Programowanie, część II

Rafał J. Wysocki

Instytut Fizyki Teoretycznej, Wydział Fizyki UW

25 marca 2011

Biblioteki i podsystemy graficzne

Biblioteki graficzne (*ang. graphics library*)

Są potrzebne, aby można było programować grafikę w C++, ponieważ (niestety) we współczesnych systemach komputerowych podsystemy graficzne (*ang. graphics subsystem*) są *bardzo* skomplikowane.

Biblioteki i podsystemy graficzne

Biblioteki graficzne (*ang. graphics library*)

Są potrzebne, aby można było programować grafikę w C++, ponieważ (niestety) we współczesnych systemach komputerowych podsystemy graficzne (*ang. graphics subsystem*) są *bardzo* skomplikowane.

Przykład – GNU/Linux

- Sprzęt (*ang. hardware*) – wyświetlanie, przetwarzanie scen 3D, komunikacja z użytkownikiem
- Sterowniki w jądrze – operowanie sprzętem (niski poziom)
- Serwer X Windows – komunikacja ze sterownikami, system okien, udostępnianie przez sieć, zdarzenia
- Menedżer okien (*ang. window manager*) – dekoracje okien, komunikacja z aplikacjami i serwerem X Windows, zdarzenia

Obsługa zdarzeń

Zdarzenie (*ang. event*)

Sytuacja potencjalnie wymagająca reakcji ze strony co najmniej jednego z programów korzystających z podsystemu graficznego.

Obsługa zdarzeń

Zdarzenie (*ang. event*)

Sytuacja potencjalnie wymagająca reakcji ze strony co najmniej jednego z programów korzystających z podsystemu graficznego.

Źródła zdarzeń

1. Serwer X Windows
   - Sprzęt (klawiatura, mysz, ekran dotykowy)
   - Interakcje między oknami (zachodzenie na siebie itp.)
2. Menedżer okien
   - Komunikaty z menu itp.
   - Komunikaty od aplikacji

Obsługa zdarzeń

Zdarzenie (*ang. event*)

Sytuacja potencjalnie wymagająca reakcji ze strony co najmniej jednego z programów korzystających z podsystemu graficznego.

Źródła zdarzeń

1. Serwer X Windows
   - Sprzęt (klawiatura, mysz, ekran dotykowy)
   - Interakcje między oknami (zachodzenie na siebie itp.)
2. Menedżer okien
   - Komunikaty z menu itp.
   - Komunikaty od aplikacji

Sygnalizowanie zdarzenia

Sprzęt $\Leftrightarrow$ Serwer X Windows $\Leftrightarrow$ Menedżer okien $\Leftrightarrow$ Aplikacja

Związki między zdarzeniami

Jedno zdarzenie może powodować wystąpienie kolejnych zdarzeń.

Związki między zdarzeniami

Jedno zdarzenie może powodować wystąpienie kolejnych zdarzeń.

Przykład: ukrywanie okna

1. Zdarzenie – kliniecie myszą w obszarze przycisku ukrywania okna.
2. Serwer X Windows przekazuje informację o tym zdarzeniu do menedżera okien.
3. Menedżer okien inicjuje operację ukrywania okna. W jej wyniku inne okna mogą stać się widoczne.
4. Serwer X Windows wykrywa te sytuacje i traktuje je jako zdarzenia. Przekazuje informacje o nich do menedżera okien.
5. Menedżer okien przekazuje informacje o zdarzeniach do aplikacji korzystających z „odsłoniętych” okien.

Graficzny interfejs użytkownika

GUI (*ang. Graphical User Interface*)

Umożliwia użytkownikowi sterowanie komputerem (systemem operacyjnym i aplikacjami) za pośrednictwem podsystemu graficznego.

Graficzny interfejs użytkownika

GUI (*ang. Graphical User Interface*)

Umożliwia użytkownikowi sterowanie komputerem (systemem operacyjnym i aplikacjami) za pośrednictwem podsystemu graficznego.

Sterowanie z wykorzystaniem zdarzeń

Poczynania użytkownika „wytwarzają” zdarzenia (np. ruch myszą, wciśnięcie klawisza), które są interpretowane (na wielu poziomach) i skutkują przeprowadzeniem odpowiednich działań.

Graficzny interfejs użytkownika

GUI (*ang. Graphical User Interface*)

Umożliwia użytkownikowi sterowanie komputerem (systemem operacyjnym i aplikacjami) za pośrednictwem podsystemu graficznego.

