на запит. Якщо після SELECT вказати знак «*», то результуюча таблиця буде вміщувати всі стовбці. Вказані конкретні імена залишає в проекції тільки ті стовбці
2. Після ключового слова FROM задається таблиця, до якої адресовано запит
3. Ключове слово WHERE є оператором вибірки. Вибірка виконується відповідно до умов вказаних зразу ж після оператора. Цю частину команди можна упускати, але як наслідок будуть вибрані всі стовбці. Например, для получения адреса электронной почты мисс Piggy нужно сделать следующее:

Використання мови SQL в бібліотеці Qt

Для використання баз даних в бібліотеках Qt представлено окремий модуль QtSql. Для його використання необхідно оповістити про це – в проектний файл потрібно додати наступний рядок:

`QT += sql` (або ж дописати до вже існуючого «sql» `QT += core gui sql`)

А для того, щоб мати можливість працювати з класами цього модуля необхідно включити матафайл QtSql `#include <QtSql>`. Класи цього модуля розділяються на 3 рівня:

1. Рівень драйверів.
2. Програмний рівень
3. Рівень користувацького інтерфейсу

До першого рівня відносяться класи для отримання даних на фізичному рівні. Це такі класи як QSqlDriver, QSqlDriverCreator, QSqlDriverCreatorBase, QSqlDriverPlugin і QSqlResult. Класи другого рівня є програмним інтерфейсом для звернення до бизи данихю До цього рівня належать: QSqlDatabase, QSqlQuery, QSqlError, QSqlField, QSqlIndex і QSqlRecord. Третій рівень представляє моделі для відображення результатів запитів. До цих класів відноситься: QSqlQueryModel, QSqlTableModel і QSqlRelationalTableModel.

Класи першого рівня вам не прийдеться використовувати, якщо ви не збираєтеся писати свій власний драйвер для менеджера бази даних.

Ідентифікатор	Опис
QOCI	Доступ до баз даних Oracle через Oracle Call Interface (OCI). Підтримуються версії 7, 8 і 9
QoDBC	ODBC (Open Database Connectivity, відкритий інтерфейс доступу до бази даних) - стандартний ODBC-драйвер для Microsoft SQL Server. IBM DB2. Sybase SQL. iODBC та інших баз даних
QMYSQL	MySQL - найпопулярніший в даний час безкоштовний менеджер бази даних. Всю необхідну інформацію можна отримати на сторінці www.mysql.com
QTDS	Sybase Adaptive Server
QPSQL	Бази даних PosgreSQL з підтримкою SQL92 / SQL3
QSQLITE2	SQLite версії 2
QSQLITE	SQLite версії 3
QIBASE	Borland InterBase
QDB2	DB / 2 - платформонезалежна база даних, розроблена IBM

Рис. 2 – Приклад драйверів

В більшості випадків всі обмежуються використанням вже існуючих драйверів (Рис. 2) конкретної СУБД, яка підтримується в Qt. Наданий момент існує декілька десятків СУБД.

З'єднання з базою даних (другий рівень)

Для з'єднання з базою даних перш за все необхідно активізувати драйвер. Для цього викликається статичний метод `QSqlDatabase::addDatabase()`. В нього необхідно передати рядок з ідентифікатором драйвера СУБД. `QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE")` (Лістинг. 3):

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("MyDB");
db.setUserName("I_am");
db.setHostName("localhost");
    db.setPassword("12323");
          if(db.open())   qDebug()<<"open"; else qDebug()<<"error";
```

Лістинг. 3 – Функція з'єднання з базою даних

Як правило, код з лістингу 3 вписують в статичну булеву функцію:

```
static bool createConnection () {
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("MyDB");
db.setUserName("I_am");
db.setHostName("localhost");
    db.setPassword("12323");
```

```
    if(db.open())  {qDebug()<<"open"; return true;} else {qDebug()<<"error";
return false;}
       }
```

Для того щоб підключитися до бази даних потрібно чотири наступних параметри:

1. Ім'я бази даних – передается в метод `QSqlDatabase:: db. setDatabaseName ();`
2. ім'я користувача – передається в метод `QSqlDatabase:: setUserName();`
3. ім'я комп'ютера – передається в метод `QSqlDatabase:: setHostName();`
4. пароль – передается в метод `QSqlDatabase::setPassword()`.

Методи повинні викликатися із об'єкта, створеного з допомогою статичного метода `QSqlDatabase::addDatabase()`. Саме з'єднання виконується методом `QSqlDatabase::open()`. У випадку виникнення помилки, інформацію про неї можна отримати з допомогою методу `QSqlDatabase::lastError()`, який повертає об'єкт класу QSqlError. Саму помилку можна вивести на екран з допомогою методу qDebug().

Можна переглянути наявні таблиці в базі даних до якої підключились, використовуючи метод tables():

```
QStringList lst = db.tables();
foreach (QString str, lst) {
qDebug() << "Table:" << str;}
```

Для виконання команд SQL після встановлення з'єднання можна використовувати клас QSqlQuery. Запити оформляються у вигляді звичайного рядка, який передається в конструктор або в метод `QSqlQuery::exec()`. При передачі запиту в конструктор запуск команди буде виконуватися автоматично при створенні об'єкта.

Клас QSqlQuery надає можливість навігації. Наприклад, після виконання запиту SELECT можна переміщатися по відібраних даних за допомогою методіцв `next()` – наступний рядок, `previous()` – попередній рядок, `first()` – перший рядок, `last()` – останній рядок і `seek()` – встановлює поточному рядку даних цілочисельний індекс, який вказано в параметрі. Кількість рядків даних можна отримати викликом методу `size()`.

Складності виникають із запитом `INSERT`, оскільки потрібно додавати дані. Для цього запиту користуються двома методами `prepare()` і `bindValue()`. В методі `prepare()` ми задаємо шаблон, данні в який представлені методами `bindValue()`

```
QSqlQuery query;
    query.prepare("INSERT INTO addressbook (number, name, phone, email)"
    "VALUES(:number, :name, :phone, :email);");
    query.bindValue(":number ", 1);
    query.bindValue (": name", "Yurii");
    query.bindValue(":phone", "+380676644318");
     query.bindValue (" :email", "yurii@cym.org");
```

```
query.exec();
```

Лістинг. 4 – Додавання даних.

Можна також використовувати безіменні параметри:

```
query.prepare("INSERT INTO addressbook (number, name, phone, email)"
"VALUES (?, ?, ?, ?) ; ");
query. addBindValue (2);
query. addBindValue ( "Piggy");
query. addBindValue ("+49 631322187");
      query. addBindValue ( "piggy@mega.de");
query.exec();
```

Третій спосіб відразу прописати дані для введення:

```
QSqlQuery query("INSERT INTO addressbook (number, name, phone, email)"
      "VALUES (1, iliash, '+3807777777', 'yurii@cym.org')");
query.exec();
```

У випадку вдалого з'єднання з базою даних з допомогою методу `createConnection()` створюється рядок , який містить в собі команду SQL для створення таблиці. Цей рядок передається в метод `exec()` об'єкту класу QSqlQuery. Якщо створити таблицю не вийде, то на консоль буде виведено попередження.

Є і четвертий варіант – можна скористатися класом QString, а саме, методом QString::arg(), з допомогою якого з'являється можливість виконати підстановку значень даних (Рис 5.).

