Programowanie generyczne w Qt

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UMCS

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Projekt „Programowa i strukturalna reforma systemu kształcenia na Wydziale Mat-Fiz-Inf”.
Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego.

Człowiek-najlepsza inwestycja

UNIWERSYTET MARII CURIE-SKŁODOWSKIEJ
WYDZIAŁ MATEMATYKI, FIZYKI I INFORMATYKI
INSTYTUT INFORMATYKI

Programowanie generyczne w Qt

Paweł Mikołajczak

LUBLIN 2012

Instytut Informatyki UMCS
Lublin 2012

Paweł Mikołajczak

PROGRAMOWANIE GENERYCZNE W QT

Recenzent: Jakub Smołka
Opracowanie techniczne: Marcin Denkowski
Projekt okładki: Agnieszka Kuśmierska

Praca współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Publikacja bezpłatna dostępna on-line na stronach
Instytutu Informatyki UMCS: informatyka.umcs.lublin.pl

Wydawca
Uniwersytet Marii Curie-Skłodowskiej w Lublinie
Instytut Informatyki
pl. Marii Curie-Skłodowskiej 1, 20-031 Lublin
Redaktor serii: prof. dr hab. Paweł Mikołajczak
www: informatyka.umcs.lublin.pl
email: dyrii@hektor.umcs.lublin.pl

Druk
FIGARO Group Sp. z o.o. z siedziba w Rykach
ul. Warszawska 10
08-500 Ryki
www: www.figaro.pl

ISBN: 978-83-62773-33-6

SPIS TREŚCI

PRZEDMOWA

Platforma programistyczna Qt wykorzystywana jest do tworzenia eleganckich i bardzo zaawansowanych aplikacji. Platforma programistyczna Qt została stworzona przez norweskich programistów pracujących w firmie Quasar Technolgies, w latach późniejszych firma zmieniała nazwę na Trolltech. Firma Trolltech została wykupiona przez znany koncern fiński – Nokia. Po przejęciu firma najpierw otrzymała nową nazwę Qt Software, w roku 2009 przemianowana została na Qt Development Frameworks. Aktualna wersja Qt nosi numer 4 i jest rozprowadzana w dwóch wersjach – komercyjnej i wolnej.
Strona domowa Qt znajduje się na stronie http://qt.nokia.com, społeczne wsparcie techniczne Qt znajduje się na stronie http://www.qtcentre.org, bardzo przydatne darmowe aplikacje bazujące na Qt znajdują się na stronie http://www.qt-apps.org.
Biblioteka Qt jest dostępna w systemach MS Windows, Linux i Mac OS, a także dla urządzeń przenośnych.

Wielu programistów wykorzystuje programowanie obiektowe tworzone przy pomocy języka C++ do tworzenia aplikacji. Paradygmat programowania obiektowego nie jest jedynym paradygmatem wykorzystywanym w tworzeniu wydajnych aplikacji. Język C++ jest językiem o bardzo silnym zróżnicowaniu typów. To podejście wymaga za każdym razem tworzenia procedur w zasadzie wykonujących to samo zadanie, ale musimy tworzyć funkcje operujące na konkretnym typie danych. Realizacja na przykład potęgowania wymaga napisania odrębnej funkcji dla typu int i nowej funkcji dla typu double. Już dość dawno zauważono, że wiele procedur wykorzystywanych w programach jest powtarzanych – nie ma sensu, aby programista za każdym razem od nowa tworzył nowy kod. Klasycznym przykładem jest wykorzystanie algorytmu sortowania. Oczywiście, mamy wiele typów algorytmów sortowania, z punktu widzenia wytwórców oprogramowania, chodzi, aby była dostępna procedura, która w miarę dobrze wykona sortowanie, chcemy mieć jedną, dobrą funkcję biblioteczna wykonującą żądane zadanie. Te potrzeby spowodowały, że powstała koncepcja stworzenie nowego paradygmatu programowania – programowanie uogólnione (ang. *generic programming*). Dzięki wysiłkowi wielu informatyków (głównie A. Stepanova) powstała praktyczna koncepcja programowania ogólnego oraz narzędzie do realizacji tego programowania – Standardowa Biblioteka Wzorców, STL (ang. ***S**tandard **T**emplate **L**ibrary*).

Platforma Qt umożliwia pisanie programów realizujących paradygmat programowania ogólnego lub a także pozwala w aplikacjach pisanych obiektowo korzystać z elementów biblioteki STL.

Wiele aplikacji wymaga opracowania skomplikowanych procedur. Do celów badawczych i testowania algorytmów, procedury konsolowe są bardzo przydatne.

Podstawą programów ogólnych są klasy kontenerowe. W Qt możemy wykorzystywać klasyczne klasy kontenerowe pochodzące z STL a także opracowane specjalnie dla Qt klasy kontenerowe Qt (QTL). Kontenery Qt i inne elementy do obsługi tych klas działają bardzo podobnie, jak te pochodzące z STL, ale są też subtelne różnice.

W niniejszym podręczniku omówimy tworzenie procedur korzystających z STL oraz QTL pisanych dla konsoli. Omówimy także zasady tworzenia procedur korzystających z klas kontenerowych Qt. Należy pamiętać, że Qt jest to bardzo rozbudowana platforma. Mamy możliwość pisania programów korzystających tylko z kontenerów STL, tylko z kontenerów QTL, możemy w programie zamieścić także kontenery STL i QTL. Komplikacje mogą być jeszcze większe, ponieważ w programie możemy używać iteratorów i algorytmów STL i QTL (w Qt mamy możliwość używania dwóch różnych typów iteratorów). Szczególna uwaga musi być zachowana w plikach nagłówkowych (plik do obsługi kontenera QTL zaczyna się zawsze od litery „Q") oraz przy deklaracjach obiektów klas QTL (nazwa kontenera także zaczyna się zawsze od litery „Q"). Gdy chcemy wykorzystać algorytm STL musimy włączyć plik nagłówkowy:

```
#include <algorithm>
```

lub

```
#include <algorithm.h>
```

W przypadku korzystania z algorytmu Qt musimy włączy plik nagłówkowy:

```
#include <QtAlgorithms>
```

Autor jest wdzięczny dr Marcinowi Denkowskiemu i mgr Krzysztofowi Dmitrukowi z Zakładu Technologii Informatycznych Instytutu Informatyki UMCS w Lublinie, którzy przeczytali i skomentowali wstępną wersję skryptu. Ich komentarze i proponowane uzupełnienia były inspirujące.

ROZDZIAŁ 1

QT- TWORZENIE APLIKACJI KONSOLOWEJ

1.1. Wstęp

Qt jest doskonałą platforma programistyczną głównie przeznaczona do tworzenia programów komputerowych obsługiwanych przez graficzne interfejsy użytkownika (GUI). Istnieje bogata literatura poświecona tworzeniu okienkowych aplikacji, Qt, ale dość trudno jest znaleźć informacje poświecone tworzeniu aplikacji konsolowych. W tym rozdziale omówimy tworzenie aplikacji konsolowych w Qt korzystając z podstawowego narzędzia biblioteki Qt, jakim jest ***kreator Qt***. Aplikacje będą tworzone w ramach wolnej wersji Qt. Wszystkie programy są kompilowane w systemie Windows 7, ale powinny być także uruchamiane w systemie Linux.

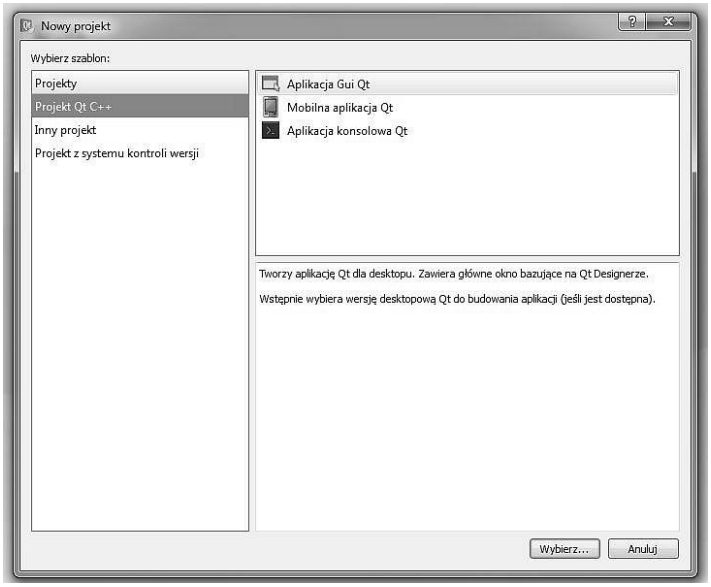
1.2. Szablon aplikacji konsolowej

Zgodnie z powszechnie przyjętymi zasadami nauki języka programowania i stosowania platform programistycznych, pokażemy jak napisać prosty program wyświetlający na ekranie monitora komputerowego żądany tekst.

Uruchamiamy pakiet Qt, na ekranie monitora pojawia się okno systemu Qt (rys.1).

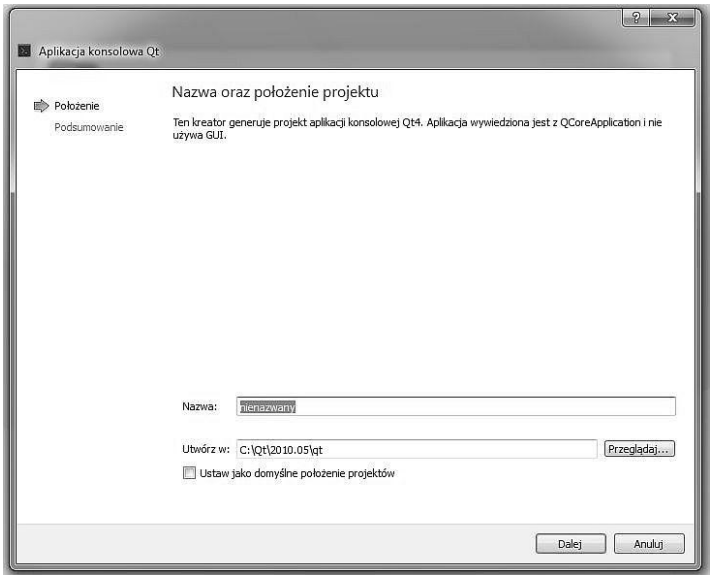
Rys. 1. Pierwsze okno kreatora Qt (wersja: Qt Creator 2.0.1, platforma Qt 4.7.0 (32 bit), lipiec, 2010)

W oknie kreatora Qt wybieramy opcję „Utwórz projekt”. W wyniku tej operacji pojawi się drugie okno (rys.2.) pozwalające na wybór typu aplikacji.



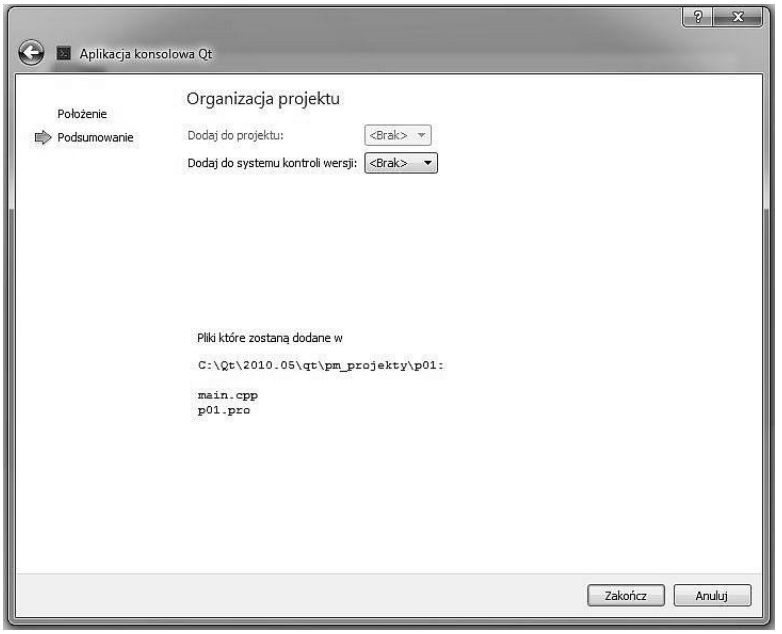
Rys.2. Wybór typu aplikacji

W tym oknie wybieramy opcję „Aplikacja konsolowa Qt” i wybieramy przycisk „Wybierz…”. Pojawia się kolejne okno (rys.3).



Rys.3. Nazwa i umiejscowienie projektu

W tym oknie podajemy nazwę projektu oraz miejsce, gdzie będzie przechowywany nasz projekt (możemy utworzyć specjalny katalog o nazwie pm_projekty). W okienku „Nazwa: „ wpisujemy nazwę naszego projektu, może to być projekt o nazwie p01. Po naciśnięciu przycisku „Dalej” pokazuje się kolejne okno (rys. 4).



Rys.4. Nazwa i lokalizacja projektu

W oknie „Aplikacja konsolowa Qt” podane są informacje związane z organizacją projektu.
Zgodnie z informacją podaną w oknie „Aplikacja konsolowa Qt” (rys. 4) kreator utworzył dwa pliki:

main.cpp
p01.pro

Plik main.cpp ma następującą treść:

Wydruk 1.1. Plik main.cpp (wyprodukowany przez kreator Qt)

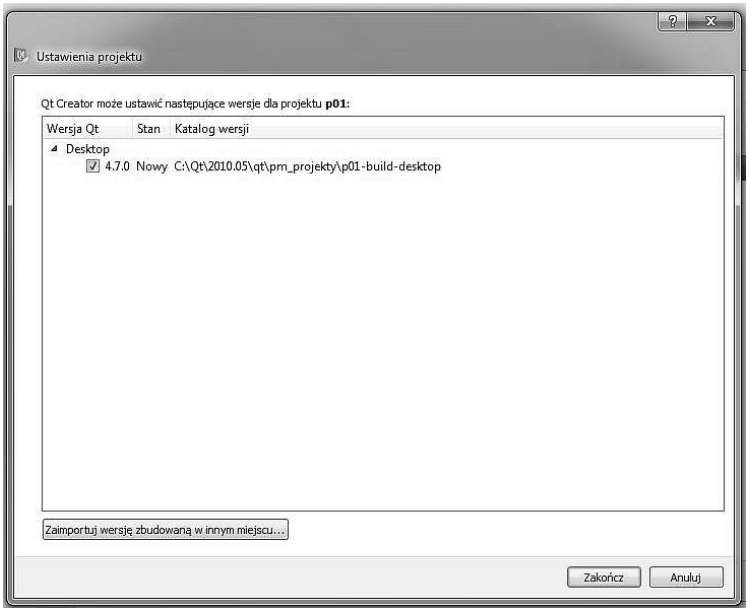
```
#include <QtCore/QCoreApplication>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    return a.exec();
}
```

Drugi plik wyprodukowany przez kreator Qt o nazwie p01.pro (nazwa pliku podawana jest przez programistę, w naszym przypadku jest to nazwa p01) ma następującą treść:

Wydruk 1.2. Plik p01.pro (wyprodukowany przez kreator Qt)

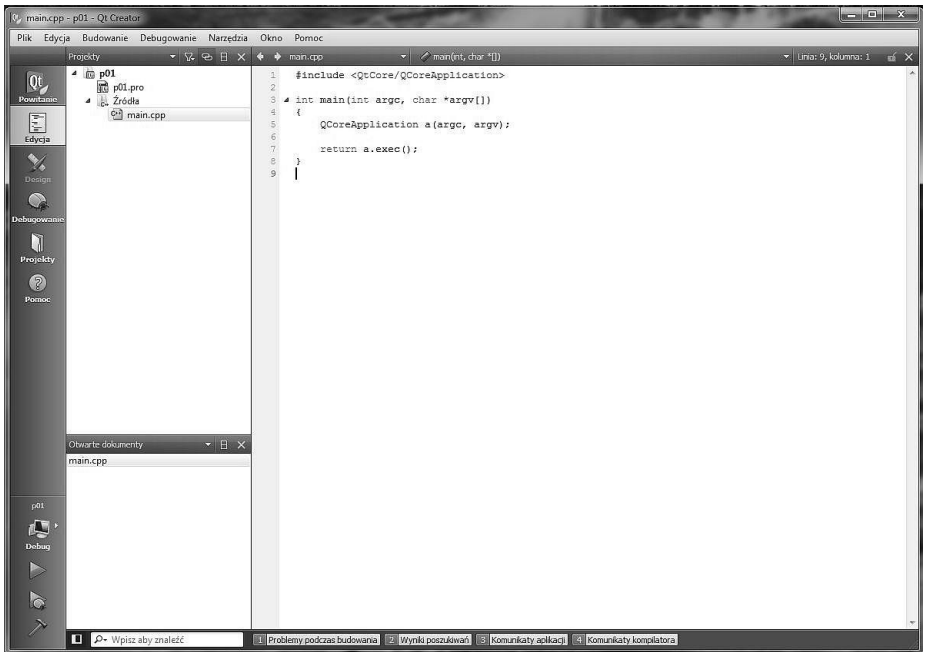
```
#-------------------------------------------------
#
# Project created by QtCreator 2012-06-07T13:45:09
#
#-------------------------------------------------
QT       += core
QT       -= gui
TARGET    = p01
CONFIG   += console
CONFIG   -= app_bundle
TEMPLATE  = app
SOURCES  += main.cpp
```

Jest to plik projektu Qt, potrzebny do obsługi naszej aplikacji, w którym nic nie musimy zmieniać.
Jeżeli wszystko się zgadza należy w oknie „Aplikacja konsolowa Qt” (rys. 4) nacisnąć przycisk „Zakończ”. Pojawi się kolejne okno kreatora Qt (rys.5) z dodatkowymi informacjami.



Rys.5. Okno ustawienia projektu

Jeżeli nie msą potrzebne inne akcje, należy nacisnąć przycisk „Zakończ”. Pojawi się okno edytora (rys.6)



Rys.6. Okno środowiska Qt po wygenerowaniu szablonu programu Qt

Jak widać jest to treść pokazanego na wydruku 1.1. pliku main.cpp.
W tym oknie piszemy tekst programu i mamy możliwość jego kompilacji i uruchomienia. Z lewej strony u dołu głównego okna znajdują się ikony do uruchamiania programu (rys.7).

Rys. 7. Ikony do kompilacji i wykonania programu

Dla nas w tym momencie najważniejszą jest ikona przedstawiająca zielony trójkąt. Służy ona do wykonania programu.
W oknie edytora pojawia się szablon pliku main.cpp, który pokazany jest na wydruku 1.1. Ponieważ jest to wstępny szablon aplikacji konsolowej, ten program nic nie robi. Jest jeszcze jedna cecha charakterystyczna tego programu – po uruchomieniu będzie wykonywał się bez końca!

1.3. Prosty program konsolowy

Mając gotowy szablon, musimy zmodyfikować go tak, aby powstał program wykonujący żądane zadanie. Chcemy, aby wynikiem działania naszego programu było pokazanie się żądanego napisu na ekranie monitora. Po modyfikacji, nasz program przybiera następującą postać:

Wydruk 1.3. Prosty program konsolowy

```
#include <QtCore/QCoreApplication>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    cout << "witaj w Qt,aplikacja konsolowa "<<endl;
    return a.exec();
}
```

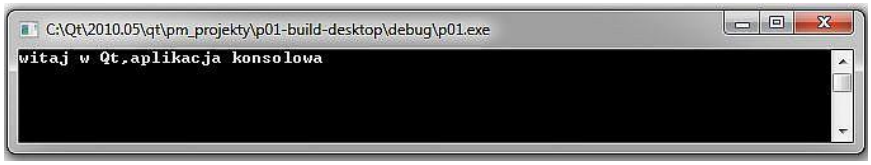
W wygenerowanym automatycznie przez Qt szablonie umieściliśmy następujące dyrektywy i instrukcje:

```
#include <iostream>
using namespace std;
```

oraz

```
cout << "witaj w Qt,aplikacja konsolowa "<<endl;
```

Wykorzystując przycisk uruchomienia (zielona ikona w kształcie trójkąta) lub kombinację klawiszy ctrl+R, powodujemy wykonanie programu, wynik programu pokazany jest w oknie wyników (rys.8).



Rys.8. Wynik wykonania pierwszego programu

Pokazany prosty program można rozbudować o nowe elementy.
Mamy możliwość sterowania czasem wyświetlania wyniku – konkretnie ustalmy czas pracy programu. W tym celu uruchamiamy program i po ustalonym czasie wysyłamy sygnał, aby go zatrzymać. Sterowanie czasem wykonania programu realizuje metoda, singleShot(), która wymaga dołączenia pliku nagłówkowego <QTimer>. Klasa QTimer dostarcza programowalny interfejs stopera, (ang. timers). Funkcja singleShot() wymaga podania czasu w milisekundach. W większości systemów, klasa stoper w klasie, QTimer ma rozdzielczość jednej milisekundy.

1.4. Obsługa operacji wejścia/wyjścia

Kolejnym rozszerzeniem poprzedniego programu jest wykorzystanie metody qDebug() do obsługi wyjścia. Metoda qDebug() w systemie Windows wysyła wiadomość na konsolę w przypadku aplikacji konsolowej, w innych przypadkach wiadomość jest wysyłana do debuggera. Jeżeli do funkcji qDebug() będzie przekazany łańcuch i lista argumentów, funkcja będzie zachowywała się bardzo podobnie do funkcji printf() znanej z języka C. Funkcja qDebug wymaga pliku nagłówkowego <QDebug>.

Wydruk 1.4. Program konsolowy, metoda qDebug()

```
#include <QtCore/QCoreApplication>
#include <QTimer>
#include <QDebug>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int t = 10000;     //argument singleShot()
    QCoreApplication a1(argc, argv);
    cout << "wyjscie - strumien cout\n";
    qDebug() << "wyjscie - metoda qDebug\n";
    QTimer :: singleShot(t,&a1,SLOT(quit()));
    return a1.exec();
}
```

Wynikiem uruchomienia programu jest komunikat:

Rys.9. Wynik wykonania programu z wydruku 1.4

W instrukcji:

```
QCoreApplication a1(argc, argv);
```

wygenerowana przez kreator Qt nazwa aplikacji **app** została zamieniona na **a1**.
Do obsługi operacji wejścia i wyjściowych mamy jeszcze jedną użyteczną klasę – QTextStream. Jest ona bardzo wydajna w obsłudze wejścia/wyjścia aplikacji konsolowych. Klasa QTextStream wymaga dołączenia pliku nagłówkowego <QTextStream>. Przy pomocy strumienia tekstowego QTextStream możemy umieścić informację w standardowym wyjściu – **stdout**. W programie utworzyliśmy obiekt **qout**:

```
QTextStream qout(stdout);
qout << "wyjscie - QTextStream\n";
```

Wykorzystanie klasy QTextStream (także klasy iostream i metody qDebug) pokazane jest na kolejnym wydruku.

Wydruk 1.5. Program konsolowy, klasa QTextStream

```
#include <QtCore/QCoreApplication>
#include <QTimer>
#include <QDebug>
#include <iostream>
#include <QTextStream>
using namespace std;

int main(int argc, char *argv[])
{   int t=10000;
    QCoreApplication a1(argc, argv);
    QTextStream qout(stdout);
    qout << "wyjscie - QTextStream\n";
    cout << "wyjscie - strumien cout\n";
    qDebug() << "wyjscie - metoda qDebug\n";
    QTimer :: singleShot(t,&a1,SLOT(quit()));
    return a1.exec();
}
```

Po uruchomieniu programu otrzymujemy następujący komunikat:

Rys.10. Wynik wykonania programu z wydruku 1.5

Jak wyraźnie widać na rys.10. spodziewanej informacji o zastosowaniu klasy QTextStream nie otrzymaliśmy na wyjściu. Na wyjściu powinien pojawić się napis:

wyjscie – QTextStream

To nieoczekiwane zachowanie się strumienia tekstowego klasy QTextStream spowodowane jest faktem, że QTextStream jest buforowane. Dopóki bufor strumienia nie zostanie opróżniony, żaden komunikat nie pojawi się na wyjściu. Opróżnienie bufora można wykonać albo przy pomocy wyspecjalizowanej metody **flush()** albo zastępując znak **\n** na końcu łańcucha manipulatorem biblioteki IOStream – **endl**. Uważa się, że zamiast polecenia qout.flush() bardziej wydajne jest użycie manipulatora endl, ponieważ bufor jest natychmiast opróżniany. Należy też pamiętać, że qDebug() ma wbudowany manipulator endl i dzięki temu zachowuje się zgodnie z naszym oczekiwaniem. Poprawiony program pokazuje kolejny listing.

Wydruk 1.6. Klasa QTextStream, endl

```
#include <QtCore/QCoreApplication>
#include <QTimer>
#include <QDebug>
#include <iostream>
#include <QTextStream>

using namespace std;

int main(int argc, char *argv[])
{   int t=10000;
    QCoreApplication a1(argc, argv);
    QTextStream qout(stdout);
    qout << "wyjscie - QTextStream"<<endl;
    cout << "wyjscie - strumien cout\n";
    qDebug() << "wyjscie - metoda qDebug\n";
    QTimer :: singleShot(t,&a1,SLOT(quit()));
    return a1.exec();
}
```

Teraz otrzymamy oczekiwany wynik:

Rys.11. Wynik wykonania programu z wydruku 1.6

1.5. Obsługa plików

Dane można wprowadzać zarówno z klawiatury jak i z pliku. Kolejny przykład ilustruje obsługę plików w Qt. Program czyta dane z pliku i wyświetla je na ekranie monitora. Dane zapisane są w pliku tekstowym test_1.txt umieszczonym na dysku **C** w katalogu Users\mikfiz\. Jak wiemy, aby można było obsługiwać pliki zapisane na dowolnym nośniku informacji, musi być ustanowiona komunikacja pomiędzy plikiem i programem. Nawiązanie komunikacji nazywamy „otwarciem pliku". Otwieranie pliku obsługują odpowiednie funkcje.

Wydruk 1.7. Klasa QTextStream, QFile

```
#include <QtCore/QCoreApplication>
#include <QFile>
#include <QTextStream>
#include <iostream>
#define np "c:\\Users\\mikfiz\\test_1.txt"
using namespace std;
int main(int argc, char *argv[])
{   QCoreApplication a(argc, argv);
    QFile plik(np);
    plik.open(QIODevice::ReadOnly);
    QTextStream out(&plik);
    QString file_text = out.readAll();
    cout << file_text.toAscii().data();
    return a.exec();
}
```

Wynikiem uruchomienia programu jest następujący komunikat:

Rys.12. Wynik wykonania programu z wydruku 1.7

Do obsługi plików w Qt potrzebna jest klasa, QFile, która dostarcza niezbędne metody i inne narzędzia do czytania i zapisywania informacji w plikach. Klasa QFile obsługuje pliki tekstowe i binarne. W obsłudze plików przydatne są dwie klasy: QTextStream i QDataStream. Klasa QTextStream obsługuje strumienie tekstowe. Klasa QDataStream wykorzystywana jest do obsługi plików binarnych. Producent twierdzi, że strumień danych obsługiwany przez klasę QDataStream jest w stu procentach przenaszalny (nie zależy ani od systemu operacyjnego ani od CPU).

W linii:

```
#define np "c:\\Users\\mikfiz\\test_1.txt"
```

podana została ścieżka dostępu do pliku test_1.txt, z którego chcemy odczytać zapisane dane, w naszym przykładzie są one tekstowe. Należy pamiętać, że podając ścieżkę dostępu nie można użyć separatora '\', QFile oczekuje separatora w postaci '\\' lub '/'! Można wykorzystać metodę exists(), żeby sprawdzić, czy plik istnieje a także wykorzystać metodę, remove() aby istniejący plik usunąć. Plik może być obsługiwany, gdy jest otwarty (tzn., gdy zostanie zbudowana komunikacja pomiędzy plikiem a programem. Plik jest otwarty przy pomocy metody open(), zamykany przy pomocy metody close() i opróżniany przy pomocy metody flush().
W linii:

```
plik.open(QIODevice::ReadOnly);
```

ustawiany jest tryb obiektu QFile. Dane zwykle obsługiwane są przez QDataStream lub QTextStream, ale można bezpośrednio wykorzystać funkcje klasy, QIODevice takie jak read(), readLine(), readAll(), write(). Klasa QIODevice jest klasa bazową dla wszystkich urządzeń input/output w Qt. Dostarcza narzędzi do czytania i zapisywania pakietów danych w obiektach klas QFile, QBuffer i QTCPSocket. W obiektach QFile można także wykorzystać metody getChar(), putChar() i ungetChar(), obsługują one pojedyncze znaki.
W tabeli 1.1. umieszczone są wszystkie tryby otwierania plików w Qt.

Tabela 1.1. Tryby otwierania plików.

Tryb	Opis
`NotOpen`	Plik nie jest otwary
`ReadOnly`	Plik jest otwarty do czytania
`WriteOnly`	Plik jest otwarty do zapisywania
`ReadWrite`	Plik jest otwarty do czytania i zapisywania
`Append`	Plik jest otwarty, indeks pliku ustawiony jest na końcu, co oznacza, że informacja zostanie dopisana do informacji już zapisanej.
`Truncate`	W tym trybie, plik zostanie obcięty, oznacza to, że cała wcześniejsza zawartość pliku zostanie utracona
`Text`	Podczas czytania terminator końca linii zostanie zamieniony na specyfikator '\n'. Gdy plik jest zapisywany, terminator końca linii jest zamieniany na lokalny symbol, np. na '\r\n' dla Win32
`Unbuffered`	Bufor w pliku jest pominięty

W linii:

```
QTextStream out(&plik);
```

tworzony jest obiekt klasy QTextStream (w przykładzie jest to ***out***). Obiekt ten będzie czytał tekst z pliku.
W linii

```
QString file_text = out.readAll();
```

Tworzony jest obiekt QString o nazwie file_text. Przy pomocy metody readAll() ze strumienia tekstowego czytane są dane i umieszczane w file_text. Metod readAll() czyta cały strumień, nie należy nadużywać tej metody, gdy plik jest bardzo duży. Rekomenduje się stosować metodę readLine(), która czyta tekst linia po linii, co daje większą kontrolę nad ilością przesyłanych danych.
W ostatniej linii:

```
cout << file_text.toAscii().data();
```

występuje polecenie wydruku tekstu umieszczonego w pliku test_1.txt na ekranie monitora. Metoda toAscii() zwraca 8-bitową reprezentacje łańcucha, jako obiekt klasy QByteArray. Metoda data() klasy QByteArray zwraca wskaźnik do danych przechowywanych w tablicy bitów (ang. *the byte array*). Klasa QByteArray obsługuje tablice bitów. Obiekty QByteArray mogą być wykorzystane do przechowywania bitów i tradycyjnych łańcuchów (tzn. zbiorów znaków, kodowanych na ośmiu bitach, tak jak to jest w przypadku typu ***char*** w C++). Stosowanie QByteArray jest znacznie wygodniejsze niż używanie specyfikatora ***const char ****. Obiekt QByteArray przechowuje łańcuch zakończony terminatorem '\0'. Pamiętamy, że oprócz obiektów klasy QByteArray, dane mogą być przechowywane także w obiektach klasy QString. Rekomenduje się używanie raczej obiektów QString, gdyż przechowują one 16-bitowe znaki Unicode. Zaletą stosowania obiektów klasy QByteArray jest fakt, że do przechowywania danych potrzebują mniej pamięci niż obiekty klasy QString.

1.6. Program konsolowy z widgetami.

Programy konsolowe mogą wykorzystywać pewne elementy biblioteki graficznej Qt bez pełnego obciążania systemu graficznym interfejsem. Możemy budować proste programy konsolowe korzystające z koncepcji slotów i sygnałów oraz zdarzeń, tak charakterystycznych dla eleganckich programów wykorzystujących GUI tworzonych na platformie Qt. W programach konsolowych wykorzystamy widety w bardzo prostej postaci, minimalna ich obsługą. Przypomnijmy, że ***widget*** jest graficznym obiektem, wykorzystywanym w programach GUI.

Klasyczne widgety to okienka z poleceniami czy wszelkiego typu paski przewijania. Nazwa ***widget*** jest skrótem od **wi**ndow ga**dget**.

> UWAGA!
> W programach konsolowych z widgetami należy usunąć w pliku ***nazwa.pro*** (generowanym przez Qt Creator) polecenie:
>
> QT -= gui
>
> Polecamy symbol, # który w tym pliku jest symbolem komentarza:
>
> #QT -= gui

Pokażemy prosty przykład wykorzystania widgeta. Program wyświetli komunikat w okienku. Będziemy mieli możliwość manipulowania okienkiem, np. można je przesuwać po ekranie, powiększać czy po prostu zamknąć.

