Programowanie wieloplatformowe

Typy danych i kolekcje

Tłumaczenie i adaptacja materiałów: dr Tomasz Xięski.
Na podstawie prezentacji udostępnionych przez Digia Plc. na licencji CC.

© 2012 Digia Plc.

The enclosed Qt Materials are provided under the Creative Commons Attribution-Share Alike 2.5 License Agreement.

The full license text is available here:
http://creativecommons.org/licenses/by-sa/2.5/legalcode.

Digia, Qt and the Digia and Qt logos are the registered trademarks of Digia Plc. in Finland and other countries worldwide.

Obsługa ciągów znaków

- W języku C tablica znaków wykorzystuje wyłącznie aktualną (lokalną) stronę kodową.

```
char *text = "Hello world!";
```

- Qstring jest klasą uwzględniającą szereg współczesnych wymagań dotyczących łańuchów
  - Obsługuje Unicode oraz mechanizmy transkrypcji
  - Niejawne współdzielenie (implicit sharing) poprawiające wydajność

```
QString s("Hello world!");
QString s2 = s //ten sam obszar pamięci
```

QString

- Dane reprezentowane są jako Unicode co umożliwia przechowanie napisu praktycznie w każdym języku
- Umożliwia konwersję między różnymi standardami (stronami kodowymi)

```
QString::toLatin1 - QString::toLocal8Bit
```

- Udostępnia dogodne API do przetwarzania i modyfikacji łańcuchów tekstowych

Tworzenie napisów

- Istnieją 3 metody tworzenia napisów:
- Metoda wykorzystująca `operator+`

```
QString res = "Hello “ + name +
    “, the value is " + QString::number(42);
```

- Metody klasy `QStringBuilder`

```
QString res = "Hello “ % name %
    “, the value is " % QString::number(42);
```

- Wykorzystanie `arg`

```
QString res = QString("Hello %1, the value is %2")
    .arg(name)
    .arg(42);
```

QStringBuilder

- Użycie operatora + do łączenia ciągów powoduje konieczność wielokrotnej alokacji pamięci
- Lepszym rozwiązaniem jest zastosowanie `QStringBuilder` i wykorzystanie operatora `%`
- `QStringBuilder` najpierw określa całkowitą długość tekstu wynikowego, a następnie dokonuje pojedynczej alokacji pamięci

```
QString res = "Hello " % name %
    ", the value is %" % QString::number(42);
```

```
QString temp = "Hello ";
temp = temp % name;
temp = temp % ", the value is %"
temp = temp % QString::number(42);
```

Łączenie w taki sposób obniża wydajność aplikacji

QString::arg

- Metoda arg zastępuje `%1-99` odpowiednimi wartościami

```
"%1 + %2 = %3, the sum is %3"
```

Wszystkie wyrażenia %n są zastępowane

- Konwersja działa dla napisów, znaków, liczb

```
...).arg(qulonglong a)
...).arg(short a)
...).arg(ushort a)
...).arg(QChar a)
...).arg(char a)
...).arg(double a)
...).arg(QString, ... QString)
...).arg(int a)
...).arg(uint a)
...).arg(long a)
...).arg(ulong a)
...).arg(qlonglong a)
```

Do 9 argumentów dla każdego arg

- Można konwertować pomiędzy systemami liczbowymi

```
...).arg(value, width, base, fillChar);
...).arg(42, 3, 16, QChar('0')); // Results in 02a
```

Podciągi

- Podciąg można wyłuskać metodami `left`, `right`

```
QString s = "Hello world!";
r = s.left(5); // "Hello"
r = s.right(1); // "!"
r = s.mid(6,5); // "world"
```

- Gdy nie podamy długości dla `mid`, zwracany jest ciąg począwszy od wskazanego znaku do końca

```
r = s.mid(6); // "world!"
```

- Metoda `replace` znajduje i zastępuje podciąg

```
r = s.replace("world", "universe"); // "Hello universe!"
```

Wyświetlanie w konsoli

- Qt jest stworzony przede wszystkim dla aplik--- --okienkowych, stąd nie ma w nim typowej linii komend
- Do wypisania danych na konsole (debugowanie) wykorzystywana jest funkcja `qDebug`
  - Jest zawsze dostępna, nie będzie jednak wyświetlać danych prze kompilowaniu w trybie Release
  - Działa jak funkcja `printf` (ale automatycznie dodaje \n)

```
qDebug("Integer value: %d", 42);
qDebug("String value: %s", qPrintable(myQString));
```

  - Może być wykorzystywana z operatorem strumieniowym <<

```
#include <QtDebug>

qDebug() << "Integer value:" << 42;
qDebug() << "String value:" << myQString;
qDebug() << "Complex value:" << myQColor;
```

Konwersja z i na liczby

- Konwersja liczby do napisu

```
QString::number(int value, int base=10); QString
twelve = QString::number(12); // "12" QString
oneTwo = QString::number(0x12, 16); // "12"

QString::number(double value, char format='g', int precision=6);
QString piAuto = QString::number(M_PI); // "3.14159"
QString piScientific = QString::number(M_PI,'e'); // "3.141593e+00"
QString piFixedDecimal = QString::number(M_PI,'f',2); // "3.14"
```

