Qt – hierarchia elementów graficznych

Bogdan Kreczmer

bogdan.kreczmer@pwr.wroc.pl

Katedra Cybernetyki i Robotyki
Wydział Elektroniki
Politechnika Wrocławska

Kurs: Wizualizacja danych sensorycznych

Niniejsza prezentacja została wykonana przy użyciu systemu składu PDFLaTeX *oraz stylu* `beamer`, *którego autorem jest Till Tantau.*

Strona domowa projektu `Beamer`:
`http://latex-beamer.sourceforge.net`

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Ogólna struktura funkcji `main`

Przykład prezentowany w pliku `hamlet.cpp` nie pokazuje typowej konstrukcji aplikacji pisanej z wykorzystaniem biblioteki Qt. Zwykle w funkcja `main` każdej z aplikacji tworzonej na bazie Qt możemy wyróżnić kilka kluczowych operacji:

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów z linii wywołania
2. Utworzenie okna aplikacji
3. Wymuszenie ukazania się okna
4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji.

Nie jest to najogólniejszy schemat (o ile taki w ogóle istnieje) funkcji `main`. Niemniej zawiera najważniejsze elementy jej konstrukcji.

Przykład nieprzyzwoicie prostej aplikacji

Przykład nieprzyzwoicie prostej aplikacji

Nazwa pojawiająca się w nagłówku okienka jest w tym przypadku nazwą pliku wykonywalnego.

Nieprzyzwoicie prosta aplikacja w szczegółach

```
#include <QApplication>
#include <QWidget>

int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    QWidget      Okno;

    Okno.show();

    return App.exec();
}
```

Nieprzyzwoicie prosta aplikacja w szczegółach

```
#include <QApplication>
#include <QWidget>

int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    QWidget      Okno;

    Okno.show();

    return App.exec();
}
```

1. Utworzenie obiektu klasy **QApplication** i wstępne przetworzenie argumentów wywołania

Nieprzyzwoicie prosta aplikacja w szczegółach

```
#include <QApplication>
#include <QWidget>

int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    QWidget      Okno;

    Okno.show();

    return App.exec();
}
```

2. Utworzenie okna aplikacji

Nieprzyzwoicie prosta aplikacja w szczegółach

```
#include <QApplication>
#include <QWidget>

int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    QWidget      Okno;

    Okno.show();

    return App.exec();
}
```

3. Wymuszenie ukazania się okna

Nieprzyzwoicie prosta aplikacja w szczegółach

```
#include <QApplication>
#include <QWidget>

int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    QWidget      Okno;

    Okno.show();

    return App.exec();
}
```

4. Uruchomienie obsługi pętli zdarzeń dla całej aplikacji

Nieprzyzwoicie prosta aplikacja w szczegółach

```
#include <QApplication>
#include <QWidget>

int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    QWidget      Okno;

    Okno.show();

    return App.exec();
}
```

Na czym polega przetwarzanie argumentów wywołania aplikacji?

Przetwarzanie argumentów wywołania

Biblioteka Qt umożliwia rozpoznanie i odpowiednie przetworzenie opcji z linii wywołania, które są standardowe dla danej platformy.

W systemie X Window należą do nich m.in.: `-geometry`, `-display`, `-title`.

Nie są obsługiwane opcje takie jak: `-fg`, `-bg`, `-fn`.

Przetworzenie argumentów tych opcji powoduje nadanie domyślnych wartości zasobom graficznym, którymi dysponuje aplikacja. Programista może zawsze jawnie wymusić wybór przez siebie zadanych wartości.

Przetwarzanie argumentów wywołania

Biblioteka Qt umożliwia rozpoznanie i odpowiednie przetworzenie opcji z linii wywołania, które są standardowe dla danej platformy.

W systemie X Window należą do nich m.in.: `-geometry`, `-display`, `-title`.

Nie są obsługiwane opcje takie jak: `-fg`, `-bg`, `-fn`.

Przetworzenie argumentów tych opcji powoduje nadanie domyślnych wartości zasobom graficznym, którymi dysponuje aplikacja. Programista może zawsze jawnie wymusić wybór przez siebie zadanych wartości.

Przetwarzanie argumentów wywołania

Biblioteka Qt umożliwia rozpoznanie i odpowiednie przetworzenie opcji z linii wywołania, które są standardowe dla danej platformy.

W systemie X Window należą do nich m.in.: `-geometry`, `-display`, `-title`.

Nie są obsługiwane opcje takie jak: `-fg`, `-bg`, `-fn`.

Przetworzenie argumentów tych opcji powoduje nadanie domyślnych wartości zasobom graficznym, którymi dysponuje aplikacja. Programista może zawsze jawnie wymusić wybór przez siebie zadanych wartości.

Przetwarzanie argumentów wywołania

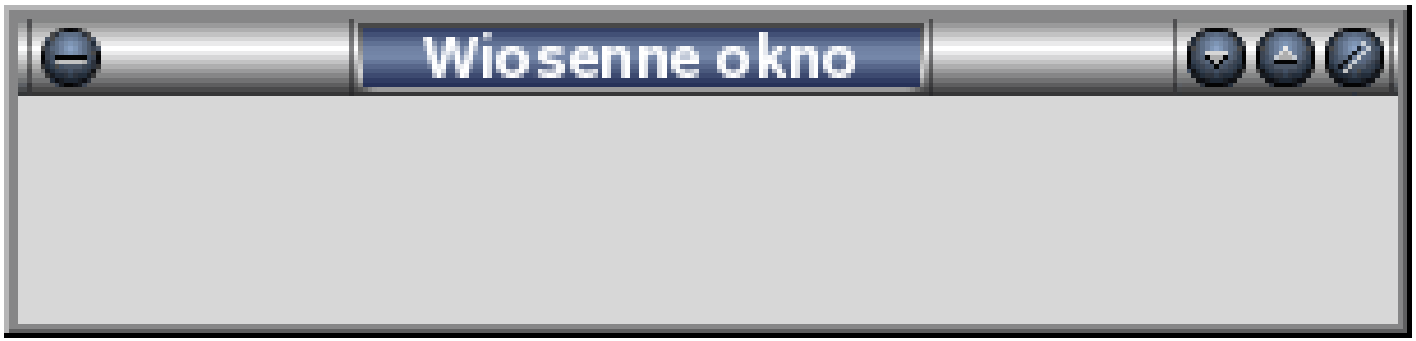
Biblioteka Qt umożliwia rozpoznanie i odpowiednie przetworzenie opcji z linii wywołania, które są standardowe dla danej platformy.