```
QString strF = "INSERT INTO addressbook (number, name, phone,
email) "
    "VALUES(%1, '%2', '%3', '%4');";
   QString str = strF.arg("5")
    .arg ( "Piggy")
    .arg("+49 631322187")
    .arg("piggy@mega.de");

         query.exec(str);
```

Лістинг 5 – Викоистання методу QString::arg()

Коли база даних створена, і всі дані внесені в таблицю, виконується запит SELECT, що поміщає всі рядки і стовпці таблиці в об'єкт query. Виведення значень таблиці на консоль здійснюється в циклі. При першому виклику методу next () цей об'єкт вказує на найпершу рядок таблиці. Наступні виклики приведуть до переміщення покажчика на наступні рядки. У тому випадку, якщо записів більше немає, метод next () поверне значення false, що призведе до виходу з циклу.

Для отримання результату запиту слід викликати метод QSqlQuery:: value (), в якому необхідно передати номер стовпчика. Для цього ми скористаємося методом record (). Цей метод повертає об'єкт класу QSqlRecord, який містить інформацію, яка відноситься до запиту SELECT. Викликаючи метод QSqlRecord :: indexOf (), ми отримуємо індекс стовпця.

Метод value () повертає значення типу QVariant. QVariant - це спеціальний клас, об'єкти якого можуть містити в собі значення різних типів. Тому в нашому прикладі отримане значення потрібно перетворити до необхідного типу, скориставшись методами QVariant:: toInt () і QVariant:: toString ().

```
if ( !query.exec("SELECT * FROM addressbook; " ) ){
    qDebug() << " Unable to execute query - exiting11" ;
    } else qDebug() << " Select connect";
    //------
    QSqlRecord rec = query.record();
    int nNumber = 0;
    QString strName;
    QString strPhone;
    QString strEmail;

    while (query.next()){
    nNumber =query.value(rec.indexOf( "number" )).toInt();
    strName =query.value(rec.indexOf( "name" )) .toString();
    strPhone= query.value(rec.indexOf( "phone" )) .toString();
    strEmail =query.value(rec.indexOf( "email" )) .toString();
    qDebug() << nNumber << " " << strName << " ;\t"
         << strPhone << " ;\t" << strEmail;}
```

або

```
if ( !query.exec("SELECT * FROM addressbook; " ) ){
    qDebug() << " Unable to execute query - exiting11" ;
    } else qDebug() << " Select connect";
    //------
    int nNumber = 0;
    QString strName;
    QString strPhone;
    QString strEmail;

    while (query.next()){
    nNumber =query.value(0).toInt();
    strName =query.value(1) .toString();
    strPhone= query.value(2) .toString();
    strEmail =query.value(3) .toString();
    qDebug() << nNumber << " " << strName << " ;\t"
                    << strPhone << " ;\t" << strEmail;}
```

Тепер повернемося до об'єкту класу QSqlRecord. Важливо розуміти, що в коді вище він просто містить копію реального запису, і якщо в базі станеться будь-яка зміна, то значення копії залишиться незмінним. У разі коли, наприклад, про структуру самого запису ми нічого б не знали, то могли б викликати метод QSqlRecord:: count () який поверне кількість полів в записі і за допомогою методу QSqlRecord :: fieldName () послідовно опитати всі імена полів запису, передавши йому чисельний індекс. Звертатися до кожного полю окремо можна також на ім'я, передавши його в метод QSqlRecord :: field (). Цей метод повертає об'єкт

класу QSqlField, який дасть всю необхідну інформацію про основні характеристики запису, таких як, наприклад, ім'я (метод QSqlField:: name ()), тип (метод QSqlField:: type ()), довжина (метод QSqlField :: length ()) і значення (метод QSqlField:: value ()). Перевірити ж існування в запису поля з певним ім'ям можна за допомогою методу QSqlRecord:: contains ().

Класи SQL-моделей для інтерв'ю (третій рівень)

Модуль QtSql підтримує концепцію інтерв'ю. Використання інтерв'ю – це найпростіший спосіб відобразити дані таблиці. В такому випадку немає необхідності використання циклів для проходження порядках таблиці. Бібліотека Qt представляє три різних моделі – це модель запиту і таблична модель.

Модель запиту

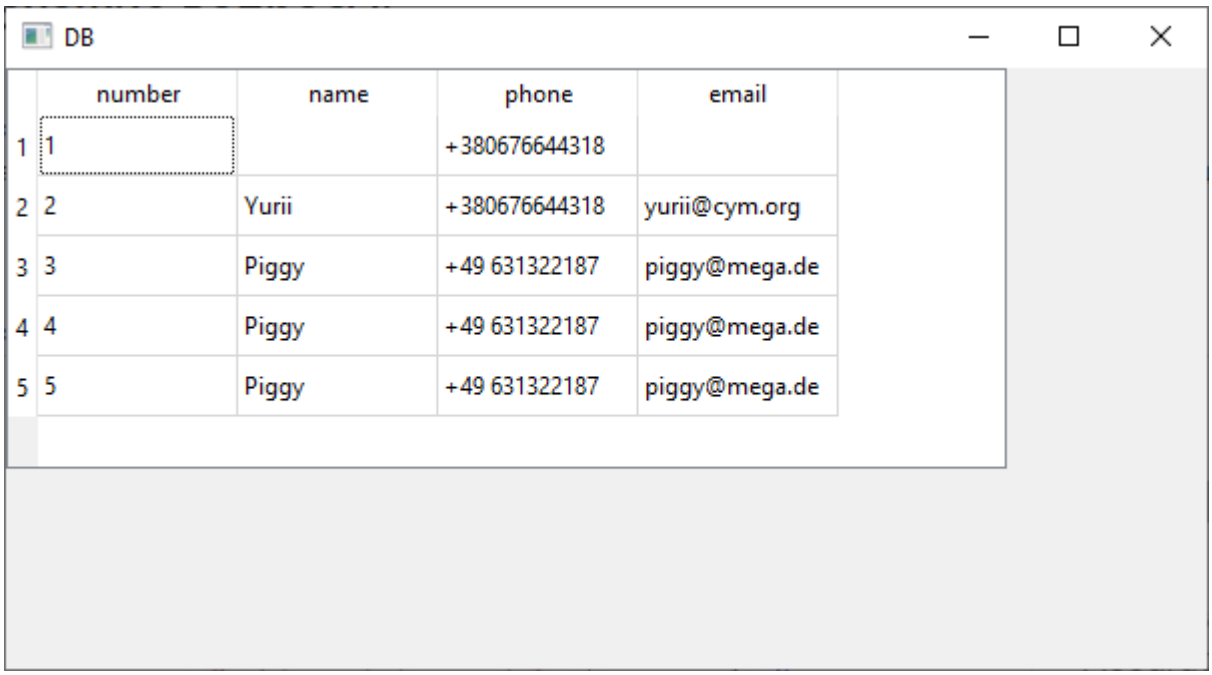
Якщо вам необхідно відобразити дані якого-небудь конкретного запиту SELECT, то для цього можна скористатися класом QSqlQueryModel. Ця модель призначена тільки для зчитування даних і з її допомогою ви можете швидко відображати результати запитів без можливості редагування.

З конструктора форми mainwindow.cpp:

```
QTableView *view = new QTableView(this); view-
>setGeometry(0,0,500,200);
        QSqlQueryModel *model = new QSqlQueryModel;
        model->setQuery("SELECT * FROM addressbook; " );
        if (model->lastError().isValid()) {
           qDebug() << model->lastError();
        }
            view->setModel(model);
```

З main.cpp

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setGeometry(0,0,600,300);
    w.show();

    return a.exec();
      }
```



Таблична модель

Клас табличної моделі QSqlTableModel успадковується від попереднього. Він має всі його можливості, дозволяє працювати з талблицями баз даних на більш високому рівні. Крім того дає можливість редагувати дані. Відображення таблиці бази даних відбувається повністю, а якщо необхідно обмежити стовбці для відображення то необхідно передати необхідні назви стовбців в методи `removeColumn()`.