- Konwersja napisu do liczby

```
bool ok;
QString i = "12";
int value = i.toInt(&ok);
if(ok) {
    // Converted ok
}
```

```
bool ok;
QString d = "12.36e-2";
double value = d.toDouble(&ok);
if(ok) {
    // Converted ok
}
```

Praca z std::(w)string

- Konwersja z oraz na ciągi znaków wykorzystywane w STL jest pomocna podczas wykorzystywania zawartych tam algorytmów
- Konwersja z ciągu STL

```
std::string ss = "Hello world!";
std::wstring sws = "Hello world!";

QString qss = QString::fromStdString(ss);
QString qsws = QString::fromStdWString(sws);
```

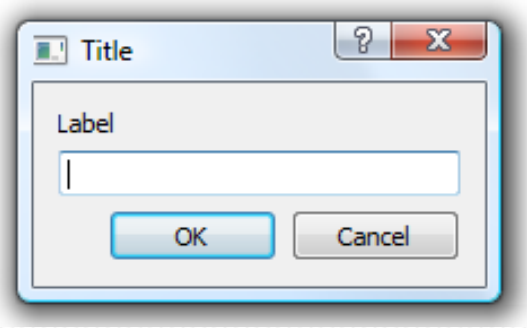
- Konwersja na ciąg STL

```
QString qs = "Hello world!";
std::string ss = qs.toStdString();
std::wstring sws = qs.toStdWString();
```

Puste ciągi czy null?

- `QString` może być null (nic nie zawierać)

```
QString n = QString();

n.isNull();  // true
n.isEmpty(); // true
```



- Może też być pusty, tj. Zawierać pusty ciąg

```
QString e =  "";

e.isNull();  // false
e.isEmpty(); // true
```

Rozdzielanie i łączenie

- `QString` może być podzielony na elementy

```
QString whole = "Stockholm - Copenhagen - Oslo - Helsinki";
QStringList parts = whole.split(" - ");
```

- W wyniku otrzymujemy obiekt `QStringList`, którego elementy można połączyć tworząc `QString`

```
QString wholeAgain = parts.join(", ");
    // Results in "Stockholm, Copenhagen, Oslo, Helsinki"
```

QStringList

- The `QStringList` jest specjalizowaną listą
- Zaprojektowana do przechowywania napisów

  Dostarcza dogodnego API do pracy z listami napisów

- Klasa wykorzystuje niejawne współdzielenie danych (implicit sharing)
  - Kopiowanie realizowane jest dopiero przy modyfikacji zawartości
  - Szybkie przy przekazywaniu wartości

Operacje na listach napisów

- Operator `<<` służy do dodawania elementów

```
QStringList verbs;
verbs = "running" << "walking" << "compiling" << "linking";
```

- Metoda `replaceInStrings` pozwala na przeszukiwanie i zastępowanie elementów całej `QStringList`.

```
qDebug() << verbs; // ("running", "walking", "compiling", "linking")
verbs.replaceInStrings("ing", "er");
qDebug() << verbs; // ("runner", "walker", "compiler", "linker")
```

Sortowanie i filtrowanie

- `QStringList` można posortować...

```
qDebug() << capitals; // ("Stockholm", "Oslo", "Helsinki", "Copenhagen")
capitals.sort();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

- ...filtrować...

Zwraca uwagę na wielkość znaków

```
QStringList capitalsWithO = capitals.filter("o");
qDebug() << capitalsWithO; // ("Copenhagen", "Oslo", "Stockholm")
```

- ...usunąć duplikaty

```
capitals << capitalsWithO;
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm",
                      //  "Copenhagen", "Oslo", "Stockholm")
capitals.removeDuplicates();
qDebug() << capitals; // ("Copenhagen", "Helsinki", "Oslo", "Stockholm")
```

Iteracyjne odwołanie do elementów listy

- Operator `operator[]` i metoda `length` pozwalają na odwołanie się do elementów `QStringList` jak dla klasycznej tablicy

```
QStringList capitals;
for(int i=0; i<capitals.length(); ++i)
    qDebug() << capitals[i];
```

- Innym sposobem jest wykorzystanie metody `at()` umożliwiającej odczyt danego elementu
- Metoda `at()` uniemożliwia kopiowanie elementu, co może `operator[]`
- Dostępne jest również makro `foreach`

```
QStringList capitals;
foreach(const QString &city, capitals)
    qDebug() << city;
```

Kolekcje w Qt

- `QList` jest jednym z wielu wzorców (kolekcji):
  - `QLinkedList` – lista dynamiczna
  - `QVector` – lista dynamiczna (ciągły obszar pamięci)
  - `QStack` – LIFO, last in – first out
  - `QQueue` – FIFO, first in – first out
  - `QSet` – zbiór niepowtarzalnych elementów
  - `QMap` – tablica asocjacyjna, sortowana po kluczach
  - `QHash` – tablica asocjacyjna z haszowaniem (szybsza niż `QMap`)
  - `QMultiMap` – tablica asocjacyjna z wieloma wartościami dla klucza
  - `QMultiHash` – tablica asocjacyjna z wieloma wartościami dla klucza (wymaga funkcji haszującej)

Dodawanie do listy

- Za pomocą operatora <<