W systemie X Window należą do nich m.in.: `-geometry`, `-display`, `-title`.

Nie są obsługiwane opcje takie jak: `-fg`, `-bg`, `-fn`.

Przetworzenie argumentów tych opcji powoduje nadanie domyślnych wartości zasobom graficznym, którymi dysponuje aplikacja. Programista może zawsze jawnie wymusić wybór przez siebie zadanych wartości.

Efekt wywołania aplikacji



Wywołanie aplikacji:

```
./puste_okno -geometry 100x30+10 -title "Wiosenne okno"
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Określa monitor do komunikacji użytkownika z serwerem `X Window`, tzn. ekran, klawiatura, urządzenie wskazujące (np. mysz). Przykład użycia opcji:

```
xterm -display sequoia.ict.pwr.wroc.pl:0.0
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Definiuje geometrię okienka, tzn. szerokość, wysokość, przesunięcie x i y względem górnego rogu ekranu. Przykład użycia opcji:

```
xterm -geometry 200x80+10-2
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Definiuje domyślny kolor tekstu poprzez nazwę (zbiór nazw kolorów patrz: `/usr/X11R6/lib/X11/rgb.txt`) lub wartości RGB. Przykład użycia opcji:

```
xterm -fg DarkSlateGray
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Definiuje domyślny kolor tła poprzez nazwę (zbiór nazw kolorów patrz: `/usr/X11R6/lib/X11/rgb.txt`) lub wartości RGB. Przykład użycia opcji:

```
xterm -bg "light gray"
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Definiuje domyślny czcionkę. Lista nazw czcionek (i innych o nich informacji) dostępny jest poprzez polecenie `xlsfonts`. Przykład użycia opcji:

```
xterm -fn "lucidasans-bold-10"
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Definiuje domyślny kolor przycisków. Przykład użycia opcji:

```
./hamlet -btn red
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Przyporządkowuje aplikacji nazwę. Pozwala to zdefiniować dla niej zasoby w `~/.Xresorces`. Przykład użycia opcji:

```
./hamlet -name "XHamlet"
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Zmienia tytuł nagłówka okienka.
Przykład użycia opcji:

```
xterm -title "Moj XTerm"
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Dotyczą wyświetlaczy 8-bitowych.

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`.

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`:
`-nograb`, `-dograb` (tylko dla `X11`), `-sync` (tylko dla `X11`).

Ustala styl aplikacji (postać klawiszy, ikon, suwaków itd.). Dopuszczalne style to: `motif`, `windows`, `platinum`. Przykład użycia opcji:

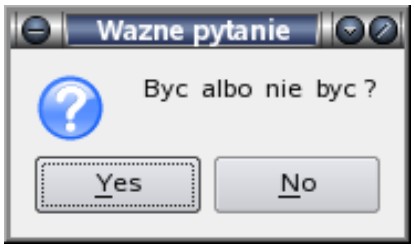
```
./hamlet -style motif
```

Style GUI

domyślny motif windows platinum

Style GUI

domyślny motif windows platinum

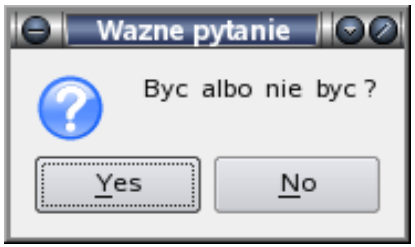


Sposób wywołania aplikacji:

```
./hamlet
```

Style GUI

domyślny motif windows platinum

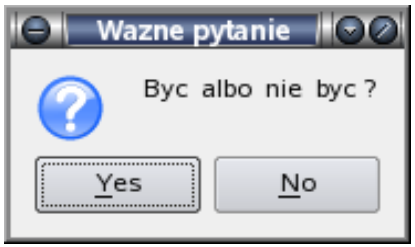


Sposób wywołania aplikacji:

```
./hamlet -style motif
```

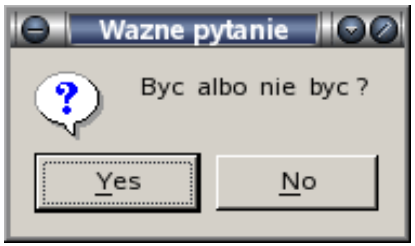
Style GUI

domyślny



motif

windows



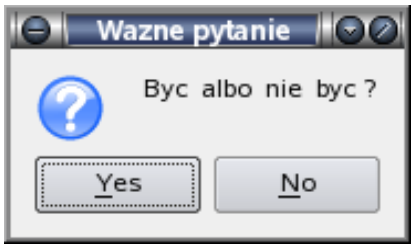
platinum

Sposób wywołania aplikacji:

```
./hamlet -style windows
```

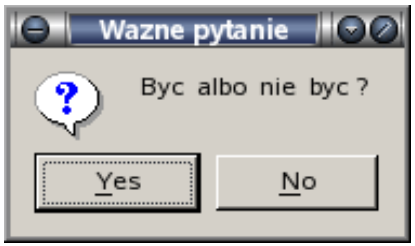
Style GUI

domyślny

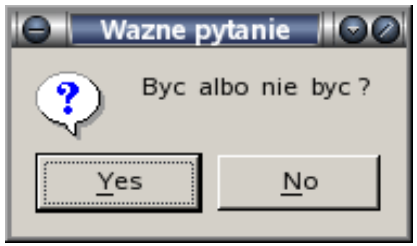


motif

windows



platinum



Sposób wywołania aplikacji:

```
./hamlet -style platinum
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Przywraca wygląd aplikacji z podanej sesji.