Jak pamiętamy po uruchomieniu Creatora pojawi się szablon pliku źródłowego. W celu zastosowania widgetów w programie konsolowym musimy dokonać kilku zmian. Po pierwsze musimy zmienić treść pliku z rozszerzeniem ***pm10.pro***. Nazwa ***pm10*** została nadana przez programistę na żądanie Creatora.

Wygenerowny szablon pm10.pro po zmianach ma następującą postać:

Wydruk 1.8. Program konsolowy z widgetem, ***pm10.pro***

```
#-------------------------------------------------
#
# Project created by QtCreator 2012-06-14T18:20:29
#
#-------------------------------------------------

QT       += core
#QT      -= gui

TARGET    = pm10
CONFIG   += console
CONFIG   -= app_bundle
TEMPLATE  = app

SOURCES  += main.cpp
```

Więcej zmian wymaga szablon pliku źródłowego pm10.cpp wygenerowany przez Creator. Ma on postać:

Wydruk 1.9. Program konsolowy z widgetem, ***pm10.cpp***

```
#include <QtCore/QCoreApplication>
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    return a.exec();
}
```

W programach konsolowych z widgetami wykorzystamy klasę QApplication, która jest podstawą aplikacji wykorzystujących model zdarzeniowy Qt. Polecenie szablonu:

```
#include <QtCore/QCoreApplication>
```

zamieniamy na polecenie:

```
#include <QApplication>
```

Plik zawiera definicje klasy QApplication. Tą klasę używają wszystkie programy, które korzystają z graficznych elementów biblioteki Qt. Ta klasa inicjalizuje oraz kończy działanie programu, przetwarza pętlę zdarzeń i umożliwia ich obsługę, gdy korzystamy z elementów graficznych interfejsu. Klasa ta dziedziczy po klasie QCoreApplication. Pamiętamy, że klasa QCoreApplication obsługuje wszystkie programy konsolowe. Z kolei ta klasa jest potomkiem klasy QObject, która jest klasą bazową wszystkich klas w pakiecie Qt.
Tworzenie okienka wymaga dołączenia pliku nagłówkowego z definicją klasy QLabel:

```
#include <QLabel>
```

Obiekt klasy QLabel tworzy prostą kontrolkę wyświetlającą wyspecyfikowany tekst ***st***:

```
#define st "Programujemy w C++"
QLabel *label = new QLabel(QObject :: tr(st));
```

Wykorzystana w konstruktorze klasy QLabel statyczna metoda klasy QObject:

```
QString QObject :: tr(const char* tekst)
```

zwraca przetłumaczoną wersję napisu ***tekst***.

Przy pomocy metody show() obiekt label jest wyświetlany na ekranie:

```
label -> show();
```

Program rozpoczyna działanie dzięki poleceniu:

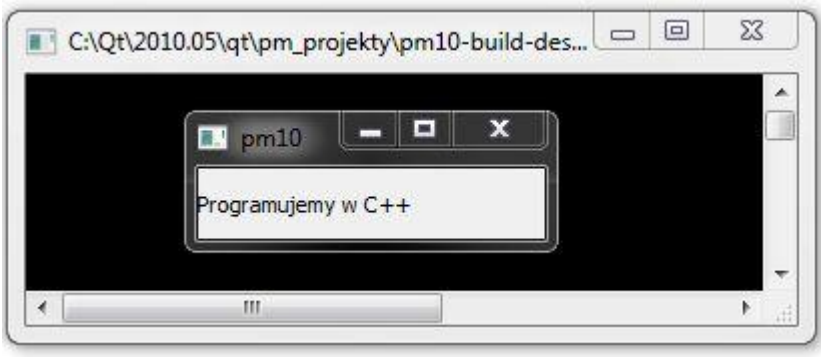
```
return a.exec();
```

Obiekt ***a*** przechodzi w stan nieskończonej pętli obsługi zdarzeń. Zdarzenia mogą być generowane zarówno przez system, jak i przez użytkownika aplikacji. Zakończenie programu uzyskujemy po zatrzymaniu pętli. Po opisanych modyfikacjach program przyjmuje postać:

Wydruk 1.10. Program konsolowy z widgetem, QLabel

```
// widget nr 1
#include <QApplication>
#include <QLabel>
#define st "Programujemy w C++"
int main(int argc, char *argv[])
{  QApplication a(argc, argv);
   QLabel *label = new QLabel(QObject :: tr(st));
   label -> show();
   return a.exec();
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:



Rys.13. Obiekt klasy QLabel - okienko z tekstem

Okienko przy pomocy myszki można powiększać, całe okno może być przemieszczane po ekranie, trzy przyciski z prawej strony (ukrycie, powiększenie, zakończenie) działają. Pokazany przykład jest prosty, ale pokazuje jak potężnym narzędziem jest pakiet Qt.

Widzimy, że przy pomocy niewielkiej liczby poleceń tworzymy eleganckie okno z tekstem i przyciskami, obsługującymi wygenerowane okienko.
Biblioteka Qt jest bardzo rozbudowana, programista ma do dyspozycji wiele możliwości. Pokażemy kolejny program, który podobnie jak poprzedni wygeneruje okienko z tekstem, ale będą wykorzystane inne obiekty.
Przy pomocy Creatora tworzymy kolejny program:

Wydruk 1.11. Program konsolowy, QPushButton

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QPushButton okienko("Program z widgetem");
    okienko.resize(100,50);
    okienko.show();
    return a.exec();
}
```

Pamiętamy, aby zmodyfikować plik ***.pro*** generowany przez kreatora, tak, aby można było wykorzystać widgety w programie konsolowym. Plik nagłówkowy:

```
#include <QPushButton>
```

zawiera definicję klasy QPushButton. Ta klasa tworzy wykorzystywany w programie standardowy przycisk. W naszym przykładzie w okienku (obiekt ***okienko***) przycisku umieszczamy wyspecyfikowany przycisk:

```
QPushButton okienko("Program z widgetem");
```

Możemy ustalić wymiary okienka, przy pomocy metody ***resize()***:

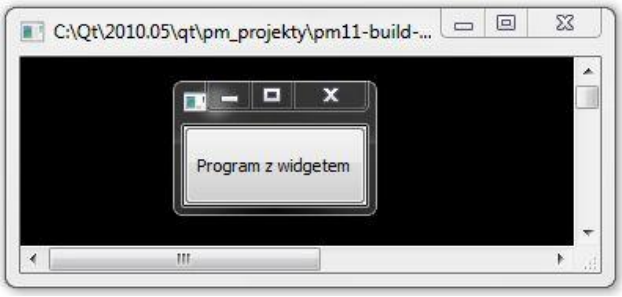
```
okienko.resize(100,50);
```

Metoda ***resize()*** pochodzi z klasy **QWidget**. Ta klasa jest klasą bazową wszystkich wizualnych elementów GUI w pakiecie Qt. Metoda ***show():***

```
okienko.show();
```

jest metodą klasy QWidget, jej zadaniem jest wyświetlanie bieżących kontrolek i wszystkich elementów pochodnych. Należy pamiętać, że każdy obiekt utworzony z klasy QWidget lub z jej klas pochodnych, dziedziczy pełną obsługę zdarzeń generowanych przez użytkownika, oznacza to możliwość obsługi naszej aplikacji przez polecenia generowane przez klawiaturę i mysz.

Po uruchomieniu programu otrzymujemy następujący rezultat:



Rys.14. Obiekt klasy QPushButton - okienko z tekstem

Jak widać na zademonstrowanym przykładzie, rezultat jest identyczny jak w poprzednim programie. Bardziej rozbudowany program prezentujemy na kolejnym listingu:

Wydruk 1.12. Program konsolowy, widgety

```
//w pliku .pro mamy:    # QT       -= gui
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QSlider>
#include <QSpinBox>
#include <QHBoxLayout>
int main(int argc, char *argv[])
{QApplication app (argc, argv);
 QWidget *mainWidget = new QWidget();
 mainWidget->setWindowTitle(QObject::tr("aplikacja
                                          konsolowa"));
 QPushButton *exitButton = new
            QPushButton(QObject::tr("KONIEC"),mainWidget);
 QSlider *slider = new QSlider(Qt::Horizontal, mainWidget);
 int mi = 0;
 int ma = 30;
 slider->setRange(mi,ma);
 QSpinBox *spin = new QSpinBox(mainWidget);
 QHBoxLayout *layout = new QHBoxLayout (mainWidget);
 layout ->addWidget(exitButton);
 layout ->addWidget(slider);
 layout ->addWidget(spin);
 QObject :: connect(exitButton,SIGNAL(clicked()),&app,
                                          SLOT(quit()));
 QObject :: connect(slider,SIGNAL(valueChanged(int)),spin,
                                    SLOT(setValue(int)));
 mainWidget->show();

 return app.exec();
}
```

Po uruchomieniu programu pokazuje się okno naszej aplikacji na ekranie monitora (Rys.15).

Rys.15. Wynik działania programu z kilkoma widgetami

Główne okno naszej aplikacji tworzone jest przy pomocy polecenia:

```
QWidget *mainWidget = new QWidget();
```

Obiekt ***mainWidget*** jest głównym oknem aplikacji i jednocześnie jest kontenerem, w którym umieszczone są inne kontrolki. W naszym programie są to trzy kontrolki: przycisk (QPushButton), suwak (QSlider) oraz kontrolka o nazwie QSpinBox. Te trzy kontrolki umieszczone w oknie głównym wymagają dołączenia trzech plików nagłówkowych:

```
#include <QPushButton>
#include <QSlider>
#include <QSpinBox>
```

Okno główne wymaga dołączenia pliku nagłówkowego <QWidget>. Tytuł okna głównego umieszczany jest przy pomocy metody ***setWindowTitle()***:

```
mainWidget->setWindowTitle(QObject::tr("aplikacja konsolowa"));
```

Kontrolki QSlider oraz QSpinBox umożliwiają wprowadzanie wartości całkowitoliczbowych.
Suwak obsługiwany jest myszą, przesuwając znacznik zmieniamy wartości w zakresie (mi, ma), w naszym przykładzie jest to zakres (0,30):

```
QSlider *slider = new QSlider(Qt::Horizontal,
                                    mainWidget);
int mi = 0;
int ma = 30;
slider->setRange(mi,ma);
```

Kontrolka QSpinBox umożliwia wprowadzenie wartości całkowitych przy pomocy klawiatury lub strzałek:

```
QSpinBox *spin = new QSpinBox(mainWidget);
```

Sposobem rozmieszczenia kontrolek zarządza obiekt typu „layout". Jest to obiekt klasy QHBoxLayout:

```
QHBoxLayout *layout = new QHBoxLayout (mainWidget);
layout ->addWidget(exitButton);
layout ->addWidget(slider);
layout ->addWidget(spin);
```

Trzy kontrolki umieszczone są w linii poziomej, w kolejności ich umieszczenia w programie, u nas ta kolejność to: ***exitButton, slider, spin***.
Qt dostarcza narzędzia potrzebne do obsługi danych obrazów graficznych. Mamy do dyspozycji cztery klasy:

- QImage
- QPixmap
- QBitmap
- QPicture

Biblioteka Qt ma możliwość obsługiwania kilku formatów, SA to popularne formaty o dużym znaczeniu praktycznym.

Tabela 1.2. Qt – formaty plików graficznych

Nazwa	Symbol	read	write
Bitmap Windows	BMP	tak	tak
Graphic Interchange Format	GIF	tak	nie
Joint Photographic Experts Group	JPG	Tak	tak
Joint Photographic Experts Group	JPEG	Tak	tak
Portable Network Graphics	PNG	Tak	tak
Portable Bitmap	PBM	Tak	nie
Portable Graymap	PGM	Tak	nie
Portable Pixmap	PPM	Tak	tak
Tagged Image File Format	TIFF	Tak	tak
X11 Bitmap	XBM	Tak	tak
X11 Pixmap	XPM	Tak	tak

Klasa QImage jest zoptymalizowana pod katem obsługi operacji wejścia/wyjścia (odczytywanie i zapisywanie różnego formatu plików graficznych) oraz do operacji wykonywanych na pikselach obrazów.

Klasa QPixmap w zasadzie służy do obsługi wyświetlania obrazów na ekranie. Klasa QBitmap jest klasą pochodną po klasie QPixmap i jest przeznaczona wyłącznie do obsługi obrazów monochromatycznych (1-bitowych).
Klasa QPicture odbiera i realizuje polecenia klasy QPainter (klasy QPainter, QPaintEngine oraz QPaintDevice są podstawa systemu graficznego w bibliotece Qt). W klasie QPainter są zdefiniowane metody, dzięki którym rysowane są prymitywy graficzne. Qt obsługuje wybrane formaty plików graficznych, nie wszystkie formaty można czytać i zapisywać. Tabela prezentuje akceptowane formaty graficzne w Qt.
Prostą obsługę plików graficznych zilustrujemy kolejnym przykładem konsolowym. Dysponujemy kultowym obrazem Lena zapisanym na dysku w formacie JPG. Zadaniem programu jest wczytanie pliku graficznego i wyświetlenie go w oknie. Program pokazany jest na wydruku.

Wydruk 1.12. Wyświetlany obraz JPG, klasa QPixmap

```
#include <QtCore>
#include <QApplication>
#include <QLabel>
#include <QPixmap>
#define obraz "c:\\Users\\mikfiz\\lenablack.jpg"

int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  QLabel img("");
  img.setPixmap(QPixmap(obraz));
  img.show();

  return a.exec();
}
```

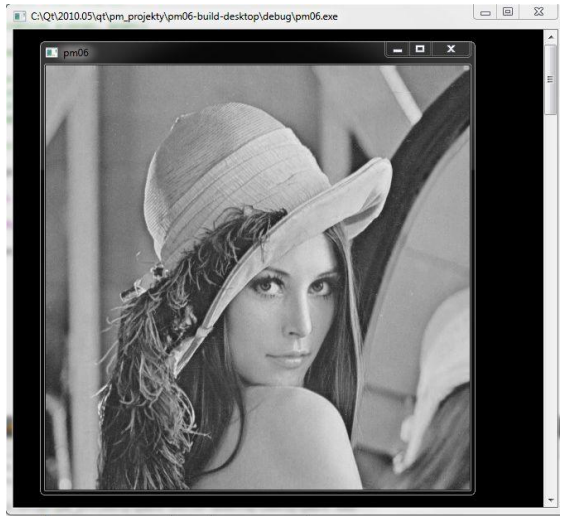
Odczyt i wyświetlanie obrazów realizowane jest w Qt bardzo prosto. Należy utworzyć obiekt klasy QPixmap dla obrazka a następnie narysować go przy pomocy wybranej metody klasy. Obiekt klasy QPixmap może być prosto wyświetlony wykorzystując obiekt QLabel. Metoda show() wyświetla obraz odczytany z dysku. Utworzenie obiektu klasy QPixmap i QLabel wymaga dołączenia plików nagłówkowych:

```
#include <QLabel>
#include <QPixmap>
```

W naszym przykładzie obraz zapisany jest w pliku:

```
"c:\\Users\\mikfiz\\lenablack.jpg"
```

Na rysunku pokazano wynik działania naszego programu. Ten przykład pokazuje jasno, że programowanie w Qt jest bardzo efektywne, kilka linijek kodu powoduje, że obrazy w wybranych formatach mogą być wyświetlane na ekranie monitora bez kłopotów.



Rys.16. Wynik działania programu, obraz JPG, klasa QPixmap

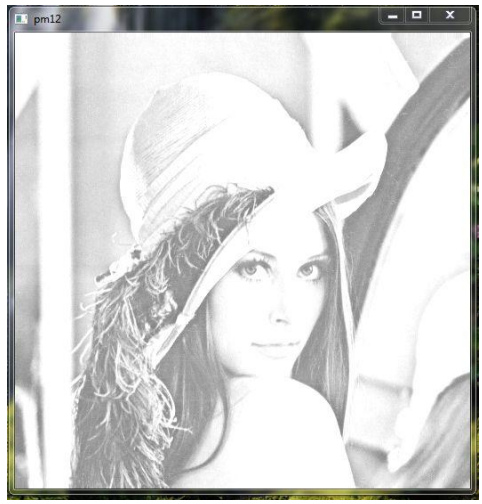
Stosunkowo prosto możemy wykonywać operacje przetwarzania obrazów cyfrowych w kolejnym programie pokażemy operację rozjaśniania obrazu – jest to lokalna operacja zwiekszania jasności każdego piksela obrazu. Program pokazany jest na kolejnym wydruku.

Wydruk 1.13. Przetwarzanie obrazu JPG,

```
#include <QtCore>
#include <QApplication>
#include <QLabel>
#include <QPixmap>

#define obraz "c:\\Users\\mikfiz\\lenablack.jpg"

int main(int argc, char *argv[])
{
  QApplication a(argc, argv);
  QImage img(obraz); //czyta obraz z pliku
  short bri=100;   //wartosc, o jaka zmieniamy jasnosc
  //przegladamy kolejne linie obrazu
  for(int y=0; y<img.height(); y++) {
      //zapisuje wskaznik do aktualnej linii obrazu
        QRgb *line=(QRgb*)img.scanLine(y);
```

```
        //w ramach linii przegladamy kolejne piksele
    for(int x=0; x<img.width(); x++) {
      //zapisuje wartosc koloru w aktualnym pikselu
      QRgb color=line[x];
      color=qRgb(qRed(color)+bri>255?255:qRed(color)+bri,
              qGreen(color)+bri>255?255:qGreen(color)+bri,
              qBlue(color)+bri>255?255:qBlue(color)+bri);
      //nowy kolor jest ustawiany w aktualnym pikselu
      line[x]=color;
    }
  }
  //tworzy obiekt etykiety i ustawia mu pixmape
   QLabel label("");
   label.setPixmap(QPixmap::fromImage(img));
   //wyswietlam etykiete
   label.show();

   return a.exec();
}
```

Po uruchomieniu program otrzymujemy przekształcony obraz.



Rys.17. Wynik działania programu, rozjaśniony obraz JPG, klasa QPixmap

Widzimy, że obraz został zdecydowanie rozjaśniony. Zgodnie z przyjętą konwencją wartość 0 jasności piksela oznacza minimalną jasność (czerń) a wartość 255 – oznacza maksymalną jasność (biel). Ponieważ do wartości jasności każdego piksela oryginalnego obrazu dodajemy wartość 100, to nic dziwnego, że obraz został rozjaśniony.

Obraz, o nazwie ***lenablack.jpg***, który znajduje się w katalogu c:

```
#define obraz "c:\\Users\\mikfiz\\lenablack.jpg"
```

jest wczytany do programu poleceniem:

```
QImage img(obraz); //czyta obraz z pliku
```

Ustalamy wartość, o jaką ma się zmienić jasność każdego piksela w obrazie:

```
short bri=100;   //wartosc, o jaka zmieniamy jasność
```

W naszym przypadku do aktualnej wartości jasności dodajemy wartość 100. Obraz jest dwuwymiarowy o rozmiarach img.height() x img.width(), dlatego stosujemy dwie pętle zagnieżdżone, aby zmodyfokować każdy piksel obrazu:

```
for(int y=0; y<img.height(); y++) {
   ……..
   for(int x=0; x<img.width() ; x++) {
   ……..
   }
}
```

Obraz zapisany jest zgodnie z modelem RGB, mamy trzy kanały (Red, Green Blue), musimy obsłużyć te trzy kanały. Ponieważ jasnośc piksela nie może przekroczyć wartości 255 (kodowanie na ośmiu bitach) w momencie sprawdzenia, że nowa warość jasności po operacji dodawania jest większa niż 255, zostaje ona zmienione do wartości 255:

```
color=qRgb(qRed(color)+ bri >255?255:qRed(color)  +bri,
           qGreen(color)+bri>255?255:qGreen(color)+bri,
           qBlue(color)+ bri>255?255:qBlue(color) +bri);
```

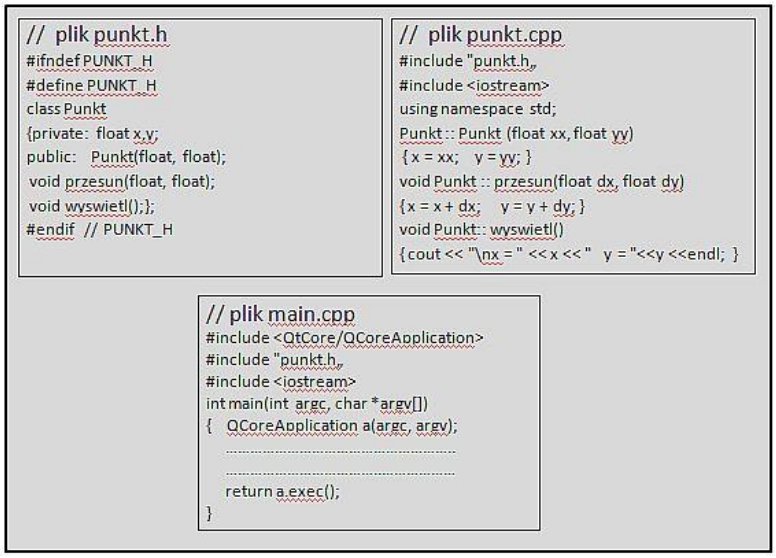
Wyswietlanie obrazu obsługują polecenia:

```
QLabel label("");
label.setPixmap(QPixmap::fromImage(img));
label.show();
```

1.7. Programy wieloplikowe

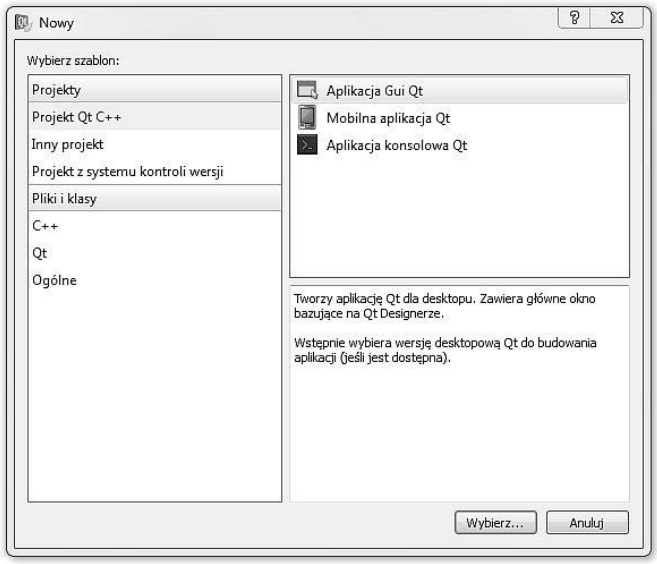
Podczas pisania dużych programów komputerowych zaleca się, aby rozdzielać interfejs od implementacji. Praktycznie oznacza to pisanie kodu w wielu plikach. Deklarację klasy zaleca się umieszczać w pliku nagłówkowym. Zwyczajowo ten plik ma rozszerzenie .h. Definicje funkcji składowych klasy powinny być umieszczane w odrębnym pliku źródłowym, zwyczajowo plik taki ma rozszerzenie .cpp, a nazwą taką samą jak plik z deklaracją klasy. Do programu głównego (w Qt jest to plik o nazwie main.cpp) włączamy wszystkie potrzebne, utworzone przez nas pliki nagłówkowe. Zintegrowane środowiska programistyczne mają specjalne narzędzia do obsługi programów wieloplikowych. W Qt tworzenie programów wieloplikowych jest stosunkowo proste, aczkolwiek musimy pamiętać, jakie opcje z rozwijanego menu w Kreatorze należy wybrać. Procedurę tworzenia programu wieloplikowego opiszemy krok po kroku. Przykładowy program tworzy klasę ***Punkt*** obsługującą punkty na płaszczyźnie. Utworzymy trzy pliki:

- punkt.h - deklaracja klasy Punkt
- punkt.cpp - definicje funkcji składowych klasy Punkt
- main.cpp - definicja funkcji main()



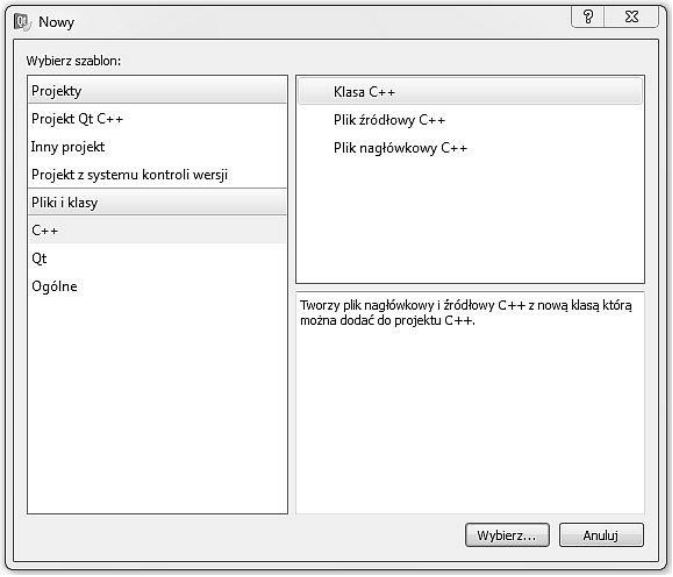
Rys. 18. Fizyczne diagramy komponentów i programu testującego (main.cpp)

Po typowym uruchomieniu Kreatora i podaniu nazwy projektu otrzymamy ekran edytora (Rys. 6). Z menu górnego: ***Plik, Edycja, Budowanie,……*** wybieramy opcję ***Plik***. Rozwija się podmenu, z niego wybieramy opcję: ***Nowy plik lub projekt…,*** otrzymamy nowe okno (rys.19).



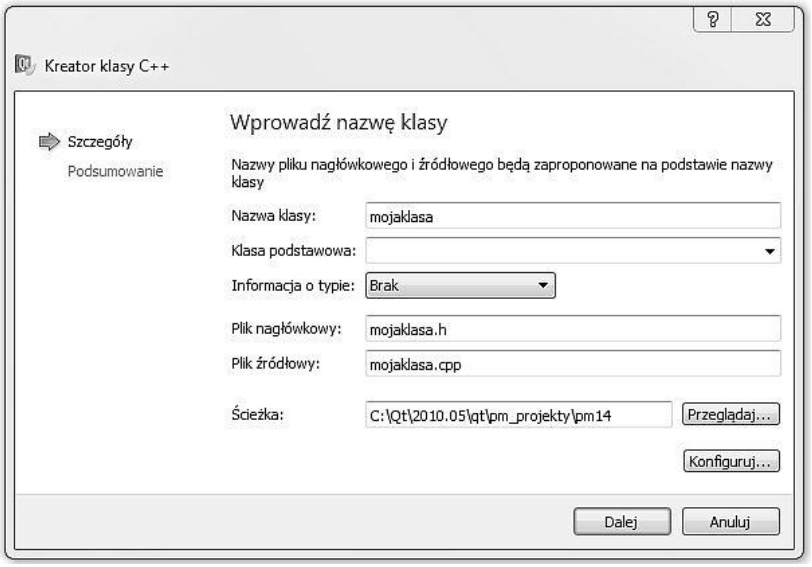
Rys. 19. Kreator Qt, wybór szablonu do projektu wieloplikowego

W grupie ***Pliki i klasy***, wybieramy opcję ***C++*** , pojawia się nowe okno.



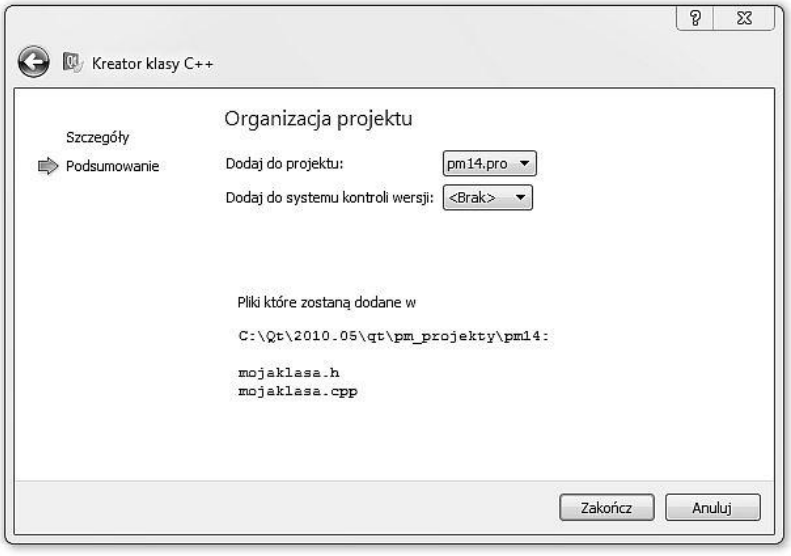
Rys. 20. Kreator Qt, wybór klasy

Wybieramy opcję ***Klasa C++*** i zatwierdzamy opcją Wybierz. Pojawia się kolejne okno.



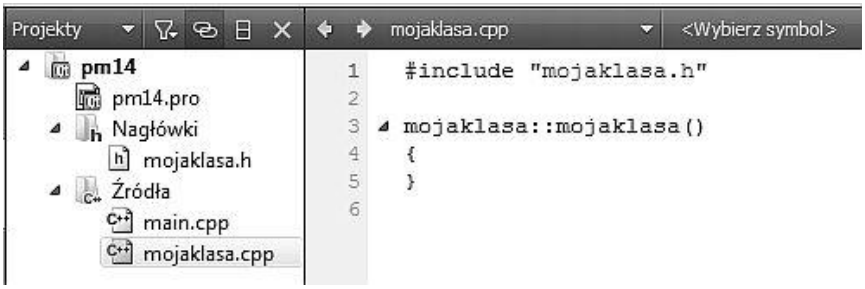
Rys.21. Kreator Qt, zapisanie nazwy klasy

Po wybraniu opcji ***Dalej*** pojawia się następne okno.



Rys.22. Kreator Qt, organizacja projektu

Po wybraniu opcji ***Zakończ*** mamy w końcu ekran edytora, z lewej strony ekranu widzimy wygenerowane pliki naszego projektu wieloplikowego.



Rys. 23. Kreator Qt, edytor i schemat organizacji projektu.

Widzimy, że utworzone zostały wszystkie potrzebne pliki, możemy teraz przystąpić do kodowania kolejnych plików. Dostęp do wybranych plików realizujemy klikając myszą na nazwie pliku w edytorze. Na rysunku 23 widać, że plik nagłówkowy ma nazwę ***mojaklasa.h***, jest to przykład. W praktyce w każdym projekcie będziemy nazwy plików nagłówkowych zmieniać. Nasz poglądowy program posiada klasę ***Punkt***, stąd nazwa pliku nagłówkowego to ***punkt.h***. Teraz musimy stworzyć potrzebne pliki, pokazane są one na kolejnych wydrukach.