Sterowanie z wykorzystaniem zdarzeń

Poczynania użytkownika „wytwarzają" zdarzenia (np. ruch myszą, wciśnięcie klawisza), które są interpretowane (na wielu poziomach) i skutkują przeprowadzeniem odpowiednich działań.

Jeśli każda akcja w systemie komputerowym jest skutkiem jakiegoś zdarzenia (np. związanego z GUI), to mamy do czynienia ze środowiskiem sterowanym zdarzeniami (*ang. event-driven environment*).

Komunikacja z użytkownikiem w GUI

Machanizm obsługi zdarzeń pozwala użytkownikowi wpływać na zachowanie się komputera (systemu operacyjnego i aplikacji), ale potrzebne jest też przekazywanie informacji zwrotnych (*ang. feedback*) użytkownikowi.

Komunikacja z użytkownikiem w GUI

Machanizm obsługi zdarzeń pozwala użytkownikowi wpływać na zachowanie się komputera (systemu operacyjnego i aplikacji), ale potrzebne jest też przekazywanie informacji zwrotnych (*ang. feedback*) użytkownikowi.

W GUI do tego celu wykorzystywana jest grafika – wyświetlana na ekranie reprezentuje informacje przeznaczone dla użytkownika.

Komunikacja z użytkownikiem w GUI

Machanizm obsługi zdarzeń pozwala użytkownikowi wpływać na zachowanie się komputera (systemu operacyjnego i aplikacji), ale potrzebne jest też przekazywanie informacji zwrotnych (*ang. feedback*) użytkownikowi.

W GUI do tego celu wykorzystywana jest grafika – wyświetlana na ekranie reprezentuje informacje przeznaczone dla użytkownika.

Może ona zawierać tekst, ale w GUI tekst jest wyświetlany „w towarzystwie" elementów graficznych (tzn. tekst jest traktowany jako część grafiki).

Podstawowe elementy grafiki (*ang. graphics primitives*)

Każdy rysunek jest zbudowanych z podstawowych elementów, takich jak pojedyncze punkty, linie proste, łuki, prostokąty itp.

Podstawowe elementy grafiki (*ang. graphics primitives*)

Każdy rysunek jest zbudowanych z podstawowych elementów, takich jak pojedyncze punkty, linie proste, łuki, prostokąty itp.

Zwykle sprzęt (tzw. karta graficzna) może być zaprogramowany do umieszczania takich podstawowych elementów w odpowiednich miejscach ekranu.

Podstawowe elementy grafiki (*ang. graphics primitives*)

Każdy rysunek jest zbudowanych z podstawowych elementów, takich jak pojedyncze punkty, linie proste, łuki, prostokąty itp.

Zwykle sprzęt (tzw. karta graficzna) może być zaprogramowany do umieszczania takich podstawowych elementów w odpowiednich miejscach ekranu.

Pamięć obrazu (*ang. video memory*)

Pamięć dostępna dla procesora karty graficznej (zwykle wchodzi w skład samej karty), zawierająca informacje o tym, co ma być wyświetlane.

Podstawowe elementy grafiki (*ang. graphics primitives*)

Każdy rysunek jest zbudowanych z podstawowych elementów, takich jak pojedyncze punkty, linie proste, łuki, prostokąty itp.

Zwykle sprzęt (tzw. karta graficzna) może być zaprogramowany do umieszczania takich podstawowych elementów w odpowiednich miejscach ekranu.

Pamięć obrazu (*ang. video memory*)

Pamięć dostępna dla procesora karty graficznej (zwykle wchodzi w skład samej karty), zawierająca informacje o tym, co ma być wyświetlane.

Podstawowe elementy grafiki są zapisywane (w odpowiedni sposób i we właściwej kolejności) w pamięci obrazu, dzięki czemu (później) stają się widoczne na ekranie wyświetlacza (np. monitora).

Piksele

Grafika rastrowa (*ang. raster graphics*)

Obraz wyświetlany jest w postaci siatki (*ang. raster*) złożonej z punktów o różnych kolorach.

Piksele

Grafika rastrowa (*ang. raster graphics*)

Obraz wyświetlany jest w postaci siatki (*ang. raster*) złożonej z punktów o różnych kolorach.

Piksel (*ang. picture element*)

Odpowiada pojedynczemu punktowi o ustalonym kolorze na wyświetlanym obrazie.