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Po zamknięciu aplikacji wyświetla liczbę nie zniszczonych widget'ów i maksymalną ich liczbę w trakcie pracy. Przykład użycia opcji:

```
./hamlet -widgetcount
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Po zamknięciu aplikacji wyświetla liczbę nie zniszczonych widget'ów i maksymalną ich liczbę w trakcie pracy. Przykład użycia opcji:

```
./hamlet -widgetcount
Widgets left: 0   Max widgets: 7
```

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Powoduje ustawienie stylu rozkładu tekstu i elementów aplikacji "z prawa na lewo" (patrz **Qt::RightToLeft**).

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Zabrania aplikacji zawłaszczenia klawiatury i myszki. Operacja zawłaszczenia powoduje, że wszystkie zdarzenia generowane w systemie okienkowym przechwytywane są przez aplikację, która dokonuje zawłaszczenia. Operacja ta może uniemożliwiać debugowanie aplikacji.

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Aplikacje pracujące pod debugerem mogą być niejawnie ustawiane w trybie `nograb`. Opcja `-dograb` powoduje unieważnienie tego trybu i jawne zezwolenie dla aplikacji na dokonywanie zawłaszczenia klawiatury i myszki.

Ważniejsze rozpoznawane opcje wywołania

Opcje `X11` — tradycyjne opcje systemu `X Window`:
`-display`, `-geometry`, `-fg`, `-bg`, `-fn`, `-btn`, `-name`, `-title`, `-visual`, `-ncols`, `-cmap`.

Opcje `Qt` — są one rozpoznawane przez każdy program pisany z wykorzystaniem biblioteki `Qt`: `-style`, `-session`, `-widgetcount`, `-reverse`

Opcje debugowania — dostępne jeśli `Qt` zostało skompilowane ze zdefiniowanym symbolem `QT_DEBUG`: `-nograb`, `-dograb` (tylko `X11`), `-sync` (tylko `X11`).

Przełącza aplikację w tryb pracy synchronicznej na potrzeby debugowania.

Ustawianie wartości parametrów

Wartości parametrów poszczególnych opcji wywołania aplikacji można zignorować zadając własne wartości. Operację tę zawsze należy wykonać po przetworzeniu argumentów wywołania przez obiekt klasy **QApplication**. Zazwyczaj jest również konieczne, aby wykonać tę operację po wywołaniu metody `show`.

Ustawianie wartości parametrów

Wartości parametrów poszczególnych opcji wywołania aplikacji można zignorować zadając własne wartości. Operację tę zawsze należy wykonać po przetworzeniu argumentów wywołania przez obiekt klasy **QApplication**. Zazwyczaj jest również konieczne, aby wykonać tę operację po wywołaniu metody `show`.

Ustawianie wartości parametrów

Wartości parametrów poszczególnych opcji wywołania aplikacji można zignorować zadając własne wartości. Operację tę zawsze należy wykonać po przetworzeniu argumentów wywołania przez obiekt klasy **QApplication**. Zazwyczaj jest również konieczne, aby wykonać tę operację po wywołaniu metody `show`.

Ustawianie wartości parametrów

```
#include <QApplication>
#include <QWidget>

int main( int argc,  char * argv[ ] )
 {
   QApplication  App(argc,argv);
   QWidget       Okno;

   Okno.show();
   Okno.setWindowTitle("Bardzo proste okno");
   Okno.setGeometry(0, 100, 300, 150);

   return App.exec();
 }
```

Ustawianie wartości parametrów

```
#include <QApplication>
#include <QWidget>

int main( int argc,  char * argv[ ] )
 {
   QApplication  App(argc,argv);
   QWidget       Okno;

   Okno.show();
   Okno.setWindowTitle("Bardzo proste okno");
   Okno.setGeometry(0, 100, 300, 150);

   return App.exec();
 }
```

Jawnie zadany zostaje tytuł okienka oraz jego geometria, tzn. położenie względem lewego górnego rogu ekranu oraz rozmiar.

Efekt wywołania aplikacji

Wywołanie aplikacji:

```
./puste_okno -geometry 300x5 -title "Wiosenne okno"
```

Mimo zadania parametrów, tytuł i geometria okienka nie uległy zmianie.

Ustawianie własnych parametrów – zalecenie

Z punktu widzenia programisty łatwiej jest zignorować ustawienia użytkownika i nadać aplikacji stałe cechy.

Jednak z punktu widzenia użytkownika znacznie zmniejsza to jej funkcjonalność (wystarczy sobie wyobrazić co byłoby gdyby `xterm` pracował tylko z jedną paletą barw i jednym rozmiarem czcionki).

Z tego powodu, o ile jest to tylko możliwe, należy zachować funkcjonalności związanę ze zmianą parametrów GUI przez użytkownika, tak aby mógł on dostosować wygląd interfejsu do swoich potrzeb.

Ustawianie własnych parametrów – zalecenie

Z punktu widzenia programisty łatwiej jest zignorować ustawienia użytkownika i nadać aplikacji stałe cechy.

Jednak z punktu widzenia użytkownika znacznie zmniejsza to jej funkcjonalność (wystarczy sobie wyobrazić co byłoby gdyby `xterm` pracował tylko z jedną paletą barw i jednym rozmiarem czcionki).

Z tego powodu, o ile jest to tylko możliwe, należy zachować funkcjonalności związanę ze zmianą parametrów GUI przez użytkownika, tak aby mógł on dostosować wygląd interfejsu do swoich potrzeb.

Ustawianie własnych parametrów – zalecenie

Z punktu widzenia programisty łatwiej jest zignorować ustawienia użytkownika i nadać aplikacji stałe cechy.

Jednak z punktu widzenia użytkownika znacznie zmniejsza to jej funkcjonalność (wystarczy sobie wyobrazić co byłoby gdyby `xterm` pracował tylko z jedną paletą barw i jednym rozmiarem czcionki).

Z tego powodu, o ile jest to tylko możliwe, należy zachować funkcjonalności związanę ze zmianą parametrów GUI przez użytkownika, tak aby mógł on dostosować wygląd interfejsu do swoich potrzeb.

Przetwarzania argumentów wywołania – kilka pytań

- Co się dzieje z przetwarzanymi argumentami z linii wywołania (o ile w ogóle coś się z nimi dzieje)?
- Jeżeli będziemy chcieli użyć również własnych argumentów wywołania, to jak je odróżnić od tych, które są właściwe dla danej platformy?
- Czy można w jakiś sposób odróżnić te argumenty, które zostały przetworzone od tych, które nie zostały przetworzone?

Przetwarzania argumentów wywołania – kilka pytań

- Co się dzieje z przetwarzanymi argumentami z linii wywołania (o ile w ogóle coś się z nimi dzieje)?
- Jeżeli będziemy chcieli użyć również własnych argumentów wywołania, to jak je odróżnić od tych, które są właściwe dla danej platformy?
- Czy można w jakiś sposób odróżnić te argumenty, które zostały przetworzone od tych, które nie zostały przetworzone?

Przetwarzania argumentów wywołania – kilka pytań

- Co się dzieje z przetwarzanymi argumentami z linii wywołania (o ile w ogóle coś się z nimi dzieje)?
- Jeżeli będziemy chcieli użyć również własnych argumentów wywołania, to jak je odróżnić od tych, które są właściwe dla danej platformy?
- Czy można w jakiś sposób odróżnić te argumenty, które zostały przetworzone od tych, które nie zostały przetworzone?

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "==========================" << endl;
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << endl;
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "==========================" << endl;
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << endl;
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "==========================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "==========================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
==========================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "==========================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
==========
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
====================
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
====================

      ????
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
====================
argv[0] = ./testargs
argv[1] = -x
argv[2] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
====================
argv[0] = ./testargs
argv[1] = -x
argv[2] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "==========================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
====================
argv[0] = ./testargs
argv[1] = -x
argv[2] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

testargs.cpp – przetwarzania argumentów wywołania