У лістингу 41.4 після з'єднання з базою даних, створюються об'єкт табличного представлення QTableView І об'єкт табличній моделі QSqlTableModel. ВИКЛИКОМ методу setTable() ми встановлюємо в моделі актуальну базу. Виклик методу select() виконує заповнення даними.

Лістинг 41.4. Використання табличній моделі - функція main () (файл main.cpp)

```
int main (int argc, char ** argv)
{QApplication app (argc, argv);
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("MyDB");
db.setUserName("I_am");
db.setHostName("localhost");
    db.setPassword("12323");
    if(db.open())  {qDebug()<<"open"; } else {qDebug()<<"error"; }


QTableView view;
QSqlTableModel model;
model.setTable ( "addressbook");
model.select ();
model.setEditStrategy (QSqlTableModel :: OnFieldChange);
view.setModel (&model);
view.show ();
```

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

При використанні головного вікна, та розміщеного на ньому віджету таблиці, потрібно оголошувати динамічні вказівники на об'єкти (код прописаний в конструкторі головного вікна):

```
QTableView *view = new QTableView(this); view->setGeometry(0,0,500,200);

        QSqlTableModel *model=new QSqlTableModel;
        model->setTable ("addressbook");
        model->select ();
        model->setEditStrategy (QSqlTableModel :: OnFieldChange);
           view->setModel (model);
```

Тепер настав час трохи розповісти про можливості редагування та запису даних. Клас QSqlTableModel надає для цього три наступні стратегії редагування, які встановлюються за допомогою методу setEditStrategy ():

♦ onRowChange - запис даних виконується, як тільки користувач перейде до іншої рядку таблиці;

♦ OnFieidchange- запис даних відбувається після того, як користувач перейде до іншої клітинки таблиці;

♦ OnManualSubmit - дані записуються при виклику слота submitAll (). Якщо викликається слот revert All(), то дані повертаються в початковий стан.

У лістингу 41.4 використовується стратегія QSqlTableModel::OnFieldChange, викликаючи метод setEditStrategy() з цим аргументом. Тепер дані нашої моделі можна змінювати після подвійного клацання на комірці. На завершення викликом методу setModel () в поданні встановлюємо модель.

Перебір по рядках моделі

Метод rowCount () повертає кількість всіх рядків набору даних. Їм можна скористатися, наприклад, щоб «пройтися» по рядках всієї таблиці.

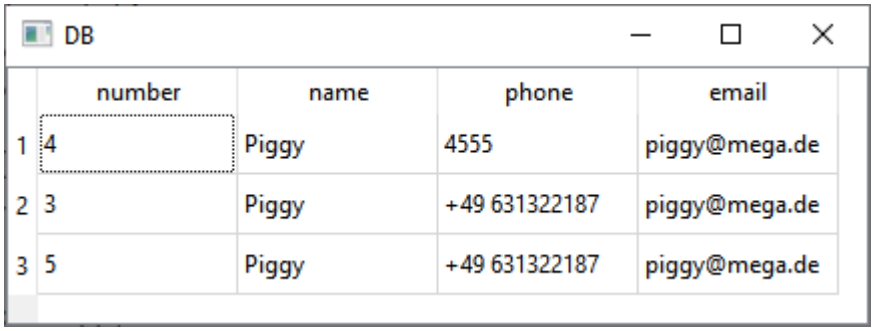
```
for (int nRow = 0; nRow <model.rowCount(); ++ nRow)
```

```
{QSqlRecord rec            = model.record (nRow);
int                nNumber = rec.value ( " number ").toInt ();
     QString        strName = rec.value (" name ").toString();}
```

Фільтрація і сортування

Таблична модель для усунення показу непотрібних рядків записів дозволяє встановлювати фільтри. Для цього існує метод set Filter(),в який ми можемо передати рядок з використовуваним в SQL умовним виразом where. метод setsort () дозволяє виконувати сортування по потрібному стовпці таблиці. Встановимо наш фільтр і проведемо сортування в порядку спадання за номером телефону:

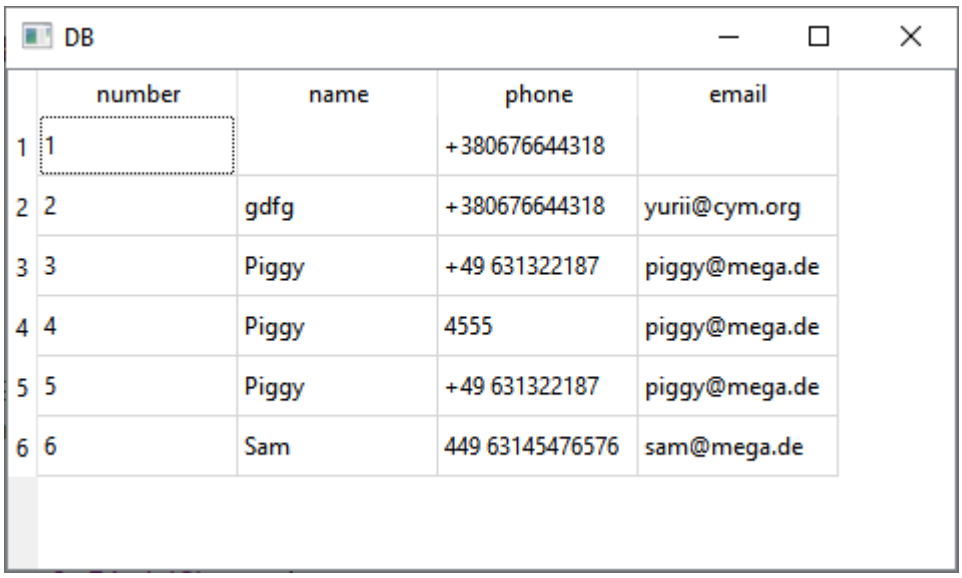
```
model.setFilter( "name = 'Piggy'");
model.setSort (2, Qt :: DescendingOrder);
model.select();
```



Вставка нових записів

Для того щоб вставити в таблицю нові записи, треба викликати метод insertRow (), в першому параметрі цього методу вказати індекс рядка, у другому - кількість нових рядків, а потім викликами методів setData () в нові рядки внести необхідні дані. Запам'ятайте, що при використанні стратегій поновлення OnFieldChange і OnRowChange за один раз можна вставити тільки один рядок.

```
model.insertRows (0, 1);
model.setData (model.index(0, 0), 4);
model.setData (model.index (0, 1), "Sam ");
model.setData (model.index (0, 2), "449 63145476576");
model.setData (model.index (0, 3), "sam@mega.de");
if (! model.submitAll ())
     {qDebug()<<" Insertion error!";}
```