Wydruk 1.14a. Plik nagłówkowy: punkt.h

```
#ifndef PUNKT_H
#define PUNKT_H
class Punkt
{private:
  float x,y;
 public:
    Punkt(float, float);
    void przesun(float, float);
    void wyswietl();
};
#endif // PUNKT_H
```

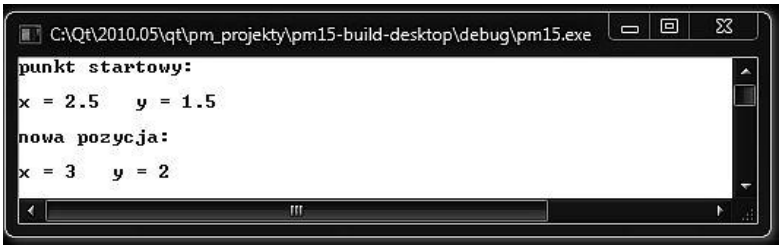
Wydruk 1.14b. Plik źródłowy: punkt.cpp

```
#include "punkt.h"
#include <iostream>
using namespace std;
Punkt :: Punkt (float xx, float yy)
 { x = xx;     y = yy;
 }
void Punkt :: przesun(float dx, float dy)
 { x = x + dx;      y = y + dy;
 }
void Punkt:: wyswietl()
 { cout << "\nx = " << x << "    y = "<<y <<endl;
 }
```

Wydruk 1.14c. Plik źródłowy: main.cpp

```
#include <QtCore/QCoreApplication>
#include "punkt.h"
#include <iostream>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
     Punkt p1 (2.5, 1.5);
     std::cout <<"punkt startowy: \n"
     p1.wyswietl();
     p1.przesun(0.5,0.5);
     std::cout <<"\nnowa pozycja: \n";
     p1.wyswietl();

    return a.exec();
}
```

Po uruchomieniu programu otrzymujemy wynik:



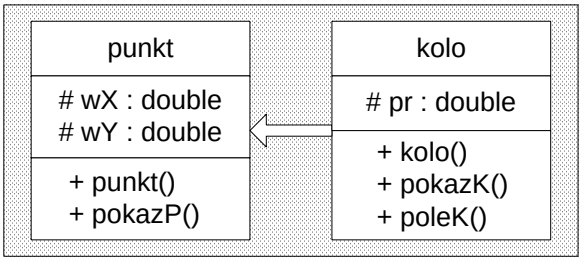
Rys. 23. Wynik działania programu wieloplikowego

Na wydruku 14.a mamy następujące dyrektywy preprocesora (wygenerowane przez Kreator):

```
#ifndef PUNKT_H
#define PUNKT_H
.....
.....
#endif // PUNKT_H
```

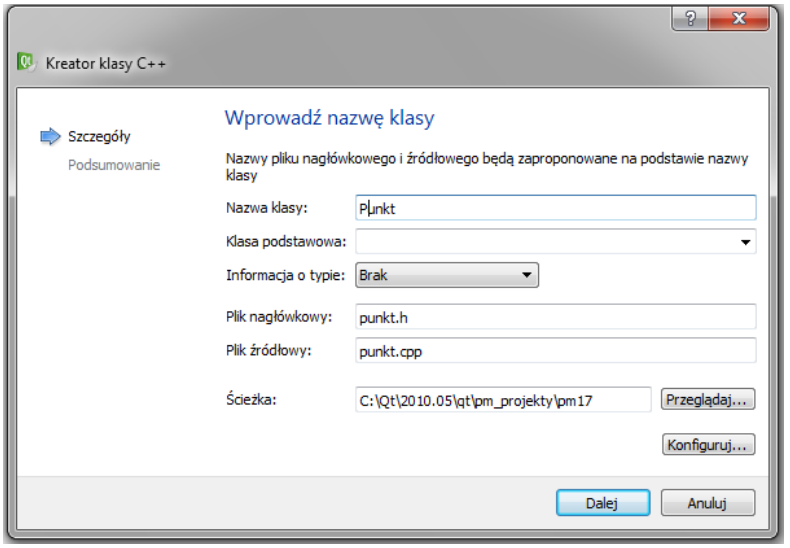
Pokazane dyrektywy, które są generowane przez Qt automatycznie, zapobiegają wielokrotnemu włączaniu pliku ***punkt.h*** (a także każdego innego) do naszego projektu, zaleca się pozostawienie ich w tym pliku.
Ważną cechą programowania obiektowego jest możliwość wykorzystania dziedziczenia. Zazwyczaj klasę bazową i klasę pochodną zapisujemy w oddzielnych plikach. W takiej sytuacji powstaje projekt wielokilowy. W

kolejnym przykładzie pokażemy sposób tworzenia projektu złożonego z pięciu plików. Utworzymy program, w którym korzystając z klasy Punkt (klasa bazowa) będziemy mogli obsługiwać obiekt typu Kolo. Na początku tworzymy klasę Punkt a potem na podstawie tej klasy tworzymy klasę Kolo. Realizujemy model dziedziczenia prostego. Hierarchia klas pokazana jest na rysunku 24.



Rys. 24. Diagram klasy ***Punkt*** i klasy ***Kolo*** w notacji UML.
Klasa ***Kolo*** dziedziczy składowe klasy ***Punkt.***

Procedurę tworzenia projektu opiszemy krok po kroku. Po typowym uruchomieniu Kreatora i podaniu nazwy projektu otrzymamy ekran edytora (Rys. 6). Z menu górnego: ***Plik, Edycja, Budowanie,……*** wybieramy opcję ***Plik***. Rozwija się podmenu, z niego wybieramy opcję: ***Nowy plik lub projekt…,*** otrzymamy nowe okno (rys.25).



Rys. 25. Kreator Qt, zapisanie nazwy klasy

Po wpisaniu nazwy klasy (w naszym przykładzie Punkt), Kreator Qt wygeneruje dwa szablony plików: punkt.h i punkt.cpp.

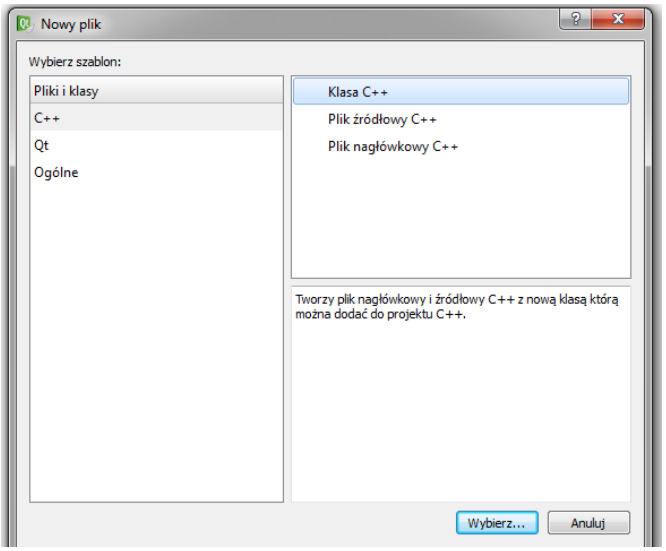
Fragment obszaru edytora Qt ma postać:

Rys. 26. Schemat projektu pm17

W naszym nowym projekcie tworzymy pięć plików:

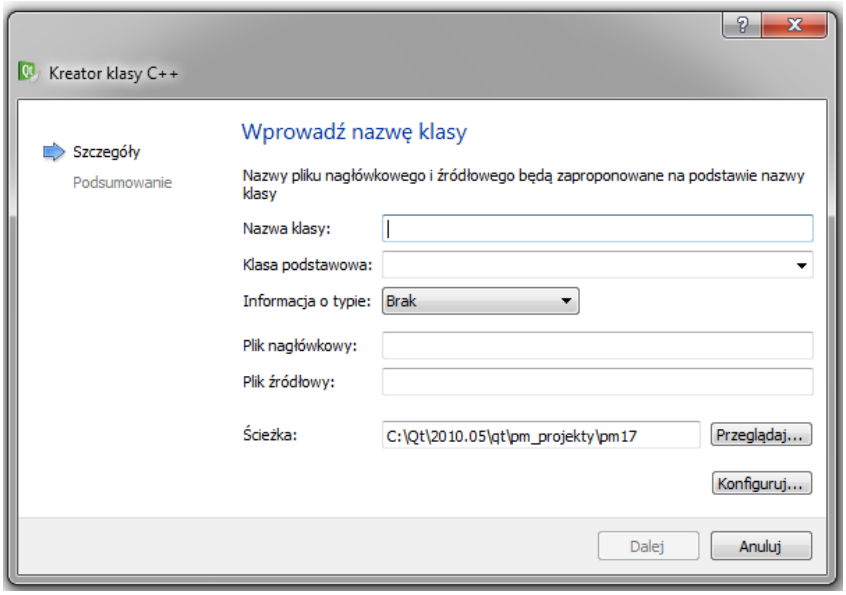
- punkt.h - deklaracja klasy Punkt
- punkt.cpp - definicje funkcji składowych klasy Punkt
- kolo.h - deklaracja klasy Kolo
- kolo.cpp - definicje funkcji składowych klasy Kolo
- main.cpp - definicja funkcji main()

W celu dopisania nowych plików do projektu musimy wykorzystać Kreator Qt. Należy prawym przyciskiem myszy (kursor myszy jest umieszczony na nazwie projektu, w naszym przypadku na nazwie pm17) otworzyć nowy dialog, z menu wybieramy opcję **Dodaj nowy…,** co spowoduje pojawienie się okna:



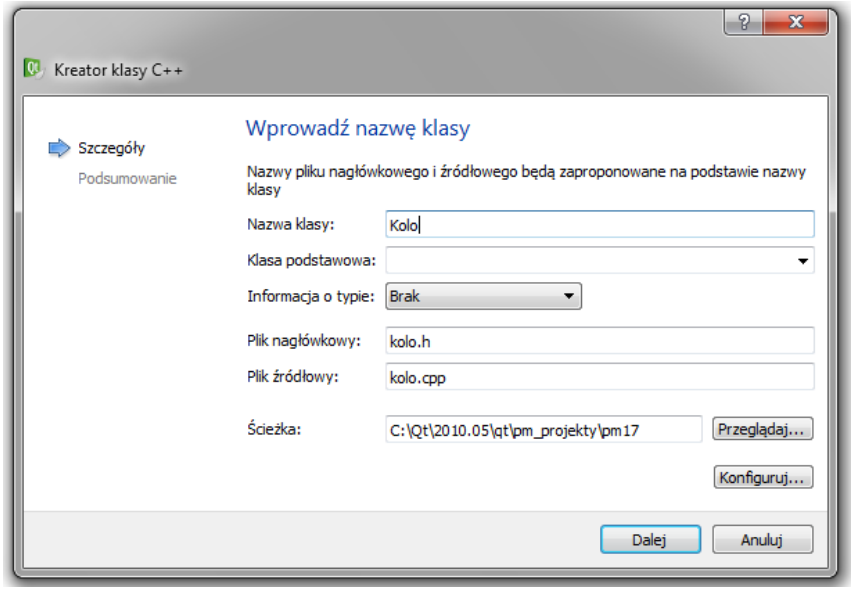
Rys.27. Wybór typu pliku

Po wybraniu opcji **Klasa C++** i opcji **Wybierz…** pojawi się kolejne okno:



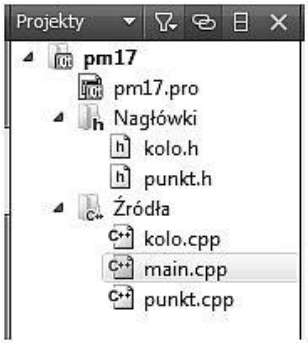
Rys. 28.Szablon do wpisywania nazwy klasy

Wpisujemy nazwę klasy Kolo, ekran ma postać:



Rys. 29. Wprowadzona nazwa klasy i wygenerowane pliki

Przy pomocy Kreatora Qt generowane są nowe pliki: pliki kolo.h i kolo.cpp
Struktura naszego projektu ostatecznie ma postać:



Rys. 30. Schemat projektu pm17.

W wygenerowanym szablonie należy teraz uzupełnić treść poszczególnych plików. Utworzone pliki pokazane są w kolejnych wydrukach.

Wydruk 1.15a. Plik nagłówkowy: punkt.h

```
#ifndef PUNKT_H
#define PUNKT_H
class Punkt
 { public:
    Punkt(double = 0.0, double = 0.0);
    void pokazP();
   protected:
    double wX, wY;
 };
#endif // PUNKT_H
```

Wydruk 1.15b. Plik nagłówkowy: kolo.h

```
#ifndef KOLO_H
#define KOLO_H
#include "Punkt.h"
class Kolo : public Punkt
{ public:
    Kolo(double r=0.0, double wX=0.0, double wY=0.0);
    void pokazK();
    double poleK() const;
  protected:
   double pr;
};
#endif // KOLO_H
```

Wydruk 1.15c. Plik źródłowy: punkt.cpp

```
#include "punkt.h"
#include <iostream>
Punkt :: Punkt (double xx, double yy)
 { wX = xx;    wY = yy;
 }
void Punkt :: pokazP()
{ std::cout << "\nx = " << wX << "   y = "<<wY;
}
```

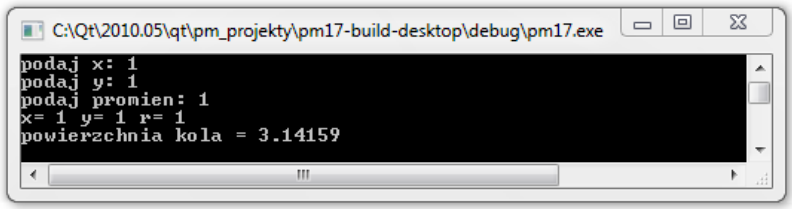
Wydruk 1.15d. Plik źródłowy: kolo.cpp

```
#include "kolo.h"
#include <iostream>
Kolo::Kolo(double r, double wX, double wY):
Punkt(wX,wY)
 { pr=r;
 }
void Kolo ::pokazK()
 {std::cout<<"x= "<<wX<<" y= "<<wY<<" r= "<<pr;
 }
double Kolo::poleK() const
 {return 3.1415926*pr*pr;
 }
```

Wydruk 1.15e. Plik źródłowy: main.cpp

```
#include <QtCore/QCoreApplication>
#include "Punkt.h"
#include "Kolo.h"
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{   QCoreApplication a(argc, argv);
    double wX, wY, pr;
    cout <<"podaj x: ";
    cin >> wX;
    cout <<"podaj y: ";
    cin >> wY;
    cout <<"podaj promien: ";
    cin >> pr;
    Kolo k1(pr,wX,wY);
    k1.pokazK();
    std::cout<<"\npowierzchnia kola = "<<k1.poleK();
    return a.exec();
}
```

Po uruchomieniu program otrzymujemy następujący wynik:



Rys. 31. Wynik wykonania programu pm17

ROZDZIAŁ 2

STL – STANDARD JĘZYKA C++

2.1. Wstęp

Wielu programistów uważa, że jednym z najważniejszych rozszerzeń dodanych do języka C++ jest biblioteka standardowych wzorców – STL (w polskiej literaturze przedmiotu spotkamy także określenie *standardowa biblioteka szablonów*, określenie angielskie: Standard Template Library). Biblioteka STL została opracowana przez Alexandra Stepanowa i Menga Lee, pracowników amerykańskiej firmy Hewlett Packard.

Biblioteka STL zawiera szablony klas oraz funkcje implementujące. W zestawie szablonów miedzy innymi mamy kontenery, iteratory, obiekty funkcyjne oraz algorytmy. Elegancka biblioteka STL umożliwiła realizację kolejnego paradygmatu programowania – programowania ogólnego (ang. *generic programming*). W dużym uproszczeniu możemy powiedzieć, że programowanie ogólne polega na wykorzystywaniu wzorców (szablonów).

2.2. Programowanie ogólne (generyczne)

Realizując paradygmat programowania obiektowego (język C++), łączymy w klasie dane i metody (funkcje). Stosując proste techniki, metody są parametryzowane wartościami (nawet w przypadku stosowania przeciążenia funkcji). Dopiero wykorzystanie szablonów (język C++) pozwala na parametryzowanie wartościami oraz typami. Dzięki temu możemy tworzyć kod klasy, która będzie w stanie obsłużyć żądany typ.

Należy zauważyć, że stosowanie technik zawartych w bibliotece STL nie jest najlepszym przykładem programowania obiektowego, realizować w zamian możemy tak zwane programowanie ogólne (*programowanie generyczne).* Podstawą programowania ogólnego jest maksymalne wykorzystywanie szablonów. W przeciwieństwie do kodów realizujących programowanie obiektowe (konstrukcja klasy), koncepcja biblioteki STL oparta jest na rozdzieleniu danych i metod (operacji). Dane przechowywane są w kontenerach a operacje na danych wykonywane są przy pomocy uogólnionych algorytmów. Operując na danych algorytmy wykorzystują iteratory. Dzięki rozdzieleniu danych i algorytmów mamy możliwość operowania dowolnego algorytmu na dowolnym kontenerze, niezależni od typu danych. Jest to istota programowania ogólnego. Kontenery z danymi oraz działające na tych danych algorytmy są ogólne, obsługują dowolne typy oraz klasy.

Zgodnie B. Stroustropem przyjmiemy następującą definicję:

> *Programowanie ogólne*: pisanie kodu, który może działać z różnymi typami przekazywanymi do niego, jako argumenty, pod warunkiem, że typy te spełniają określone wymagania syntaktyczne i semantyczne.

Istnieje wiele istotnych różnic pomiędzy programowaniem obiektowym i ogólnym. Bardzo ważny jest fakt, że wyboru funkcji do wywołania kompilator

dokonuje w czasie kompilacji, natomiast w programowaniu obiektowym wywołanie to odbywa się w czasie wykonywania programu. *Programowanie ogólne* realizowane jest wykorzystując szablony, wywoływanie metod odbywa się w czasie kompilacji, *programowanie obiektowe* wykorzystuje hierarchię klas oraz funkcje i klasy wirtualne, wywoływanie metod odbywa się w czasie wykonywania programu.

2.3. Elementy biblioteki STL

Zgodnie z opisem podanym przez Alexandra Stepanov'a i Menga Lee (A.Stepanov, M.Lee, The Standard Template Library, Hewlett-Packard Company, 1994) biblioteka STL dostarcza zbiór dobrze skonstruowanych generycznych składników (ang. *well structured generic C++ components*), które działają razem bezproblemowo (ang. *in a seamless way*). Biblioteka STL dostarcza narzędzi do zarządzania kolekcjami danych przy użyciu wydajnych algorytmów. Biblioteka STL zawiera szablony klas ogólnego przeznaczenia (czyli daje możliwość obsługi dowolnych typów danych) oraz funkcje implementujące. Dzięki bibliotece STL mamy możliwość realizacji kolejnego paradygmatu programowania – możemy realizować ***programowanie generyczne*** (ang. *generic programming*). W odróżnieniu od programowania obiektowego, które koncentruje się na danych i konstruowaniu typów użytkownika, w programowaniu generycznym główny nacisk kładzie się na algorytmy.

Nicolai Josuttis w swoim doskonałym podręczniku o STL (The C++ Standard Library: A Tutorial and Reference, 1999) za najważniejsze składniki STL uważa kontenery, iteratory i algorytmy.

- Kontenery służą do obsługi kolekcji obiektów, istnieje wiele rodzajów kontenerów. Najpopularniejszym kontenerem jest klasa *vecto*r, ta klasa zawiera definicję tablicy dynamicznej. Klasa *vector*, podobnie jak klasa *list* czy klasa *deque* są nazywane kontenerami sekwencyjnymi (ang. *sequence containers*). Sekwencyjny kontener jest takim typem kontenera, który organizuje skończony zbiór obiektów tego samego typu, powiązanych w ściśle liniowym porządku. Istnieją też kontenery stowarzyszone, inna nazwa kontenery asocjacyjne (ang. *associative containers*). Przykładem takiej klasy jest klas *set* czy *map*. Tego typu kontenery umożliwiają efektywne operowanie wartościami na podstawie kluczy.
- Iteratory są uogólnieniem wskaźników. Dzięki iteratorom (ang. *iterators*) programista może pracować z różnymi strukturami danych (kontenerami) w jednorodny i uporządkowany sposób. Dzięki nim można poruszać się po zawartości kontenera w podobny sposób jak dzięki wskaźnikom przemieszczamy się w obrębie tablicy. Iteratory zachowują się bardzo podobnie do

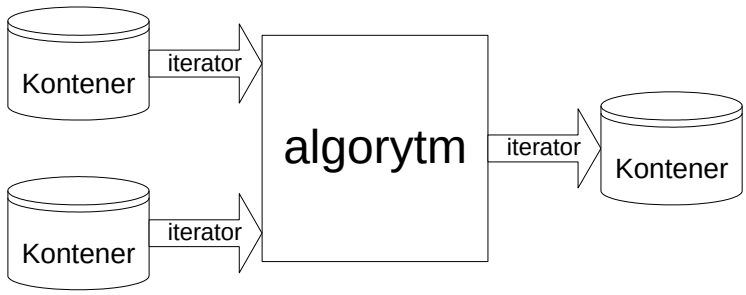
wskaźników. Możliwa jest ich inkrementacja i dekrementacja. Iteratory deklaruje się za pomocą typu **iterator** zdefiniowanego w różnych kontenerach. Istnieje pięć różnych typów iteratorów.

- Algorytmy operują na kontenerach. Dzięki iteratorom mogą operować na elementach kontenerów niezależnie od struktury danych. Algorytmy używają szablonów, dzięki czemu można stosować generyczne typy danych. Dzięki iteratorom określamy zakres przetwarzanych danych oraz określamy miejsce do którego należ wysłać wyniki. Algorytmy pozwalają w sposób stosunkowo prosty na przykład sortować dane bez względu na typ, algorytm *sort()* sortuje zakres bez względu czy mamy zbiór liczb całkowitych czy zbiór napisów. Aby uzyskać dostęp do algorytmów STL należy włączyć do programu plik nagłówkowy <algorithm>.

Biblioteka algorytmów w STL dzieli się na cztery grupy:

- Operacje sekwencyjne nie zmieniające wartości
- Operacje sekwencyjne zmieniające wartości
- Operacje sortowania i pokrewne
- Uogólnione operacje numeryczne

Relacje pomiędzy głównymi składnikami STL pokazane są na rysunku 2.1.



Rys. 2.1. Relacje miedzy głównymi składnikami STL (M. Josuttis)

Algorytmy przeznaczone do przetwarzania danych numerycznych najczęściej wymagają pliku nagłówkowego <numeric>.

Z kolei A.Stepanov i M. Lee wyróżniają pięć głównych składników w STL:

- Algorytmy (ang. *algorithm*)

- Kontenery (ang. *container*)
- Iteratory (ang. *iterator*)
- Obiekty funkcyjne (ang. *function object*)
- Adaptatory (ang. *adaptor*)

Obiekty funkcyjne są to klasy, dla których zdefiniowano operator (). W przypadku definiowania obiektu funkcyjnego korzysta się z szablonów zdefiniowanych w pliku <functional>. W wielu przypadkach obiekty funkcyjne są wykorzystywane, jako wskaźniki do funkcji. Jednym z najczęściej używanych obiektów funkcyjnych jest *less*, dzięki któremu możemy określić czy dany obiekt jest mniejszy od innego.

Adaptatory są szablonami klas, dzięki którym można mapować interfejsy. Mówiąc bardziej prosto, dzięki adaptatorom wykorzystujemy inne klasy do tworzenia klas z nowymi właściwościami. Klasycznym przykładem jest tworzenie stosu wykorzystując klasy *vector*, *list* lub *deque*.

2.4. Przykłady użycia elementów STL

Biblioteka STL jest bardzo duża, składnia w niej stosowana, przynajmniej na początku wydaje się bardzo skomplikowana, ale praktyczne stosowanie elementów tej biblioteki nie jest zbyt trudne. Biblioteka STL udostępnia zestaw wzorców reprezentujących kontenery, iteratory, obiekty funkcyjne oraz algorytmy. Jako przykład pokażemy użycie jednego z najbardziej ogólnych kontenerów, jakim jest **vector**. Kontener, podobnie jak tablica pozwala na przechowywanie grupy wartości, i podobnie jak tablica może obsługiwać tylko wartości tego samego typu. Klasa (kontener) **vektor** obsługuje tablice dynamiczne (należy zauważyć, że nazwa kontenera jest trochę myląca, klasa **vektor** ma niewiele wspólnego z matematycznym pojęciem wektora). Obiekt szablonu **vector** przechowuje zestaw wartości tego samego typu o dostępie swobodnym, wobec tego możemy operować poszczególnymi wartościami przy pomocy indeksu. Po utworzeniu obiektu **vector** przy pomocy przeciążonego operatora [ ] oraz indeksu uzyskujemy dostęp do poszczególnych elementów, tak samo jak w przypadku tablic.

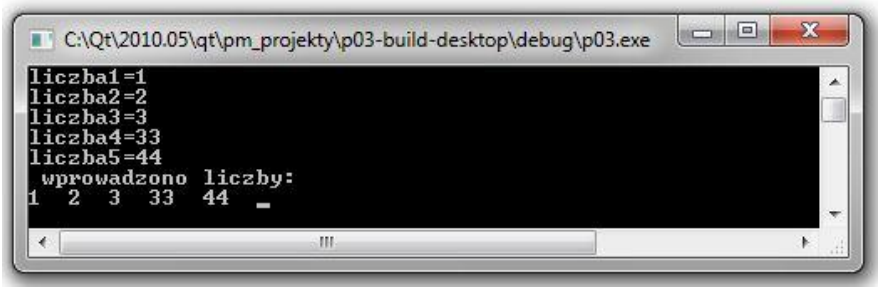
Kolejny program pokazuje sposób tworzenia obiektu szablonu **vector**. W programie tworzymy wektor (obiekt szablonu **vector**) liczb całkowitych, wprowadzimy 5 liczb całkowitych przy pomocy klawiatury, a następnie program wyświetli wprowadzone liczby. W celu utworzenia obiektu klasy **vector** musimy dołączyć plik **vector** oraz zadeklarować kolekcję (obiekt szablonu **vector**), co w praktyce sprowadza się do naśladowania posługiwania się zwykłymi szablonami klas.

Wydruk 2.1. Przykład użycia wzorca **vector**, typ **int**

```
#include <QtCore/QCoreApplication>
#include <iostream>
```

```
#include <vector>

using namespace std;

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    vector<int> liczby(5);
    for(int i =0; i<5; i++)
    {cout << "liczba" << i+1 << "=";
        cin >> liczby[i];
    }
    cout << " wprowadzono liczby: "<<endl;
    for(int i=0; i<5; i++)
        cout << liczby[i]<<"  ";
    return a.exec();
}
```

Po uruchomieniu programu mamy następujący wydruk:



Plik nagłówkowy potrzebny do obsługi wektorów dołączamy poleceniem :

```
#include <vector>
```

Deklarując kolekcję (wektor, obiekt szablonu vector) musimy określić typ elementów:

```
vector < int > liczby( ILE );
```

Został utworzony konkretny rodzaj kontenera, w tym przypadku jest to wektor o nazwie *liczby*, którego elementy są typu **int**, kontener może początkowo posiadać 5 elementów (zmienna ILE). Wektor *liczby* jest inicjalizowany liczbami całkowitymi wprowadzonymi z klawiatury. Zastosowano klasyczną składnię operacji na tablicach, wykorzystując indeksy.
W następującym fragmencie programu trudno zorientować się, że obsługujemy obiekty klasy **vector** a nie zwykłe tablice:

```
for (int i=0; i < ILE; i++)
```

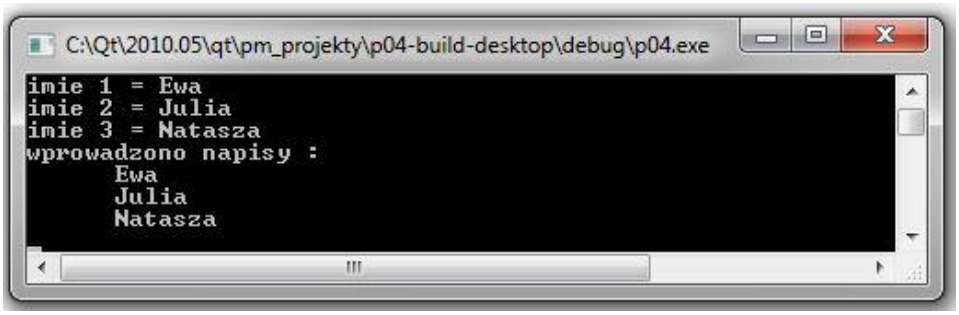
```
        { cout << "liczba " << i+1 << " = " ;
          cin >> liczby[i];
        }
```

Główną zaletą biblioteki STL jest fakt, że nie musimy zajmować się szczegółami implementacji struktur danych i zarządzaniem pamięcią podczas przetwarzania kolekcji.
W kolejnym przykładzie utworzymy obiekt **vector** dla typu **string**.

Wydruk 2.2. Użycie wzorca **vector**, typ **string**

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <vector>
#include <string>
using namespace std;
const int ILE = 3 ;
int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  vector<string> imiona(ILE);  //wektor typu string
    for (int i=0; i < ILE; i++)
         { cout << "imie " << i+1 << " = " ;
           getline(cin, imiona[i]);
         }
  cout << "wprowadzono napisy : " << endl;
    for (int i=0; i < ILE; i++)
           cout << "      " << imiona[i] << endl ;
  return a.exec();
}
```

Wydruk z programu ma następującą postać:



W programie, aby utworzyć kolekcję imion (typ string) musimy włączyć dwa pliki:

```
#include <vector>
#include <string>
```

Definicja wektora napisów ma postać:

```
vector<string> imiona(ILE);
```

Pętla do wprowadzania napisów (w naszym przypadku imion) ma postać:

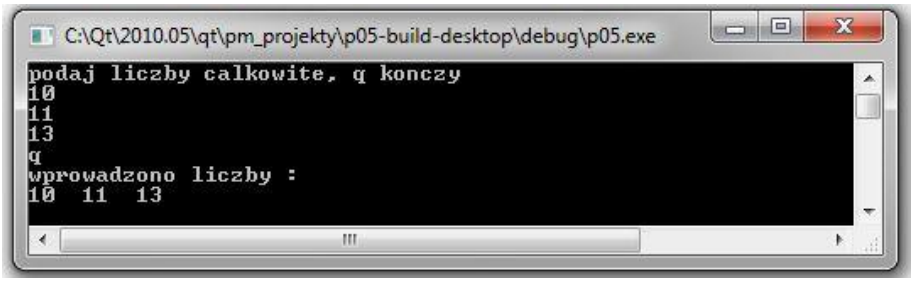
```
for (int i=0; i < ILE; i++)
{ cout << "imie " << i+1 << " = " ;
  getline(cin, imiona[i]);
}
```

Wszystkie kontenery biblioteki STL udostępniają potrzebne metody. Do najważniejszych metod możemy zaliczyć metodę **size(),** która umożliwia określenie liczby elementów umieszczonych w kontenerze oraz dwie metody **begin()** oraz **end().** Metoda **begin()** zwraca iterator (iterator jest obiektem bardzo przypominającym w działaniu wskaźnik, umożliwia poruszanie się po zawartości kontenera, tak jak wskaźnik pozwala przemieszczać się po zawartości tablicy). Szablon klasy **vector** ma dodatkowo metodę **push_back(),** która dodaje elementy na koniec wektora. Kolejny program ilustruje wykorzystanie metody **push_back().**

Wydruk 2.3. Użycie wzorca **vector**, metody kontenerów

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  vector<int> v;      // utworzenie pustego
                      // wektora(kontenera),typ: int
  int we;
  cout << "podaj liczby calkowite, q konczy " << endl;
  while (cin >> we)  //wprowadzanie liczb z klawiatury
     v.push_back (we); // umieszczanie w kontenerze
  int n = v.size();
  cout << "wprowadzono liczby : \n" ;
  for (int i = 0; i < n; i++)
     cout << v[i] << "  " ;
  return a.exec();
}
```

Po uruchomieniu programu, mamy wydruk:



W pokazanym programie wykorzystano metodę **push_back(),** która dodaje element na koniec wektora:

```
vector<int> v;      // utworzenie pustego wektora
int we;
cout << "podaj liczby calkowite, q konczy " << endl;
while (cin >> we)   // wprowadzanie liczb z klawiatury
v.push_back (we);   // umieszczanie w kontenerze
```

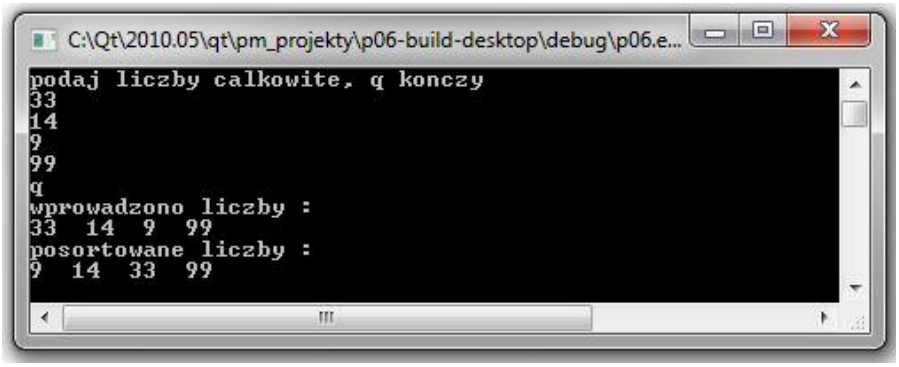
Każde wykonanie pętli wymusza dodanie nowego elementu do wektora **v**. Zauważmy, że nie określono na początku liczby elementów, jakie ma przechować wektor **v**. Wektor zwiększa swój rozmiar automatycznie, tzn. pamięć jest przydzielana automatycznie. Dopóki program będzie posiadał dostęp do odpowiedniej ilości pamięci, będzie powiększał swój rozmiar. W programie wykorzystano także metodę **size()** do wyliczenia ilości elementów umieszczonych w kontenerze **v**. Prawdziwe korzyści wynikające ze stosowania szablonów STL możemy zilustrować programem sortującym tablice liczb całkowitych.