```
void WyswietlArgumenty( int argc, char* argv[ ] )
{
   cout << "============================" << e
   for (int i=0; i < argc; ++i)
        cout << "argv[" << i << "] = " << argv[ i ] << e
}

int main( int argc, char * argv[ ] )
{
   WyswietlArgumenty(argc, argv);
   QApplication App(argc, argv);
   WyswietlArgumenty(argc, argv);
}
```

Wynik działania programu:

```
====================
argv[0] = ./testargs
argv[1] = -display
argv[2] = rab:0
argv[3] = -title
argv[4] = Idzie wiosna
argv[5] = -x
argv[6] = moja_opcja
====================
argv[0] = ./testargs
argv[1] = -x
argv[2] = moja_opcja
```

Rozważmy następujące wywołanie programu:

```
./testargs -display rab:0 -title "Idzie wiosna"  -x moja_opcja
```

QApplication – najważniejsze cechy

- W aplikacji może być tylko jeden obiekt tej klasy.
- Jest on odpowiedzialny za zainicjowanie zasobów domyślnych dla całej aplikacji, na podstawie aktualnych ustawień desktopu.
- Przetwarza argumenty z linii wywołania.
- Realizuje obsługę zdarzeń dla całej aplikacji rozsyłając informacje o zdarzeniach do odpowiednich okienek.
- Dostarcza informacji o lokalizacji napisów, które są wyświetlane poprzez funkcję `translate()`.
- Udostępnia specjalne obiekty poprzez metody `desktop()` i `cliboard()`.

QApplication – najważniejsze cechy

- W aplikacji może być tylko jeden obiekt tej klasy.
- Jest on odpowiedzialny za zainicjowanie zasobów domyślnych dla całej aplikacji, na podstawie aktualnych ustawień desktopu.
- Przetwarza argumenty z linii wywołania.
- Realizuje obsługę zdarzeń dla całej aplikacji rozsyłając informacje o zdarzeniach do odpowiednich okienek.
- Dostarcza informacji o lokalizacji napisów, które są wyświetlane poprzez funkcję `translate()`.
- Udostępnia specjalne obiekty poprzez metody `desktop()` i `cliboard()`.

QApplication – najważniejsze cechy

- W aplikacji może być tylko jeden obiekt tej klasy.
- Jest on odpowiedzialny za zainicjowanie zasobów domyślnych dla całej aplikacji, na podstawie aktualnych ustawień desktopu.
- Przetwarza argumenty z linii wywołania.
- Realizuje obsługę zdarzeń dla całej aplikacji rozsyłając informacje o zdarzeniach do odpowiednich okienek.
- Dostarcza informacji o lokalizacji napisów, które są wyświetlane poprzez funkcję `translate()`.
- Udostępnia specjalne obiekty poprzez metody `desktop()` i `cliboard()`.

QApplication – najważniejsze cechy

- W aplikacji może być tylko jeden obiekt tej klasy.
- Jest on odpowiedzialny za zainicjowanie zasobów domyślnych dla całej aplikacji, na podstawie aktualnych ustawień desktopu.
- Przetwarza argumenty z linii wywołania.
- Realizuje obsługę zdarzeń dla całej aplikacji rozsyłając informacje o zdarzeniach do odpowiednich okienek.
- Dostarcza informacji o lokalizacji napisów, które są wyświetlane poprzez funkcję `translate()`.
- Udostępnia specjalne obiekty poprzez metody `desktop()` i `cliboard()`.

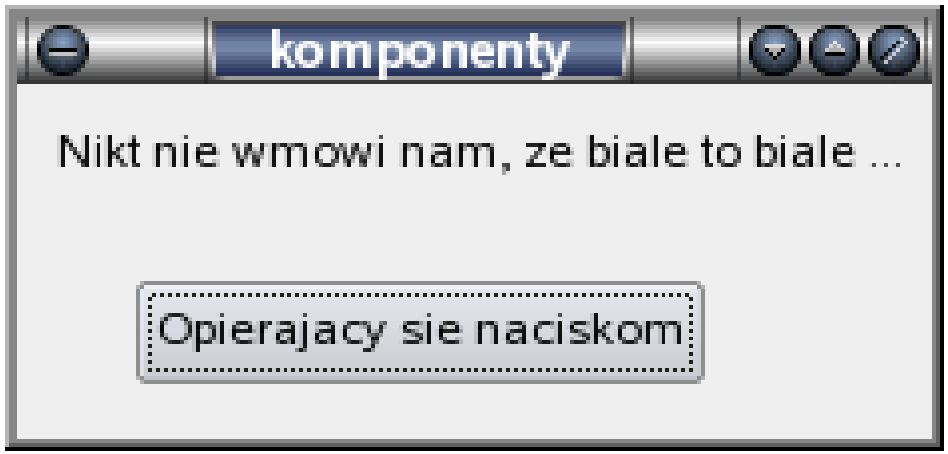
QApplication – najważniejsze cechy

- W aplikacji może być tylko jeden obiekt tej klasy.
- Jest on odpowiedzialny za zainicjowanie zasobów domyślnych dla całej aplikacji, na podstawie aktualnych ustawień desktopu.
- Przetwarza argumenty z linii wywołania.
- Realizuje obsługę zdarzeń dla całej aplikacji rozsyłając informacje o zdarzeniach do odpowiednich okienek.
- Dostarcza informacji o lokalizacji napisów, które są wyświetlane poprzez funkcję `translate()`.
- Udostępnia specjalne obiekty poprzez metody `desktop()` i `cliboard()`.

QApplication – najważniejsze cechy

- W aplikacji może być tylko jeden obiekt tej klasy.
- Jest on odpowiedzialny za zainicjowanie zasobów domyślnych dla całej aplikacji, na podstawie aktualnych ustawień desktopu.
- Przetwarza argumenty z linii wywołania.