```
QList<int> fibonacci;
fibonacci << 0 << 1 << 1 << 2 << 3 << 5 << 8;
```

- Za pomocą metod `prepend`, `insert` **oraz** `append`

```
QList<int> list;
list.append(2);
```

index 0: **2**

```
list.append(4);
```

index 0: **2**
index 1: **4**

```
list.insert(1,3);
```

index 0: **2**
index 1: **3**
index 2: **4**

```
list.prepend(1);
```

index 0: **1**
index 1: **2**
index 2: **3**
index 3: **4**

Usuwanie

- Usuwanie elementów z `QList` metodami `removeFirst`, `removeAt`, `removeLast`

```
while(!list.isEmpty())
    list.removeFirst();
```

- Za pomocą `takeFirst`, `takeAt`, `takeLast`

```
QList<QWidget*> widgets;
widgets << new QWidget << new QWidget;
while(!widgets.isEmpty())
    delete widgets.takeFirst();
```

- Usuwanie elementów o konkretnej wartości metodą `removeAll` lub `removeOne`

```
QList<int> list;
list << 1 << 2 << 3 << 1 << 2 << 3;
list.removeAll(2); // Leaves 1, 3, 1, 3
```

Dostęp do elementów

- Elementy `QList` indeksowane są od `0` do `length-1`
- Do określonych elementów listy można uzyskać dostęp funkcją `at()` lub operatorem `[]`

```
for(int i=0; i<list.length(); ++i)
    qDebug("At: %d, []: %d", list.at(i), list[i]);

for(int i=0; i<100; ++i)
    qDebug(“Value: %d”, list.value(i));
```

- Operator`[]` zwraca referencję (do odczytu i zapisu)

```
for(int i=0; i<list.length(); ++i)
    list[i]++;
```

Iteratory – wersja Java

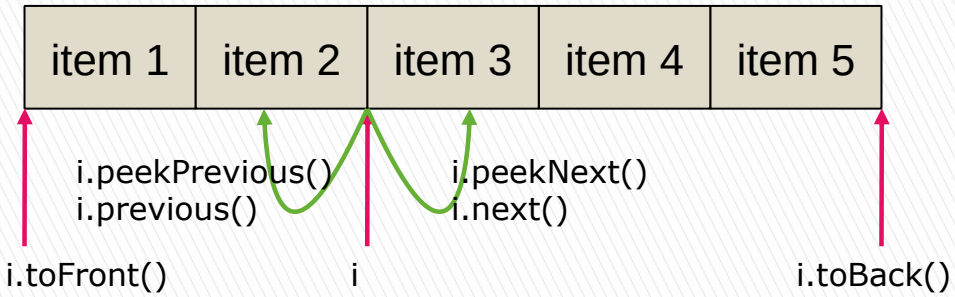
- Qt wspiera iteratory w konwencji Java:

```
QListIterator<int> iter(list);
        while(iter.hasNext())
            qDebug("Item: %d", iter.next());
```

Zwróć wartość i przejdź do następnego elementu

Użyj QMutableListIterator jeżeli modyfikujesz elementy listy

- Wykorzystanie iteratorów:
  - toFront przenosi iterator przed pierwszy element
  - toBack przenosi iterator za ostatni element
  - Do elementów odwołuj się przez peekNext oraz peekPrevious
  - Przesuwaj się używając next lub previous



Iteratory – wersja STL

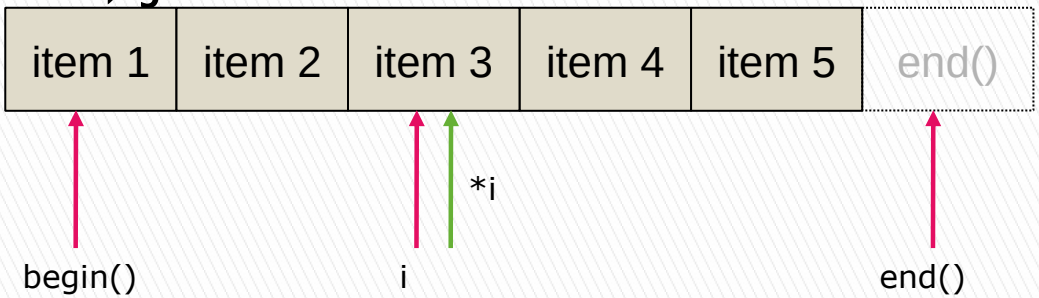
- Qt wspiera iteratory w konwencji STL:

```
for(QList<int>::const_iterator iter=list.begin();
    iter!=list.end(); ++iter)
    qDebug("Item: %d", *iter);
```

Użyj klasę Iterator jeżeli modyfikujesz elementy

Zarówno Iterator|iterator jak i ConstIterator|const_iterator mogą być użyte

- Iteratory STL wskazują dany element, wykorzystując nieistniejące elementy (null) jako wskaźnik końca
  - Pierwszy element dostępny jest przez begin
  - Wskaźnik końca zwracany jest przez end
  - Operator * odwołuje się do wartości elementu
  - Przesuwając się do tyłu najpierw należy zmienić pozycję iteratora a dopiero później odwołać się do wartości elementu



Iteracyjne przeglądanie listy

- Do iteracyjnego przeglądania elementów kolekcji można wykorzystać pętlę `foreach`

```
QStringList texts;
foreach(QString text, texts)
    doSomething(text);
```

Użycie referencji do stałej zwiększa wydajność, równocześnie uniemożliwia jednak zmianę wartości elementu

```
QStringList texts;
foreach(const QString &text, texts)
    doSomething(text);
```

Interakcja z STL

- Obiekt klasy QList może zostać przekonwertowany do std::list

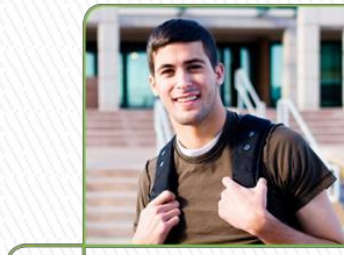
```
QList<int> list;
list << 0 << 1 << 1 << 2 << 3 << 5 << 8 << 13;

std::list<int> stlList = list.toStdList();

QList<int> otherList = QList<int>::fromStdList(stlList);
```

Z listy Qt do STL

Z listy STL do Qt

- Konwersja z i do listy STL oznacza tworzenie pełnej kopii listy – nie zostanie wykorzystany mechanizm współdzielenie (ang. *implicit sharing*)

Inne kolekcje



- Jakie są alternatywy dla `Qlist?`
- `QLinkedList`
  - Wolna podczas swobodnego dostępu (po indeksach)
  - Szybka, podczas używania iteratorów
  - Szybki (stały) czas dodania do środka listy
- `QVector`
  - Używa ciągłego obszaru pamięci
  - Wolne wstawianie na początek i do środka

Inne kolekcje

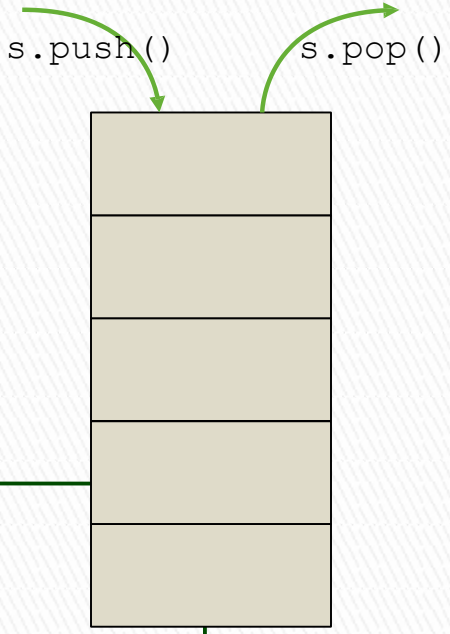
Collection	Index access	Insert	Prepend	Append
`QList`	O(1)	O(n)	Amort. O(1)	Amort. O(1)
`QLinkedList`	O(n)	O(1)	O(1)	O(1)
`QVector`	O(1)	O(n)	O(n)	Amort. O(1)

- Amort oznacza brak gwarancji do takiej złożoności
- Inne kolekcje
  - `QStringList` – bazuje na `QList`
  - `QStack` – bazuje na `QVector`
  - `QQueue` – bazuje na `QList`
  - `QSet` – bazuje na `QHash`

Stos- QStack

- Kontener typu LIFO
  last in, first out

- Dodawanie: `push()`
- Usuwanie: `pop()`
- Podgląd elementu na górze: `top()`

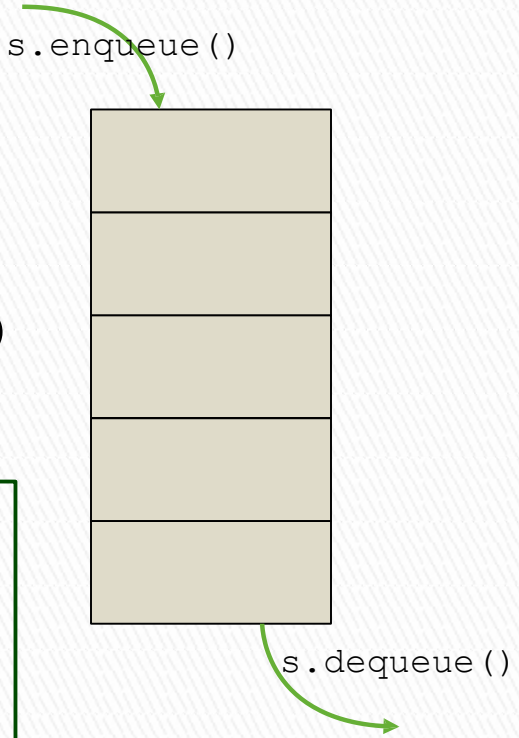