Видалення записів

Щоб видалити записи, нам потрібно спочатку здійснити їх вибір - для цього використовується метод фільтра, а потім викликати метод removeRows(). Наприклад, видалення всіх записів з ім'ям Piggy могло б виглядати так:

```
model.setFilter ( "name = 'Piggy'"); model.select ();
     model.removeRows(0, model.rowCount ()); model.submitAll();
```

Реляційна модель

Реляційна модель являє собою більш високий рівень реалізації, ніж той, який надає таблична модель. Вона володіє механізмами зв'язування таблиць за допомогою первинних (primary) і/або вторинних ключів (foreign keys). Це дозволяє моделі шукати інформацію відразу в декількох таблицях і відображати їх в одній. Клас для реляційної моделі в Qt-QSqlRelationalTableModel. Він успадкований від щойно розглянутого класу QSqlTableModel. Цей клас до успадкованихм методів додає тільки три нових методи, які призначені лише для роботи зі зв'язками таблиць: гемщвудlationModel(), relation () і setRelation().Для демонстрації доповнимо нашу базу даних "addressbook" ще однією таблицею "status", яка буде містити інформацію про те, одружений / заміжній контакт чи ні.

```
   QString str = "CREATE TABLE status"
" ( number INTEGER PRIMARY KEY NOT NULL, married VARCHAR (5) );";
    QSqlQuery query;
   if (!query.exec(str)) {
    qDebug() << "Unable to create a table";}
```

Внесемо в неї дані:

INSERT INTO status (number, married) VALUES (1, 'YES');

INSERT INTO status (number, married) VALUES (2, 'NO');

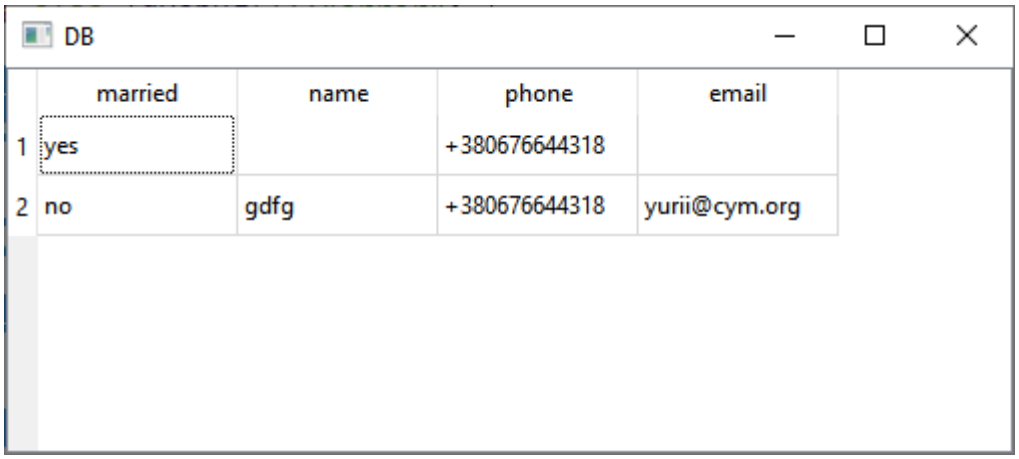
```
query.prepare("INSERT INTO status (number, married)"
"VALUES(:number, :married);");
query.bindValue(":number", 1);
query.bindValue (":married", "yes");
query.exec();

query.prepare("INSERT INTO status (number, married)"
"VALUES(:number, :married);");
query.bindValue(":number", 2);
query.bindValue (":married", "no");
     query.exec();
```

Тепер виконаємо (лістинг 41.5) відображення даних з обох таблиць.

Лістинг 41.5 відрізняється від лістингу 41.3 тільки викликом методу setReiation(). У цьому методі першим параметром ми вказуємо номер стовпця таблиці " addressbook ", в якому розташовані первинні ключі. у другому параметрі передаємо об'єкт QSqlRelation. у ньому ми вказуємо таблицю "status ", яку хочемо об'єднати з таблицею "addressbook ", ім'я стовпця первинних ключів таблиці "status " і третім параметром вказуємо ім'я, яке повинно бути відображено для користувача.

Лістинг 41.5. Використання реляційної моделі - функція main () (файл main.cpp)

```
QTableView view;
QSqlRelationalTableModel model;
model.setTable ( "addressbook");
model.setRelation (0, QSqlRelation ( "status", "number",
"married") ) ;
model.select ();
model.setEditStrategy (QSqlTableModel :: OnFieldChange);
view.setModel (&model);
     view.show ();
```



Про SQLITE

SQLite дещо відрізняється від «звичайних» баз даних, таких як MySQL тим, що «не володіє» клієнт-серверної архітектурою. Тобто движок БД не є окремо працюючим процесом, з яким взаємодіє програма. SQLite є бібліотекою, з якої компонується ваша програма і, таким чином, движок стає складовою частиною програми. Тобто уявіть ви вирішили зберігати всі дані, з якими «стикається» ваша програма в звичайний файл. В один прекрасний день ви вирішуєте зберігати дані у файлі, але організувавши це з «реляційної» точки зору. Після цього ви зрозуміли, що нова структура файлу повинна «розпізнаватися особливим чином». З цього, як мінімум, слід, що вам потрібно надати певний API, що забезпечує зв'язок між цим файлом даних з додатком. Загалом, слідуючи логічної постановці наведеного сценарію у вас народжується система БД, яка не потребує сервера БД і власне, клієнта. Виходить досить швидка в порівнянні з «клієнт-серверної» БД система, і сама програма спрощується.

Створення меню

Меню є важливою і невід'ємною частиною практично будь-якої програми. Головне меню, як правило, знаходиться у верхній частині головного вікна програми і являє собою секцію для розташування великої кількості команд, з яких користувач може вибирати потрібну йому. У додатках використовуються меню чотирьох основних типів:

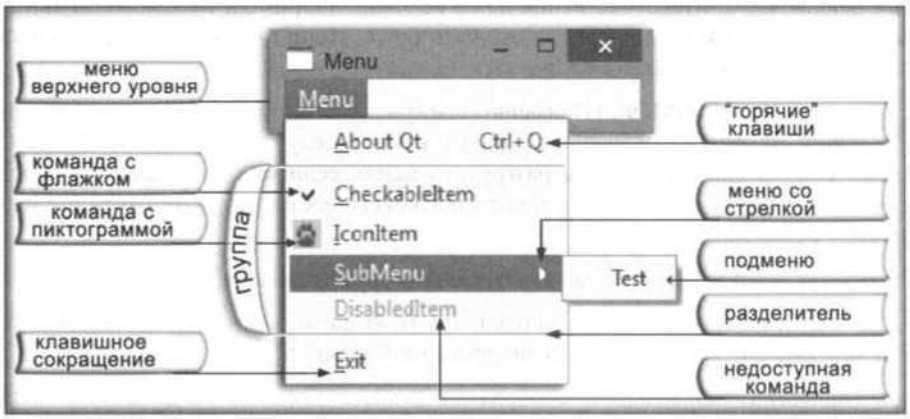
- меню верхнього рівня;
- спливаюче меню;
- відривне меню;
- контекстне меню.