- Realizuje obsługę zdarzeń dla całej aplikacji rozsyłając informacje o zdarzeniach do odpowiednich okienek.
- Dostarcza informacji o lokalizacji napisów, które są wyświetlane poprzez funkcję `translate()`.
- Udostępnia specjalne obiekty poprzez metody `desktop()` i `cliboard()`.

Przykład nieco bardziej złożonej aplikacji

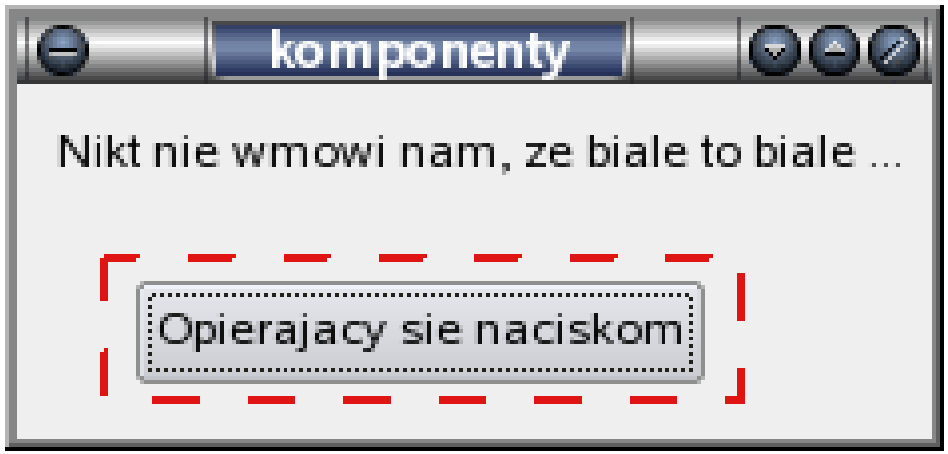


Aplikacja zawiera dwa komponenty, które są *położone* na bazowym obiekcie klasy **QWidget**. Są nimi: obiekt klasy **QLabel** zawierający napis oraz obiekt klasy **QPushButton** tworzący przycisk.

Przykład nieco bardziej złożonej aplikacji

Aplikacja zawiera dwa komponenty, które są *położone* na bazowym obiekcie klasy **QWidget**. Są nimi: obiekt klasy **QLabel** zawierający napis oraz obiekt klasy **QPushButton** tworzący przycisk.

Przykład nieco bardziej złożonej aplikacji



Aplikacja zawiera dwa komponenty, które są *położone* na bazowym obiekcie klasy **QWidget**. Są nimi: obiekt klasy **QLabel** zawierający napis oraz obiekt klasy **QPushButton** tworzący przycisk.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie obiektu klasy **QApplication** i przetworzenie argumentów wywołania aplikacji.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie widget'u, który będzie kanwą dla następnych elementów.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie widget'u, który będzie kanwą dla następnych elementów.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie etykiety z napisem (etykieta może być również skojarzona z obrazkiem lub animacją, patrz metoda `setPicture`). Etykieta zostaje położona na widget'cie `Okno` (staje się jego potomkiem).

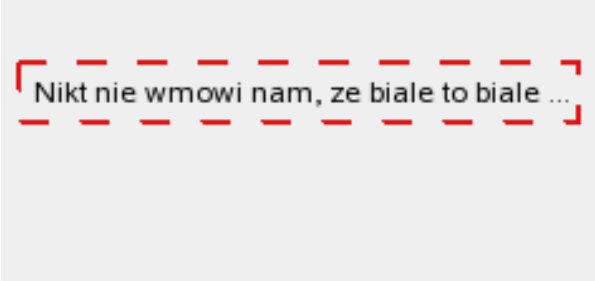
Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie etykiety z napisem (etykieta może być również skojarzona z obrazkiem lub animacją, patrz metoda `setPicture`). Etykieta zostaje położona na widget'cie `Okno` (staje się jego potomkiem).

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```



Utworzenie etykiety z napisem (etykieta może być również skojarzona z obrazkiem lub animacją, patrz metoda `setPicture`). Etykieta zostaje położona na widget'cie `Okno` (staje się jego potomkiem).

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Ulokowanie etykiety w pożądanym miejscu.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie przycisku z napisem (przycisk może być również skojarzona z ikoną poprzez wywołanie odpowiedniego konstruktora). Przycisk zostaje położona na widget'cie `Okno` (staje się jego potomkiem).

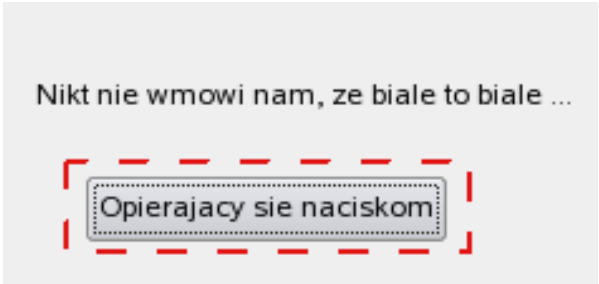
Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Utworzenie przycisku z napisem (przycisk może być również skojarzona z ikoną poprzez wywołanie odpowiedniego konstruktora). Przycisk zostaje położona na widget'cie `Okno` (staje się jego potomkiem).

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```



Utworzenie przycisku z napisem (przycisk może być również skojarzona z ikoną poprzez wywołanie odpowiedniego konstruktora). Przycisk zostaje położona na widget'cie `Okno` (staje się jego potomkiem).

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Przesunięcie przycisku do pożądanego miejsca.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Ustalenie nowego rozmiaru okienka.

Przykład nieco bardziej złożonej aplikacji

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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Wymuszenie ukazania się okienka.

Przykład nieco bardziej złożonej aplikacji


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

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Wymuszenie ukazania się okienka.

Przykład nieco bardziej złożonej aplikacji

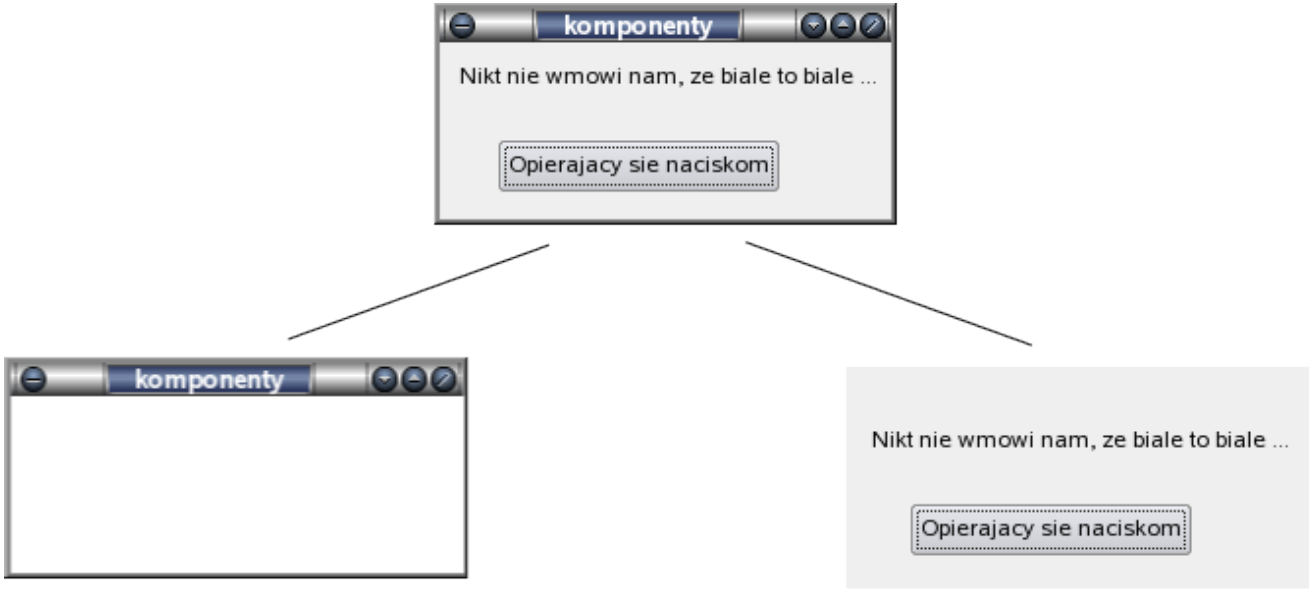