```
QStack<int> stack;
stack.push(1);
stack.push(2);
stack.push(3);
qDebug("Top: %d", stack.top()); // 3
qDebug("Pop: %d", stack.pop()); // 3
qDebug("Pop: %d", stack.pop()); // 2
qDebug("Pop: %d", stack.pop()); // 1
qDebug("isEmpty? %s", stack.isEmpty()?"yes":"no");
```

Kolejka – QQueue

- Kontener typu FIFO
  first in, first out

- Dodawanie: `enqueue()`
- Usuwanie: `dequeue()`
- Pobranie pierwszego elementu: `head()`




```
QQueue<int> queue;
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
qDebug("Head: %d", queue.head());    // 1
qDebug("Pop: %d", queue.dequeue()); // 1
qDebug("Pop: %d", queue.dequeue()); // 2
qDebug("Pop: %d", queue.dequeue()); // 3
qDebug("isEmpty? %s", queue.isEmpty()?"yes":"no");
```

Zbiór – QSet

- Wartości w ramach zbioru muszą być unikalne.
- Łatwe sprawdzenie czy wartość jest w zbiorze:

```
QSet<int> primes;
primes << 2 << 3 << 5 << 7 << 11 << 13;
for(int i=1; i<=10; ++i)
    qDebug("%d is %sprime", i, primes.contains(i)?"":"not ");
```

- Iterowanie po wszystkich elementach:

```
foreach(int prime, primes)
    qDebug("Prime: %d", prime);
```

- Można przekonwertować `QList` do `QSet`

```
QList<int> list;
list << 1 << 1 << 2 << 2 << 2 << 3 << 3 << 5;
QSet<int> set = list.toSet();
qDebug() << list; // (1, 1, 2, 2, 2, 3, 3, 5)
qDebug() << set;  // (1, 2, 3, 5)
```

Kolekcje do przechowywania danych typu klucz-wartość

- Klasy `QMap` i `QHash` umożliwiają tworzenie tablic asocjacyjnych

```
QMap<QString, int> map;

map["Helsinki"] = 1310755;
map["Oslo"] = 1403268;
map["Copenhagen"] = 1892233;
map["Stockholm"] = 2011047;

foreach(const QString &key,
map.keys())
    qDebug("%s", qPrintable(key));

if(map.contains("Oslo"))
{
    qDebug("Oslo: %d",
                map.value("Oslo"));
}
```

```
QHash<QString, int> hash;

hash["Helsinki"] = 1310755;
hash["Oslo"] = 1403268;
hash["Copenhagen"] = 1892233;
hash["Stockholm"] = 2011047;

foreach(const QString &key, hash.keys())
    qDebug("%s", qPrintable(key));
```

Wykorzystanie QMap

- Klasa QMap wymaga zdefiniowania operatora operator< dla typu klucza
  Operator ten odpowiada za kolejność kluczy
- Dodawanie elementów zrealizowano przez operator[] lub insert

```
map["Stockholm"] = 2011047;

map.insert("London", 13945000);
```

- Do odczytu należy wykorzystać value w połączeniu z contains

```
if(map.contains("Oslo"))
    qDebug("Oslo: %d",
           map.value("Oslo"));

qDebug("Berlin: %d", map.value("Berlin",42));
```

Użyj value zamiast [] by przypadkiem nie dodać elementu

Opcjonalna wartość domyśla

Jednokierunkowe funkcje skrótu

- QMap typ klucza zależy od definicji wykorzystanej podczas tworzenia QMapy
- QHash wykorzystuje wewnętrzne wartości typu uint
- Klucz za pomocą haszowania zmieniany jest w wartość typu uint
- Wykorzystanie wartości uint zwykle poprawia wydajność kodu
- Wykorzystanie haszowania oznacza, że klucze pamiętane są w dowolnej kolejności
- Funkcja haszująca musi być odporna na występowanie kolizji i efektywna obliczeniowo

Wykorzystanie QHash

- Należy stworzyć funkcję qHash dla klucza i przeciążyć operator operator==

```
uint qHash(const Person &p)
{
    return p.age() ^ qHash(p.name());
}

bool operator==(const Person &first, const Person &second)
{
    return ((first.name() == second.name()) &&
            (first.age() == second.age()));
}
```

Funkcja powinna mieć niską złożoność obliczeniową

- Dodawanie elementów odbywa się identycznie jak w przypadku QMap

Wiele wartości przypisanych do jednego klucza

- `QMultiMap` i `QMultiHash` dostarczają implementację tablic asocjacyjnych, w których wiele wartości może być przypisanych do tego samego klucza

Brak `[]`, trzeba użyć `insert`

`QMap` i `QHash` też to obsługują funkcją `insertMulti()`