У бібліотеці Qt меню реалізується класом **`QMenu.`** Цей клас визначено в заголовку QMenu. Основне призначення класу-це розміщення команд в меню. Кожна команда представляє об'єкт дії-клас **`QAction`**. Всі дії або самі команди меню можуть бути з'єднані зі слотами для виконання відповідного коду при виборі команди користувачем. Наприклад, якщо користувач виділив команду меню, то і меню, і об'єкти дій відправляють сигнал triggered(), і якщо вам потрібно в цей момент виконати будь-які дії, то їх потрібно з'єднати з відповідним слотом.

«Анатомія» меню

Користувачеві буде легше звикнути до роботи з новою програмою, якщо її меню буде схоже на меню інших програм. На рис. 31.1 показані складові типового меню.

Основною відправною точкою меню є *меню верхнього рівня.* Воно являє собою постійно видимий набір команд, які, в свою чергу, можуть бути обрані за допомогою вказівника миші або клавіш клавіатури (клавіші <Alt> і клавіш управління курсором). Команди меню верхнього рівня призначені для відображення *меню, що* випадає, тому їх не слід використовувати для інших цілей, так як це може неабияк спантеличити користувача. Намагайтеся логічно групувати команди і об'єднувати їх в одному випадаючому меню, яке, в свою чергу, буде викликатися при виборі відповідної команди меню верхнього рівня. Клас **`QMenuBar`** відповідає за меню верхнього рівня і визначено в заголовку QMenuBar.



Таблиця 31.1. *Деякі стандартні комбінації для «гарячих» клавіш*

«Гарячі» клавіші- це, по суті, певні комбінації клавіш, за допомогою яких виконується та ж сама дія, що і при виборі відповідної команди меню. Наприклад, для відображення вікна **AboutQt** (Про Qt) в прикладі, показаному на рис. 31.1, використовується комбінація клавіш <Ctrl> + <Q>. Намагайтеся використовувати для «гарячих» клавіш стандартні комбінації. Деякі з них наведені в табл. 31.1.

Клавіші	Опис	Клавіші	Опис
<Esc>	Скасувати поточну операцію	<Ctrl> + <Z>	Скасувати попередню дію
<F1>	Викликати файл допомоги	<Ctrl> + <X>	Вирізати
<Shift> + <F1>	Викликати контекстну допомогу	<Ctrl> + <C>	Копіювати
<Ctrl> + <N>	Створити	<Ctrl> + <V>	Вставити
<Ctrl> + <0>	Відкрити	<Ctrl> + <F4>	Закрити активний документ MDI-програми
<Ctrl> + <P>	Друк	<Ctrl> + <F6>	Активувати вікно перегляду наступного документа MDI-програми
<Ctrl> + <S>	Зберегти	<Shift> + <Ctrl> + <F6>	Активувати вікно перегляду попереднього документа MDI-додатки

По можливості, для всіх пунктів меню повинні бути визначені *клавіші швидкого виклику*. Це дозволить користувачеві вибирати команди не тільки за допомогою миші, але і за допомогою клавіатури, натиснувши підкреслену букву (в назві команди) спільно з клавішею <Alt>. Наприклад, для вибору команди **Exit** (Вихід) потрібно натиснути <Alt> + <E> (див. Рис. 3l.l). Основні відмінності подібного роду комбінацій для швидкого виклику від «гарячих» клавіш полягають у наступному:

- такі комбінації складаються з клавіші <Alt> і буквеної клавіші;
- вони зустрічаються не тільки в меню, але і в діалогових вікнах;
- вони реалізують контекстне виконання команд. Наприклад, щоб викликати діалогове вікно **About Qt** (Про Qt), потрібно відкрити меню **Menu** (Меню) комбінацією клавіш <Alt> + <M>, а потім натиснути <Alt> + <A>.

Стрілка у команди **SubMenu** (Підменю) (див. Рис. 31.1) говорить про те, що при виборі цієї команди з'явиться *вкладене* підменю, в нашому випадку - **Test** (Тест). Вкладене підменю зручно для того, щоб розвантажити меню, якщо воно містить велику кількість команд. Для зручності розуміння програми рекомендується, щоб ступінь вкладеності не перевищувала двох.

Розділювач - це риса, яка відокремлює одну групу команд від іншої.

Команда з *прапорцем* служить для управління режимами роботи програми. Встановлений прапорець сигналізує про активність команди меню.

Значок (піктограма) команди відображає команду меню в графічному вигляді. Це дуже хороший прийом для додаткової ілюстрації дій самої команди.

Іноді зустрічаються команди, які не можна виконати в певний момент часу. У таких випадках застосування повинне робити такі команди *недоступними,* тобто зробити їх вибір неможливим. Такі команди меню відображаються іншим кольором - як правило, сірим.

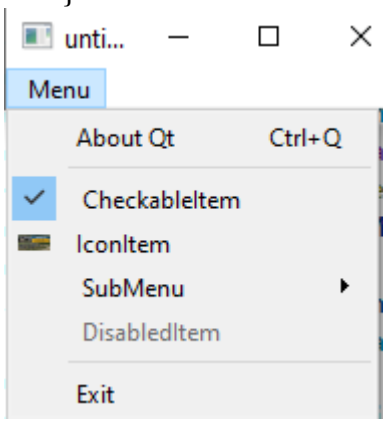
Також пам'ятайте, що у випадках, коли команда меню викликає діалогове вікно, в кінці її назви прийнято додавати три крапки. Це правило, правда, не поширюється на виклик простих вікон повідомлень.

У лістингу 31.1 реалізується меню, показане на рис. 31.1.

Лістинг 31-1. Приклад реалізації меню (файл mam.cpp)

```
#include <QtWidgets>
int main (int argc, char ** argv)
{QApplication
app (argc, argv);
QMenuBar mnuBar;
QMenu * pmnu = new QMenu ( "& Menu");
pmnu-> addAction( "&About Qt", &app, SLOT(aboutQt()), Qt::CTRL+Qt::Key_Q);
```

```
    pmnu-> addSeparator ();
    QAction * pCheckableAction = pmnu-> addAction ( "& CheckableItem");
    pCheckableAction-> setCheckable (true);
    pCheckableAction-> setChecked (true);
    pmnu-> addAction (QPixmap ( ":/img4.png"), "&IconItem");
    QMenu * pmnuSubMenu = new QMenu ( "& SubMenu", pmnu);
    pmnu-> addMenu (pmnuSubMenu);
    pmnuSubMenu-> addAction ( "& Test");
    QAction *pDisabledAction = pmnu-> addAction ( "&DisabledItem");
    pDisabledAction-> setEnabled (false);
    pmnu-> addSeparator ();
    pmnu-> addAction ( "&Exit", &app, SLOT (quit()));
    mnuBar.addMenu (pmnu);
    mnuBar.show ();
    return app.exec ();
    }
```



Якщо розбити на класи програму, матимемо наступний код:

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
QMenuBar *mnuBar = new QMenuBar(this);
    QMenu *pmnu = new QMenu ( "&Menu");
    pmnu-> addAction( "&About Qt", qApp, SLOT(aboutQt()), Qt::CTRL+Qt::Key_Q);
    pmnu-> addSeparator ();
    QAction * pCheckableAction = pmnu-> addAction ( "&CheckableItem");
    pCheckableAction-> setCheckable (true);
    pCheckableAction-> setChecked (true);
    pmnu-> addAction (QPixmap ( "123.jpg"), "&IconItem");
    QMenu * pmnuSubMenu = new QMenu ( "&SubMenu", pmnu);
    pmnu-> addMenu (pmnuSubMenu);
    pmnuSubMenu-> addAction ( "&Test");
    QAction *pDisabledAction = pmnu-> addAction ( "&DisabledItem");
    pDisabledAction-> setEnabled (false);
    pmnu-> addSeparator ();
    pmnu-> addAction ( "&Exit", this, SLOT(close()));
pmnu->setStyleSheet("background-color: Red");
mnuBar->addMenu (pmnu);
//this->setMenuBar(mnuBar);
      }