```
int main( int argc, char * argv[ ] )
{
  QApplication App( argc, argv );
  QWidget      Okno;

  QLabel *wEtyk =
        new QLabel(QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk =
        new QPushButton(QObject::tr("Opierajacy sie naciskom"), &Okno);
  wPrzycisk->move(30,50);

  Okno.resize( 230, 90 );   Okno.show( );

  return App.exec( );
}
```

Uruchomienie pętli obsługi zdarzeń.

Diagram obiektów graficznych

Pojawiające się okienko jest komponentem co najmniej dwu składników.

Diagram obiektów graficznych

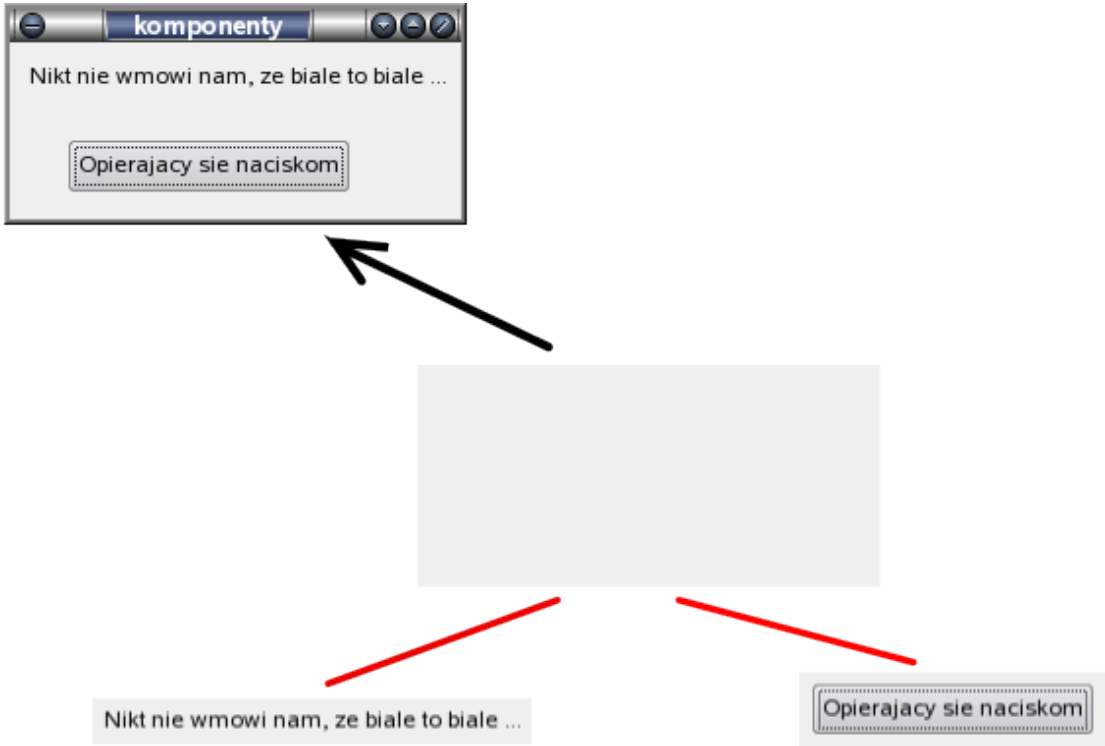


Są nimi: okienko samej aplikacji i obramowanie dostarczane przez *Window Manager*.

Diagram obiektów graficznych

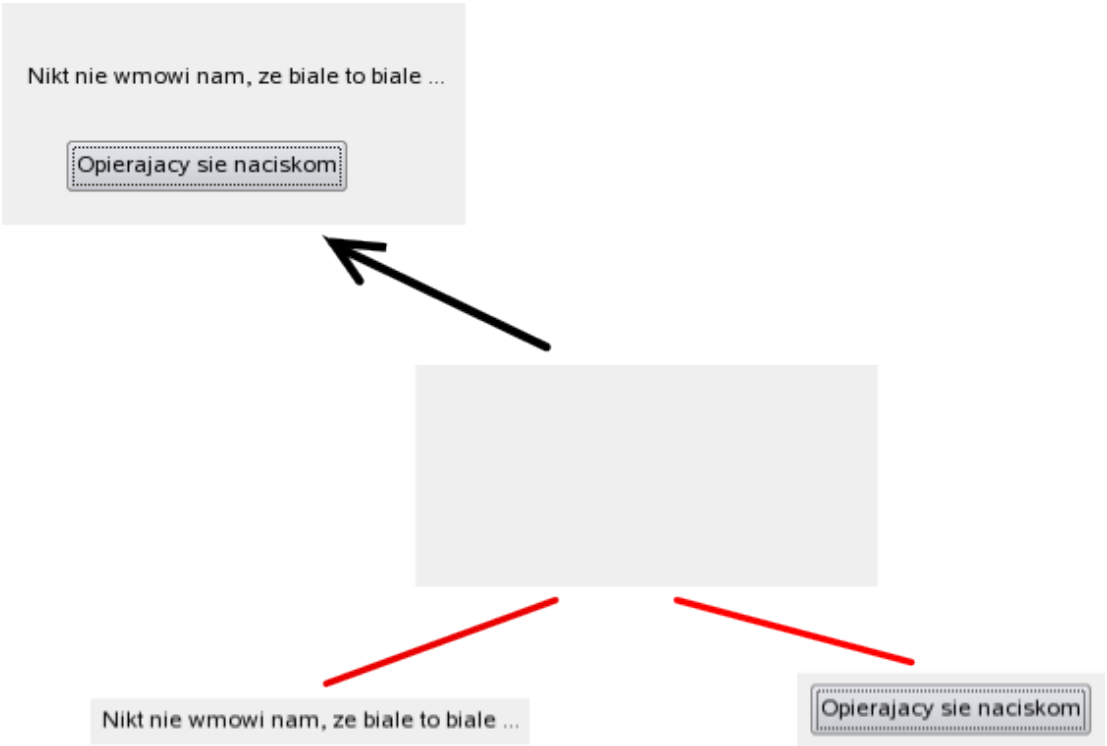
Samo okienko aplikacji jest także komponentem wielu elementów graficznych. Ich sposób organizacji ma strukturę drzewiastą, przy czym elementy potomne leżą w obrębie swojego przodka.

Diagram obiektów graficznych



W ten sposób logiczna reprezentacja układu komponentów pozwala utworzyć reprezentację graficzną finalnego okienka.

Diagram obiektów graficznych



Należy jednak pamiętać, że obramowanie okienka nie jest częścią tej reprezentacji.

Modelowanie obiektów graficznych

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

    QLabel *wEtyk =
          new QLabel( QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
    wEtyk->move(10,10);

    QPushButton *wPrzycisk =
          new QPushButton(tr("Opierajacy sie naciskom"), &Okno);
    wPrzycisk->move(30,50);

    Okno.resize( 230, 90 );    Okno.show( );

    return App.exec( );
}
```