Piksele

Grafika rastrowa (*ang. raster graphics*)

Obraz wyświetlany jest w postaci siatki (*ang. raster*) złożonej z punktów o różnych kolorach.

Piksel (*ang. picture element*)

Odpowiada pojedynczemu punktowi o ustalonym kolorze na wyświetlanym obrazie.

RGB (*ang. Red, Green, Blue*)

Najczęściej wykorzystywany zapis pikseli w pamięci, w którym każdy piksel jest reprezentowany przez 3 B (24 b) danych określających natężenie poszczególnych składowych światła widzialnego (czerwonej, zielonej i niebieskiej). Dla każdej składowej można używać wartości od 0 do 255.

Bufor klatki i przyspieszanie grafiki 2D

Bufor klatki

Obszar pamięci obrazu reprezentujący całą (dwuwymiarową) scenę do wyświetlenia. Jego rozmiar zależy od rozdzielczości wyświetlacza i reprezentacji pikseli. Dla wyświetlacza 1280x1024 i pikseli w 24-bitowej reprezentacji RGB wynosi on $3932160\,B = 3840\,KiB = 3,75\,MiB$.

Bufor klatki i przyspieszanie grafiki 2D

Bufor klatki

Obszar pamięci obrazu reprezentujący całą (dwuwymiarową) scenę do wyświetlenia. Jego rozmiar zależy od rozdzielczości wyświetlacza i reprezentacji pikseli. Dla wyświetlacza 1280x1024 i pikseli w 24-bitowej reprezentacji RGB wynosi on $3932160\,B = 3840\,KiB = 3,75\,MiB$.

Manipulowanie pikselami

Praktycznie każda karta graficzna umożliwia modyfikowanie pojedynczych pikseli bezpośrednio w buforze klatki i odczytywanie zawartości bufora klatki.

Bufor klatki i przyspieszanie grafiki 2D

Bufor klatki

Obszar pamięci obrazu reprezentujący całą (dwuwymiarową) scenę do wyświetlenia. Jego rozmiar zależy od rozdzielczości wyświetlacza i reprezentacji pikseli. Dla wyświetlacza 1280x1024 i pikseli w 24-bitowej reprezentacji RGB wynosi on $3932160\,B = 3840\,KiB = 3,75\,MiB$.

Manipulowanie pikselami

Praktycznie każda karta graficzna umożliwia modyfikowanie pojedynczych pikseli bezpośrednio w buforze klatki i odczytywanie zawartości bufora klatki.

Przyspieszanie grafiki 2D (*ang. 2D graphics acceleration*)

Sprzętowe wspomaganie przez kartę graficzną operacji na zawartości bufora klatki, takich jak rysowanie linii prostych, łuków, prostokątów, wypełnianie wzorem (*ang. pattern*) itp.

Tworzenie rysunków poza buforem klatki

Piksmapa (*ang. pixmap*)

Odwzorowany w pamięci operacyjnej komputera fragment bufora klatki odpowiadający prostokątnemu obszarowi na ekranie.

Tworzenie rysunków poza buforem klatki

Piksmapa (*ang. pixmap*)

Odwzorowany w pamięci operacyjnej komputera fragment bufora klatki odpowiadający prostokątnemu obszarowi na ekranie.

Piksmapy można wykorzystywać do tworzenia ryskunków, podobnie jak bufor klatki, ale będą one wyświetlane dopiero po skopiowaniu zawartości piksmapy do bufora klatki.

Tworzenie rysunków poza buforem klatki

Piksmapa (*ang. pixmap*)

Odwzorowany w pamięci operacyjnej komputera fragment bufora klatki odpowiadający prostokątnemu obszarowi na ekranie.

Piksmapy można wykorzystywać do tworzenia ryskunków, podobnie jak bufor klatki, ale będą one wyświetlane dopiero po skopiowaniu zawartości piksmapy do bufora klatki.

Niektóre karty graficzne wspomagają sprzętowo kopiowanie piksmap, zarówno między buforem klatki i pamięcią główną, jak i w obrębie bufora klatki.

Tworzenie rysunków poza buforem klatki

Piksmapa (*ang. pixmap*)

Odwzorowany w pamięci operacyjnej komputera fragment bufora klatki odpowiadający prostokątnemu obszarowi na ekranie.

Piksmapy można wykorzystywać do tworzenia ryskunków, podobnie jak bufor klatki, ale będą one wyświetlane dopiero po skopiowaniu zawartości piksmapy do bufora klatki.