- - - - - -

int main (int argc, char ** argv)
{QApplication app (argc, argv);
MainWindow w;
w.show ();
return app.exec ();
      }
```

Розширений варіант прив’язки дії до пункту меню виглядає нуступним чином:

```
   QAction* exitAction = pmnu->addAction( "Exit" );
```

```
connect( exitAction, SIGNAL( triggered() ), qApp, SLOT(quit() ) );
```

Скорочено записано:

```
pmnu-> addAction ( "&Exit", this, SLOT(close()));
```

Щоб створити повноцінне меню, потрібно до кожної команди меню верхнього рівня приєднати відповідне *спливаюче меню*. За спливаючі меню відповідає клас **QMenu.** Отже, для створення меню необхідно мати віджет класу **QMenuBar** - вказівник **pmnuBar** і, щонайменше, один віджет класу **QMenu** - покажчик **pmnu** (див. Лістинг 31.1).

Для додавання спливаючого меню до меню верхнього рівня потрібно передати в метод **addAction**() назва команди. Кожен метод **addAction ()** повертає покажчик на об'єкт дії **QAction.** Користуючись цим покажчиком, можна отримати доступ до команди меню. Виклик методу **setcheckabie ()** об'єкта дії (в нашому випадку **pCheckableAction)** надає можливість установки прапорця. Подальший виклик методу **setchecked** () встановлює стан прапорця. У нашому прикладі в цей метод передається значення **true,** а це значить, що прапорець буде перебувати під «включеному» стані.

Метод **addAction** () приймає чотири параметри. Перший задає назву команди меню, в якому можна вказати букву для швидкого виклику, поставивши перед нею символ &. Не упускайте з виду, що різні команди меню повинні використовувати різні літери для швидкого виклику, в іншому випадку одна з них виявиться недоступною. Другим параметром передається покажчик на об'єкт, що містить слот, який викликається при виборі цієї команди. Сам слот передається третім параметром. Останній параметр задає комбінацію для «гарячої» клавіші. У нашому прикладі для відображення діалогового вікна **About Qt** (Про Qt) використовується комбінація клавіш <Ctrl> + <Q>. Натискання цієї комбінації клавіш призведе до того ж дії, що і вибір відповідної команди меню, а саме - буде викликаний слот об'єкта додатка **aboutQt** (). Для складання комбінацій «гарячих» клавіш можна скористатися табл. 14.2.

Виклик методу **addSeparator** () додає роздільник в меню.

Покажчиком на об'єкт дії **(pDisabieAction)** можна скористатися також і для того, щоб зробити деякі з команд меню недоступними - за допомогою методу **setEnabled ().**

У метод **addAction** () першим параметром можна передавати об'єкти растрових зображень для установки значка команди.

QAction *QMenuBar::addAction(const QString &*text*)

Ця зручна функція створює нову дію з *текстом*. Функція додає новостворену дію до списку дій меню та повертає її.

приклад - addAction("&CheckableItem");

QAction *QMenuBar::addAction(const QString &*text*, const QObject **receiver*, const char **member*)

Це перевантажена функція.

Ця зручна функція створює нову дію із заданим *текстом*. Спрацьовує сигнал triggered(), який викликає підключений до receiver слот. Функція додає новостворену дію до списку дій меню та повертає її.

приклад - addAction ("&Exit", this, SLOT(close()));

addAction("&About Qt", qApp, SLOT(aboutQt()), Qt::CTRL+Qt::Key_Q);

QAction *QMenuBar::addAction(const QString &*text*, const Obj **receiver*, PointerToMemberFunctionOrFunctor *method*)

Це перевантажена функція.

Ця зручна функція створює нову дію із заданим *текстом*. Дія Спрацьовує сигнал triggered(), який викликає підключений до receiver слот. Функція додає новостворену дію до списку дій меню та повертає її.

QMenuBar приймає право власності на повернутий QAction.

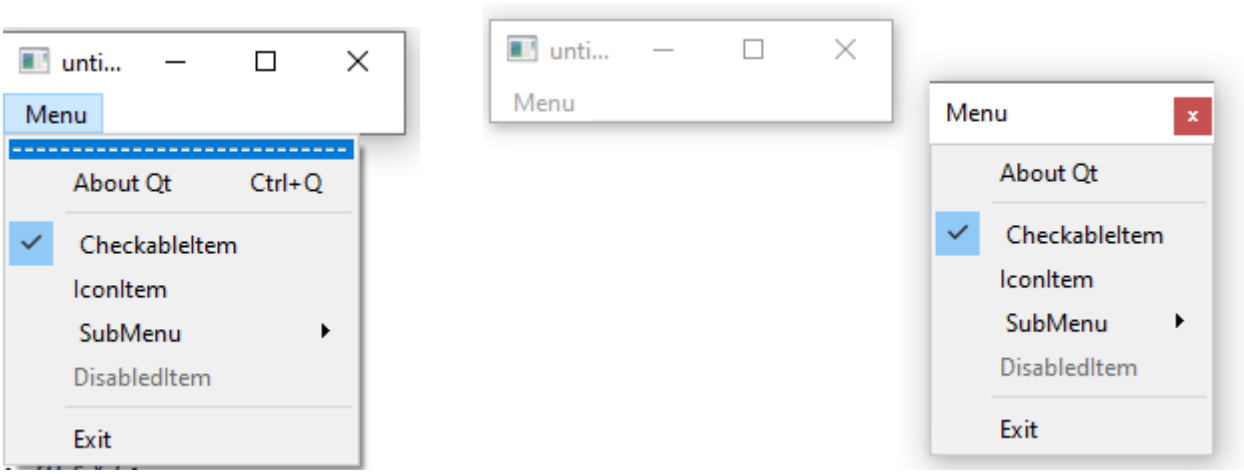
Відривне меню.

Qt надає можливість реалізації відривних меню (tear-off menu). Натискання мишею на переривчасту лінію призводить до того, що спливаюче меню відділяється від меню верхнього рівня, перетворюючись в окреме вікно, яке вільно переміщається. Таке меню дуже зручно, наприклад, для настройки конфігурацій програми.

Щоб задати відривний меню, спочатку необхідно викликати з віджета меню pmnu метод setTearOffEnabled(), передавши йому значення true - це відобразить лінію відриву на верхньому бордюрі спливаючого меню.

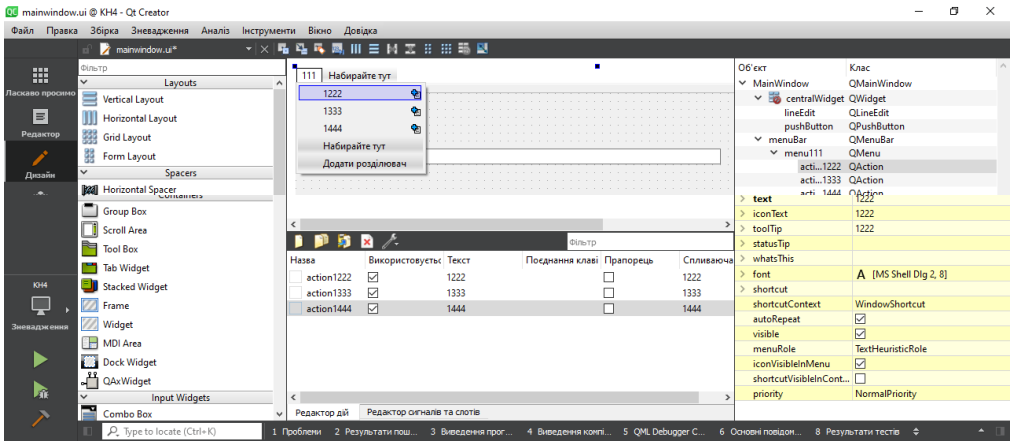
Для прикладу візьмем лістинг 31.1
Добавимо метод setTearOffEnabled()