Wydruk 2.4. Użycie wzorca **vector**, sortowanie

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <vector>
#include <algorithm>   // dla sort()

using namespace std;

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  vector<int> v;    // utworzenie pustego
                    // wektora(kontenera), typ: int
  int we;
  cout << "podaj liczby calkowite, q konczy " << endl;
```

```
  while (cin >> we) // wprowadzanie liczb z klawiatury
       v.push_back (we); // umieszczanie w kontenerze
  int n = v.size();
  cout << "wprowadzono liczby : \n" ;
  for (int i = 0; i < n; i++)
          cout << v[i] << "  " ;
  sort(v.begin(), v.end());
  cout << "\nposortowane liczby : \n" ;
  for (int i = 0; i < n; i++)
          cout << v[i] << "  " ;

  return a.exec();
}
```

Po uruchomieniu programu mamy następujący wydruk:



Jak wiadomo w standardowej bibliotece języka C oraz C++ mamy funkcje **qsort(),** która może być wykorzystana do sortowania elementów. Zamiast funkcji **qsort()** wykorzystamy procedurę **sort()** z biblioteki STL. Jest ona przykładem jednego z wielu algorytmów, jakie dostarcza nam biblioteka STL. Procedura **sort()** jest generyczna, tzn. może sortować wiele różnych typów danych przechowywanych w kontenerach. Algorytm **sort()** wykorzystuje iteratory. Operacje sortowania na elementach kontenera wykonywane są bardzo często przy pomocy iteratorów. Przypominamy, że iteratory są obiektami, które umożliwiają przeglądanie zawartości kontenerów. Każdy iterator reprezentuje pozycję w kontenerze. Kontenery posiadają odpowiednie metody umożliwiające obsługę iteratorów. Najważniejsze metody to **begin()** oraz **end().**
Należy zwrócić uwagę, że **begin()** zwraca iterator reprezentujący pozycje pierwszego elementu w kontenerze, a **end()** zwraca iterator reprezentujący pozycję za ostatnim elementem w kontenerze. Sortowanie elementów kontenera wykonane jest dzięki instrukcji:

```
sort( v.begin(), v.end() );
```

Algorytm **sort()** wymaga dołączenia pliku nagłówkowego:

```
#include <algorithm>   //dla sort()
```

Algorytm **sort()** przyjmuje, jako argumenty dwa iteratory, które definiują sortowany zakres.
Użycie klasy ciągu znaków (ciągu tekstowego) zamiast tablicy znakowej bądź wskaźników było doskonałym pomysłem twórców biblioteki STL. Klasa ciągu tekstowego w bibliotece STL:

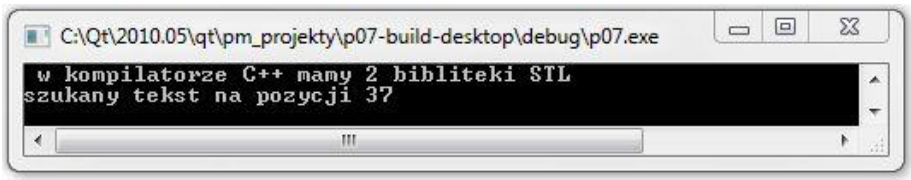
```
std:: string
```

pozwala na proste manipulowanie napisami i jest bardzo odporna na błędy. W celu skorzystania z klasy **string** należy do programu włączyć plik nagłówkowy <string>. Klasa **string** ma wiele metod, kilka konstruktorów oraz duży zestaw przeciążonych operatorów. Porosty przykład pokazujący wykorzystanie klasy **string** do manipulowania napisami pokazuje poniższy wydruk programu.

Wydruk 2.5. Przykład użycia klasy string

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <string>
using namespace std;
int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  string n1 ( " w kompilatorze C++");
  string n2 ( " mamy 2 bibliteki STL");
  string n12;
  n12 = n1 + n2;
  cout << n12 << endl;
  size_t  poz = n12.find("STL",0);
  if (poz != string:: npos)
       cout <<"szukany tekst na pozycji"<<poz;
  else
       cout <<" nie ma takiego tekstu"<<endl;
  return a.exec();
}
```

Wynikiem wykonania programu jest komunikat:



Prosta inicjalizacja i przypisanie ciągu tekstowego do zwykłego obiektu **n1** ma postać:

```
string n1 ("w kompilatorze C++");
```

Łączenie dwóch napisów jest proste:

```
n12 = n1 + n2;
```

Klasa **string** posiada wiele metod. Obiekt STL **string** zawiera między innymi metodę **find(),** dzięki której możemy wyszukiwać znak lub sekwencję znaków w napisie.

ROZDZIAŁ 3

KONTENERY C++ STL

3.1. Wstęp

Klasy kontenerowe (ang. *container classes*), nazywane kontenerami lub zasobnikami służą do zarządzania zbiorami elementów. Wyróżniamy dwa typy kontenerów:

- Kontenery sekwencyjne
- Kontenery asocjacyjne

Kontenery sekwencyjne są kolekcjami uporządkowanymi, pozycja każdego elementu jest określona w momencie wstawiania i jest niezależna od wartości elementu. Kontenery asocjacyjne są kolekcjami sortowanymi, pozycja każdego elementu zależy od jego wartości zgodnie z kryterium sortowania.
Krótkie opisy kontenerów oraz pliki nagłówkowe, które muszą być włączane do programu pokazane są w tabeli I. Stosując kontenery bitset należy pamiętać, że ta klasa nie zachowuje się jak typowa klasa biblioteki STL, należy, więc podejść z dużą starannością do tworzenia obiektów klasy bitset i technikami operowania tymi obiektami.

Wszystkie kontenery posiadają określone właściwości i udostępniają określone operacje. W STL nie jest zdefiniowana ściśle konkretna implementacja żadnego kontenera. Standard określa jedynie zachowanie i złożoność. W praktyce poszczególne implementacje różnią się jedynie drobnymi szczegółami.

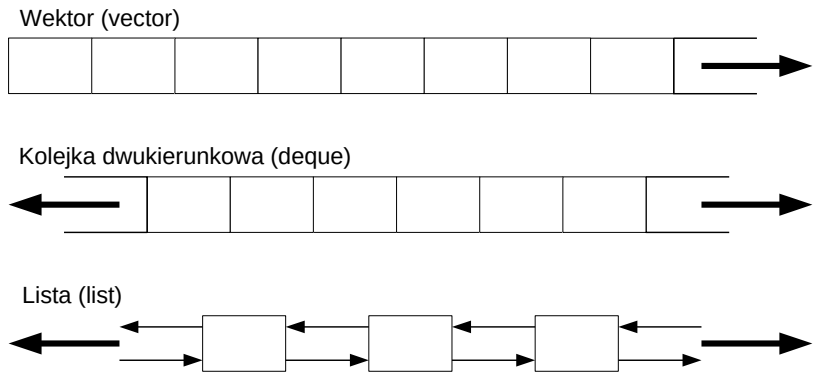
Tabela I. Opisy kontenerów i pliki nagłówkowe

Lp	Kontener	Opis	Plik
1	vector	Tablica dynamiczna	<vector>
2	deque	Kolejka dwukierunkowa	<deque>
3	list	Lista liniowa	<list>
4	set	Zbiór elementów unikatowych	<set>
5	multiset	Zbiór, elementy nie muszą być unikatowe	<set>
6	map	Przechowuje pary klucz/wartość, klucz powiązany z jedna wartością	<map>
7	multimap	Przechowuje pary klucz/wartość, klucz powiązany z wieloma wartościami	<map>
8	bitset	Zbiór bitów	<bitset>
9	stack	Stos	<stack>
10	queue	Kolejka	<queue>
11	priority_queue	Kolejka z priorytetami	<queue>

Biblioteka STL jest rozbudowana i dość skomplikowana, ale wykorzystanie jej składników nie jest zbyt trudne. Przede wszystkim należy określić, jaki typ kontenera będzie najlepszy dla rozwiązania postawionego zadania. Każdy kontener ma swoje zalety. Jeżeli tworzymy obiekty typu tablicowego i chcemy mieć dostęp swobodny do nich oraz nie przewidujemy zbyt wielu operacji wstawiania i usuwania to najlepiej jest zastosować kontener *vector*. Jeżeli przewidujemy wykonywanie dużo operacji wstawiania i usuwania obiektów, wtedy najlepiej jest wykorzystać kontener *list*. Za pomocą metod możemy dodawać do kontenera żądane elementy, modyfikować je, usuwać i wykonywać inne potrzebne operacje. Należy pamiętać, że kontenery są automatycznie powiększane, gdy dodawany jest nowy element i zmniejszane, gdy element jest usuwany. Ta właściwość nie dotyczy kontenera *bitset*. Najczęściej, aby uzyskać dostęp do elementów w kontenerze stosujemy iteratory. Wszystkie kontenery maja metody begin() i end(), które zwracają iteratory wskazujące na początek i koniec kontenera. Mając ustalony początek kontenera przy pomocy inkrementacji, iteratora możemy przeglądać kolejne elementy. Jeżeli w kontenerze są przechowywane dane, to możemy nimi manipulować przy pomocy algorytmów w sensie STL.

3.2. Kontenery sekwencyjne C++

Kontenery sekwencyjne C++ są kontenerami, które obsługują skończony zbiór elementów, wszystkie muszą być tego samego typu, elementy musza być uporządkowane ściśle liniowo. W STL mamy trzy rodzaje takich kontenerów : vector, list oraz deque. Uporządkowanie liniowe oznacza, że istnieje pierwszy element i ostatni. Każdy element pomiędzy pierwszy i ostatnim ma ściśle jeden element poprzedzający i jeden element za nim. Kontenery są użyteczne, gdy można w nich umieszczać obiekty i je usuwać. Opracowując składniki STL zwrócono szczególną uwagę na wydajność metod i algorytmów.



Rys. 3.1. Typy kontenerów sekwencyjnych STL

Wektor (vector) jest jednym z najczęściej używanych kontenerów sekwencyjnych. W zasadzie zastępuje tak często wykorzystywane w C++ tablice. Wektor zarządza swoimi elementami dynamicznie. Pozwala wykorzystać przeciążony operator indeksu [ ] aby mieć bezpośredni dostęp do swojego dowolnego elementu. Podobnie jak w tablicach języka C nie jest sprawdzany automatycznie zasięg (nie sygnalizuje automatycznie sięgnięcie po indeks spoza tablicy), za to posiada metodę at() która pozwala na taką kontrolę. Każdy obiekt przechowywany w kontenerze vector musi mieć zdefiniowane operator < oraz operator ==.

Szablon klasy **deque** (wymawiaj jako „diik" albo jako „dek") znajduje się w pliku <deque> i reprezentuje kolejkę o dwóch końcach. Klas deque jest bardzo pożytecznym zasobnikiem gdyż łączy w sobie cechy kontenerów vector i list. Termin „deque" jest skrótem „kolejki dwukierunkowej" (ang. double–ended queue). W swoim działaniu deque jest bardzo podobnym kontenerem do wektora. Zasadnicza różnica polega na tym, że wstawianie i usuwanie elementów z początku kontenera deque jest operacją o stałej złożoności, podczas gdy dla wektora są to operacje o złożoności liniowej. Tak jak przypadku wektora dostęp do wybranych elementów w deque jest szybki (stały czas). Wstawianie i usuwanie elementów w środku kontenera deque jest podobne do wykonywania tych operacji w kontenerze wektor (czas liniowy) a nawet trochę wolniejsze, ponieważ kontener deque ma bardziej złożoną strukturę niż wektor. Kontener deque powinno się używać zamiast kontenera wektor, gdy planowane jest częste wstawianie i usuwanie elementów na początku i końcu kontenera oraz potrzebny jest odczyt wszystkich elementów.

Klasa deque stosuje obsługę iteratorów o dostępie bezpośrednim, wszystkie algorytmy STL mogą być stosowane do działania na obiektach deque. Najczęściej kontener deque jest stosowany przy realizacji kolejki elementów FIFO („pierwszy wszedł, pierwszy wyszedł', ang. first-in-first-out).

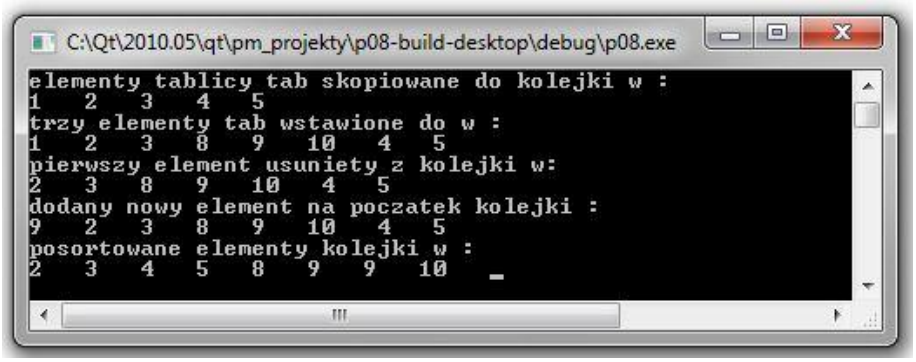
Dla kontenerów deque mamy dostępne metody begin(), end(), rbegin() oraz rend(), które zwracają iteratory o takim samym znaczeniu jak w innych kolekcjach sekwencyjnych. Dostępne są także metody związane z rozmiarem: size(), max size(), resize(), empty() o takim samym znaczeniu jak w innych kolekcjach sekwencyjnych.

W kolejnym przykładzie ilustrujemy obsługę kolejki deque i omawiamy podstawowe operacje. Kontener deque wymaga dołączenia pliku <deque>. W programie utworzona jest tablica tab[ ] zainicjalizowana wartościami typu int. Elementy tablicy tab[ ] są umieszczone w kontenerze deque. Funkcje insert() i pop_front() służą do umieszczania elementów w kolejce oraz do usuwania elementów z kolejki.

Wydruk 3.1. Kontener deque, insert() i sort()

```
#include <QtCore/QCoreApplication>
#include <deque>
#include <iterator>                //ostream_iterator
#include <algorithm>               //copy(), sort()
#include <iostream>

using namespace std;
const int MAX = 10;

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  int tab[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
  deque<int> w(tab, tab+5);
  cout<<"elementy tablicy tab skopiowane do kolejki w :" << endl;
  ostream_iterator <int> output (cout, "   ");
  copy (w.begin(), w.end(), output);                  // 1 2 3 4 5
  cout << endl << "trzy elementy tab wstawione do w :" << endl;
  w.insert(w.begin()+3, tab+7, tab+10);
  copy (w.begin(), w.end(), output);       // 1 2 3 8 9 10 4 5
  cout<< endl << "pierwszy element usuniety z kolejki w:" << endl;
  w.pop_front();
  copy (w.begin(), w.end(), output);       // 2 3 8 9 10 4 5
  cout<<endl<<"dodany nowy element na poczatek kolejki :" << endl;
  w.push_front(9);
  copy (w.begin(), w.end(), output);       // 9 2 3 8 9 10 4 5
  cout << endl << "posortowane elementy kolejki w : " <<  endl;
  sort ( w.begin(), w.end() );                        // sortowanie
  copy (w.begin(), w.end(), output);

  return a.exec();
}
```

Wydruk wyników działania programu ma następująca postać:



W programie utworzona została klasyczna tablica tab[ ] oraz utworzony została kolejka deque o nazwie **w** do której skopiowano elementy tablicy:

```
int tab[MAX] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
deque < int > w ( tab, tab+5);
```

W celu wydrukowanie elementów kolejki w zastosowano instrukcję:

```
ostream_iterator <int> output (cout, "   ");
```

która tworzy ostream_iterator o nazwie output, który może być użyty do wysłania na wyjście liczb całkowitych oddzielonych spacjami przez cout. Stosowanie iteratora wyjściowego (wymagany jest plik nagłówkowy <iterator>) jest bezpiecznym mechanizmem. Pierwszy argument dla konstruktora określa strumień wyjściowy, a drugi jest napisem określającym znaki rozdzielające dla wartości wyjściowych.
Kolejna instrukcja :

```
copy (w.begin(), w.end(), output);
```

wykorzystuje algorytm copy() (wymagany jest plik nagłówkowy <algorithm>) do wyprowadzenia całej zawartości obiektu w klasy deque do standardowego wyjścia. Algorytm copy() kopiuje wartości z zakresu wskazanego przez [w.begin(), w.end()). Elementy kopiowane są do miejsca określonego przez iterator wyjściowy (trzeci argument). W pokazanym przykładzie iteratorem wyjściowym jest ostream_iterator output, który jest dowiązany do cout, więc elementy są skopiowane do standardowego wyjścia.
W instrukcji:

```
w.insert(w.begin()+3, tab+7, tab+10);
```

wykorzystana została metoda insert().

Funkcja insert() w ogólnej postaci:

```
w.insert(pos, beg, end)
```

wstawia na pozycji iteratora pos kopie wszystkich elementów z zakresu [beg, end). W naszym przypadku wstawiane są elementy tablicy tab[ ] do kolejki w na pozycji czwartej. Wstawiane są trzy elementy tablicy: ósmy, dziewiąty i dziesiąty. Elementy kolejki: czwarty i piąty są przesuwane na koniec kolejki.
Kolejne metoda:

```
w.pop_front();
```

usuwa pierwszy element kolejki.

Metoda:

```
w.push_front(9);
```

wstawia liczbę 9 na początek kolejki. Na koniec algorytm sort() sortuje elementy kolejki w:

```
sort ( w.begin(), w.end() );
```

Zasobnik sekwencyjny **list** jest standardową listą powiązaną, wymaga pliku nagłówkowego <list>. Podobnie jak kontenery vector czy deque służy do zapisywania ciągu elementów. Istotna cecha kontenera list jest to, że nie musi zajmować w pamięci ciągłego obszaru. Podobnie jak w klasycznej liście, każdy element kontenera list wskazuje gdzie są elementy następny i poprzedni (wykorzystujemy do tego celu wskaźniki, element pierwszy i ostatni są traktowane inaczej). Taka lista nosi nazwę listy podwójnie powiązanej. Ten fakt umożliwia klasie list obsługę dwukierunkowych iteratorów, dzięki którym można przeglądać kontener do przodu jak i w odwróconym porządku. Wiele algorytmów STL może operować na obiekcie typu list. W porównaniu do kontenera vector czy deque, dostęp do elementów jest wolny (liniowy). Zaletą kontenera list jest szybkość wstawiania i usuwania elementów w dowolnym miejscu (czas stały).

Listy stosujemy w przypadku, gdy często wstawiamy i usuwamy elementy a rzadko przeglądamy listę. Taki przypadek zachodzi, gdy obsługujemy biuro informacyjne dla klientów. Klienci często kontaktują się z operatorem a potem się odłączają, rzadko operator musi przeglądać listę rozmówców. W przeciwieństwie do kontenera vector, kontener ***list*** nie obsługuje indeksowania ani nie umożliwia dostępu swobodnego. Iteratory wektora działają inaczej niż iteratory listy. W wektorze po wstawieniu nowego elementu, iterator dalej wskazuje to samo miejsce, co oznacza, że gdy wstawimy nowy element na początku kontenera, iterator pokaże inną wartość (wszystkie elementy są przesuwane o jedno miejsce, aby zrobić miejsce dla nowego elementu). Gdy wstawiany jest nowy element do listy, ta operacja nie powoduje przesunięcia pozostałych elementów, może tylko zmienić informacje wiążące elementy. Iterator wskazujący na dany element dalej wskazuje na ten sam element. Do dodawania i usuwania elementów kontenera ***list*** posiada następujące metody: push_back(), pop_back(), trzy formy insert(), dwie formy erase() oraz clear(). W kontenerze list obsługiwane są następujące metody związane z rozmiarem: size() oraz empty(). Kontener ***list*** nie obsługuje metody resize() oraz capacity().

Kolejny program ilustruje działanie kontenera ***list*** przechowującego elementy typu string. Tworzymy listę kompozytorów i sprawdzamy czy wprowadzone z klawiatury nazwisko znajduje się na liście, a jeżeli jest na liście, to sprawdzamy, na jakim miejscu. Wykorzystano ogólny algorytm find().

W programie deklarujemy listę i wstawiamy elementy:

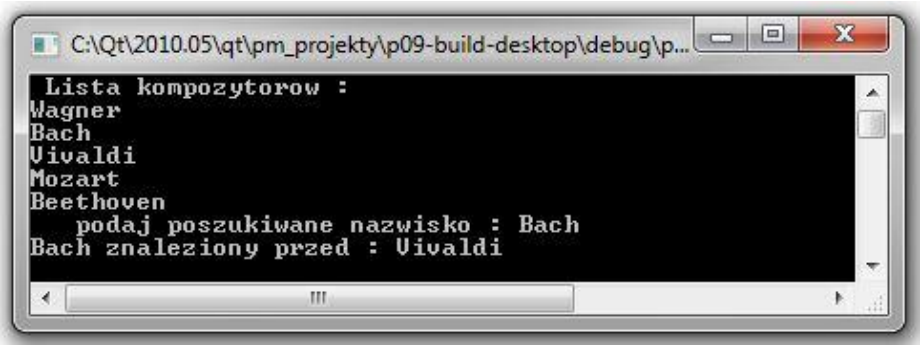
```
list<string> kompozytor;
kompozytor.push_back("Wagner");
kompozytor.push_back("Bach");
kompozytor.push_back("Vivaldi");
kompozytor.push_back("Mozart");
kompozytor.push_back("Beethoven");
```

Jak zwykle metoda push_back() pozwala na bezpieczne wstawienie wszystkich żądanych elementów na listę.

Wydruk 3.2. Kontener list, find()

```
#include <QtCore/QCoreApplication>
#include <list>
#include <algorithm>
#include <string>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  string n;
  list<string> kompozytor;
  kompozytor.push_back("Wagner");
  kompozytor.push_back("Bach");
  kompozytor.push_back("Vivaldi");
  kompozytor.push_back("Mozart");
  kompozytor.push_back("Beethoven");
  list<string>::iterator p;
  cout << " Lista kompozytorow :" << endl;
  for (p=kompozytor.begin(); p!=kompozytor.end(); ++p)
       cout << *p << endl;
  cout << "   podaj poszukiwane nazwisko : " ;
  cin >> n;
  p = find(kompozytor.begin(), kompozytor.end(), n);
  if (p == kompozytor.end())
      cout<< "nie znaleziono nazwiska : " << n << endl;
  else if (++p != kompozytor.end())
      cout<< n << " znaleziony przed : "<< *p << endl;
  else
      cout<< n <<" znaleziony na koncu." << endl;

 return a.exec();
}
```

Działanie programu jest następujące:



W instrukcjach:

```
list < string > :: iterator p ;
cout << " Lista kompozytorow :" << endl;
for (p = kompozytor.begin(); p! = kompozytor.end(); ++p)
   cout << *p << endl;
```

wykorzystano iterator do obsługi kontenera list. Pierwsza pokazana instrukcja definiuje iterator p. Przy pomocy iteratorów otrzymujemy najprostszy sposób uzyskiwania dostępu do elementów przechowywanych w kontenerach. Wszystkie kontenery maja funkcje składowe begin() i end(), które zwracają iteratory wskazujące na początek i koniec kontenera. Iteratory, tak samo jak wskaźniki można inkrementować, dekrementować, stosować operator dereferencji *. W pętli for wykorzystując dereferencje iteratora p drukujemy na ekranie monitora nazwiska wprowadzonych na listę kompozytorów.
W instrukcjach:

```
cout << "   podaj poszukiwane nazwisko : " ;
cin >> n;
p = find(kompozytor.begin(), kompozytor.end(), n);
if (p == kompozytor.end())
   cout<< "nie znaleziono nazwiska : " << n << endl;
else if (++p != kompozytor.end())
   cout<< n << " znaleziony przed : "<< *p << endl;
else
   cout<< n <<" znaleziony na koncu." << endl;
```

realizujemy zadanie wprowadzenia z klawiatury nazwiska poszukiwanego kompozytora a następnie wykorzystując algorytm STL find() sprawdzamy, czy wprowadzone nazwisko znajduje się na liście i na jakiej pozycji. Algorytm find(), jak każdy algorytm zdefiniowany w STL operuje na szablonach za pomocą iteratorów.

Wszystkie algorytmy są szablonami funkcji, find() zadeklarowany jest następująco:

```
template < class InIter, class T >
InIter  find ( InIter start, InIter end, const T &val )
```

Algorytm find() przeszukuje kontener w zakresie od ***start*** do ***end***, poszukuje wartości określonej przez ***val***. Zwraca iterator do pierwszego wystąpienia elementu lub do ***end,*** gdy w kontenerze nie występuje szukana wartość.

3.3. Kontenery asocjacyjne C++

Kontenery asocjacyjne (znane także, jako kontenery skojarzeniowe, kontenery stowarzyszone lub zasobniki skojarzeniowe) umożliwiają bezpośredni dostęp do elementów kontenerów przez klucze (zwanymi także kluczami wyszukiwawczymi). Elementy nie są przechowywane w konfiguracji liniowej. Kontenery asocjacyjne zapewniają odwzorowanie kluczy na wartości. W kontenerach asocjacyjnych czas operacji wstawiania, usuwania i wyszukiwania jest porównywalny. Mamy cztery typy kontenerów asocjacyjnych:

- set
- multiset
- map
- multimap

W praktyce bardzo często tworzymy struktury danych. W firmach przechowywane są rekordy pracowników: nazwisko, rok urodzenia, miejsce urodzenia, telefon domowy, itp. Do wyszukania w bazie pracowników odpowiedniego pracownika dobrze jest mieć ustalony klucz (może to być unikalny numer pracownika albo np. PESEL). Dzięki kluczowi szybko odnajdujemy interesującą nas strukturę i dzięki temu mamy szybki dostęp do konkretnych danych o pracowniku. Praktyczne znaczenie kontenerów asocjacyjnych polega na tym, że umożliwiają bardzo szybki (można by rzec - błyskawiczny) dostęp do elementów.

Kontener set (zbiór) najczęściej używany jest do zapamiętywania i odzyskiwania kluczy, przy czym w kontenerze set sama wartość jest kluczem. Kontener nie dopuszcza do przechowywania powtarzających się wartości, dane przechowywane są w postaci uporządkowanej. Klasa set obsługuje iteratory dwukierunkowe (ale nie iteratory o dostępie bezpośrednim, set nie posiada operatora [ ]. W zbiorach realizowane jest automatyczne sortowania. Wprowadza to istotne ograniczenia – nie można bezpośrednio zmieniać wartości elementów. Oczywiście mamy możliwości zmiany elementów zbioru, ale wymaga to dość skomplikowanych zabiegów.

Na zbiorach STL możemy wykonać wiele operacji znanych dla matematycznych zbiorów, takich jak suma, przecięcie czy różnica. Suma dwóch

zbiorów to zbiór zawierający elementy dwóch zbiorów z wyłączeniem powtórzeń. Przecięcie dwóch zbiorów to zbiór, który zawiera ich wspólne elementy. Różnica dwóch zbiorów to zbiór elementów pierwszego zbioru, które nie pojawiły się w drugim zbiorze. W bibliotece STL znajdują się algorytmy do obsługi wymienionych operacji na zbiorach takie jak na przykład set_union(), set_intersection() czy set_difference().W operowaniu kontenerami asocjacyjnymi bardzo przydatna jest klasa narzędziowa pair(). Zdefiniowana jest ona w pliku nagłówkowym <utility>. Typ pair używany jest wszędzie tam, gdzie zachodzi potrzeba traktowania dwóch wartości, jako pojedynczego elementu. Typ pair zadeklarowany został w STL, jako struktura, żeby wszystkie jego składowe były publiczne. Deklaracja:

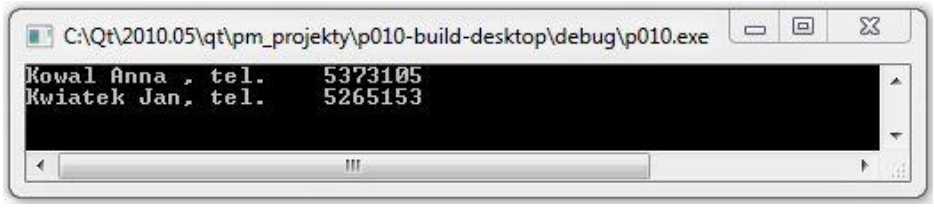
```
pair < int, double > p
```

powoduje inicjalizację wartości pary p za pomocą konstruktorów int() oraz double(), co w rezultacie daje wartości zerowe. Jeżeli istnieją obiekty klasy pair(), możemy uzyskać dostęp do jej składników poprzez pola ***first*** oraz ***second.*** Użycie obiektu pair() pokazane jest w kolejnym programie, parę o składowych typu ***int*** oraz ***string*** tworzą numer telefonu i nazwisko.

Wydruk 3.3. Obiekt klasy pair<>

```
#include <QtCore/QCoreApplication>
#include <utility>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  pair <int, string> a1(5373105,"Kowal Anna ");
  pair <int, string> a2(5265153,"Kwiatek Jan");
  cout << a1.second <<", tel.    " << a1.first<<endl;
  cout << a2.second <<", tel.    " << a2.first<<endl;
  return a.exec();
}
```

Po uruchomieniu tego programu mamy następujący wydruk:



Obiekt struktury ***pair*** może być bardzo skomplikowany, tworzy się go dla konkretnych typów, (np. dla kontenerów asocjacyjnych), czasami te obiekty są

niezbędne do wykonania poszczególnych operacji kontenerach. W szczególności utworzona para dla kontenera set w postaci:

```
pair <set<int> :: iterator, bool> stan;
```

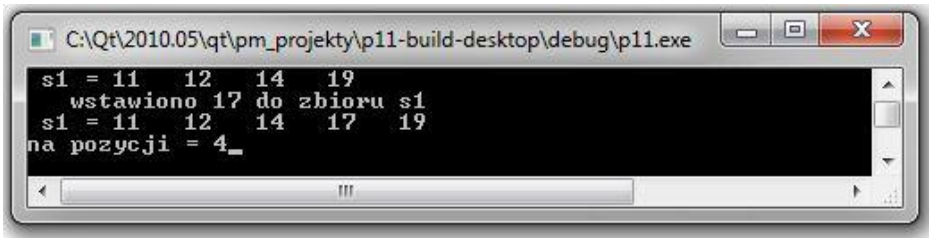
deklaruje parę składająca się z iteratora dla set<int> i wartości typu bool. Jeżeli np. obiekt stan będzie przechowywał wynik wywolania do set funkcji insert(), to w wyniku otrzymamy parę stan, która zawiera iterator stan.first wskazujący na wartość typu int w obiekcie set oraz wartość bool (stan.second), która jest prawdą (true), jeżeli wartość została wstawiona lub fałszem (false), gdy wartość nie została wstawiona (taka możliwość istnieje, jeżeli w zbiorze istnieje już wstawiana wartość).

Nie jest sprawą trywialną ustalenie pozycji w kontenerach asocjacyjnych. Istnieje specjalna funkcja pomocnicza dla iteratorów - distance(), która zwraca odległość pomiędzy dwoma iteratorami. Pobiera ona dwa argumenty – iteratory wejściowe pos1 oraz pos2 i zwraca odległość pomiędzy nimi. Obydwa iteratory muszą odnosić się do elementów tego samego kontenera. Dla iteratorów dostępu swobodnego funkcja distance() zwraca po prostu wynik wyrażenia pos2 – pos1. W przypadku pozostałych kategorii iteratorów, iterator pos1 jest inkrementowany tak długo, aż osiągnie pozycję iteratora pos2, a zwracana jest liczba tych inkrementacji. W kolejnym programie zastosujemy obiekt typu pair oraz funkcje składową distance() do ustalenia pozycji, na jakiej umieszczamy wstawianą wartość do zbioru.

Wydruk 3.4 Kontener set,insert(),distance()

```
#include <QtCore/QCoreApplication>
#include <set>
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{QCoreApplication a(argc, argv);
 set<int> s1;
 set <int> :: iterator p;
 pair <set<int> :: iterator, bool> stan;
   s1.insert(12);
   s1.insert(14);
   s1.insert(19);
   s1.insert(11);
 int x = 17;
 cout <<" s1 = ";
 for (p=s1.begin(); p!=s1.end(); ++p) cout << *p << "   ";
 cout <<"\n   wstawiono " << x << " do zbioru s1\n s1 = ";
 stan = s1.insert(x);
 for (p=s1.begin(); p!=s1.end(); ++p) cout << *p << "   ";
 cout<<"\nna pozycji = "<<distance(s1.begin(),stan.first)+1;
    return a.exec();
}
```

Wynik wykonania programu ma postać:



W przedstawionym programie tworzony jest zbiór o nazwie s1 i inicjalizowany wartościami typu int. W instrukcji:

```
pair <set<int> :: iterator, bool> stan;
```

deklarujemy parę składającą się z iteratora i wartości typu bool. W kolejnej instrukcji:

```
stan = s1.insert(x);
```

używamy funkcji insert() do umieszczenia wartości 17 w obiekcie set. Zwracana para stan zawiera iterator i wartość typu bool. W instrukcji:

```
cout<<"\nna pozycji = "<<distance(s1.begin(),stan.first)+1;
```

wykorzystano funkcję distance() do ustalenia pozycji wstawianej wartości 17 w zbiorze s1. Wynik wyświetlony jest na ekranie monitora.