```
QMultiMap<QString,int> multiMap;

multiMap.insert("primes", 2);
multiMap.insert("primes", 3);
multiMap.insert("primes", 5);
    ...
multiMap.insert("fibonacci", 8);
multiMap.insert("fibonacci", 13);

foreach(const QString &key, multiMap.uniqueKeys())
{
    QList<int> values = multiMap.values(key);
    QStringList temp;
    foreach(int value, values)
        temp << QString::number(value);

    qDebug("%s: %s", qPrintable(key), qPrintable(temp.join(",")));
}
```

Typy w Qt

- C++ nie definiuje wielkości typów dla typów wbudowanych

sizeof(void*) = ?

ARM = 4 bytes
x86 = 4 bytes
IA64 = 8 bytes
...

Zależne od architektury CPU, systemu operacyjnego, kompilatora

- W przenośności kodu istotne jest zachowanie tego samego rozmiaru typu na każdej platformie

Przenośne typy

Type	Size	Minimum value	Maximum value
uint8	1 byte	0	255
uint16	2 bytes	0	65 535
uint32	4 bytes	0	4 294 967 295
uint64	8 bytes	0	18 446 744 073 709 551 615
int8	1 byte	-128	127
int16	2 bytes	-32 768	32 767
int32	4 bytes	-2 147 483 648	2 147 483 647
int64	8 bytes	-9 223 372 036 854 775 808	9 223 372 036 854 775 807
quintptr	“pointer sized”	n/a	n/a
qptrdiff	“pointer sized”	n/a	n/a
qreal	fast real values	n/a	n/a

Wszystkie typy zdefiniowane są w pliku nagłówkowym `<QtGlobal>`

Typy złożone w Qt

- Qt posiada wiele typów złożonych

QFont

QList

QColor

QString QRect QPen

QBrush

QSize

QPoint QImage

QPixmap

QByteArray

QVariant

- Typy wbudowane obsługiwane są przez konstruktor i metody `to`*`Typ`*

```
QVariant v;
int i = 42;
qDebug() << "Before:" << i; // Before: 42
v = i;
i = v.toInt();
qDebug() << "After:" << i;  // After: 42
```

- Typy złożone obsługiwane są za pomocą metody `setValue` oraz szablonowej `value<`*`type`*`>`

```
QVariant v;
QColor c(Qt::red);
qDebug() << "Before:" << c; // Before: QColor(ARGB 1, 1, 0, 0)
v.setValue(c);
c = v.value<QColor>();      // After: QColor(ARGB 1, 1, 0, 0)
qDebug() << "After:" << c;
```

Własny typ złożony

- Implementacja prostej klasy opisującej osobę

```
class Person
{
public:
    Person();
    Person(const Person &);
    Person(const QString &, int);

    const QString &name() const;
    int age() const;

    void setName(const QString &);
    void setAge(int);

    bool isValid() const;

private:
    QString m_name;
    int m_age;
};
```

Nie musi być QObject.

```
Person::Person() : m_age(-1) {}

...

void Person::setAge(int a)
{
    m_age = a;
}

bool Person::isValid() const
{
    return (m_age >= 0);
}
```

QVariant przechowujący obiekty klasy Person

- Wykorzystanie klasy Person w połączeniu z typem QVariant nie zadziała:

```
qmetatype.h:200: error: 'qt_metatype_id' is not a member of 'QMetaTypeId<Person>'
```

- Deklaracja takiego typu w warstwie metadanych rozwiązuje ten problem:

```
class Person
{
  ...
};

Q_DECLARE_METATYPE(Person)

#endif // PERSON_H
```

QVariant przechowujący obiekty klasy Person

- Jeżeli typ został zarejestrowany w systemie metadanych, Qt może go przechowywać jako QVariant

```
QVariant var;
var.setValue(Person("Ole", 42));
Person p = var.value<Person>();
qDebug("%s, %d", qPrintable(p.name()), p.age());
```

- Wymagania dla własnych obiektów:
  - Publiczny konstruktor domyślny
  - Publiczny konstruktor kopiujący
  - Publiczny destruktor

Ale jest jeszcze jeden problem...

- Większość połączeń z wykorzystaniem mechanizmu slotów i sygnałów jest połączeniami bezpośrednimi
  - Przy połączenia bezpośrednich wszystko działa
  - Ale istnieją połączenia kolejkowane, tj. asynchroniczne (nie-blokujące) – wówczas takie podejście nie zadziała (np. między wątkami)

```
connect(src, SIGNAL(), dest, SLOT(), Qt::QueuedConnection);

...

QObject::connect: Cannot queue arguments of type 'Person'
(Make sure 'Person' is registered using qRegisterMetaType().)
```

Błąd podczas wykonywania programu

Registering the type

- Typ błędu jednoznacznie sugeruje co trzeba zrobić
- Funkcja qRegisterMetaType musi zostać wywołana przed stworzeniem połączenai (zwykle w funkcji main)

```
int main(int argc, char **argv)
{
    qRegisterMetaType<Person>();
    ...
```

Pliki i systemy plików

- Sposób dostępu do systemu plików zależy od systemu operacyjnego, co rodzi następujące problemy:
  - System może mieć napędy (dyski) lub katalog root
  - Systemy mogą mieć różne separatory ścieżki dostępu “/” lub “\”
  - Systemy mogą mieć różną lokalizację katalogu do przechowywania danych tymczasowych
  - Różną lokalizację dla dokumentów użytkownika
  - Różną lokalizację dla folderu aplikacji

Ścieżki

- Klasa `QDir` dedykowana jest do obsługi ścieżek