```
pmnu->setTearOffEnabled(true);
```

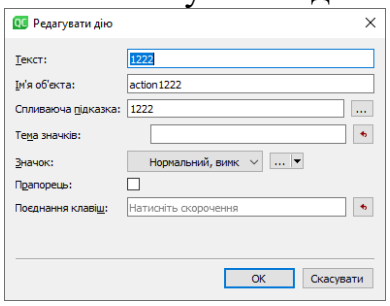


Візуальне налаштування меню за допомогою UI компоненти.

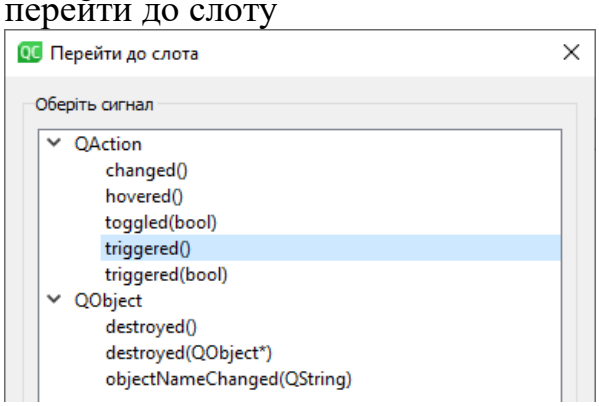
Для кожного меню встановлюється відповідна дія в вікні редактора подій.



Коли активувати відповідну дію то відкриється діалогове вікно редактора події



Для добавлення дій до елементів меню, в вікні редактора дій на відповідній дії ПКМ-перейти до слоту



Потрібно вибираємо сигнал **triggered()** – сигнал, який генерується при активації підпункту меню.

Контекстні меню, у випадку центрально віджету

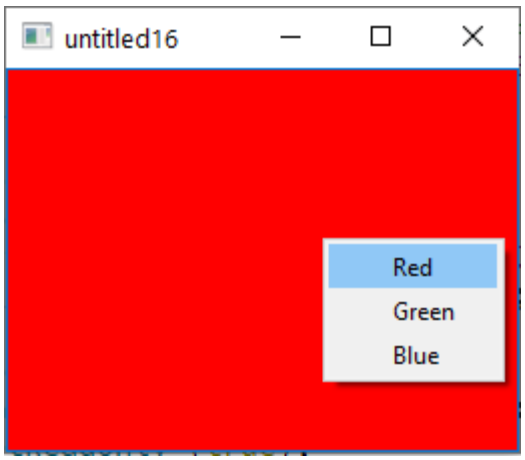


Рис. 31.2. Контекстне меню для вибору кольору вікна

Візитною карткою професійного додатку є наявність контекстного меню. *Контекстне меню* - це меню, яке відкривається при натисканні правої кнопки миші. Для

його реалізації, як і в разі спливаючого меню, використовується клас **QMenu.** Різниця полягає лише в тому, що це меню не приєднується до віджету **QMenuBar.** На рис. 31.2 показано вікно програми (лістинг 31.2) і контекстне меню, що відображається при натисканні правої кнопки миші.

У цьому меню користувач може вибрати одну з трьох команд: Red (Червоний), Green (Зелений) або Blue (Синій), які задають відповідний колір фону вікна.

У конструкторі класу **ContextMenu** лістингу 31.2 створюється віджет контекстного меню - вказівник **m_pmnu.** За допомогою методу **addAction ()** додаються команди меню. Метод **connect ()** з'єднує сигнал меню **triggered (QAction *) зі слотом slotActivated (QAction *).** Сигнал відправляється кожен раз при виборі користувачем однієї з команд меню. Цей слот отримує вказівник на об'єкт дії. Завдяки тому, що наш клас **ContextMenu** успадкований від класу **QTextEdit,** ми можемо встановлювати колір фону за допомогою рядка в форматі HTML - потрібно тільки викликати метод **QTextEdit::setHtml**(). Сам колір встановлюється відповідно до імені обраної команди, з якого видаляється символ &, для чого викликається метод **QString::remove().** Рядок з кольором записується в змінну **strColor.**

Показ контекстного меню виконується з методу обробки події контекстного меню **QWidget:: contextMenuEvent** () і повинен здійснюватися на місці (в координатах) вказівника миші при натисканні її правої кнопки. Для цього потрібно передати в метод **exec** () значення, що повертається методом **globalPos ()** об'єкта події контекстного меню. Цей метод повертає об'єкти класу **QPoint,** що містять координати вказівника миші щодо верхнього лівого кута екрана.

Лістинг 31.2. Контекстне меню (файл mainwindow.h)

```
#include <QtWidgets>
// ================================ ====================================
class ContextMenu: public QTextEdit
{Q_OBJECT
private:
QMenu * m_pmnu ;
protected:

virtual void contextMenuEvent (QContextMenuEvent * pe)
{m_pmnu->exec(pe->globalPos ());
}


public:
ContextMenu (QWidget * pwgt =0):QTextEdit(pwgt)
{setReadOnly (true);
    m_pmnu = new QMenu (this);
    m_pmnu-> addAction("&Red");
    m_pmnu-> addAction("&Green");
    m_pmnu-> addAction("&Blue");
    connect (m_pmnu,SIGNAL (triggered (QAction *)), SLOT (slotActivated (QAction*)));
}
public slots:
void slotActivated (QAction * pAction)
{QString strColor = pAction->text().remove( "&");
setHtml (QString ( "<BODY BGCOLOR = %1> </ BODY>").arg(strColor));

}};
```

файл main.cpp

```
#include <QtWidgets>
#include "mainwindow.h"
```

```
int main (int argc, char ** argv)
{QApplication app (argc, argv);
ContextMenu w;
w.show ();
return app.exec ();
}
```

Організація контекстного меню Qt

Є два підходи до створення контекстного меню в додатках на Qt:

- полягає у попередній підготовці обробки подій QWidget :: contextMenuEvent () для вашого класу та отримання контекстного меню в ньому.
- полягає у встановленні спеціальної політики вибору контекстного меню для потрібного віджету та очікування від нього сигналів customContextMenuRequest ()

Перший спосіб.

Заголовочний файл:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QtWidgets>

class MainWindow : public QMainWindow
{
    Q_OBJECT
QMenu * m_pmnu ;
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
//public slots:
//    void slotActivated (QAction * pAction);
protected:
void contextMenuEvent (QContextMenuEvent * pe);


};

#endif // MAINWINDOW_H
```

Контекстне меню, представлене полем m_pmnu, яке є екземпляром класу QMenu. Безпосередня обробка кліку правої кнопок миші на формі віджета відбувається в захищеній віртуальній функції-члені contextMenuEvent () базового класу QWidget, яку ми перевизначаємо. Це означає, що контекстне меню буде вказуватися саме для цього віджету MainWindow.