Kontener **multiset** służy do szybkiego zapamiętywania i odzyskiwania kluczy, ale w przeciwieństwie do kontenera set dopuszcza duplikowanie kluczy. Podobnie jak w zbiorze set, elementy są uporządkowane (porządek rosnący jest domyślnym). Kontener multiset obsługuje wszystkie operacje dostępne w set.

Kontener **map** uważany jest za jeden z bardziej przydatnych kontenerów. Przechowuje on pary klucz – wartość. Operacje wstawiania, wyszukiwania i usuwania korzystają z wartości klucza, Nazwa map związana jest z funkcjonalnością kontenera – odwzorowuje (mapuje) klucze na wartości. Duplikaty kluczy nie są możliwe w obiekcie map, (ale są dopuszczalne w kontenerze multimap). Jest to mapowanie jeden-do jednego. Kontener map przechowuje elementy w postaci uporządkowanej względem kluczy. Istotną zaletą map jest umożliwienie wyszukiwania wartości na podstawie klucza. Bardzo często tworzymy spis znajomych oraz ich numerów telefonów. Możemy utworzyć mapę, służącą do przechowywania nazwisk i numerów telefonów, nazwisko będzie kluczem.

Kontener asocjacyjny **multimapa** (ang. multimap), który działa podobnie jak kontener mapa jest także użytecznym zasobnikiem. Przechowuje pary klucz – wartość, ale klucze mogą się powtarzać.

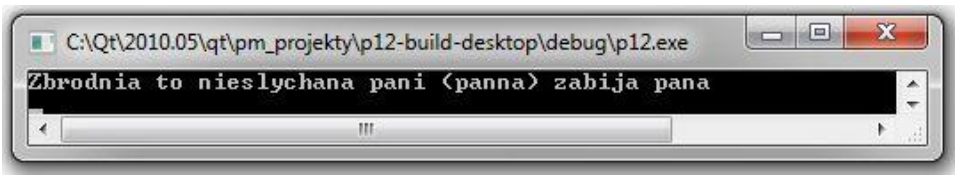
Elementy kontenera multimapa są automatycznie sortowane, gdy określimy kryterium sortowania wobec bieżącego klucza. Multimapa posiada dwukierunkowy iterator.

Kolejny program ilustruje wykorzystanie kontenera multimap. Do wstawiania elementów do kontenera multimap wykorzystano metodę ***insert()*** z użyciem funkcji ***make_pair().*** Do programu musimy dołączyć plik nagłówkowy <map> aby można było używać klasy multimap.

Wydruk 3.5 Kontener multimap,insert(), make_pair()

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main(int argc, char *argv[])
{
   QCoreApplication a(argc, argv);
   typedef multimap<int, string> AM ;
   AM lilije;
   lilije.insert(make_pair(6,"pana"));
   lilije.insert(make_pair(2,"to"));
   lilije.insert(make_pair(4,"pani"));
   lilije.insert(make_pair(1,"Zbrodnia"));
   lilije.insert(make_pair(3,"nieslychana"));
   lilije.insert(make_pair(4,"(panna)"));
   lilije.insert(make_pair(5,"zabija"));
   AM :: iterator n;
   for(n = lilije.begin(); n != lilije.end(); ++n)
        cout << n -> second << ' ';
   cout << endl;

   return a.exec();
}
```

W wyniku działania programu zostaje wyświetlony tekst:



Jest to początek znanej ballady Adama Mickiewicza zatytułowanej Lelije z roku 1820, dodatkowy napis w nawiasie (***panna***) wygenerował nasz program w celach dydaktycznych. W instrukcji

```
typedef multimap<int, string> AM ;
```

wykorzystano wyrażenie ***typedef*** do definicji typu multimap (synonim **AM**), gdzie typem klucza jest **int**, typem skojarzonej wartości jest **string**, a elementy (przez domniemanie) są uporządkowane rosnąco. Taką konstrukcje stosujemy zazwyczaj, aby zastąpić skomplikowaną nazwę typu prostszym symbolem. Nie tworzymy osobnego typu a jedynie synonim. W instrukcjach:

```
AM lilije;
lilije.insert(make_pair(6,"pana"));
```

tworzymy obiekt klasy multimap o nazwie ***lilije*** (taki tytuł nosi ballada A. Mickiewicza) oraz wykorzystując funkcję ***insert()*** dodajemy nowe pary klucz-wartość do obiektu multimap. W tej konkretnej instrukcji klucz 6 jest skojarzony z napisem „pana”. Ponieważ elementy kontenera są parami, do ich wstawiania wykorzystano wygodną funkcję ***make_pair().*** Funkcja ***make_pair()*** tworzy obiekty typu ***pair***, które reprezentują pary wartości. W kolejnych instrukcjach:

```
AM :: iterator n;
for (n = lilije.begin( ); n ! = lilije.end( ); ++n )
     cout << n -> second << ' ';
```

tworzymy iterator kontenera. Program wyświetli wszystkie elementy kontenera w pętli **for** przy pomocy iteratora kontenera **n**. Wewnątrz pętli iterator zostaje zainicjalizowany pierwszym elementem kontenera, a następnie przesuwa się po kolejnych elementach do momentu, gdy osiągnie pozycję ostatniego elementu. Operator inkrementacji kontenera jest tak zdefiniowany, że bierze pod uwagę drzewiastą strukturę kontenera. Ponieważ mamy obiekty **pair**, możemy uzyskać dostęp do jego składników używając składowych ***first*** oraz ***second***. W naszym przykładzie żądamy wyświetlenia składowej ***second***, (czyli wartości typu **string** skojarzonej z kluczem **int**).

ROZDZIAŁ 4

ITERATORY C++ STL

4.1. Wstęp

Zgodnie z definicją Stepanova ***Iteratory*** są uogólnieniem wskaźników języka C++, dzięki którym programista może pracować z różnymi kontenerami zgodnie z jednolitymi zasadami. Dzięki iteratorom można poruszać się po zawartości kontenera w podobny sposób jak dzięki wskaźnikom przemieszczamy się w obrębie tablicy. Ponieważ iteratory zachowują się bardzo podobnie do wskaźników, możliwa jest ich inkrementacja i dekrementacja. Pamiętamy, że w języku C++ mając wskaźnik ***wsk*** do elementu tablicy, dzięki wykonaniu inkrementacji: ***wsk + 1,*** otrzymamy wskaźnik do następnego elementu tablicy. Podobnie działają iteratory. Wszystkie kontenery maja zdefiniowane własne iteratory. Iteratory deklaruje się za pomocą typu **iterator** zdefiniowanego w różnych kontenerach. W zasadzie, aby operować iteratorami nie musimy włączać odrębnego pliku nagłówkowego. Jedynie, gdy programista chce wykorzystać specjalne iteratory należy włączyć plik nagłówkowy o nazwie <iterator>. Istnieje pięć różnych typów iteratorów, ich charakterystyki pokazane są w tabeli 1.

Tabela 1. Iteratory i ich charakterystyki

Typ iteratora	charakterystyka
Iterator wejściowy *(input iterator)*	Umożliwia odczytanie danych wejściowych (z kontenera, klawiatury, itp.)
Iterator wyjściowy *(output iterator)*	Umożliwia zapis danych (niekonieczny jest odczyt danych)
Iterator postępujący *(forward iterator)*	Jest to ulepszona wersja iteratorów we/wy, umożliwia odczyt i zapis danych. Używa jedynie operatora ++ do poruszania się po kontenerze, co pozwala jedynie na ruch do przodu i zapewnia odczyt tylko po jednym elemencie
Iterator dwukierunkowy *(bidirectional iterator)*	Ma te same funkcje, co iterator postępujący, ale zapewnia przeglądanie kontenera w przód i wstecz
Iterator dostępu swobodnego *(random access iterator)*	Jest to ulepszona wersja, iteratora dwukierunkowego, umożliwia odczyt i zapis z dostępem swobodnym (udostępnia operatory do wykonywania arytmetyki iteratorów, co jest analogiczne do arytmetyki wskaźników)

Każdy kontener posiada własne definicje iteratorów, ale wiele typów operacji przeprowadzonych z wykorzystaniem iteratorów daje jednolite wyniki dla wszystkich kontenerów.

Operator dereferencji * na przykład pozwala na wykonanie dereferencji iteratora, dzięki czemu możemy wykorzystać wartość elementu wskazywanego przez iterator. Podobnie inkrementacja iteratora ++ zwraca iterator do następnego elementu kontenera.

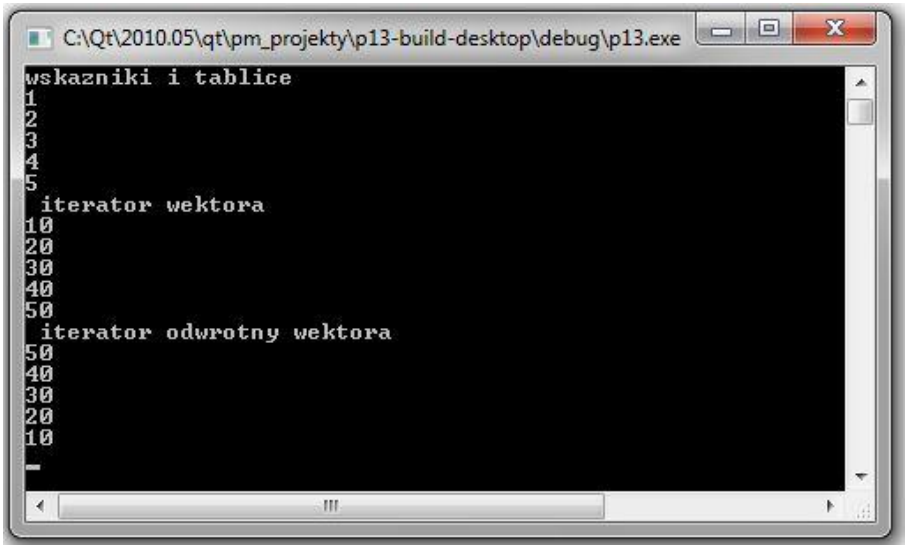
4.2. Proste wykorzystanie iteratorów

W prostym przykładzie stosowania, iteratora wykorzystamy fakt, że istnieje związek, pomiędzy iteratorem i kontenerem vector, analogiczny do związku pomiędzy wskaźnikiem i tablicą, jaki istnieje w C++. Jak wiadomo do elementu tablicy możemy odwołać się poprzez indeks albo wskaźnik. Podobnie zachowuje się iterator – umożliwia dostęp do elementu wektora. Kolejny program pokazuje analogie występujące pomiędzy iteratorami i wskaźnikami.

Wydruk 4.1 Iterator, kontener **vector**, typ **int**

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <vector>
using namespace std;

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  const int R = 5;
  int *wsk;              //deklaracja wskaznika
  int tab[R] = {1,2,3,4,5};   //klasyczna tablica
  cout << "wskazniki i tablice" << endl;
  for (wsk=tab; wsk<tab+R; wsk++)
        cout << *wsk << endl;
  vector <int> dane ;   // kontener vector
  dane.push_back(10);
  dane.push_back(20);
  dane.push_back(30);
  dane.push_back(40);
  dane.push_back(50);
  vector <int> :: iterator p1; //deklaracja iteratora
  cout << " iterator wektora" << endl;
  for (p1 = dane.begin(); p1!= dane.end(); ++p1)
       cout << *p1 << endl;
  vector <int> :: reverse_iterator p2;  //iterator
  cout << " iterator odwrotny wektora" << endl;
  for (p2 = dane.rbegin(); p2!= dane.rend(); ++p2)
       cout << *p2 << endl;

  return a.exec();
}
```

Po uruchomieniu programu mamy następujące wyniki:



Następujący fragment programu:

```
const int R = 5;
int *wsk;                    //deklaracja wskaznika
int tab[R] = {1,2,3,4,5};  //klasyczna tablica
cout << "wskazniki i tablice" << endl;
for (wsk=tab; wsk<tab+R; wsk++)
     cout << *wsk << endl;
```

tworzy zainicjalizowaną tablicę ***tab***, wykorzystując arytmetykę wskaźnikową w pętli ***for*** wyświetla wartości kolejnych elementów tablicy ***tab***, korzystając z operatora dereferencji *. Kolejno deklarujemy kontener vector o nazwie dane, w którym przy pomocy metody push_back() umieszczamy kolejno wartości całkowite:

```
vector <int> dane ;              // kontener vector
dane.push_back(10);
   ..........................
```

Tak, jak w przypadku wskaźników, iterator także musi być zadeklarowany. Deklaracja iteratora o nazwie ***p1*** działającego na kontenerze vector ma postać:

```
vector <int> :: iterator p1; //deklaracja iteratora
```

Tak deklaracja jest ważna, ponieważ każdy kontener definiuje typ o nazwie iterator, który jest charakterystyczny dla danego kontenera.

Przy pomocy pętli ***for*** wykorzystując operator dereferencji iteratora ***p1*** wyświetlane są wprowadzone do kontenera wartości:

```
for (p1 = dane.begin(); p1!= dane.end(); ++p1)
      cout << *p1 << endl;
```

W wyrażeniach pętli for wykorzystano metody begin() oraz end(). Metoda begin() zwraca iterator (pozycję) do pierwszego elementu umieszczonego w kontenerze dane . Inkrementując iterator:

```
++p1
```

uzyskujemy dostęp do kolejnych elementów kontenera vector. Należy zwrócić uwagę na wyrażenie warunkowe, wykorzystane do sprawdzenia warunku osiągnięcia końca kontenera:

```
p1!= dane.end();
```

Przy pomocy metody ***end()*** sprawdzamy, czy iterator znajduje się poza końcem sekwencji elementów umieszczonych w kontenerze. Gdy iterator osiągnie to miejsce, pętla kończy działanie. W programie pokazano także działania metod rbegin() i rend(), dzięki którym możemy przeglądać kontener w odwrotnym kierunku (wartości elementów kontenera pokazane będą w odwrotnej kolejności, pierwsza pokazana będzie ostatnia wartość kontenera), wykorzystaliśmy tzw. iterator odwrotny:

```
vector <int> :: reverse_iterator p2; //deklaracja
                                     //iteratora odwrotnego
cout << " iterator odwrotny wektora" << endl;
for (p2 = dane.rbegin(); p2!= dane.rend(); ++p2)
     cout << *p2 << endl;
```

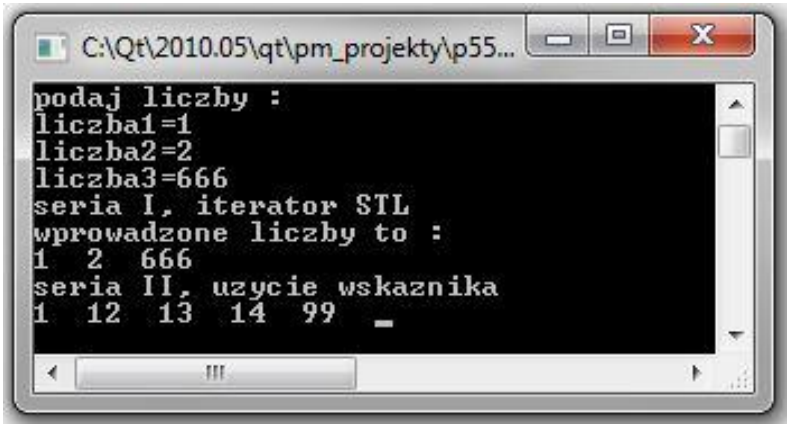
Przy pomocy iteratorów możemy prosto manipulować elementami w kontenerach STL, szczególnie przydatne są iteratory w obsłudze algorytmów STL. Iteratory STL działają bardzo podobnie do wskaźników. Dzięki temu można stosować algorytmy STL obsługiwane przez wskaźniki C++. To zagadnienie ilustruje kolejny program. W programie przypomniano także zastosowanie klasycznych iteratorów kontenera wektor.

Wydruk 4.2 Iteratory STL i wskaźniki

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
```

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    vector<int> liczby(3);
    cout <<"podaj liczby :"<<endl;
    for(int i =0; i<3; i++)
    {cout << "liczba" << i+1 << "=";
        cin >> liczby[i];
    }
    cout << "seria I, iterator STL"<<endl;
    vector<int>::iterator p;
    cout << "wprowadzone liczby to :"<< endl;
    p=liczby.begin();
    while(p!=liczby.end())
    {cout << *p << "  ";
      p++;
    }
    cout << "\nseria II, uzycie wskaznika"<<endl;
    int ar[5]={12,13,14,1,99};
    int *begin = &ar[0];
    int *end = &ar[5];
    sort(begin,end);
    for(int i=0; i<5; i++)
        cout << ar[i]<<"  ";

    return a.exec();
}
```

Wynikiem wykonania programu jest następujący komunikat:



W programie tworzymy wektor liczb całkowitych i z klawiatury wprowadzamy trzy elementy.

W celu wydrukowania wartości elementów tworzymy iterator p do kontenera vector i wykorzystujemy metody begin() i end() :

```
vector<int>::iterator p;   // iterator p
cout << "wprowadzone liczby to :"<< endl;
p=liczby.begin();  //ustawia iterator na 1 elemencie
while(p!=liczby.end())     //metoda end()
{cout << *p << "  ";       //dereferencja iteratora
  p++;
}
```

Następnie tworzymy tablicę elementów typu int i definiujemy dwa wskaźniki tej tablicy (begin i end), wykorzystujemy algorytm STL o nazwie sort() do uporządkowania elementów tablicy:

```
int ar[5]={12,13,14,1,99};
int *begin = &ar[0];
int *end = &ar[5];
sort(begin,end);
```

Widzimy, że algorytm STL może także być obsługiwany przez zwykle wskaźniki (nie tylko przez odpowiednie iteratory). Tabela 2 pokazuje podstawowe operacje wykonywane na iteratorach.

Tabel 2. Opis podstawowych operacji wykonywanych na iteratorach STL.

operacja	opis
*p	Zwraca wartość bieżącego element (dereferencja)
++p	Przemieszcza iterator do następnego elementu
p+= n	Przesuwa iterator o n elementów
--p	Przesuwa iterator z powrotem o jeden element
p -= n	Przesuwa iterator z powrotem o n elementów
p – p1	Zwraca liczbę elementów pomiędzy p and p1

Należy pamiętać, jak to dowcipnie napisali N. Solter i S. Kleper (C++, zaawansowane programowanie) „iteratory są niemal tak bezpieczne jak wskaźniki, czyli są niezwykle niebezpieczne”. Przykładem może być niepoprawne użycie metody ***end().*** Pamiętamy, że iterator zwracany przez ***end()*** wskazuje na miejsce poza ostatnim elementem kontenera. Dereferencja tego elementu jest nieokreślona, co może spowodować przerwanie działania programu.

4.3. Typy iteratorów

Zgodnie z definicją Stepanova ***Iteratory*** są uogólnieniem wskaźników języka C++, dzięki którym programista może pracować z różnymi kontenerami zgodnie z jednolitymi zasadami. Dzięki iteratorom można poruszać się po zawartości kontenera w podobny sposób jak dzięki wskaźnikom przemieszczamy się w obrębie tablicy. Ponieważ iteratory zachowują się bardzo podobnie do
Potrzeba istnienia kilku kategorii iteratorów wynika ze względów praktycznych – każdy kontener ma inne wymagania względem obsługi pozycji elementów w kontenerze. W tabeli 3 pokazano typy iteratorów związane z konkretnym kontenerem.

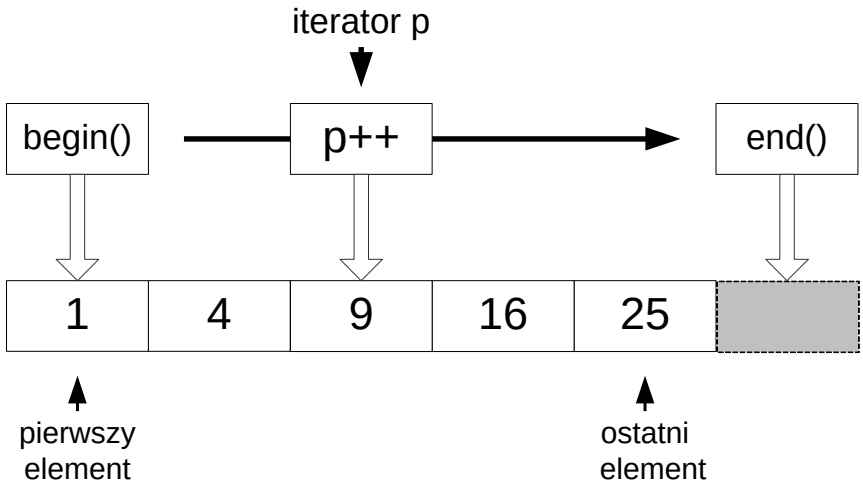
Tabela 3. Kontenery i dostępne iteratory

kontener	Typ iteratora
vector	Iterator dostępu swobodnego
list	Iterator dwukierunkowy
deque	Iterator dostępy swobodnego
set	Iterator dwukierunkowy
multiset	Iterator dwukierunkowy
map	Iterator dwukierunkowy
multimap	Iterator dwukierunkowy

Iterator jest obiektem, dzięki któremu możemy wykonywać operacje przemieszczania się od jednego elementu do innego w kontenerze. Z praktycznego punktu widzenia iterator reprezentuje określoną pozycję elementu w kontenerze. Klasy kontenerowe udostępniają wiele funkcji, dzięki którym można wykonywać operacje związane z przemieszczaniem się iteratora po elementach.

Wydaje się, że dwie funkcje są najczęściej wykorzystywane:

- begin() – zwraca iterator reprezentujący początek elementów w kontenerze (pozycje pierwszego elementu w sekwencji)
- end() – zwraca iterator, który określa pozycję potrzebną do ustalenia pozycji ostatniego elementu w sekwencji, jest to pozycja za ostatnim elementem w kolekcji (rys. 4.1).



Rys. 4.1. Pozycje wskazywane przez metody begin() oraz end()

W bibliotece STL są zaimplementowane trzy dodatkowe funkcje operujące na iteratorach:

- advance()
- distance()
- iter_swap()

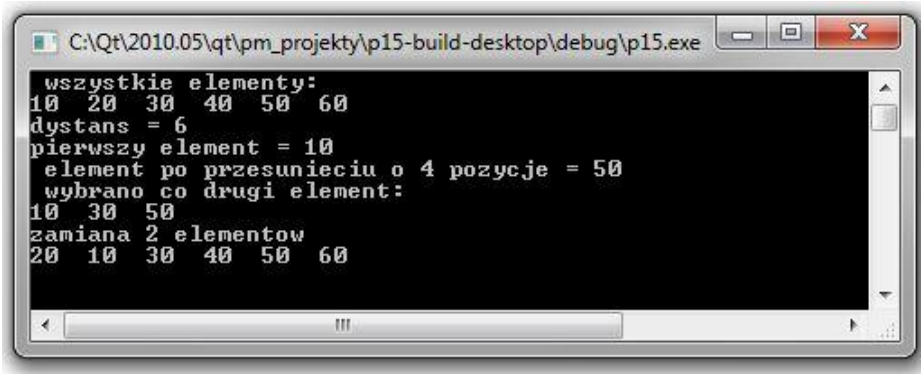
Użycie tych funkcji wymaga dołączenia pliku <iterator>. Funkcja advance(), umożliwia przesunięcie iteratora ***p*** o **n** elementów. Argument **n** może być dodatni lub ujemny. Funkcja distance zwraca odległość pomiędzy iteratorami p1i p2. Kolejną funkcją pomocniczą jest funkcja ***iter_swap().***Funkcja iter_swap() wykorzystywana jest do zamiany dwóch iteratorów. Iteratory nie muszą być tego samego typu. W prostym przykładzie pokażemy zastosowanie funkcji pomocniczych operujących na iteratorach.

Wydruk 4.3 Użycie advance() oraz iter_swap()

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <deque>
using namespace std;

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  deque<int> dane ;                // kontener deque
  dane.push_front(60);
  dane.push_front(50);
  dane.push_front(40);
  dane.push_front(30);
```

```
 dane.push_front(20);
dane.push_front(10);
deque <int> :: iterator p1,p2;  //deklaracja iteratora
cout << " wszystkie elementy: " << endl;
for (p1 = dane.begin(); p1!= dane.end(); ++p1)
     cout << *p1 << "  ";
cout<<"\ndystans = "<<dane.end()- dane.begin()<< endl;
p2 = dane.begin() ;
cout << "pierwszy element = " << *p2 << endl;
advance(p2, 4);
cout<<" element po przesunieciu o 4 pozycje = "<<*p2 << endl;
cout << " wybrano co drugi element: " << endl;
for (p1 = dane.begin(); p1< dane.end()-1; p1 +=2)
     cout << *p1 << "  ";
cout << "\nzamiana 2 elementow"<< endl;
iter_swap(dane.begin(),dane.begin()+1);
for (p1 = dane.begin(); p1!= dane.end(); ++p1)
     cout << *p1 << "  ";
return a.exec();
}
```

Wynik działania programu ma postać:



Działanie funkcji advance() pokazane jest w następującym fragmencie programu:

```
p2 = dane.begin() ;
cout << "pierwszy element = " << *p2 << endl;
advance(p2, 4);
cout<<"element po przesunieciu o 4 pozycje = "<<*p2<<endl;
```

Na początku iterator p2 wskazuje na pierwszy element sekwencji (funkcja begin()), co potwierdza pierwszy wydruk *p2. Następnie wywołujemy funkcję advance() z parametrami (p2, 4) . Oznacza to, że żądamy przesunięcia iteratora p2 o 4 miejsca. Kolejny wydruk *p potwierdza, że p2 wskazuje na 5 element w naszej kolekcji, jego wartość jest równa 50.

Sposób zamiany dwóch elementów kolekcji pokazany jest w instrukcji:

```
iter_swap(dane.begin(),dane.begin()+1);
```

Funkcja iter_swap() pobiera dwa argumenty – iteratory (pozycje) dwóch elementów, które mają się wymienić położeniem. W naszym przykładzie zamieniamy miejscami element pierwszy (begin()) oraz drugi (begin()+1). W wyniku element drugi staje się elementem pierwszym kolekcji, a element pierwszy staje się drugim.

ROZDZIAŁ 5

ALGORYTMY C++ STL

5.1. Wstęp

Według wielu specjalistów najważniejszymi elementami STL są kontenery, iteratory i algorytmy**.** Cenną zaletą STL jest fakt, że zawiera ona wiele typowych algorytmów. W praktyce programistycznej bardzo często sortujemy dane, wyszukujemy odpowiednie elementy, itp. Wykorzystując algorytmy STL programista oszczędza mnóstwo czasu. Algorytmy STL są odseparowane od kontenerów, wykorzystują iteratory do operowania na elementach kontenera**.** Dzięki takiej koncepcji algorytmy wykonują swoje działania na wielu kontenerach, bez względu na typ danych. STL zawiera znaczą ilość algorytmów. Doskonały przegląd wszystkich algorytmów STL można znaleźć w podręczniku „C++, biblioteka standardowa, podręcznik programisty autorstwa N. Josuttisa. W zależności od implementacji STL zawiera od 60 do 70 standardowych algorytmów. N. Josuttis (2003 rok) wprowadził następująca klasyfikację:

- Algorytmy niemodyfikujące (ang. *nonmodifying algorithms*)
- Algorytmy modyfikujące (ang. *nonmodifying algorithms*)
- Algorytmy usuwające (ang. *removing algorithms*)
- Algorytmy mutujące (ang. *mutating algorithms*)
- Algorytmy sortujace (ang**.** *sorting algorithms*)
- Algorytmy przeznaczone dla zakresów posortowanych ((ang. *sorted range algorithms*)
- Algorytmy numeryczne (ang**.** numeric *algorithms*)

Należy pamiętać, że użycie algorytmów wymaga dołączenia pliku nagłówkowego z definicjami algorytmów:

```
#include <algorithm>
```

Niektóre algorytmy numeryczne wymagają dołączenia pliku nagłówkowego numerycznego:

```
#include <numeric>
```

Gdy używane będą obiekty funkcyjne (niektóre algorytmy wymagają tego) należy włączyć także odpowiedni plik z definicjami obiektów funkcyjnych i adaptatorów funkcji:

```
#include <functional>
```

Zasadniczo algorytmy STL wykonują operacje na kontenerach wykorzystując iteratory. Kilka algorytmów może działać nie wymagając elementów zapisanych w kontenerach, może działać także tylko na wartościach.

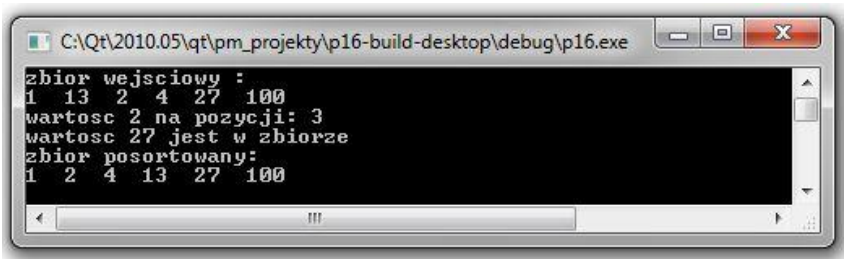
5.2. Algorytmy wyszukiwania i sortowania

W kolejnym przykładzie demonstrujemy użyteczne algorytmy wyszukiwania elementu w zbiorze (algorytm find() i binary_search() oraz algorytm sort() służący do porządkowania elementów w sekwencji.

Wydruk 5.1 Algorytmu find(), sort(), binary_search()

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

int main(int argc, char *argv[])
{
 QCoreApplication a(argc, argv);
 int tab[6] = {1,13, 2, 4, 27,100 };
 vector <int> v(tab, tab+6);
 vector <int> :: iterator p1;
 cout << "zbior wejsciowy : "<< endl;
 for (p1=v.begin(); p1 != v.end(); ++p1)
      cout << *p1 << "  ";
 cout << endl;
 p1 = find(v.begin(), v.end(), 2);
 cout<< "wartosc 2 na pozycji: " <<(p1 - v.begin()+1)<<endl;
 if (binary_search(v.begin(), v.end(), 27))
     cout << "wartosc 27 jest w zbiorze";
 else
     cout << "wartosci tej nie ma w zbiorze" << endl;
 cout <<endl;
 sort(v.begin(),v.end());
 cout << "zbior posortowany:"<< endl;
 for (p1=v.begin(); p1 != v.end(); ++p1)
      cout << *p1 << "  ";
 cout << endl;

 return a.exec();
}
```

Po uruchomieniu tego programu mamy następujący komunikat:



W programie zainicjalizowano wektor v tablicą tab:

```
int tab[6] = {1,13, 2, 4, 27,100 };
vector <int> v(tab, tab+6);
```

przeciążony konstruktor vector, korzystając z dwóch iteratorów (pamiętamy, że wskaźniki do tablicy mogą być użyte, jako iteratory) tworzy wektor v zainicjowany elementami tablicy tab od lokalizacji tab do lokalizacji mniejszej niż tab + 6. W programie definiujemy iterator p1:

```
vector <int> :: iterator p1;
```

Wykorzystamy algorytm find() do sprawdzenia czy wyspecyfikowany element jest w zbiorze elementów wektora v:

```
p1 = find(v.begin(), v.end(), 2);
cout << "wartosc 2 na pozycji: " << (p1 - v.begin()+1) << endl;
```

W pokazanej instrukcji, algorytm find(p1,p2,w) zwraca pozycję pierwszego wystąpienia elementu o wartości w z zakresu p1 i p2 w kontenerze v. Gdy wyspecyfikowana wartość nie będzie znaleziona zostanie zwrócona pozycja p2 (koniec sekwencji). Do stwierdzenia faktu, czy wyspecyfikowana wartość znajduje się w zbiorze elementów można wykorzystać algorytm binary_search(). Formalnie ten algorytm wymaga uporządkowanego zbioru (można to osiągnąć korzystając z algorytmu sort()), ale w większości implementacji, algorytm binary_search() działa poprawnie, tak jak to pokazano w naszym przykładzie. Rekomenduje się wykorzystanie algorytmu binary_search(), ponieważ działa bardzo szybko. Jeżeli szukany element jest w zbiorze, to algorytm binary_search zwraca wartość typu bool (true), w przeciwnym przypadku zwróci wartość false.

```
if (binary_search(v.begin(), v.end(), 27))
    cout << "wartosc 27 jest w zbiorze";
else
     cout << "wartosci tej nie ma w zbiorze" << endl;
```

Do posortowania elementów w kontenerze wykorzystaliśmy algorytm sort(p1,p2), który ustawia elementy w zakresie od p1 do p2 (ale z wyłączeniem p2) w obiekcie v w porządku rosnącym:

```
sort(v.begin(),v.end());
```

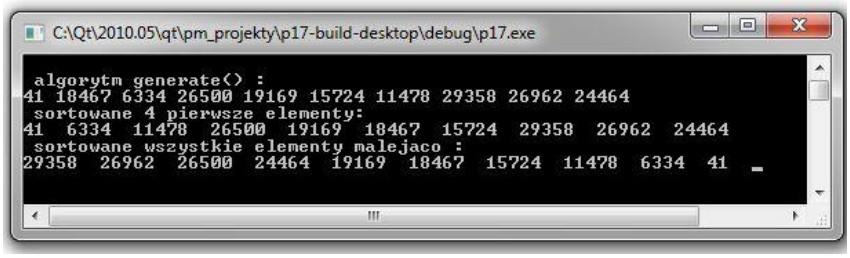
5.3. Algorytmy sortowania częściowego

Sortowanie kolekcji można wykonać także częściowo, w tym celu korzystamy z algorytmu partial_sort(). To zagadnienie ilustruje kolejny program.