Nie jest dobrym rozwiązaniem tworzenie wszystkich elementów niezależnie. W takim przypadku *gubią* się związki logiczne między poszczególnymi komponentami.

Modelowanie obiektów graficznych

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

    QLabel *wEtyk =
          new QLabel( QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
    wEtyk->move(10,10);

    QPushButton *wPrzycisk =
          new QPushButton(tr("Opierajacy sie naciskom"), &Okno);
    wPrzycisk->move(30,50);

    Okno.resize( 230, 90 );    Okno.show( );

    return App.exec( );
}
```

Zdecydowanie lepszym rozwiązaniem jest uwypuklenie wspomnianych związków logicznych w postaci osobnej klasy, która będzie modelowała dany obiekt graficzny.

Modelowanie obiektów graficznych


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

    QLabel *wEtyk =
          new QLabel( QObject::tr("Nikt nie wmowi nam, ze biale to biale ..."), &Okno);
    wEtyk->move(10,10);

    QPushButton *wPrzycisk =
          new QPushButton(tr("Opierajacy sie naciskom"), &Okno);
    wPrzycisk->move(30,50);

    Okno.resize( 230, 90 );    Okno.show( );

    return App.exec( );
}
```

W takim przypadku kod odpowiedzialny za utworzenie okna powinien znaleźć się w konstruktorze danej klasy.

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
    QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
    wEtyk->move(10,10);

    QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
    wPrzycisk->move(30,50);

    resize(230,90);
}
```

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```

Klasa *ProsteOkno* definiujemy jako specjalizację klasy **QWidget**.

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
    QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
    wEtyk->move(10,10);

    QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
    wPrzycisk->move(30,50);

    resize(230,90);
}
```

Definiując konstruktor nie powinno się wykluczać możliwości *położenia* danego elementu graficznego na innym elemencie.

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```

Konstruktor elementu graficznego należy jawnie wywołać w liście inicjalizacyjnej, aby móc mu przekazać odpowiedni parametr (w tym przypadku wskaźnik na rodzica).

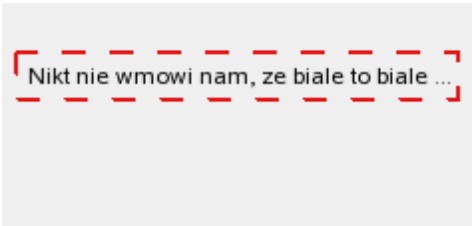
Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```

Uruchomienie konstruktora tworzy właściwy obiekt graficzny, który będzie rodzicem dla wszystkich innych elementów graficznych utworzonych w ciele konstruktora.

Modelowanie obiektów graficznych

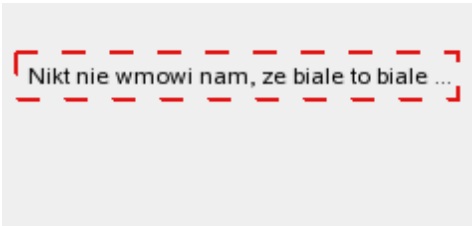
```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```



Tworzenie etykiety i odpowiednie jej ulokowanie.

Modelowanie obiektów graficznych

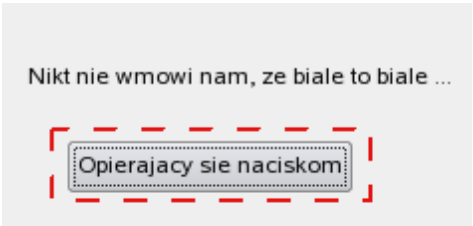
```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```



Tym razem rodzicem jest tworzony obiekt, gdyż zawiera on (dzięki dziedziczeniu) podobiekt klasy **QWidget**.