Файл реалізації:

```
#include "mainwindow.h"
#include <QFile>
#include<QMessageBox>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    m_pmnu = new QMenu (this);
    QAction* exitAction = m_pmnu->addAction( "Exit" );
        connect( exitAction, SIGNAL( triggered() ), qApp, SLOT( quit() ) );

}
```

```
MainWindow::~MainWindow()
{

}

void MainWindow::contextMenuEvent(QContextMenuEvent *pe) {
    if( m_pmnu ) {
        m_pmnu->exec(pe->globalPos());
    }
}
```

Додавання пунктів до контекстного меню здійснюється за допомогою addAction (). На вхід ця функція-член приймає рядок з текстом, який відповідає назві цього пункту. Окрім тексту можна передати іконку QIcon, яка буде відображатися поруч із надписом. Аналогічним способом ви можете додати стільки пунктів, скільки потрібно.

У найпростішому випадку для обробки сигналів натискання на пункт меню можна підключити сигнал, який ініціюється () об'єктом QAction із власним слотом. У представленому вище застосунку під натисканням на пункт меню Exit буде вибраний слот quit () додатків qApp, який буде введено до завершення роботи програм.

Також не забудьте про реалізацію contextMenuEvent (), інакше меню не з'являється. Для відображення контекстного меню досить викликати функцію-член exec(), яка у аргументі приймає координати, щодо яких, меню повинно бути виведено на екран.

Другий спосіб.

Заголовочний файл:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QtWidgets>

class MainWindow : public QMainWindow
{
    Q_OBJECT
    QTextEdit *txtedt;
QMenu * m_pmnu ;
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
//public slots:
//    void slotActivated (QAction * pAction);
public slots:
void showContextMenu(QPoint);};
#endif // MAINWINDOW_H
```

cpp – файл

```
#include "mainwindow.h"
#include <QFile>
#include<QMessageBox>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    txtedt = new QTextEdit(this);
    txtedt->setReadOnly(true);

txtedt->setContextMenuPolicy(Qt::CustomContextMenu);
connect(txtedt, SIGNAL(customContextMenuRequested(QPoint)),
    this, SLOT(showContextMenu(QPoint)));
}
MainWindow::~MainWindow()
{}
```

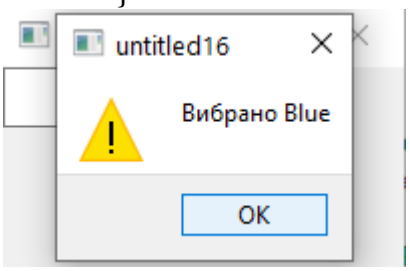
```
void MainWindow::showContextMenu(QPoint pos) {
   QPoint globalPos = txtedt->mapToGlobal(pos);
   m_pmnu = new QMenu ();
   m_pmnu-> addAction ( "Red");
   m_pmnu-> addAction ( "Green");
   m_pmnu-> addAction ( "Blue");
   QAction* exitAction = m_pmnu->addAction( "Exit" );
       connect( exitAction, SIGNAL( triggered() ), qApp, SLOT( quit() ) );
       m_pmnu->exec(globalPos);
      }
```

Відображення меню може ініціалізуватися за допомогою двох методів - popup () і exec (). Метод popup () - не блокуючий, і в разі його використання для обробки вибраних пунктів меню потрібно для кожного QAction, прописати обробник сигналів `triggered()`.

```
connect(action, SIGNAL(triggered()), this, SLOT(action()));
```

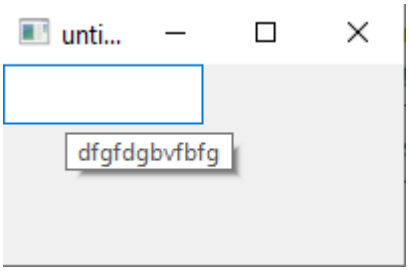
Метод exec () отримує від користувача вибраний пункт меню і повертає відповідний QAction.

```
 QAction* selectedItem = m_pmnu->exec(globalPos);
       if (selectedItem->text()=="Red") {
           QMessageBox::warning(this, "", QString::fromUtf8("Вибрано
%1").arg(selectedItem->text()));
       }
       if (selectedItem->text()=="Green") {
                   QMessageBox::warning(this, "", QString::fromUtf8("Вибрано
%1").arg(selectedItem->text()));
               }
       if (selectedItem->text()=="Blue") {
                   QMessageBox::warning(this, "", QString::fromUtf8("Вибрано
%1").arg(selectedItem->text()));
               }
```



Випливаюча підказка при наведенні на віджет встановлюється, за допомогою методу setToolTip()

```
  txtedt = new QTextEdit(this);
txtedt->setToolTip("dfgfdgbvfbfg");
```



Існує можливість використання в кнопках які натискаються кнопку контекстного меню (мал. 8.2). Подібні кнопки можна зустріти, наприклад, в браузері Microsoft Internet Explorer. Кнопка **Start** (Пуск) панелі задач ОС Windows також є такою кнопкою. Додати меню можна, викликавши метод `setMenu()` і передавши курсор на об'єкт спливаючого меню. Подібні кнопки можуть використовуватися в якості альтернативи для списку

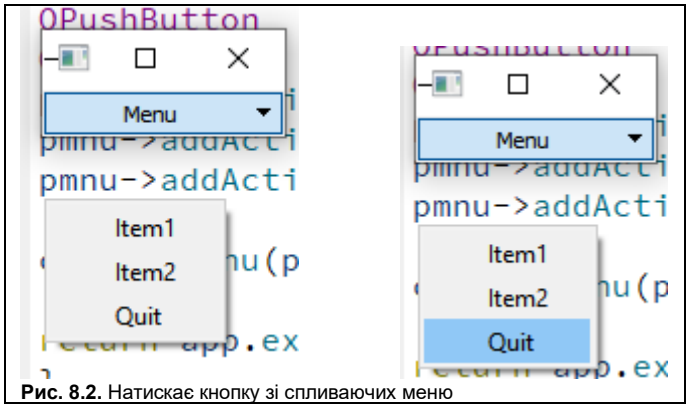


Рис. 8.2. Натискає кнопку зі спливаючих меню

Лістинг 8.2. Файл main.cpp

```
#include <QApplication>
#include <QtWidgets>

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

QPushButton  cmd("Menu");
QMenu*  pmnu = new QMenu(&cmd);
pmnu->addAction("Item1");
pmnu->addAction("Item2");
pmnu->addAction("&Quit", &app, SLOT(quit()));

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

return app.exec();
}
```

У лістингу 8.2 створюються віджети кнопки `cmd` і спливаючого меню `pmnu`. Викликом методу `addAction()` додається елемент меню. Остання команда **Quit** (Вихід) з'єднується зі слотом `quit()` з'єднується зі слотом об'єкта додатка, що дозволяє користувачеві завершити роботу, вибравши цю команду з меню. Установка створеного меню в кнопці яка натискається здійснюється викликом методу `setMenu()`.