Wydruk 5.2 Algorytmu partial_sort()

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>      //ostream_iterator

using namespace std;

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  int i;
  vector <int> k1(10);
  cout<< "\n algorytm generate() :" << endl;
  generate(k1.begin(), k1.end(), rand);
  for (i=0; i<k1.size(); i++)
       cout << k1[i]<< ' ';
  cout << "\n sortowane 4 pierwsze elementy: "<< endl;
  partial_sort(k1.begin(), k1.begin()+3, k1.end());
  copy(k1.begin(),k1.end(),ostream_iterator<int>(cout,"  "));
  cout<< "\n sortowane wszystkie elementy malejaco :" << endl;
  partial_sort(k1.begin(), k1.end(), k1.end(), greater<int>());
  copy(k1.begin(),k1.end(),ostream_iterator<int>(cout, "  "));

  return a.exec();
}
```

W wyniku działania programu otrzymamy następujący komunikat:



Program generuje losowo liczby całkowite i umieszcza je w kontenerze k1:

```
generate(k1.begin(), k1.end(), rand);
```

W kolejnej instrukcji:

```
partial_sort(k1.begin(), k1.begin()+3, k1.end());
```

wywoływany jest algorytm partial_sort(). Przy pomocy algorytmu partial_sort() sortujemy elementy, inaczej niż to wykonuje algorytm sort().

Sortownie przebiega do momenty aż osiągniemy posortowane elementy do pozycji wskazywanej przez drugi argument. W naszym przypadku jest to

```
k1.begin()+3
```

to znaczy chcemy otrzymać tylko cztery pierwsze posortowane elementy kolekcji. Sortowanie algorytmem partial_sort() może przebiegać także zgodnie z ustaleniami predykatu dwuargumentowego, który przekazywany jest opcjonalnie jako czwarty parametr:

```
partial_sort(k1.begin(),k1.end(),k1.end(),greater<int>());
```

Obiekt funkcyjny greater<typ> jest wykorzystywany jako kryterium sortowania, spowoduje on, że porządek w kolekcji będzie rosnący. Domyślne kryterium sortowania jest określone przez obiekt funkcyjny less<typ>, dzięki któremu porządek sortowania jest rosnący.

ROZDZIAŁ 6

KONTENERY QT

6.1. Wstęp

Dla Qt opracowano własne kontenerowe klasy, dzięki czemu użytkownik może w swoich programach pisanych w języku C++ wykorzystywać klasyczne kontenery STL a także kontenery Qt. Głównym argumentem za stosowaniem kontenerów Qt jest gwarancja, że wszystkie platformy i wersje Qt mają identyczne biblioteki oraz że współpraca z innymi elementami Qt jest zoptymalizowana. Dodatkowa Qt wprowadziło nowy typ iteratora wzorowany na koncepcji Javy, który według producentów nie generuje tak wiele problemów jak iterator STL. Koncepcja klas kontenerowych spełnia wszystkie wymogi nałożone na klasy kontenerowe STL, oczywiście mamy do dyspozycji także iteratory i algorytmy dostosowane do obsługi kontenerów Qt. Użytkownik ma wybór, mówi się, że STL jest bardziej rozbudowany i jej elementy działają szybciej, natomiast klasy kontenerowe Qt są bardziej proste w użyciu.
Kontenery QT są zoptymalizowane ze względu na szybkość działania, zapotrzebowanie na pamięć, dają mniejszy kod wykonywalny. Przeglądanie kontenera wykonywać można przy pomocy dwóch typów iteratorów: iteratorów w stylu Javy oraz iteratorów w stylu STL. Twierdzi się, że iteratory typu Java są bardziej proste w użyciu, iteratory STL są bardziej wydajne. Iteratory STL mogą być wykorzystane w generycznych algorytmach Qt. Możemy także wykorzystywać konstrukcję ***foreach*** do łatwego przeglądania zawartości kontenerów.

Qt dostarcza następujące kontenery sekwencyjne: QList, QLinkedList, QVector, QStack i QQueue. Najczęściej wykorzystywany jest kontener QList. Pakiet Qt dostarcza następujące kontenery asocjacyjne: QMap, QMultiMap, QHash, QMultiHash, i QSet. Są jeszcze specjalne klasy takie jak QCache oraz QContiguousCache. W tabeli 1 pokazano kontenery Qt i ich krótkie charakterystyki.

Tabela 1. Klasy kontenerowe Qt

Klasa Qt	opis
QList<T>	Przechowuje listę wartości typu T, które są obsługiwane najczęściej przez indeks.
QLinkedList<T>	Przechowuje listę wartości typu T, które są obsługiwane najczęściej przez iterator.
QVector<T>	Przechowuje tablicę wartości
QStack<T>	Jest to podklasa QVector do realizacji koncepcji LIFO. Posiada funkcje: push(), pop(), and top().
QQueue<T>	Jest to podklasa QList do realizacji

	koncepcji FIFO. Posiada funkcje enqueue(), dequeue(), and head().
QSet<T>	Kontener umożliwia obsługę zbiorów
QMap<Key, T>	Kontener, którego elementami są pary klucz-wartość. Każdy klucz obsługuje jedną wartość.
QMultiMap<Key, T>	Jest to podklasa QMap. Klucz może obsługiwać wiele wartości.
QHash<Key, T>	Jest to szybszy odpowiednik QMap.
QMultiHash<Key, T>	Jest to podklasa QHash, dla realizacji złożonego haszowania (multi-valued hashes).

Kontenery mogą być zagnieżdżane. Przykładem może być następująca konstrukcja:

```
QMap<QString, QList<int> >
```

gdzie typem klucza jest typem QString, a typem wartości jest Qlist<int>. Należy zawsze pamiętać, aby pomiędzy ostatnimi znakami większości (> >) umieścić spację, bo inaczej spowoduje to błąd krytyczny. Wykorzystywanie kontenerów wymaga dołączenia plików nagłówkowych o tych samych nazwach, co kontener. Na przykład o do obsługi kontenera QList musimy dołączyć plik:

```
#include <QList>
```

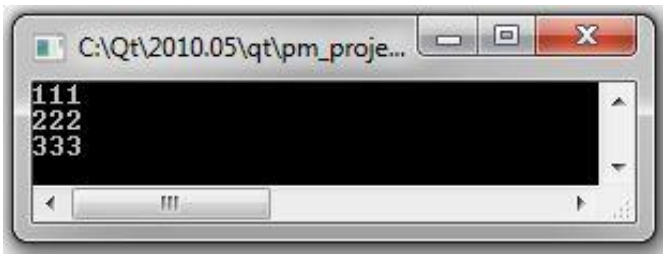
Niektóre kontenery wymagają dostarczenia odpowiednich funkcji, dlatego zawsze należy sprawdzić dokumentację opisującą konkretną klasę kontenerową. Z drugiej strony, gdy żądane warunki nie są spełnione, kompilator wygeneruje komunikat błędu.
Iteratory w stylu STL są bardzo dobre w obsłudze kontenerów Qt. W kolejnym programie wykorzystamy takie iteratory do obsługi listy Qt.

Wydruk 6.1 Kontener Qt, iteratory STL

```
#include <QtCore/QCoreApplication>
#include <QDebug>

int main(int argc, char *argv[])
{   QCoreApplication a(argc, argv);
    QList<int> lista;
    lista << 111 << 222 << 333;
```

```
    QList<int>::iterator p = lista.begin();
    while(p!=lista.end())
    {qDebug()<< *p;
     ++p;
    }
    return a.exec();
}
```

Wynikiem program jest następujący wydruk:



W programie utworzona została lista zawierająca trzy elementy typu int. Następnie został zdefiniowany iterator p w stylu STL dla kontenera QList i ustawiony na pierwszy element listy. W pętli while korzystając z dereferencji wskaźnika (*p) spowodowaliśmy wyprowadzenie wartości elementów na standardowe wyjście (ekran monitora):

```
QList<int>:: iterator p = lista.begin();
while(p!=lista.end())
{qDebug()<< *p;
 ++p;
}
```

6.2. Kontenery sekwencyjne Qt

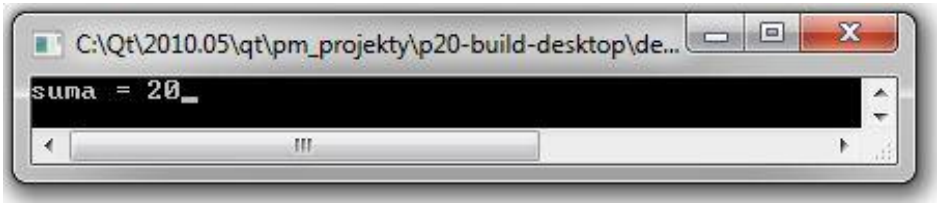
Bardzo prostym kontenerem jest QVector. QVector<T> jest strukturą bardzo podobną do tablicy, umieszcza elementy w ciągłym fragmencie pamięci. Zasadnicza różnica między kontenerem vector i tablicą C++ jest fakt, że vector zna własny rozmiar i w miarę potrzeby może być powiększany, aby pomieścić nowe elementy. Kolejny program demonstruje użycie kontenera QVector. W programie inicjalizujemy wektor wartościami i obliczamy ich sumę.

Wydruk 6.2 Kontener QVector

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <QVector>
using namespace std;
```

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QVector<int> v1(10);
    v1[0]=1;
    v1[1]=2;
    v1.append(2);
    v1.append(4);
    v1 << 5 << 6 ;
    int sum = 0;
    for(int i=0; i <v1.count(); ++i)
        sum += v1[i];
    cout << "suma = "<< sum;

    return a.exec();
}
```

Wynik wykonania programu ma postać:



W linii

```
QVector<int> v1(10);
```

deklarujemy wektor v1 o rozmiarze 10, który będzie przechowywał elementy typu int. Gdy na początku wiemy ile elementów będzie przechowywał wektor, możemy ustalić jego rozmiar, w przeciwnym przypadku tego nie robimy. Kontener wektor obsługiwany jest przez operator indeksu []. W programie pokazano trzy możliwości inicjalizacji wektora v1:

```
v1[0]=1;
v1.append(2);
v1 << 5 << 6 ;
```

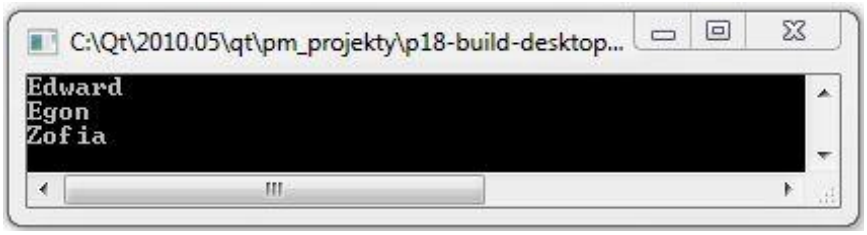
Funkcja append() umieszcza wartość na końcu wektora. Możemy użyć także przeciążonego operatora << zamiast funkcji append(). W pętli for w wyrażeniu warunkowym wykorzystaliśmy metodę count(). Ta funkcja zwraca liczbę wartości umieszczonych w kontenerze. Okazuje się, że umieszczanie elementów na początku wektora, usuwanie elementów ze środka wektora może być procesem mało wydajnym. W takim przypadku poleca się stosowanie kontenera QLinkedList lub QList. Obsługa kontenera QList jest także prosta. To

zagadnienie ilustruje następny program. W programie wprowadzamy na listę trzy imiona, algorytm qSort sortuje listę.

Wydruk 6.3 Kontener QList

```
#include <QtCore/QCoreApplication>
#include <QTextStream>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QTextStream out(stdout);
    QList<QString> list;
    list <<"Wacek"<<"Lola"<<"Ziuta";
    qSort(list);
    for(int i =0; i<list.size(); ++i)
        out << list.at(i) << endl;

    return a.exec();
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:



W programie do obsługi tekstu na konsoli wykorzystaliśmy specjalna klasę Qt 4 o nazwie QTextStream. Ta klasa dostarcza wygodny interfejs do odczytu i wydruku tekstu.
Po włączeniu pliku:

```
#include <QTextStream>
```

definiujemy obiekt ***out*** :

```
QTextStream out(stdout);
```

dzięki któremu dane mogą być wprowadzone do standardowego wyjścia.

W programie utworzono listę o nazwie ***list*** oraz wywołano algorytm sortujący qSort():

```
list <<"Wacek"<<"Lola"<<"Ziuta";
qSort(list);
```

Formalnie należałoby do programu włączyć następujące linie:

```
#include <iostream>
#include<QList>
```

W naszej instalacji korzystamy z Qt 4.7.0 (32 bit) i do edytowania programu mamy Qt Creator 2.0.1, potrzebne narzędzia włączane są przez domniemanie (tak jak w pokazanym przykładzie plik <QList>).
W pętli:

```
for(int i =0; i<list.size(); ++i)
        out << list.at(i) << endl;
```

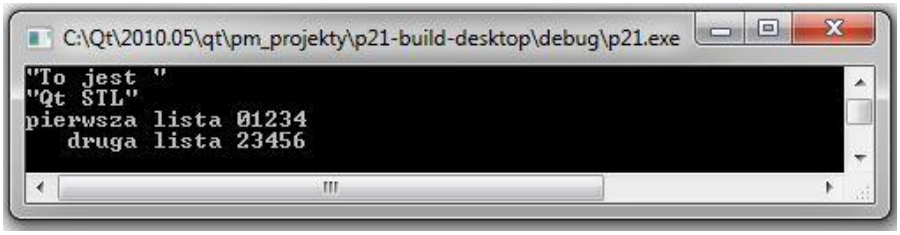
drukujemy na ekranie uporządkowane elementy naszej listy. Metoda listy at(i) zwraca element o indeksie i w legalnym zakresie, tzn. 0 <= i < size().
Kolejny przykład pokazujący obsługę kontenera QList jest bardziej skomplikowany, ponieważ wykorzystujemy kilka elementów typowych dla biblioteki Qt.

Wydruk 6.4 Kontener QList,<QDebug>

```
#include <QtCore/QCoreApplication>
#include <QDebug>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QList<QString> list;
    list <<"To jest "<<"Qt STL";
    QListIterator<QString> i(list);
    while(i.hasNext())
        qDebug() << i.next();
    QList<int>::iterator p;
    QList<int> lst;
    cout << "pierwsza lista ";
    for (int k=0;k<5;++k)
      {lst.push_back(k);
       cout << lst[k];
      }
```

```
    cout << "\n   druga lista ";
    for (p = lst.begin(); p != lst.end(); ++p)
        *p += 2;
    for (p = lst.begin(); p != lst.end(); ++p)
        cout << *p;

    return a.exec();
}
```

Wynikiem wykonania programu jest komunikat na ekranie monitora:



W programie tworzymy listę o nazwie list, w której umieszczamy dwa napisy

```
QList<QString> list;
list <<"To jest "<<"Qt STL";
```

W celu wydrukowania elementów listy wykorzystamy iterator Qt (styl Javy) oraz obsługę komunikatów QT (qDebug):

```
QListIterator<QString> i(list);  //iterator styl Javy
while(i.hasNext())
   qDebug() << i.next();
```

Wykorzystana metoda hasNext():

```
bool QListIterator::hasNext() const
```

zwraca true jeżeli znajduje sie jeden element przed iteratorem, tzn. iterator nie jest na końcu kontenera, w przeciwnym przypadku zwraca false. Tego typu iterator (zwany iteratorem w stylu Javy będzie omówiony później). Metoda next():

```
const T & QListIterator::next()
```

zwraca następny element i powiększa iterator o jedna pozycję. Obsługą wydruku teksu zajmuje się funkcja qDebug():

```
void qDebug(const char* msg,……)
```

Formalnie wywołanie funkcji qDebug() skutkuje otrzymaniem obiektu QDebug użytecznego do zapisywania informacji w obsłudze błędów. W praktyce qDebug można używać zamiast obiektu cout. Proste wykorzystanie funkcji qDebug() ma postać:

```
int x = 10;
qDebug() << "nasz wynik = " << x;
```

W następnym fragmencie programu:

```
QList<int> lst;
cout << "pierwsza lista ";
for (int k=0;k<5;++k)
 {lst.push_back(k);
   cout << lst[k];
 }
```

Tworzymy listę o nazwie lst do przechowywania elementów typu int. Umieszczanie elementów w kontenerze wykonuje metoda push_back(). Do obsługi elementów listy wykorzystujemy operator indeksowania. Korzystając ze zdefiniowanego iteratora p w stylu STL:

```
QList<int>::iterator p;
```

w kolejnym fragmencie następuje nadpisanie elementów listy:

```
for (p = lst.begin(); p != lst.end(); ++p)
    *p += 2;
for (p = lst.begin(); p != lst.end(); ++p)
    cout << *p;
```

Funkcja begin() zwraca iterator reprezentujący początek elementów w kontenerze, jest to pierwsza pozycja. Funkcja end() zwraca iterator reprezentujący koniec elementów w kontenerze. Jest to pozycja ***za ostatnim*** elementem.

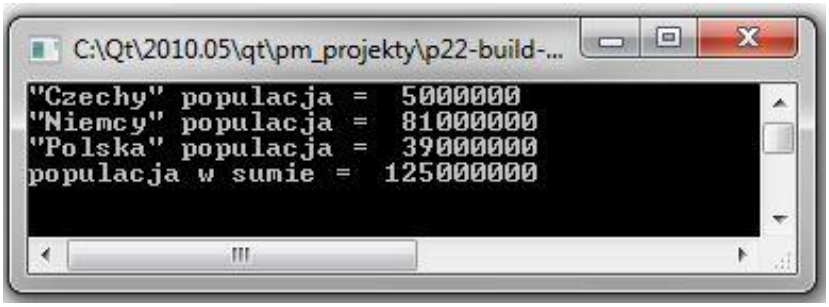
6.3. Kontenery asocjacyjne Qt

Kontenery asocjacyjne przechowują elementy tego samego typu indeksowane kluczem. Qt obsługuje dwa główne kontenery asocjacyjne: QMap<K,T> oraz QHash<K,T>.
Kontener QMap przechowuje uporządkowane pary klucz-wartość.
Kolejny przykład ilustruje obsługę kontenera QMap. Program przechowuje listę państw (klucz) i ich populacje ((wartość).

Wydruk 6.5 Kontener QMap

```
#include <QtCore/QCoreApplication>
#include <QMap>
#include <QDebug>3
int main(int argc, char *argv[])

{
  QCoreApplication a(argc, argv);
  int suma =0;
  QMap<QString,int> kraj;
  kraj["Polska"]= 39000000;
  kraj["Niemcy"]= 81000000;
  kraj["Czechy"]=  5000000;
  QMapIterator<QString,int> i(kraj);
  while(i.hasNext())
  {i.next();
   qDebug()<< i.key()<<"populacja = "<< i.value();
  }
  i.toFront();
  while(i.hasNext())
      suma += i.next().value();
  qDebug()<< "populacja w sumie = "<< suma;

  return a.exec();
}
```

Wynikiem uruchomienia programu jest komunikat:



W instrukcjach:

```
QMap<QString,int> kraj;
kraj["Polska"]= 39000000;
kraj["Niemcy"]= 81000000;
kraj["Czechy"]=  5000000;
```

tworzymy mapę o nazwie kraj.
Program przechowuje pary klucz/wartość pokazując kraj jego populację.

Kluczem jest wartość typu string, a przechowywana wartość jest typu int.
Do inicjalizacji mapy można tez wykorzystać metodę insert():

```
kraj.insert("Polska",39000000);
kraj.insert("Niemcy",81000000;
kraj.insert("Czechy",5000000;
```

W celu wydrukowania zawartości mapy korzystamy z iteratora w stylu Javy:

```
QMapIterator<QString,int> i(kraj);
```

oraz metod hasNext() i next():

```
while(i.hasNext())
 {i.next();
  qDebug() << i.key()<<"populacja = "<< i.value();
 }
```

W celu wykonania ponownego przeglądania zawartości mapy musimy iterator ustawić na początku mapy, należy wykorzystać metodę toFront()

```
i.toFront();
```

Jeżeli zachodzi potrzeba utworzenia mapy z parami klucz/wartość, w których każdy z kluczy jest powiązany z kilkoma wartościami to należy użyć multimapy QMultiMap<K,T>.

ROZDZIAŁ 7

ITERATORY QT

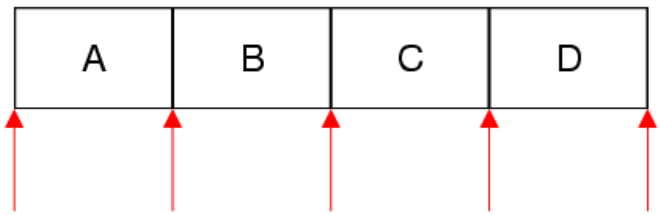
7.1. Wstęp

W bibliotece Qt 4 do obsługi kontenerów Qt opracowano nowy typ iteratora (dodatkowo do istniejących iteratorów w stylu STL) nazywany iteratorem w stylu Java (ang. *Java style iterators*). Ten typ iteratora jest standardem w obsłudze kontenerów Qt. Dla każdego kontenera mamy dwa typy iteratorów QT: jeden służący tylko do odczytu i jeden służący do odczytu i zapisu. W tabeli 1 pokazane są kontenery i opracowane dla nich iteratory.

Tabela 1. Iteratory kontenerów Qt

Kontener Qt	Iterator-Odczyt	Iterator-Odczyt/zapis
QList<T>, QQueue<T>	QListIterator<T>	QMutableListIterator<T>
QLinkedList<T>	QLinkedListIterator<T>	QMutableLinkedListIterator<T>
QVector<T>, QStack<T>	QVectorIterator<T>	QMutableVectorIterator<T>
QSet<T>	QSetIterator<T>	QMutableSetIterator<T>
QMap<Key, T>, QMultiMap<Key, T>	QMapIterator<Key, T>	QMutableMapIterator<Key, T>
QHash<Key, T>, QMultiHash<Key, T>	QHashIterator<Key, T>	QMutableHashIterator<Key, T>

Iteratory Qt w stylu Java działają całkiem inaczej niż iteratory STL, wskazują bowiem na miejsce pomiędzy dwoma elementami a nie na konkretny element. Na rysunku 7.1. naszkicowana jest zasada działania iteratorów Qt.



Rys.7.1. Pozycje, na które może wskazywać iterator w kontenerze z 4 elementami. Pozycje wskazują strzałki.

Typowe przeglądanie elementów w kontenerze QList aby je wydrukować na konsoli ma postać (na podstawie opisu menu Help Qt):

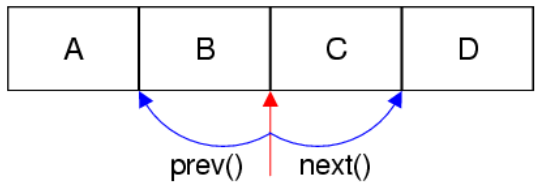
```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list);
while (i.hasNext())
qDebug() << i.next();
```

Po utworzeniu listy ***list***, jest ona przekazywana do konstruktora QListIterator. Iterator ***i*** wskazuje na pozycję przed pierwszym elementem. Następnie wywoływana jest funkcja hasNext(), której zadaniem jest sprawdzenie, czy za tą pozycja jest element. Jeżeli jest taki element to następnie wywoływana jest metoda next(), która powoduje przesuniecie iteratora do pozycji za tym elementem. Funkcja next() zwraca wartość elementu ,który przeskoczyła. Funkcja qDebug() powoduje wydruk wartości elementu. Przeglądanie listy może odbywać się w dwóch kierunkach. Takie zadanie może realizować następujący fragment programu:

```
QListIterator<QString> i(list);
i.toBack();
while (i.hasPrevious())
qDebug() << i.previous();
```

W tym fragmencie kodu, funkcja toBack() przesunie iterator na pozycję za ostatnim elementem kolekcji.

Funkcja hasPrevious() sprawdza czy istnieje element na pozycji przed iteratorem a nastepnie jest wywołana funkcja previous(). Rysunek 7.2 ilustruje działanie funkcji next() i previous().



Rys. 7.2. Szkic działania funkcji next() i previous()

W tabeli 2 pokazane są funkcje do obsługi iteratora Qt QListIterator.

Tabela 2. Funkcje iteratorów Qt

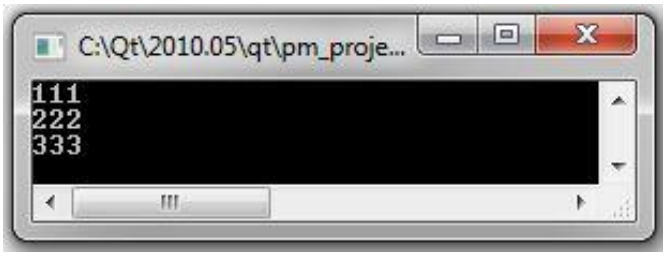
funkcja	opis
toFront()	Przesuwa iterator na początek listy
toBack()	Przesuwa iterator na koniec listy
hasNext()	Zwraca *true* jeżeli iterator nie jest na końcu listy
next()	Zwraca następny element i przesuwa iterator o jedną pozycję.
peekNext()	Zwraca następny element bez przesuwania iteratora
hasPrevious()	Zwraca *true* jeżeli iterator nie jest na początku listy
previous()	Zwraca poprzedzający element i przesuwa iterator z powrotem o jedną pozycję,
peekPrevious()	Zwraca poprzedni element bez przesuwania iteratora

Iteratory STL są bardzo dobre w obsłudze kontenerów Qt. W kolejnym programie wykorzystamy iteratory STL do obsługi listy Qt.

Wydruk 7.1 Kontener Qt, iteratory STL

```
#include <QtCore/QCoreApplication>
#include <QDebug>
int main(int argc, char *argv[])
{   QCoreApplication a(argc, argv);
    QList<int> lista;
    lista << 111 << 222 << 333;
    QList<int>:: iterator p = lista.begin();
    while(p!=lista.end())
    {qDebug()<< *p;
     ++p;
    }
    return a.exec();
}
```

Wynikiem program jest następujący wydruk:



W programie utworzona została lista zawierająca trzy elementy typu int. Następnie został zdefiniowany iterator ***p*** w stylu STL dla kontenera QList i ustawiony na pierwszy element listy. W pętli ***while*** korzystając z dereferencji wskaźnika (*p) spowodowaliśmy wyprowadzenie wartości elementów na standardowe wyjście (ekran monitora):

```
QList<int>:: iterator p = lista.begin();
while(p!=lista.end())
{qDebug()<< *p;
 ++p;
}
```

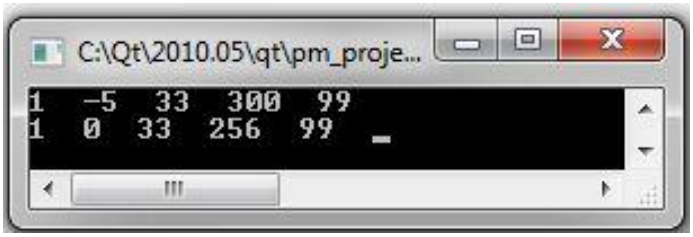
7.2. Iteratory mutujące

Iteratory Qt służące do wykonywania operacji związanych z odczytem i zapisem mają specyfikator **Mutable** w nazwie (np. QMutableListIterator<T>) . Tego typu iteratory umożliwiają wykonywanie operacji wstawiania, modyfikowania i usuwania elementów z kolekcji podczas jej przeglądania. Kolejny program ilustruje to zagadnienie. W programie tworzymy listę z różnymi elementami typu int, zadaniem programu jest wyselekcjonowanie wartości mniejszych niż 0 i zastąpieni ich wartością równą 0 oraz wyselekcjonowanie wartości większych niż 256 i zastąpienie ich wartością równą 256.

Wydruk 7.2 Kontener Qt, iteratory mutujące Qt

```
#include <QtCore/QCoreApplication>
#include<iostream>
using namespace std;
int main(int argc, char *argv[])
{   QCoreApplication a(argc, argv);
    QList<int> z1;
    z1 <<1<<-5<<33<<300<<99;
    QMutableListIterator<int> p(z1);
    while(p.hasNext())
       cout  << p.next() << "  ";
    cout << endl;
    p.toFront();
    while(p.hasNext())
    { if(p.next() > 256)
         p.setValue(256);
      if (p.peekPrevious()< 0)
          p.setValue(0);
    }
    p.toFront();
    while(p.hasNext())
       cout  << p.next() << "  ";
    return a.exec();
}
```

Wynikiem tego programu jest komunikat na ekranie monitora:



Na początku inicjalizujemy listę **z1** oraz deklarujemy mutujący iterator Qt o nazwie ***p***, który będzie obsługiwał listę **z1** przechowującą elementy typu int:

```
QMutableListIterator<int> p(z1);
```

W pętli while:

```
while(p.hasNext())
{ if(p.next() > 256)
     p.setValue(256);
  if(p.peekPrevious()< 0)
     p.setValue(0);
}
```

iterator kolejno przesuwa się od początku listy a za pomocą konstrukcji if sprawdzane są warunki. Jeżeli pierwszy warunek jest spełniony, funkcja setValue(256) wstawia żądaną wartość. W tym samym przebiegu pętli sprawdzamy też drugi warunek, musimy skorzystać z funkcji, peekPrevious() aby zbadać element bieżący, ponieważ funkcja next() przesunęła iterator do przodu. Jeżeli drugi warunek jest spełniony, funkcja setValue(0) wstawia żądaną wartość. Należy pamiętać, że iterator zawsze znajduje się na ostatniej pozycji. W celu wydrukowania wartości elementów listy możemy użyć funkcji previous() lub przesunąć iterator na początek listy:

```
p.toFront();
```

W celu usunięcie żądanego elementu możemy wykorzystać funkcję remove(), na przykład w następujący sposób :

```
QMutableListIterator<int> i(lista);
i.toBack();
while (i.hasPrevious()) {
if (i.previous() % 2 = 0)
    i.remove();
```

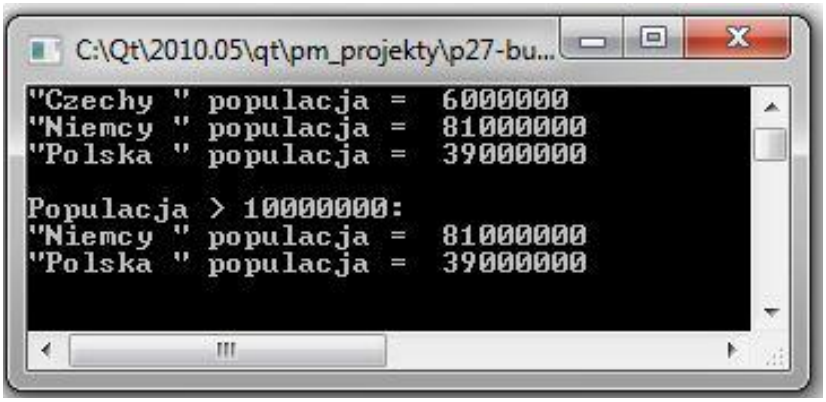
Iteratory kontenerów QVector, QSet, QLinkedList zachowują się tak samo jak iteratory QList. Zgodnie z oczekiwaniami, iteratory kontenera QMap będą zachowywały się inaczej ze względu na występowanie par ***klucz/wartość***. Iteratory QMap obsługiwane są także funkcjami toFront(), toBack(), hasNext(), next(), peekNext(), hasPrevious(), previous(), i peekPrevious(). Do obsługi klucza potrzebujemy funkcji key() a do obsługi wartości potrzebujemy funkcji value(). Te funkcje działają na obiektach zwróconych przez funkcje next(), peekNext(), previous() oraz peekPrevious().

Zagadnienie obsługi kontenera QMap przy pomocy mutujących iteratorów ilustruje kolejny przykład. W pokazanym programie tworzymy kontener przechowujący pary kraj/populacja. Zadaniem programu jest usuniecie z mapy kraju, którego populacja jest mniejsza niż zadana wartość. W naszym programie wartością graniczną jest populacja 10 milionów.