Niektóre karty graficzne wspomagają sprzętowo kopiowanie piksmap, zarówno między buforem klatki i pamięcią główną, jak i w obrębie bufora klatki.

Piksmapy pochodzące z jednego komputera nie muszą być poprawnie wyświetlane po umieszczeniu ich zawartości w buforze klatki karty graficznej na innym komputerze.

Grafika trójwymiarowa (3D)

Większość współczesnych kart graficznych jest konstruowana z myślą o grafice trójwymiarowej (3D).

Grafika trójwymiarowa (3D)

Większość współczesnych kart graficznych jest konstruowana z myślą o grafice trójwymiarowej (3D).

Pamięć obrazu jest wykorzystywana do zapisu trójwymiarowej sceny (w przestrzennym układzie współrzędnych), na podstawie której generowany jest obraz (może on składać się z wielu nakładających się na siebie części).

Grafika trójwymiarowa (3D)

Większość współczesnych kart graficznych jest konstruowana z myślą o grafice trójwymiarowej (3D).

Pamięć obrazu jest wykorzystywana do zapisu trójwymiarowej sceny (w przestrzennym układzie współrzędnych), na podstawie której generowany jest obraz (może on składać się z wielu nakładających się na siebie części).

Renderowanie (*ang. rendering*)

Operacja, w czasie której na podstawie sceny 3D zapisanej w pamięci obrazu tworzone są piksele, a następnie sygnały dla urządzeń wyświetlających. Obejmuje przekształcenia brył, nakładanie tekstur, wygładzanie krawędzi, dodawanie efektów oświetlenia, mgły, przezroczystości itp. oraz rzutowanie.

Grafika 3D w GUI

Przyspieszanie grafiki 3D (*ang. 3D graphics acceleration*)

Wspomaganie sprzętowe przez kartę graficzną przetwarzania scen 3D podczas renderowania.

Grafika 3D w GUI

Przyspieszanie grafiki 3D (*ang. 3D graphics acceleration*)

Wspomaganie sprzętowe przez kartę graficzną przetwarzania scen 3D podczas renderowania.

Współczesne karty graficzne sprawniej przyspieszają grafikę 3D, niż grafikę 2D, więc często bardziej opłaca się wykorzystywać pełne możliwości karty zamiast manipulować „tylko” zawartością bufora klatki.

Grafika 3D w GUI

Przyspieszanie grafiki 3D (*ang. 3D graphics acceleration*)

Wspomaganie sprzętowe przez kartę graficzną przetwarzania scen 3D podczas renderowania.

Współczesne karty graficzne sprawniej przyspieszają grafikę 3D, niż grafikę 2D, więc często bardziej opłaca się wykorzystywać pełne możliwości karty zamiast manipulować „tylko" zawartością bufora klatki.

Scalanie GUI (*ang. GUI compositing*)

Okna wykorzystywane przez aplikacje do komunikacji z użytkownikiem są traktowane jako elementy sceny 3D i renderowane z wykorzystaniem sprzętowego wspomagania ze strony karty graficznej. Następnie są one łączone w jedną „scenę" i prezentowane użytkownikowi.

Od czego zależy w jakim stopniu wykorzystujemy sprzęt

To, w jakim stopniu można będzie wykorzystywać możliwości sprzętu w zakresie „przyspieszania” tworzenia grafiki, zależy od sterowników w jądrze systemu i (dla GNU/Linuksa) serwera X Windows oraz menedżera okien.

Od czego zależy w jakim stopniu wykorzystujemy sprzęt

To, w jakim stopniu można będzie wykorzystywać możliwości sprzętu w zakresie „przyspieszania" tworzenia grafiki, zależy od sterowników w jądrze systemu i (dla GNU/Linuksa) serwera X Windows oraz menedżera okien.

Sterowniki karty graficznej w jądrze systemu i w serwerze X Windows muszą „wiedzieć" jak zaprogramować sprzęt do przeprowadzania poszczególnych operacji.

Od czego zależy w jakim stopniu wykorzystujemy sprzęt

To, w jakim stopniu można będzie wykorzystywać możliwości sprzętu w zakresie „przyspieszania" tworzenia grafiki, zależy od sterowników w jądrze systemu i (dla GNU/Linuksa) serwera X Windows oraz menedżera okien.

Sterowniki karty graficznej w jądrze systemu i w serwerze X Windows muszą „wiedzieć" jak zaprogramować sprzęt do przeprowadzania poszczególnych operacji.