Modelowanie obiektów graficznych

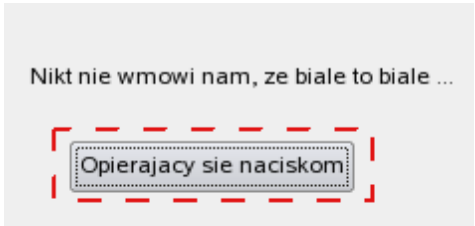
```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```



Tworzenie przycisku i odpowiednie jego ulokowanie.

Modelowanie obiektów graficznych

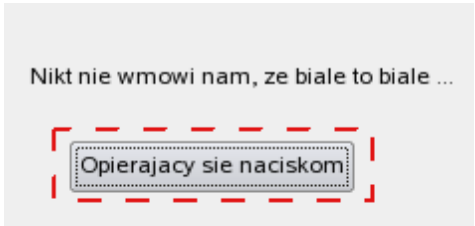
```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```



Ponownie rodzicem jest tworzony obiekt.

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
        ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```



Ustalenie rozmiaru.

Modelowanie obiektów graficznych

```
class ProsteOkno: public QWidget {
    public :
      ProsteOkno( QWidget *wRodzic = nullptr);
};

ProsteOkno::ProsteOkno(QWidget *wRodzic): QWidget(wRodzic)
{
  QLabel *wEtyk = new QLabel (tr("Nikt nie wmowi nam, ze biale to biale ..."), this);
  wEtyk->move(10,10);

  QPushButton *wPrzycisk = new QPushButton (tr("Opierajacy sie naciskom"), this);
  wPrzycisk->move(30,50);

  resize(230,90);
}
```

Jeśli utworzony element graficzny nie będzie miał rodzica, to stanie się osobnym okienkiem.

Funkcja `main` po zdefiniowaniu klasy `ProsteOkno`

```
int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    ProsteOkno Okno;

    Okno.show( );

    return App.exec( );
}
```

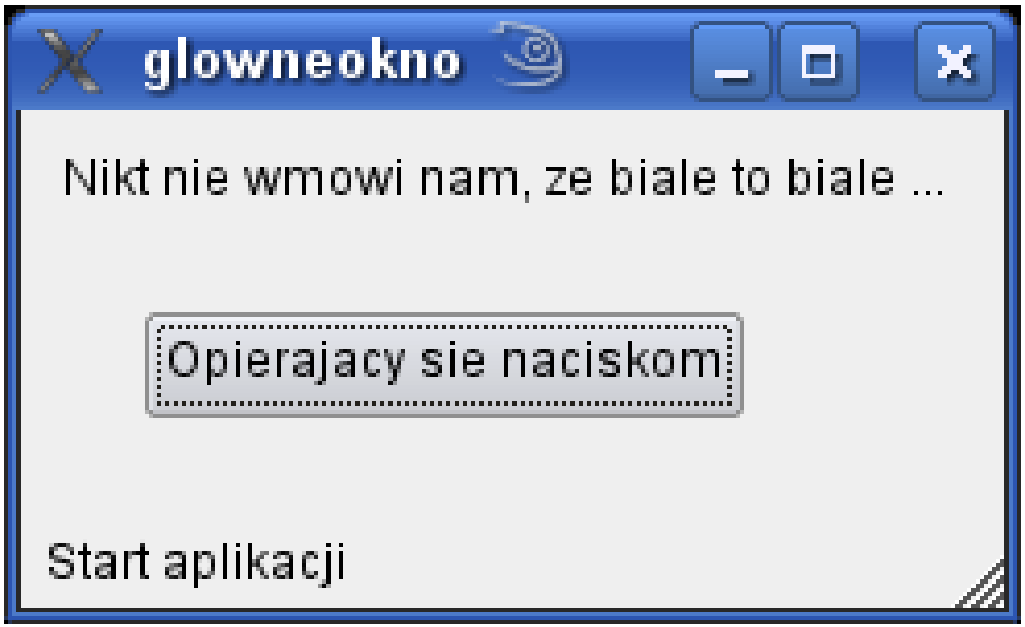
Kod funkcji `main` znacząco się upraszcza. W ten sposób wracamy do standardowego prostego schematu konstrukcji tej funkcji. Wystarczy teraz utworzyć obiekt graficzny i wymusić jego ukazanie się.

Funkcja `main` po zdefiniowaniu klasy `ProsteOkno`

```
int main( int argc, char * argv[ ] )
{
    QApplication App( argc, argv );
    ProsteOkno Okno;

    Okno.show( );

    return App.exec( );
}
```

Kod funkcji `main` znacząco się upraszcza. W ten sposób wracamy do standardowego prostego schematu konstrukcji tej funkcji. Wystarczy teraz utworzyć obiekt graficzny i wymusić jego ukazanie się.

Widget w okienku głównym



Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication     App(argc,argv);
    QMainWindow      GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Tworzymy obiekt reprezentujący główne okno aplikacji.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Tworzymy "nasz" Widget. Ważne jest to, że musi on być utworzony w sposób dynamiczny. Jest to związane z jego późniejszą destrukcją.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Dokonujemy powiązania obiektu klasy `ProsteOkno` z okienkiem głównym.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Towrzymy obiekt reprezentujący belkę statusową. Podobnie jak miało to miejsce wcześniej musi on być utworzony w sposób dynamiczny.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Dokonujemy powiązania obiektu belki statusowej z okienkiem głównym.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Wyświetlamy komunikat na belce statusowej.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Musimy pamiętać, że komunikat musi być "przygotowany" do ewentualnego tłumaczenie. Z tego powodu używamy metody `tr`.

Tworzenie głównego okna aplikacji

```
int main( int argc, char * argv[ ] )
{
    QApplication    App(argc,argv);
    QMainWindow     GlowneOkno;

    GlowneOkno.setCentralWidget(new ProsteOkno());
    GlowneOkno.setStatusBar(new QStatusBar());
    GlowneOkno.statusBar()->showMessage(QObject::tr("Start aplikacji"));
    GlowneOkno.show();

    return App.exec();
}
```

Dalszy ciąg jest już standardowy.

Koniec prezentacji