Wydruk 7.3 Kontener QMap, iteratory mutujące

```
#include <QtCore/QCoreApplication>
#include <QDebug>

int main(int argc, char *argv[])
{
 QCoreApplication a(argc, argv);
 QMap<QString,int> kraj;
 kraj["Polska "]= 39000000;
 kraj["Niemcy "]= 81000000;
 kraj["Czechy "]=  6000000;
 QMutableMapIterator<QString,int> i(kraj);
 while(i.hasNext())
  {i.next();
   qDebug() << i.key()<<"populacja = "<< i.value();
  }
 i.toFront();
 while(i.hasNext())
  {i.next();
   if(i.value()<10000000)
      i.remove();
  }
 qDebug()<<"\nPopulacja > 10000000:";
 i.toFront();
 while(i.hasNext())
  {i.next();
   qDebug() << i.key()<<"populacja = "<< i.value();
  }
 return a.exec();
}
```

Efektem wykonania programu jest następujący komunikat:



Mutujący iterator kontenera QMap zadeklarowany jest następująco:

```
QMutableMapIterator<QString,int> i(kraj);
```

Po przesunięciu iteratora na początek mapy i korzystając z pętli while sprawdzamy ustalony warunek. Gdy populacja badanego kraju jest mniejsza niż zadana wartość, kraj jest usuwany z naszej mapy:

```
i.toFront();
while(i.hasNext())
{ i.next();
  if(i.value()<10000000)
     i.remove();
}
```

Do wydrukowania nowej mapy wykorzystujemy funkcje key() i value():

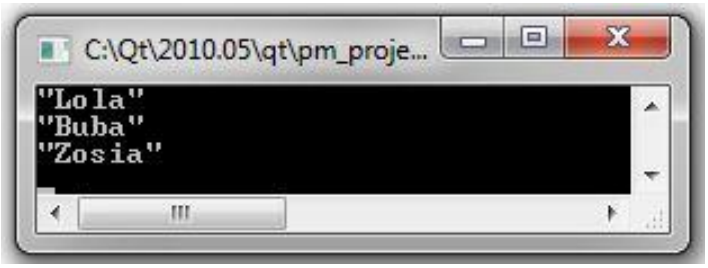
```
while(i.hasNext())
{
  i.next();
  qDebug() << i.key()<<"populacja = "<< i.value();
}
```

W Qt istnieje bardzo wygodna pętla do iterowania po elementach w kontenerach sekwencyjnych. Jest to pętla ***foreach***.
Krótki program ilustruje działanie tej pętli. Kod pętli foreach jest znacznie krótszy niż kod z użyciem iteratorów, pętla obsługuje wszystkie kontenery Qt, tego typu konstrukcja jest polecana, ze względu na wydajność..

Wydruk 7.4 Kontener QList, pętla foreach

```
#include <QtCore/QCoreApplication>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QList<QString> list;
    list <<"Lola"<<"Buba"<<"Zosia";
     QString str;
     foreach (str, list)
         qDebug() << str;

    return a.exec();
}
```

Po uruchomieniu programu otrzymujemy następujący komunikat:



ROZDZIAŁ 8

ALGORYTMY QT

8.1. Wstęp

W Qt możemy stosować klasyczne algorytmy STL (plik nagłówkowy <algorithm>) a także opracowane dla tej platformy własne algorytmy (umieszczone w pliku nagłówkowym <QtAlgorithms>). Sam producent Qt zauważa, że gdy nie ma specjalnej potrzeby rekomendowane jest używanie klasycznych algorytmów STL.

Tabela 1. Algorytmy Qt, b – begin, e – end, v – wartość, c- kontener,

Funkcja	Opis
qBinaryFind (b,e,v)	Wykonuje przeszukiwanie binarne
qBinaryFind (b,e,v,less)	Wykonuje przeszukiwanie binarne
qBinaryFind (c,v)	Wykonuje przeszukiwanie binarne
qCopy (b1,e1,b2)	Kopiuje elementy
qCopyBackward (b1,e1,e2)	Kopiuje elementy
qCount (b,e,v,n)	Zwraca liczbę elementów (n) o wartości v
qCount (c,v,n)	Zwraca liczbę elementów (n) o wartości v
qDeleteAll (b,e)	Usuwa elementy z zakresu [b,e]
qDeleteAll (c)	Usuwa elementy z kontenera
qEqual (b1,e1,b2)	Porównuje elementy z zakresów
qFill (b,e,v)	Wypełnia zakres wartościami
qFill (c,v)	Wypełnia kontener wartościami
qFind (b,e,v))	Znajduje pierwszą wartość v
qFind (c,v)	Znajduje pierwszą wartość v
qGreater ()	Zwraca funktor do qSort()
qLess ()	Zwraca funktor do qSort()
qLowerBound (b,e,v)	Szuka pierwszej pozycji wartości v
qLowerBound (b,e,v,less)	Szuka pierwszej pozycji wartości v
qLowerBound (c,v)	Szuka pierwszej pozycji wartości v
qSort (b,e) *end*)	Sortuje elementy w zakresie
qSort (b,e,less)	Sortuje elementy w zakresie
qSort (c)	Sortuje elementy w kontenerze
qStableSort (b,e)	Sortuje elementy w zakresie
qStableSort (b,e,less)	Sortuje elementy w zakresie
qStableSort (c)	Sortuje elementy w kontenerze
qSwap (v1,v2)	Wymienia elementy
qUpperBound (b,e,v)	Szuka pozycji wartości v
qUpperBound (b,e,v,less)	Szuka pozycji wartości v
qUpperBound (c,v)	Szuka pozycji wartości v

Algorytmy Qt są globalnymi funkcjami szablonowymi stosowanymi do obsługi kontenerów i elementów kontenerów, większość algorytmów stosuje iteratory w stylu STL. Biblioteka Qt, jak to widać w tabeli 1 dostarcza ograniczoną ilość algorytmów. W tabeli 1 przedstawione są wszystkie algorytmy Qt z krótkimi opisami. W celu poprawnego stosowania algorytmów Qt należy zapoznać się z dość dobrą dokumentacją techniczną zamieszczoną w menu pomocy ”Help”.
W menu pomocy zamieszczona jest dokumentacja techniczna oraz proste przykłady stosowania konkretnych algorytmów. Opisane algorytm mogą być wykorzystane we wszystkich typach kontenerów, dla których dostępne są iteratory w stylu STL, włącznie z takimi kontenerami jak: QList, QLinkedList, QVector, QMap, oraz QHash. Co ciekawsze, algorytmy generyczne mogą być wykorzystane poza kontenerami. Możliwe jest to, dlatego, że iteratory STL są modelowane podobnie jak wskaźniki w C++. W tej sytuacji możliwe jest wykorzystanie algorytmów STL i Qt z prostymi strukturami danych, takimi jak np. statyczne tablice.Do obsługi algorytmów potrzebujemy odpowiednich iteratorów.
Jak już pokazaliśmy, do dyspozycji mamy następujące typy iteratorów:

- Iteratory wejściowe (ang. input iterators)
- Iteratory wyjściowe (output iterators)
- Iteratory postępujące (forward iterators)
- Iteratory dwukierunkowe (bidirectional iterators)
- Iteratory dostępu swobodnego (ang. random access iterators)

Najczęściej używane są iteratory dostępu swobodnego. Na tych iteratorach możemy wykonać operacje pokazane w tabeli 2.

Tabela 2. Dozwolone operacje na iteratorach dostępu swobodnego

Operacja	opis
i += n	Przesuwa w przód iterator o n pozycji
i -= n	Przesuwa w tył iterator o n pozycji
i + n lub n + i	Zwraca iterator na pozycji i+n lub n+I (i – iterator, n – liczba elementów)
i - n	Zwraca iterator na pozycji i-n
i - j	Zwraca liczbę elementów pomiędzy iteratorem i oraz j
i[n]	To samo, co *(i + n)
i < j	Zwraca *true*, gdy iterator j jest za iteratorem i

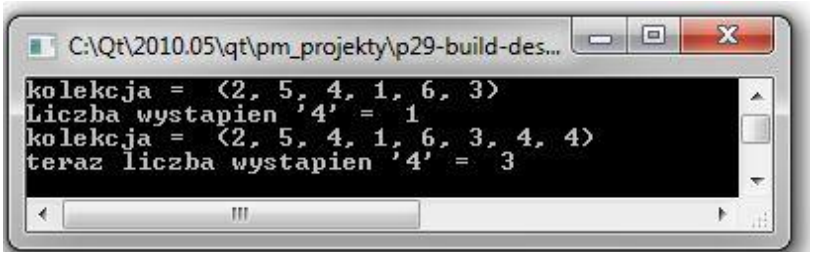
8.2. Przykłady wykorzystania algorytmów Qt

Stosowanie algorytmów Qt jest stosunkowo proste. Mają one jednak pewne specyficzne cechy, dlatego przed ich zastosowaniem najlepiej jest sprawdzić ich opis techniczny. Algorytm qCount() zwraca liczbę elementów o zadanej wartości ***value*** w kolekcji. Ilustruje to poniższy przykład.

Wydruk 8.1 Algorytm qCount()

```
#include <QtCore/QCoreApplication>
#include<QDebug>

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  QList<int> v;
  int cn = 0;
  v <<2<<5<<4<<1<<6<<3;
  qDebug() << "kolekcja = " <<v;
  qCount(v.begin(),v.end(),4,cn);
  qDebug() <<"Liczba wystapien '4' = " << cn;
  cn = 0;
  v<<4<<4;
  qDebug() << "kolekcja = " <<v;
  qCount(v.begin(),v.end(),4,cn);
  qDebug() <<"teraz liczba wystapien '4' = " << cn;
  return a.exec();
}
```

Po uruchomieniu programu otrzymujemy następujący komunikat:



Algorytm qCount() wystepuje w dwóch modyfikacjach:

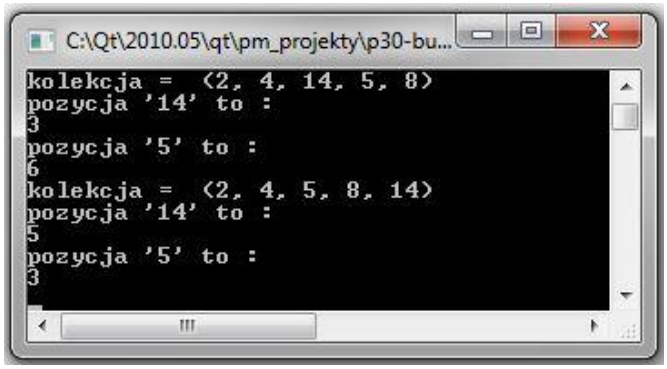
```
void qCount(InputIterator Begin, Input Iterator end,
            const T & value, Size & n)

void qCount(const Container & container,
            const T & value, Size & n)
```

Pierwszą postać algorytmu wykorzystujemy, gdy chcemy przeglądać fragment kolekcji w zakresie [begin, end], druga postać – gdy chcemy przegladać całą kolekcję. Należy pamiętać, aby zadeklarować i zainicjalizować wartość n.
W kolejnym programie ilustrujemy sposób wykorzystania algorytmu qBinaryFind().

Wydruk 8.2 Algorytm qBinaryFind(), qSort()

```
#include <QtCore/QCoreApplication>
#include<QDebug>
int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  QList<int> v;
  v << 2 << 4<< 14<< 5<< 8;
  qDebug() << "kolekcja = " << v;
  QList<int>::iterator p;
  p = qBinaryFind(v.begin(),v.end(), 14); //error
  qDebug() <<"pozycja '14' to :";
  qDebug() << p - v.begin() + 1;
  p = qBinaryFind(v.begin(),v.end(), 5);
  qDebug() <<"pozycja '5' to :";
  qDebug() << p - v.begin() + 1;
  qSort(v);
//poprawnie
  qDebug() << "kolekcja = " << v;
  p = qBinaryFind(v.begin(),v.end(), 14);
  qDebug() <<"pozycja '14' to :";
  qDebug() << p - v.begin() + 1;
  p = qBinaryFind(v.begin(),v.end(), 5);
  qDebug() <<"pozycja '5' to :";
  qDebug() << p - v.begin() + 1;
  return a.exec();
}
```

Wynik wykonania tego programu ma postać:



Funkcja qBinaryFind() ma trzy realizacje:

RandomAccessIterator qBinaryFind (RandomAccessIterator begin,
RandomAccessIterator end, const T & value)

RandomAccessIterator qBinaryFind (RandomAccessIterator begin,
RandomAccessIterator end, const T & value, LessThan lessThan)

Container :: const_iterator qBinaryFind (const Container & container,
const T & value)

Metodą przeszukiwania binarnego w przedziale [begin, end] algorytm szuka wyspecyfikowanej wartości ***value*** i zwraca pozycję znalezionej wartości. Gdy nie znajduje żadanej wartości, zwraca ***end***. Jest ważne, aby przeszukiwany zbiór był uporzadkowany wzrastająco (można to tego celu wykorzystać algorytm qSort()). Druga postać funkcji wykonuje wyszukiwanie w zbiorze posortowanym zgodnie z funktorem ***lessThan***.
W pokazanym przykladzie przy pomocy zdefiniowanego iteratora p szukamy wartośći '14' w nieuporządkowanym zbiorze:

```
QList<int>::iterator p;
p = qBinaryFind(v.begin(),v.end(), 14); //error
qDebug() <<"pozycja '14' to :";
qDebug() << p - v.begin() + 1;
```

Wynik jest prawidłowy. Kolejne szukanie wartości '5' daje błędny wynik. Jest to spowodowane faktem przeglądania nieuporządkowanego zbioru. Prawidłowy wynik uzyskamy, gdy uporządkujemy zbiór wartości algorytmem qSort().
Przy pomocy algorytmu qCopy() możemy kopiować wartości elementów jednego kontenera do innego kontenera. Algorytm przyjmuje trzy argumenty:

```
OutputIterator qCopy(InputIterator begin1,
                     InputIterator end1,
                     OutputIterator begin2)
```

Ten algorytm kopiuje element z zakresu [begin1, end1] do zakresu [begin2,…].
Bardzo użytecznym algorytmem jest algorytm qEqual(). Algorytm sprawdza czy elementy w wyspecyfikowanych zakresach są równe. Algorytm wymaga podania trzech argumentów:

```
bool qEqual (InputIterator1 begin1,
             InputIterator1 end1,
             InputIterator2 begin2)
```

Gdy elementy są równe zwraca wartość ***true***, w przeciwnym przypadku wartość ***false***. Kolejny program ilustruje wykorzystanie omówionych algorytmów.

Wydruk 8.3 Algorytm qCopy(), qEqual()

```
#include <QtCore/QCoreApplication>
#include<QStringList>
#include<QDebug>

int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  QStringList m1;
  m1 << "Bach"<< "Vivaldi" << "Wagner"<< "Mozart";
  QVector<QString> m2(4);
  qCopy(m1.begin(),m1.end(),m2.begin());
  qDebug()<< " " << m2;
  m2[3] = "Brahms";
  qDebug()<< " " << m2;
  bool wynik;
  wynik = qEqual(m1.begin(),m1.begin()+2,m2.begin());
  qDebug()<<"3 pierwsze rowne ? " << wynik;
  wynik = qEqual(m1.begin(),m1.end(),m2.begin());
  qDebug()<<"wszystkie rowne ?  " << wynik;

  return a.exec();
}
```

W programie tworzymu listę **m1** (dla typów QString) i inicjalizujemy ją nazwiskami znanych kompozytorów. Deklarujemy także wektor **m2** (także dla typow QString).
Przy pomocy algorytmu qCopy() elementy listy **m1** kopiujemy do wektora **m2**:

```
QStringList m1;
m1 << "Bach"<< "Vivaldi" << "Wagner"<< "Mozart";
QVector<QString> m2(4);
qCopy(m1.begin(),m1.end(),m2.begin());
```

Czwarty element wektora jest zmieniony:

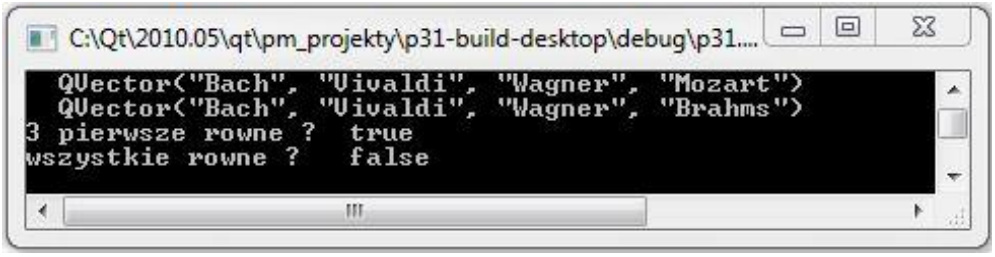
```
m2[3] = "Brahms";
```

W tym miejscu mamy nastepującą sytuację: w obu kontenerach trzy pierwsze pozycje są jednakowe.

Przy pomocy algorytmu qEqual() sprawdzamy elementy w obu kontenerach:

```
bool wynik;
wynik = qEqual(m1.begin(),m1.begin()+2,m2.begin());
qDebug()<<"3 pierwsze rowne ? " << wynik;
wynik = qEqual(m1.begin(),m1.end(),m2.begin());
qDebug()<<"wszystkie rowne ?  " << wynik;
```

Po uruchomieniu programu mamy nastepujący wynik:



ROZDZIAŁ 9

OBSŁUGA NAPISÓW W QT

9.1. Wstęp

Koncepcja obsługi napisów w języku C++ jest zapożyczona z języka C. W tym języku ciąg tekstowy (ang. *string*) jest tablicą znaków. Formalnie tablica znków deklarowana jest tak, jak każda inna tablica:

```
char napis[81];
```

W języku C++ mamy oczywiście obsługę napisów w postaci tablicy znakowej zakończonej znakiem null, ale nie jest to proste w wykorzystaniu. Problem polega na tym, że na napisach nie można prowadzic żadnych operacji wykorzystując standardowe operatory języka C++, same napisy nie mogą być elementami wyrażeń. Spowodowane jest to faktem, że napisy nie są odrębnym typem danych. Można wykonywać odpowiednie operacje jak np. konkatenacja, ale musimy użyć wyspecjalizowanych funkcji bibliotecznych (np. do konkatenacji funkcji strcat(s1,s2), a do kopiowania funkcji strcpy(s1,"Waclaw"). Tego typu obsługa jest podatna na błędy i oczywiście jest pracochłonna. W pokazanej deklaracji tablicy napis[], argument 81 oznacza, że tablica będzie przechowywała 81 znaków. Pamietamy, że do wykorzystania mamy tylko 80 znaków, ponieważ ostatni element tablicy znakowej przechowuje znak null. Gdy do tak zadeklarowanej tablicy będziemy chcieli wpisać wiecej znaków, dojdzie do przepełnienia, system tego faktu nie zasygnalizuje, zmiana statycznego rozmiaru tablicy znakowej nie jest możliwa. Można oczywiście zastosować tablice dynamiczne:

```
char *napis = New char[rozmiar];
```

Dzięki takiej deklaracji powstaje dynamiczna tablica znakowa, może ona być inicjalizowana do rozmiaru ustalonego przez argument *rozmiar* w trakcie działania programu. Gdy jednak zajdzie potrzeba zmienienia rozmiaru tablicy znakowej, należy wykonać czynności związane z dynamiczną alokacją pamięci – należy najpierw zwolnić pamięć (aby zpobiec wyciekom pamięci) a potem ponownie przydzielic nową pamięć dynamiczną.
Niedogodności obsługi napisów w języku C i C++ były zauważone dość wcześnie. Po wielu dyskusjach i ustaleniech do standardu języka C++ włączono nową klasę – klasę ciagu tekstowego. Klasa ciągu tekstowego w bibliotece STL (std::string) ma wiele zalet:

- Operownie i manipulowanie napisami jest proste
- Programista popełnia mniej błędów
- Praktycznie programista ma wrażenie, że dysponuje nowym typem string

Oczywiste zalety stosowania klasy ***string*** zostały docenione przez twórców pakietu Qt. Kompilator języka C++ oczywiście ma do dyspozycji klasę

std::string ze standardu STL C++, ale dodatkowo dla pakietu Qt opracowana została klasa ***QString.*** Zakres funkcjonalności klasy QString jest znacznie obszerniejszy od możliwości klasy std::string biblioteki STL C++. Należy pamiętać, że klasa QString przechowuje napisy w standardzie Unicode 4.0.
W pakiecie Qt mamy możliwość korzystania zarówno z klasy std:string jak i z klasy QString.

9.2. Klasa string

Klasa std:: string jest specjalizacją klasy *basic_string*. W STL występują dwie modyfikacje – klasa std::string obsługuje łańcuchy znaków 8-bitowych, klasa std:wstring obsługuje znaki rozszerzone. Omówimy w niniejszym skrypcie tylko klasę std::string. Korzystanie z klasy std:string wymaga dołączenia pliku nagłówkowego <string>. Jak w każdej klasie, w klasie string wystepuje duży zestaw konstruktorów. Najczęściej stosowane to:

- string()
- string(const char* str);
- string(const string &str);

W klasie string zdefiniowano operatory działające na obiektach (tabela 9.1).

Tabela 9.1. Operatory klasy string

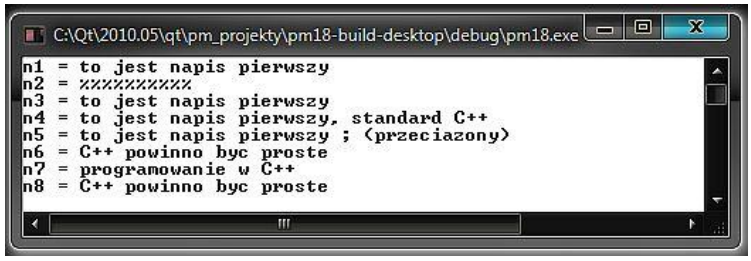
nr	operator	opis
1	=	Przypisuje nową wartość
2	+	Konkatenacja, dwa łańcuchy znakowe
3	+=	Dołącza do łańcucha znaki
4	==	Równe
5	!=	Nierówne
6	<	Mniejsze
7	<=	Mniejsze lub równe
8	>	Większe
9	>=	Większe lub równe
10	[]	Indeks, umożliwia dostęp do znaku
11	<<	Wyjście
12	>>	Wejście

Przy pomocy klasy std::string możemy łatwo obsługiwać napisy.
W kolejnym przykładzie pokażemy inicjowanie łańcuchów znakowych oraz wybrane operacje na napisach.

Wydruk 9.1 Konstruktory klasy string

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <string>
int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  using namespace std;
/*  niedozwolona obsluga napisow w C++
  char err1[81],err2[81], err3[81];
  err1 = "to";
  err2 = "jest niedozwolone";
  err3 = err1+err2;
  cout << err3<<endl;
*/
  char c = '%';
  string n1("to jest napis pierwszy");
  cout <<"n1 = "<< n1 << endl;
  string n2(10,c);
  cout <<"n2 = "<< n2 << endl;
  string n3(n1);
  cout <<"n3 = "<< n3 << endl;
  string n4(", standard C++");
  n4 = n1+n4;
  cout <<"n4 = "<< n4<< endl;
  n1 += " ; (przeciazony)";
  cout <<"n5 = "<< n1 << endl;
  char napis[]="programowanie w C++ powinno byc proste";
  string n6(napis+16, napis +38);
  cout <<"n6 = " << n6<<endl;
  string n7(napis,19);
  cout <<"n7 = " << n7<<endl;
  string n8(&napis[16], &napis[38]);
  cout <<"n8 = " << n8<<endl;
  return a.exec();
}
```

Po uruchomieniu programy otrzymujemy nastepujacy wynik:



W pokazanym programie znajduje się fragment kodu umieszczony w komentarzu:

```
/*  niedozwolona obsluga napisow w C++
  char err1[81],err2[81], err3[81];
  err1 = "to";
  err2 = "jest niedozwolone";
  err3 = err1+err2;
  cout << err3<<endl;
*/
```

Po usunięciu komentarzy i uruchomieniu programu otrzymujemy z kompilatora komunikat o błędach:

```
11    err1 = "to";
incompatible types in assignment of ‘const char[3]’ to ‘char[81]’
```

Jak to juz omawialiśmy, nie można stosować operatora przypisania do nadania nowej wartości tablicy znakowej, tak jak to pokazano w tym fragmencie kodu. Omowimy bardziej szczegółowo zastosowane w programie konstruktory.
Prosta inicjalizacja obiektu **n1** ma postać:

```
string n1("to jest napis pierwszy");
```

Obiekt klasy string można wypełnić sekwencją znaków:

```
char c = '%';
string n2(10,c);
```

W pokazanym przykladzie łańcuch **n2** bedzie skladał się z 10 znaków **%**.
Konstruktor kopiujący

```
string n3(n1);
```

w łańcuchu **n3** umieści łańcuch **n1**.
Łańcuch **n4**

```
string n4(", standard C++");
```

ponownie wykorzystuje prostą inicjalizację.
W instrukcji

```
n4 = n1+n4;
```

przy pomocy operatora + wykonujemy połączenie dwóch napisów **n4** i **n1** a wynik umieszczamy w łańcuchu **n4,** w wyniku mamy napis:

```
to jest napis pierwszy, standard C++
```

W instrukcji:

```
n1 += " ; (przeciazony)";
```

wykorzystujemy przeciążony operator **+=** , aby dołączyć nowy napis do napisu **n1** w wyniku czego mamy nowy łańcuch:

```
to jest napis pierwszy ; przeciazany
```

W instrukcjach:

```
char napis[]="programowanie w C++ powinno byc proste";
string n6(napis+16, napis +38);
```

tworzymy łańcuch **napis** w stylu języka C, a następnie konstruktor tworzy z niego łańcuch **n6** w wybranym zakresie, w wyniku czego **n6** ma postać:

```
C++ powinno byc proste
```

Kolejny konstruktor:

```
string n7(napis,19);
```

tworzy łańcuch **n7** z łańcucha **napis**, ale kopiuje tylko pierwszych 19 znaków.
Ostatni łańcuch **n8** tworzony jest podobnie jak łańcuch **n6**.
Klasa std::string posiada szereg metod służących do operowania na łańcuchach znakowych. Wybrane metody pokazane są w tabeli 9.2.

Tabela 9.2. Wybrane metody klasy string

nr	metoda	opis
1	append()	Dołącza do łańcucha znaki
2	assign()	Przypisuje nowa wartość
3	at()	Dostęp do znaku
4	compare()	Porównuje dwa łańcuchy
5	clear()	Usuwa znaki
6	empty()	Sprawdza, czy łańcuch jest pusty
7	erase()	Usuwa znaki
8	insert()	Wstawia znaki
9	length()	Zwraca liczbę znaków
10	pushback()	Dołącza znaki do łańcucha
11	replace()	Zastępuje znaki
12	resize()	Zmienia liczbę znaków
13	size()	Zwraca liczbę znaków
14	substr()	Zwraca fragment łańcucha
15	swap()	Wymienia zawartości łańcuchów

Klasa std::string dysponuje funkcjami służącymi do odnajdywania konkretnych znaków lub sekwencji znaków w podanym łańcuchu. Lista tych funkcji pokazana jest w tabeli 9.3.

Tabela 9.3. Wybrane metody wyszukiwania znaków w obiektach klasy string

Nr	metoda	Opis
1	find()	Poszukuje pierwszego wystąpienia znaku
2	rfind()	Poszukuje ostatniego wystąpienia znaku
3	find_first_of()	Poszukuje pierwszego znaku z podanej wartości
4	find_last_of()	Poszukuje ostatniego znaku z podanej wartości
5	find_first_not_of()	Poszukuje pierwszego znaku nie bedącego w podanej wartości
6	find_last_not_of()	Poszukuje ostatniego znaku nie bedącego w podanej wartości

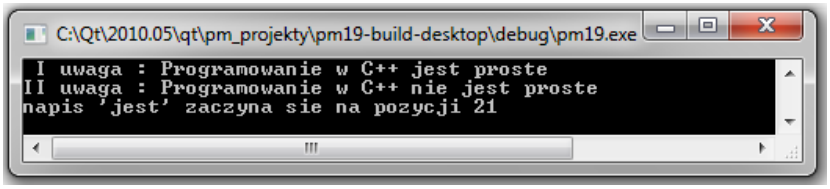
Kolejny program ilustruje wyszukiwanie sekwencji znaków w napisie. Funkcja find() poszukuje w ustalonym łańcuchu sekwencji znaków. Gdy sekwencja zostanie znaleziona, funkcja zwraca pozycję (indeks) szukanej sekwencji w badanym łańcuchu.

Wydruk 9.2 Metody klasy string, insert(), find()

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <string>
int main(int argc, char *argv[])
{
 QCoreApplication a(argc, argv);
 using namespace std;
 int n;
 string n1(" Programowanie w C++ jest proste");
 cout << " I uwaga :"<< n1<<endl;
 string n2 ("nie ");
 n = n1.find("jest");
 n1.insert(n,n2);
 cout <<"II uwaga :" << n1 <<endl;
 cout << "napis 'jest' zaczyna sie na pozycji "<<n<<endl;
 return a.exec();
}
```

Metoda insert() umożliwia dodawanie znaków do wyspecyfikowanego łańcucha. Jedna z modyfikacji tef metody wymaga dwóch argumentów: pozycji na której ma być wstawiona sekwencja oraz samej sekwencji znaków.

Po uruchomieniu pokazanego programu otrzymujemy wynik:



W linii:

```
string n1(" Programowanie w C++ jest proste");
```

inicjalizujemy badany łańcuch **n1**. W instrukcji:

```
n = n1.find("jest");
```

zlecamy wyszukanie ciągu "jest" w łańcuchu **n1**, w przypadku sukcesu zostanie zwrócony indeks pierwszego wystąpienia tej sekwencji (w naszym przypadku badana sekwencja jest na pozycji 21).
Kolejno w instrukcjach:

```
string n2 ("nie ");
n = n1.find("jest");
n1.insert(n,n2);
```

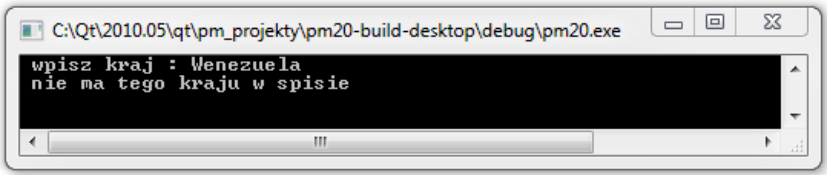
tworzymy łańcuch **n2** (łańcuch „nie"), znadujemy położenie sekwencji "jest", tuż przed nią przy pomocy metody insert() wstawiamy sekwencję "nie".
W zasadzie klasa std:string jest kontenerem. Należy pamiętać, że biblioteka STL została znakomicie opracowana, nie powinno nas dziwić, że mamy możliwość przechowywania obiektów jednego kontenera w innych kontenerach.
Kolejny przykład ilustruje umieszczanie obiektów typu string w kontenerze map. W programie tworzymy mapę Kraj – Stolica.

Wydruk 9.3 Metody klasy string, kontener map

```
#include <QtCore/QCoreApplication>
#include <iostream>
#include <map>
#include <string>
int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  using namespace std;
  map<string, string> info;
  map<string, string> :: iterator itr;
  info.insert(pair<string,string> ("Polska","Warszawa"));
```

```
    info.insert(pair<string,string> ("Niemcy","Berlin"));
    info.insert(pair<string,string> ("Rosja","Moskwa"));
    info.insert(pair<string,string> ("Litwa","Wilno"));
    info.insert(pair<string,string> ("Anglia","Londyn"));
    string kraj;
    cout << " wpisz kraj : ";
    cin >> kraj;
    itr = info.find(kraj);
    if (itr != info.end())
         cout << "stolica : " << itr->second;
    else
         cout << " nie ma tego kraju w spisie"<<endl;
    return a.exec();
}
```

Wynik działania pokazanego programu ma postać:



W omawianym programie tworzymy kontener map i deklarujemy iterator:

```
    map<string, string> info;
    map<string, string> :: iterator itr;
```

przy pomocy struktury pair< >):

```
    info.insert(pair<string,string> ("Niemcy","Berlin"));
```

umieszczamy w kontenerze map pary klucz-wartość.
Po wprowadzeniu z klawiatury nazwy kraju metoda find() przeszukuje mapę i albo znajdzie wyspecyfikowany kraj i wtedy program wyświetli jego stolicę, albo szukanego kraju nie będzie na naszej liści i wtedy pojawi się odpowiedni komunikat:

```
    itr = info.find(kraj);
    if (itr != info.end())
         cout << "stolica : " << itr->second;
    else
         cout << " nie ma tego kraju w spisie"<<endl;
```

Jak widać, po wprowadzeniu z klawiatury nazwy „Wenezuela” pojawia się komunikat, że nie znaleziono tego kraju.