```
QDir d = QDir("C:/");
```

- Posiada składowe statyczne umożliwiające określenie punktu początkowego

```
QDir d = QDir::root(); // C:/ on windows

QDir::current()  // Current directory
QDir::home()     // Home directory
QDir::temp()     // Temporary directory

// Executable directory path
QDir(QApplication::applicationDirPath())
```

Przeszukiwanie zawartości katalogu

- Metoda `entryInfoList` zwraca listę informacji o zawartości katalogu

```
QFileInfoList infos = QDir::root().entryInfoList();
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

Kolejność losowa

- W celu ograniczenia liczby elementów można zastosować filtr

```
QDir::Dirs
QDir::Files
QDir::NoSymLinks
```

Katalogi? Pliki? Dowiązania?

```
QDir::Readable
QDir::Writable
QDir::Executable
```

Jakie pliki?

```
QDir::Hidden
QDir::System
```

Jakie pliki?

Przeszukiwanie zawartości katalogu

- Można również posortować listę

```
QDir::Name
QDir::Time
QDir::Size
QDir::Type

QDir::DirsFirst
QDir::DirsLast

QDir::Reversed
```

Sortuj po...

Katalogi przed czy po plikach

Odwróć kolejność

- Lista wszystkich katalogów katalogu głównego sortowana po nazwie

Filtr

Kolejność

```
QFileInfoList infos =
    QDir::root().entryInfoList(QDir::Dirs, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

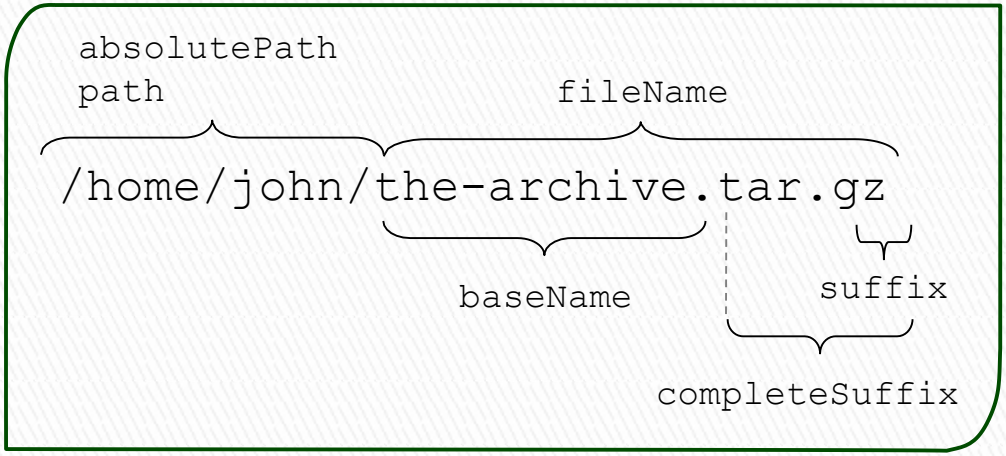
Przeszukiwanie zawartości katalogu

- Można również filtrować po rozszerzeniu / nazwie pliku

```
QFileInfoList infos =
    dir.entryInfoList(QStringList() << "*.cpp" << "*.h",
                      QDir::Files, QDir::Name);
foreach(const QFileInfo &info, infos)
    qDebug("%s", qPrintable(info.fileName()));
```

QFileInfo

- Każdy obiekt `QFileInfo` ma pewna liczbę metod
  - `absoluteFilePath` – pełna ścieżka elementu
  - `isDir` / `isFile` / `isRoot` – typ elementu
  - `isWriteable` / `isReadable` / `isExecutable` – uprawnienia



Otwieranie i odczyt plików

- Dostęp do plików realizowany jest za pomocą klasy `QFile`

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");
```

```
QByteArray data = f.readAll();
processData(data);
```

Wczytuje cały plik

```
f.close();
```

```
while(!f.atEnd())
{
    QByteArray data = f.read(160);
    processData(data);
}
```

Odczytuje do 160 bajtów za jednym wykonaniem

Zapis do plików

- By zapisywać dane do pliku należy go otworzyć w trybie `WriteOnly` i wykorzystać metodę write

```
QFile f("/home/john/input.txt");

if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QByteArray data = createData();
f.write(data);

f.close();
```

- Pliki mogą również zostać otwarte w trybie `ReadWrite`
- Flagi `Append` lub `Truncate` mogą być zastosowane w celu ustawienia trybu dopisywania danych bądź nadpisywania (plik zostanie wyczyszczony zaraz po otwarciu

```
if (!f.open(QIODevice::WriteOnly|QIODevice::Append))
```

Klasa QIODevice

- QFile dziedziczy po QIODevice
- Konstruktory dla QTextStream oraz QDataStream jako argument przyjmują również wskaźnik na QIODevice
- Istnieje szereg implementacji klasy (abstrakcyjnej) QIODevice
  - QBuffer – do odczytu i zapisu w buforach
  - QextSerialPort – do komunikacji przez RS232
  - QAbstractSocket – baza dla klas dotyczących połączeń z wykorzystaniem TCP, SSL i UDP
  - QProcess – do odczytu i zapisu standardowego wejścia/wyjścia danego procesu