9.3. Klasa QString

Do obsługi napisów Qt dostarcza dwóch wyspecjalizowanych klas – klasę QString oraz klasę QStringList. Klasa QString zbudowana jest podobnie do klasy std::string z biblioteki standardowej języka C++, ale zakres funkcjonalności ma znacznie większy. Przede wszystkim klasa QString dzięki przechowywaniu napisów w standardzie Unicode 4.0 może stosunkowo prosto obsługiwać języki narodowe. W Qt są narzędzia do konwersji obiektów QString do obiektów std::string a także do napisów znakowych w stylu języka C.
Typowe przykłady takiej konwersji:

```
QString st1 = "napis";
std::string cppstr = st1.toStdString();
const char *cstr = st1.toStdString.c_str();
```

W bibliotece Qt mamy oczywiście możliwość konwersji napisów w stylu języka C do napisów w stylu Qt:

```
const char *cstring = "napis";
QString qtstr(cstring);
```

Do obsługi napisów można także wykorzystac jeszcze jedną klasę – klasę QByteArray. Obiekty tej klasy przechowyja tradycyjne ośmiobitowe napisy zakończone znakiem końca napisu - **\0**. Według producentów Qt, stosowanie klasy QByteArray jest bardziej ekonomiczne niż wykorzystywanie zmiennych typu ***cons char*,*** tak często używanych w jezyku C.

Klasa QStringList dziedziczy z klasy QList<QString>, jest rekomendowana, gdy korzystamy z kontenera QList, ponieważ jest zoptymalizowana do obsługi napisów w tym kontenerze. Klasa QStringList korzysta z takich samych metod i innych funkcjonalności jak klasa QList.

W praktyce zaleca się używać obiektów klasy QString do obsługi napisów, argumentując, że konstruowane przy pomocy Qt API wykorzystują wyłącznie klasę QString.

Klasa QString posiada dziewięć konstruktorów i jeden destruktor (w wersji Qt 4.7). Najczęściej wykorzystywane konstruktory klasy QString to:

- QString()
- QString(contr char* str)
- QString(int size,QChar ch)
- QString(const QChar* unicode)

Klasa QString ma zdefiniowane operatory działajace na obiektach tej klasy. Lista operatorów zawiera takie same elementy jak to jest przedstawione w tabeli 9.1.

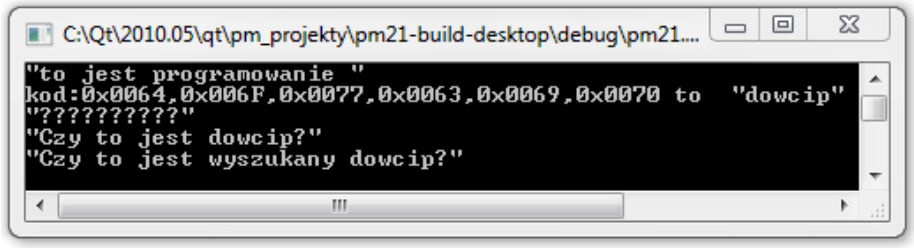
Techiniki inicjalizowania i manipulowania obiektami klasy QString zostały

przedstawione w kolejnym programie.

Wydruk 9.4 Obiekty klasy QString, wybrane metody

```
#include <QtCore/QCoreApplication>
#include<QString>
#include <QDebug>
int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  QString s1("to ");
  QString s2 = "jest ";
  QString s3 = s1+s2;
  const char *p = "programowanie ";
  QString s4(p);
  s3 += s4;
  qDebug() << s3;

  static const QChar data[6] =
     {0x0064,0x006F,0x0077,0x0063,0x0069,0x0070};
  QString s5(data, 6);

  qDebug()<<"kod:0x0064,0x006F,0x0077,0x0063,0x0069,
                                    0x0070 to "<<s5;
  QString s6(10,'?');
  qDebug()<<s6;

  s1.prepend("Czy ");
  s1.append(s2+s5+'?');
  qDebug()<<s1;

  s1.insert(12,"wyszukany ");
  qDebug()<<s1;

  return a.exec();
}
```

Wynikiem działana pokazanego programu jest komunikat:



W programie pokazano proste sposoby ustanawiania obiektu QString i jego inicjalizacji.
W liniach:

```
QString s1("to ");
QString s2 = "jest ";
```

wykorzystano konstruktor klasy QString oraz proste przypisanie. Zostały zainicjalizowane zmienne s1 i s2, które są ciągami tekstowymi.
Jak można przeczytać w technicznym opisie klasy QString (menu Help systemu Qt) prostą drogą do inicjalizacji obiektu QString jest przesłanie ***const char ** do konstruktora tej klasy. W naszym przykładzie powstal obiekt **s1** o rozmiarze 3 zawierający dane "**to** ". QString konwertuje dane ***const char ** do danych typu Unicode przy pomocy funkcji ***fromAscii()***. Opcjonalnie funkcja ***fromAscii()*** traktuje znaki powyżej 128 jako znaki Latin-1, ale to może być zmienione przez wywolanie ***QTextCodec::setCodecForCStrings().*** Wszystkie funkcje QString, które pobieraja argument w postaci ***const char ** , ten argument jest interpretowany jako klasyczny łańcuch w stylu języka C zakończony znakiem '\0'.
W linii

```
QString s3 = s1+s2;
```

wykorzystano przeciążony operator + do wykonania operacji łączenia dwóch ciagów tekstowych s1 i s2 w wyniku czego zainicjalizowana została zmienna s3.
W liniach

```
const char *p = "programowanie ";
QString s4(p);
```

został zainicjalizowany ciąg tekstowy znaków o nazwie ***p*** w stylu języka C, a przy pomocy konstruktora klasy QString utworzono kopię s4.
W linii

```
s3 += s4;
```

wykorzystano operator dołączania łańcucha s4 do łańcucha s3, pamiętamy, ze powyższa operacja może być zapisana w równoważnej postaci

```
s3 = s3 + s4;
```

Możemy także dane do łańcucha QString przekazać w postaci tablicy QChar:

```
static const QChar data[6] =
       {0x0064,0x006F,0x0077,0x0063,0x0069,0x0070};
QString s5(data, 6);
```

Mamy także mozliwość inicjalizacji obiektu klasy QString wskazanym znakiem oraz podana ilością powtórzeń tego znaku:

```
QString s6(10,'?');
```

Powstanie łańcuch zbudowany z dziesięciu znaków zapytania.
Klasa QString jest bardzo rozbudowana (znacznie bardziej niż klasa std::string). Wybrane, użyteczne metody tej klasy pokazane są w tabeli 9.4

Tabela 9.4. Wybrane metody klasy QString

nr	metoda	opis
1	append()	Dołącza do łańcucha znaki
2	at()	Pobiera znak w wybranej pozycji
3	capacity()	Zwraca liczbę znaków, jakie może zawierać łańcuch bez ponownego przydziału pamięci
4	compare()	Porównuje dwa łańcuchy
5	chop()	Usuwa n znaków z końca łańcucha
6	clear()	Usuwa znaki
7	contains()	Zwraca true, gdy łańcuch s1 zawiera s2
8	endsWith()	Sprawdza, czy łańcuch kończy się określonym znakiem
9	fill()	Tworzy napis określonym znakiem
10	insert()	Wstawia znaki
11	isEmpty()	Zwraca true gdy napis nie ma znaków
12	isNull	Zwraca true gdzy napis jest pusty
13	left()	Zwraca n znaków z lewej strony łańcucha
14	length()	Zwraca liczbę znaków
15	mid()	Zwraca n znaków ze środka łańcucha
16	prepend()	Dołącza łańcuch s1 do s2 (na początku s2)
17	push_back()	Dołącza łańcuch s1 do s2 (na końcu s2)
18	push_front()	Dołącza łańcuch s1 do s2 (na początku s2)
19	remove()	Usuwa n znaków z łańcucha
20	replace()	Zastępuje znaki
21	resize()	Zmienia liczbę znaków
22	size()	Zwraca liczbę znaków
23	split()	Dzieli łańcuch na odrębne łańcuchy
24	startWith()	Zwraca true, gdy łańcuch zaczyna się określonym znakiem
25	truncate()	Obcina łańcuch od zadanej pozycji

Porównywanie łańcuchów jest jest ważnym zadaniem w procesie przetwarzania tekstu. W klasie QString występuje metoda ***compare(),*** dzieki ktorej możemy

porównywać łańcuchy.
Prototyp tej funkcji ma postać:

```
int QString::compare (const QString &s1,const QString &s2,
                      Qt::CaseSensitivity cs) [static]
```

Funkcja ***compare()*** porównuje łańcuchy **s1** i **s2,** zwraca wartość całkowitą (typu **int**) w zależności od porownania:

- gdy s1 < s2 zwraca wartość < 0
- gdy s1 = s2 zwraca wartość 0
- gdy s1 > s2 zwraca wartość > 0

Przeciążona funkcja compare() ma postać:

```
int QString::compare (const QString &s1,const QString &s2)
```

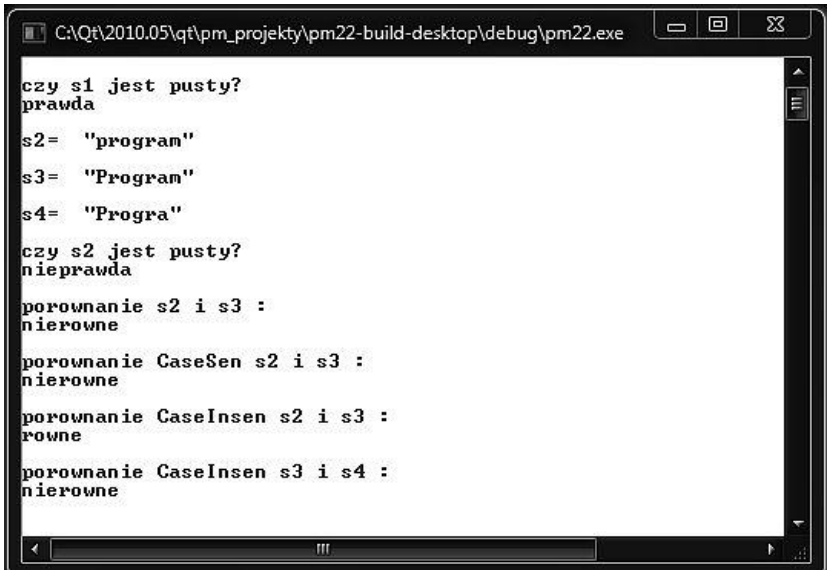
i wykonuje proste prównanie łańcuchów **s1** i **s1**.
Funkcja compare() klasy QString wystepuje w dziewięciu różnych wersjach, w celu wykorzystania najlepszej realizacji należy zapoznać się z jej opisem technicznym.
Przykład przedstawiony na kolejnym listingu pokazuje, jak można wykorzystac funkcje compare().

Wydruk 9.5 Obiekty klasy QString, metoda compare()

```
#include <QtCore/QCoreApplication>
#include<QString>
#include <QDebug>

int main(int argc, char *argv[])
{   QCoreApplication a(argc, argv);
    QString s1("");
    qDebug()<<"s1= "<<s1<<endl;
    qDebug() << "czy s1 jest pusty? ";
    if(s1.isEmpty())
       qDebug() << "prawda";
    else
       qDebug()<<"nieprawda";
    QString s2 = "program";
    QString s3 = "Program";
    QString s4 = "Progra";
    qDebug()<<"\ns2= "<<s2<<endl;
    qDebug()<<"s3= "<<s3<<endl;
    qDebug()<<"s4= "<<s4<<endl;
    qDebug() << "czy s2 jest pusty? ";
    if(s2.isEmpty())
```

```
        qDebug() << "prawda";
      else
        qDebug()<<"nieprawda";
      qDebug()<<"\nporownanie s2 i s3 : ";
      if(QString::compare(s2,s3))
        qDebug() << "nierowne";
      else
        qDebug()<<"rowne";
      qDebug()<<"\nporownanie CaseSen s2 i s3 : ";
      if(QString::compare(s2,s3,Qt::CaseSensitive))
        qDebug() << "nierowne";
      else
        qDebug()<<"rowne";
      qDebug()<<"\nporownanie CaseInsen s2 i s3 : ";
      if(QString::compare(s2,s3,Qt::CaseInsensitive))
        qDebug() << "nierowne";
      else
        qDebug()<<"rowne";
     qDebug()<<"\nporownanie CaseInsen s3 i s4 : ";
     if(QString::compare(s3,s4,Qt::CaseInsensitive))
        qDebug() << "nierowne";
     else
        qDebug()<<"rowne";
     return a.exec();
}
```

Wynikiem uruchomienia programu jest komunikat:



W sprawdzaniu łańcuchów pomocną jest metoda isEmpty(). Klasa QString ma dwie metody sprawdzające, czy łańcuch jest pusty:

```
bool QString::isEmpty() const
bool QString::isNull() const
```

Te dwie metody zwracają wartośc ***true***, gdy łańcuch jest pusty, w przeciwnym przypadku zwraca wartość ***false***. Jak podają producenci, opracowano dwie metody isEmpty() i isNull() z powodów historycznych. Zalecają używanie metody isEmpty(). Tak też mamy w naszym przykładzie:

```
QString s1("");
qDebug() << "czy s1 jest pusty? ";
if(s1.isEmpty())
   qDebug() << "prawda";
else
   qDebug()<<"nieprawda";
```

Ponieważ łańcuch **s1** nie zawiera znaków, program zwróci wartość true.
Często chcemy sprawdzić czy łańcuchy zawieraja ten sam napis, bez względu na wielkość liter. W funkcji compare() należy wprowadzić trzeci parametr QT::CaseInsensitive:

```
QString s2 = "program";
QString s3 = "Program";
qDebug()<<"\nporownanie CaseInsen s2 i s3 : ";
if(QString::compare(s2,s3,Qt::CaseInsensitive))
  qDebug() << "nierowne";
else
  qDebug()<<"rowne";
```

Funkcja zwróci wartość true.

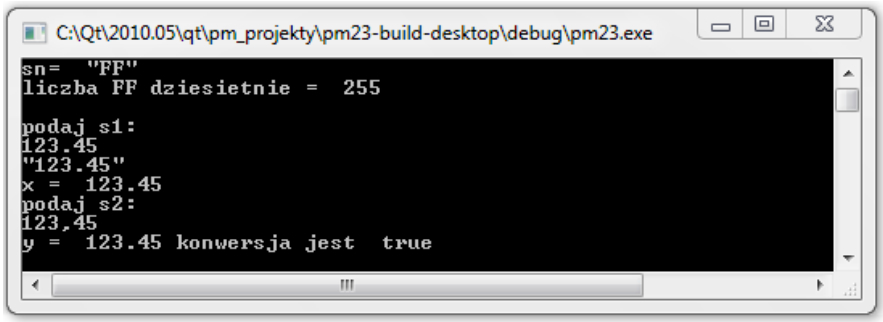
Qt posiada użyteczną klasę QTextStream, która elegancko obsługuje czytanie i zapisywanie tekstu. Ta klasa operuje na obiektach innych klas: QIODevice, QByteArray oraz QString. Dzieki tej klasie możemy prosto czytać słowa, linie danych wejściowych oraz liczby. Zaleca się, aby używać klasy QTextStream do czytania danych wejściowych z konsoli oraz do zapisywania danych wyjściowych na ekranie (ogólnie na konsoli). Klasa QTextStream automatycznie przyjmie dane ze standardowego wejścia przy użyciu standardowego kodeka. Wprowadzanie danych tekstowych obsługiwanych przez klasę QTextStream wykorzystamy w kolejnym programie, ilustrującym konwersje łańcucha do konkretnego formatu. W podręcznikach programowania zaleca się stosowanie jednego typu danych wejściowych. Oczywisci potem musimy dokonać odpowiednich konwersji. Często wygodnie jest stosować wprowadzanie danych wejściowych w postaci łańcuchów, a potem konwertować odpowiednią daną

tekstową do wartości całkowitej lub na przykład do wartości zmiennoprzecinkowej (float lub double). W klasie QString mamy odpowiednie metody do obsługi wymienionych zadań..

Wydruk 9.6 Obiekty klasy QString, konwersje

```
#include <QtCore/QCoreApplication>
#include<QTextStream>
#include<QString>
#include <QDebug>
int main(int argc, char *argv[])
{
  QCoreApplication a(argc, argv);
  QString s1,s2;
  bool ok;

  QString sn = "FF";
  qDebug()<<"sn= "<<sn;
  int liczba = sn.toInt(&ok,16);
  qDebug()<<"liczba FF dziesietnie = " << liczba;

  QTextStream in(stdin);
  qDebug()<<"\npodaj s1: ";
  in>>s1;
  qDebug()<<s1;
  double x = s1.toDouble();
  qDebug()<<"x = "<< x;

  qDebug()<<"podaj s2: ";
  in>>s2;
  double y = s2.toDouble(&ok);
  qDebug()<<"y = "<< y << "konwersja jest "<<ok;

  return a.exec();
}
```

Wynikiem pokazanego programu jest nastepujący wydruk ekranu:



W następującym fragmencie programu:

```
bool ok;
QString sn = "FF";
qDebug()<<"sn= "<<sn;
int liczba = sn.toInt(&ok,16);
qDebug()<<"liczba FF dziesietnie = " << liczba;
```

łańcuch "FF" chcemy potraktować jako wartość zapisaną w systemie szesnastkowym. Odpowiednia metoda klasy QString, funkcja ***toInt()*** dokona poprawnej kowersji. Prototyp tej funkcji ma postać:

int QString::toInt(bool *ok = 0, int base = 10) const

Metoda konwertuje łańcuch do wartości typu ***int*** wykorzystując odpowiednią bazę systemu liczbowego. Przez domniemanie podstawa jest równa 10. W naszym przypadku, ponieważ obsługujemy liczbę szesnastkową, baza musi być równa 16. Gdy konwersja się nie uda (na przykład podana jest błędna baza) to metoda zwraca wartość 0 (false). Baza musi być w zakresie (2, 36) lub 0. Gdy baza jest równa zeru to przyjmowana jest konwencja języka C, tzn. gdy łańcuch zaczyn się od napisu "0x", do konwersji użyta jest baza 16.
W kolejnym fragmencie programu:

```
QTextStream in(stdin);
qDebug()<<"\npodaj s1: ";
in>>s1;
qDebug()<<s1;
double x = s1.toDouble();
qDebug()<<"x = "<< x;
```

dane wejściowe, konsolowe, obsługiwane będą przez klasę QTextStream. Obiekt ***in*** będzie pobierał dane ze standardowego wejścia (***stdin***). Tekst konwertowany będzie do liczby zmiennoprzecinkowej typu double. Konwersję wykonuje funkcja, której prototyp ma postać:

double QString::to double(bool *ok. = 0) const

Funkcja zwraca wartość false, gdy konwersja się nie powiedzie, w przypadku powodzenia zwraca wartość true. Liczby zmiennoprzecinkowe mogą być pisane z kropką lub z przecinkiem (kraje anglosaskie zapisują wartości stosując przecinek, inne państwa, na przykład Niemcy stosują kropkę). Przez domniemanie, metoda ***toDouble*** przyjmuje typ lokalny oznaczony, jako C. Oznacza to, że łańcuch przedstawiający liczbę może być z kropką lub z przecinkiem – zawsze będzie poprawnie konwertowany do liczby zmiennoprzecinkowej, w systemach komputerowych zapisywanych z kropką.

Jak widać na zrzucie ekranu, łańcuch **s1** jest zapisany w postaci cyfr z kropką. Konwersja jest prawidłowe. W drugim podejściu łańcuch **s2** jest zapisany z przecinkiem. Konwersja także przebiegła prawidłowo. Oczywiście, możemy mieć zapis z przecinkiem, co w Niemczech nie jest prawidłowe. W takiej sytuacji możemy wymusić lokalne podejście do interpretacji liczb zmiennoprzecinkowych. Należy wykorzystać ustawienia lokalne, tak jak to jest przedstawione w systemie pomocy Qt:

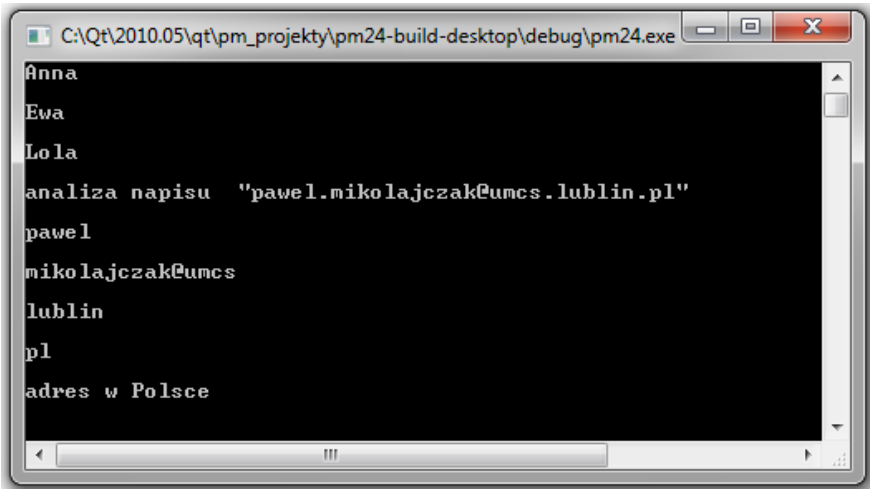
```
QLocale::SetDefault(QLocale::German);
d = QString("1234,56").toDouble(&ok);  //ok==true,d==1234.56
d = QString("1234.56").toDouble(&ok);  //ok==true,d==1234.56
```

W Qt mamy wygodną klasę QStringList do obsługi listy łańcuchów. Klasa QStringList dziedziczy z klasy QList<QString>, co oznacza, że metody klasy QList możemy stosować w klasie QStringList. Oczywiście klasa QStringList posiada także wyspecjalizowane metody, obsługujące tylko jej obiekty. Zalecene jest stosowanie tej klasy, gdy obsługujemy listę łańcuchów.
Kolejny program ilustruje wykorzystanie obiektów klasy QStringList. W programie mamy jeden łańcuch. Metoda split() pozwala na dzielenie tekstu na określone podłańcuchy.

Wydruk 9.7 Obiekty klasy QStringList, split()

```
#include <QtCore/QCoreApplication>
#include<QStringList>
#include<QString>
#include <QDebug>

int main(int argc, char *argv[])
{QCoreApplication a(argc, argv);
 QString s1 = "Anna,Ewa,Lola";
 QStringList lista_1 = s1.split(",");
 for (int i=0; i< lista_1.size(); ++i)
 qDebug()<<lista_1.at(i).toLocal8Bit().constData()<<endl;
 QString s2 = "pawel.mikolajczak@umcs.lublin.pl";
 qDebug()<<"analiza napisu "<<s2<<"\n";
 QStringList lista_2 = s2.split(".");
 for (int i=0; i< lista_2.size(); ++i)
 qDebug()<<lista_2.at(i).toLocal8Bit().constData()<<endl;
 if(lista_2.contains("pl"))
    qDebug()<< "adres w Polsce";
 else
    qDebug()<<"adres zagraniczny";

return a.exec();
}
```

Po uruchomieniu programu otrzymujemy następujący wynik:



We fragmencie programu

```
QString s1 = "Anna,Ewa,Lola";
QStringList lista_1 = s1.split(",");
for (int i=0; i< lista_1.size(); ++i)
qDebug()<<lista_1.at(i).toLocal8Bit().constData()<<endl;
```

zdefiniowano łańcuch **s1,** są w nim zapisane imiona rozdzielone przecinkiem. Zdefinowany został także obiekt klasy QStringList - **lista_1**, w tym kontenerze (w liście) przechowywane będą poszczególne imiona. Do rozdzielenia łańcucha **s1** wykorzystano metode split(), jej argumentem w naszym przypadku jest postać separator (tutaj jest to przecinek).
Prototyp metody ma postać:

QStringList QString::split(const QChar & sep, SplitBehavior behavior =
KeepEmptyParts, Qt::CaseSensitivity cs = CaseSensitive) const

Ta metoda rozdziela łańcuch na podłańcuchy zgodnie z ustalonym separatorem i zwraca listę wszystkich wykrytych podłańcuchów. Gdy nie zostaną wykryte żadne podłańcuchy, split() zwraca jednoelementową listę z całym łańcuchem.
W celu wydrukowania listy podłańcuchów, tak jak to zostało pokazane w naszym programie:

```
qDebug()<<lista_1.at(i).toLocal8Bit().constData()<<endl;
```

wykorzystano metodę at(i), której prototyp ma postać:

```
const T & QList::at(int i) const
```

Metoda at(i) zwraca obiekt listy umieszczonym pod indeksem i. Indeks musi mieć dozwoloną wartość, tzn. musi spełniać warynek (0 <= i < size()). Ta funkcja ze względu na szybkość jest zalecana. Występujaca w tej instrukcji metoda toLocal8Bit(), której prototyp ma postać:

QByteArray QString::toLocal8Bit() const

Zwraca 8-bitowa reprezentacje łańcuchaw postaci obiektu QByteArray.
Drugi analizowany łańcuch

```
QString s2 = "pawel.mikolajczak@umcs.lublin.pl";
```

Zawiera typowy adres poczty elektronicznej. Metoda split() z kropką, jako separatorem podzieli nam adres na fragmenty. Korzystajac z metody contains(), której prototyp ma postać:

bool QStringList::contains(const QString & str, Qt::CaseSensitivity
cs = Qt::CaseSensitive) const

sprawdzamy, czy w łańcuchu jest podłańcuch `"pl"`. Dzięki temu możemy na przykład identyfikować kraj właściciela adresu.
Qt posiada bardzi ciekawa klasę o nazwie QRegExp do obsługi wyrażen regularnych. Aby można było korzystać z tej klasy należy do programu włączyć plik nagłówkowy

#include <QRegExp>

Ze względu na użyteczność a także względów praktycznych, jest to bardzo rozbudowana klasa, z całą konwencją tworzenia wyrażeń regularnych. Aby efektywnie stosować wykrywanie wyrażeń regularnych zawartych w łańcuchach należy gruntownie zapoznać się z opisem technicznym zamieszczonym w systemie pomocy Qt.
Kolejny program zilustruje pewne aspekty wykorzystywania wyrażeń regularnych. W programie zanalizujemy dwa łańcuchy:

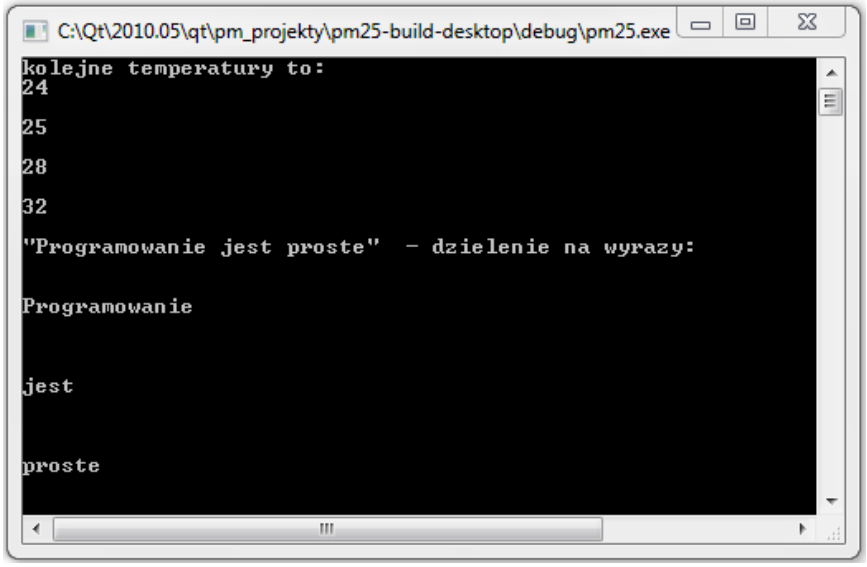
```
QString s1 = "temperatury: 24  25  28  32";
QString s2 = "Programowanie jest proste";
```

W łańcuchu pierwszym (**s1**) wyodrębnimy liczby całkowite, będące notowaniami kolejnych temperatur. W drugim łańcuchu (**s2**) wyodrębnimy poszczególne wyrazy, z których zbudowane jest zdanie.

Wydruk 9.8 Klasa QRegExp, wybrane metody

```
#include <QtCore/QCoreApplication>
#include <QRegExp>
#include<QStringList>
#include<QString>
#include <QDebug>
int main(int argc, char *argv[])
{ QCoreApplication a(argc, argv);
  QRegExp rx("(\\d+)");
  QString s1 = "temperatury: 24  25  28  32";
  QStringList lista_1;
  int pos = 0;
  while ((pos = rx.indexIn(s1,pos)) != -1)
  {lista_1 << rx.cap(1);
   pos += rx.matchedLength();
  }
  qDebug()<<"kolejne temperatury to: ";
  for (int i=0; i< lista_1.size(); ++i)
 qDebug()<<lista_1.at(i).toLocal8Bit().constData()<<endl;
  QStringList lista_2;
  QString s2 = "Programowanie jest proste";
  qDebug()<<s2<<" - dzielenie na wyrazy:";
  lista_2 = s2.split(QRegExp("\\b"));
  for (int i=0; i< lista_2.size(); ++i)
 qDebug()<<lista_2.at(i).toLocal8Bit().constData()<<endl;
  return a.exec();
}
```

Po uruchomieniu programu otrzymujemy nastepujący wynik:



Ustalamy kryterium wyszukiwania (obiekt rx) :

```
QRegExp rx("(\\d+)");
```

Zgodnie z opisem QT znacznik **\d** jest wyrażeniem regularnym do wyszukiwania cyfr, znacznik **+** oznacza kolejne cyfry.
W pętli

```
while ((pos = rx.indexIn(s1,pos)) != -1)
{lista_1 << rx.cap(1);
 pos += rx.matchedLength();
}
```

przeglądamy analizowany łańcuch. Metoda **indexIn(),** której prototyp ma postać:

int QRegExp::indexIn(const QString & str, int offset = 0,
CaretMode caretMode = CaretAtZero) const

Poszukuje wyrażenia reguralnego w łańcuchu str od pozycji danej przez offset (przez domniemanie przyjmujemy wartość 0). Jeżeli funkcja znajdzie wyrażenie regularne, zwraca jego pozycję, gdy nie znajdzie wyrażenia – zwraca wartość -1.
Metoda cap(), której prototyp ma postać:

Qstring QRegExp::cap(int nth = 0) const

zwraca wykryty podłańcuch, w naszym przykladzie, wykryty podłańcuch umieszczany jest na liście lista_1. Iteracja pozycji w łańcuchu s1 realizowana jest przy pomocy metody matchedLenght(). Prototyp tej metody ma postać:

int QRegExp::matchedLength() const

Funkcja zwraca długość ostatniego znalezionego podłańcucha, gdy go nie znajdzie , zwraca wartość -1.
W drugim łańcuchu **s2** dokonano podziału zdania na wyrazy:

```
lista_2 = s2.split(QRegExp("\\b"));
for (int i=0; i< lista_2.size(); ++i)
qDebug()<<lista_2.at(i).toLocal8Bit().constData()<<endl;
```

Argumentem metody split jest regularne wyrażenie:

```
QRegExp(("\\b"))
```

Znacznik \b służy do identyfikacji wyrazów w łańcuchu.

BIBLIOGRAFIA

[1] B. Stroustrup, *Programowanie, teoria i praktyka z wykorzystaniem C++*, Wydawnictwo Helion, Gliwice, 2010.

[2] J.Liberty, S. Rao, B. Jones, *C++*, Wydanie II, Wydawnictwo Helion, Gliwice, 2011

[3] N.Josuttus, *C++. Biblioteka standardowa*, Wydawnictwo Helion, Gliwice, 2003

[4] J. Ganczarski, *C++. Wykorzystaj potęgę aplikacji graficznych*, Wydawnictwo Helion, Gliwice, 2008

[5] D. Solin, *Poznaj programowanie przy użyciu biblioteki Qt*, Wydawnictwo Infoland, Wraszawa, 2001

[6] J.Blanchette, M. Summerfield, *C++ GUI programming with Qt 4*, Wydawnictwo Prentice Hall, London, 2010

[7] M. Dalheimer, *Programming with Qt*, Wydawictwo O'Reilly, Cambridge, Londyn, 1999

[8] S. Prata, *Język C++*, Wydawnictwo Helion, Gliwice, 2006

[9] D.Stephens, C.Diggins, J. Turkanis, J. Cogswell, *C++. Receptury*, Wydawnictwo Helion, Gliwice, 2006