Dowiązanie strumieni do pliku

- Metody read i write są niewygodne w wielu sytuacjach, szczególnie przy obsłudze bardziej złożonych typów
- Współczesne podejście zakłada używania operatorów strumieniowych
- Qt udostępnia dwa rodzaje operatorów strumieniowych
  - Do obsługi plików tekstowych
  - Do obsługi plików binarnych

QTextStream

- Klasa `QTextStream` obsługuje zapis i odczyt danych z plików tekstowych
- Klasa umożliwia
  - Określenie strony kodowej zapisu pliku
  - Określenie ilości linii i słów
  - Rozpoznawanie i przetwarzanie liczb

Zapis do strumienia tekstowego

- Do zapisu wykorzystuje się operator << (tak jak w STL)

```
QFile f(...);
if(!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QTextStream out(&f);
out << "Primes: " << qSetFieldWidth(3) << 2 << 3 << 5 << 7 << endl;
```

Results in:

```
Primes:   2  3  5  7
```

Odczyt za pomocą strumieni tekstowych

- Możliwy jest odczyt pliku linia po linii

```
QTextStream in(&f);
while(!in.atEnd())
    qDebug("line: '%s'", qPrintable(in.readLine()));
```

- Można również odczytywać ciągi i liczby

```
QTextStream in(&f);
QString s;
int i;
in >> s >> i;
```

- Metoda `atEnd` umożliwia sprawdzenie, czy osiągnięto koniec pliku

Obsługa plików binarnych

- Do obsługi plików binarnych używa się klasy `QDataStream`
  - Umożliwia określenie kolejności bajtów (domyślnie big endian)
  - Wspiera typu proste (wbudowane)
  - Wspiera typy złożone Qt
  - Umożliwia dodanie złożonych typów użytkownika

```
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream ds(&f);
ds << QString("Unicode string data");
```

Strumień danych jako format pliku

- Opieranie formatu pliku na klasie `QDataStream` wymaga pamiętania o następujących sprawach
  - Wersja formatu – jako że są zapisywane obiekty Qt konieczne jest przechowywanie informacji o wersji w której są zapisywane. Używając `QDataStream::setVersion` można wymusić konkretną wersję serializacji.
  - Informacja o typach – Qt nie dodaje informacji o typie danych, stąd należy zapisywać i odczytywać dane w określonej kolejności.
  - Kolejność bajtów – Dane zapisywane są w systemie big endian domyślnie, ale można to zmienić używając `QDataStream::setByteOrder`.

Strumień danych jako format pliku

```
QFile f("file.fmt");
if (!f.open(QIODevice::WriteOnly))
    qFatal("Could not open file");

QDataStream out(&f);
out.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

out << value;
out << text;
out << color;
```

Od wersji Qt 1.0+

Należy dopasować kolejność i typy danych

```
QFile f("file.fmt");
if (!f.open(QIODevice::ReadOnly))
    qFatal("Could not open file");

QDataStream in(&f);
in.setVersion(QDataStream::Qt_4_6);

quint32 value = ...;
QString text = ...;
QColor color = ...;

in >> value;
in >> text;
in >> color;
```

Można wykorzystać typ QVariant by nie pamiętać kolejności i typów danych.

Strumienie i własne typy danych

- Dla własnych typów danych należy zaimplementować operatory << oraz >>.

```
QDataStream &operator<<(QDataStream &out, const Person &person)
{
    out << person.name();
    out << person.age();
    return out;
}

QDataStream &operator>>(QDataStream &in, Person &person)
{
    QString name;
    int age;
    in >> name;
    in >> age;
    person = Person(name, age);
    return in;
}
```

Odczyt bezpośredni

Strumienie i własne typy danych

- Jeżeli wykorzystywany jest obiekt typu QVariant do przechowywania obiektu użytkownika trzeba zarejestrować zaimplementowane operatory:

```
qRegisterMetaTypeStreamOperators<Person>("Person");
```

- QVariant dodaje nazwę typu w swojej wewnętrznej reprezentacji

```
00: 0x00 0x00 0x00 0x7f 0x00 0x00 0x00 0x00   ____________
08: 0x07 0x50 0x65 0x72 0x73 0x6f 0x6e 0x00   _Person_
16: 0x00 0x00 0x00 0x06 0x00 0x4f 0x00 0x6c   ______O_l
24: 0x00 0x65 0x00 0x00 0x00 0x2a             e____*
```

Do sprawdzenia:

- Stwórz
  - Type::Type() – Publiczny konstruktor domyślny
  - Type::Type(const Type &other) – Publiczny konstruktor kopiujący
  - Type::~Type() – Publiczny destruktor
  - QDebug operator<< – Ułatwia późniejsze odpluskwianie
  - QDataStream operator<< i >> – Strumieniowanie
- Zarejestruj
  - Q_DECLARE_METATYPE – W nagłówku
  - qRegisterMetaType – W funkcji main
  - qRegisterMetaTypeStreamOperators – W funkcji main