Serwer X Windows udostępnia menedżerowi okien i, za jego pośrednictwem, aplikacjom, szereg standardowych funkcji, z których mogą one korzystać w celu przeprowadzania podstawowych operacji związanych z tworzeniem grafiki.

Rola biblioteki graficznej

Tworząc aplikację z reguły nie wiemy z góry na jakich systemach będzie ona uruchamiana:

- Nie wiemy jakie będą możliwości sprzętu.
- Nie wiemy jakie funkcje będzie udostępniał serwer X Windows.
- Nie wiemy jaki menedżer okien będzie używany.
- Nie wiemy na co będzie pozwalał menedżer okien.

Rola biblioteki graficznej

Tworząc aplikację z reguły nie wiemy z góry na jakich systemach będzie ona uruchamiana:

- Nie wiemy jakie będą możliwości sprzętu.
- Nie wiemy jakie funkcje będzie udostępniał serwer X Windows.
- Nie wiemy jaki menedżer okien będzie używany.
- Nie wiemy na co będzie pozwalał menedżer okien.

Biblioteka graficzna rozpoznaje możliwości podsystemu graficznego i dostosowuje do nich działanie funkcji oraz klas, z których mogą korzystać aplikacje.

Rola biblioteki graficznej

Tworząc aplikację z reguły nie wiemy z góry na jakich systemach będzie ona uruchamiana:

- Nie wiemy jakie będą możliwości sprzętu.
- Nie wiemy jakie funkcje będzie udostępniał serwer X Windows.
- Nie wiemy jaki menedżer okien będzie używany.
- Nie wiemy na co będzie pozwalał menedżer okien.

Biblioteka graficzna rozpoznaje możliwości podsystemu graficznego i dostosowuje do nich działanie funkcji oraz klas, z których mogą korzystać aplikacje.

Biblioteka graficzna pozwala korzystać z różnych podsystemów graficznych w jednolity sposób.

Biblioteki graficzne i zdarzenia

Biblioteki graficzne dostarczają narzędzi ułatwiających obsługę zdarzeń na poziomie aplikacji.

Biblioteki graficzne i zdarzenia

Biblioteki graficzne dostarczają narzędzi ułatwiających obsługę zdarzeń na poziomie aplikacji.

Widżet (*ang. widget, WIndow gaDGET*)

Obiekt reprezentujący podstawowy komponent GUI, taki jak okno, przycisk, suwak, rozwijane menu itp. W środowiskach MS Windows takie komponenty GUI nazywane są kontrolkami (*ang. control*).

Biblioteki graficzne i zdarzenia

Biblioteki graficzne dostarczają narzędzi ułatwiających obsługę zdarzeń na poziomie aplikacji.

Widżet (*ang. widget, WIndow gaDGET*)

Obiekt reprezentujący podstawowy komponent GUI, taki jak okno, przycisk, suwak, rozwijane menu itp. W środowiskach MS Windows takie komponenty GUI nazywane są kontrolkami (*ang. control*).

Każdemu widżetowi przypisuje się prostokątny obszar ekranu i zdarzenia związane z tym obszarem są interpretowane w standardowy sposób w kontekście funkcji spełnianej przez widżet (np. kliknięcie lewym przyciskiem myszy w obszarze przypisanym widżetowi reprezentującemu przycisk może być interpretowane jako wciśnięcie przycisku).

Podstawowe klasy z biblioteki *Qt*

`QApplication`

- Obiekt tej klasy reprezentuje program (aplikację).
- Inicjuje zasoby wykorzystywane przez bibliotekę.
- Do konstruktora przekazuje się argumenty funkcji `main()`.
- Wywołanie metody `exec()` uruchamia pętlę obsługi zdarzeń.

Podstawowe klasy z biblioteki *Qt*

`QApplication`

- Obiekt tej klasy reprezentuje program (aplikację).
- Inicjuje zasoby wykorzystywane przez bibliotekę.
- Do konstruktora przekazuje się argumenty funkcji `main()`.
- Wywołanie metody `exec()` uruchamia pętlę obsługi zdarzeń.

Pętla obsługi zdarzeń (*ang. event loop*)

Pętla, w której zdarzenia sygnalizowane przez różne części podsystemu graficznego są przetwarzane i przekazywane odpowiednim widżetom.

Podstawowe klasy z biblioteki *Qt*

`QApplication`

- Obiekt tej klasy reprezentuje program (aplikację).
- Inicjuje zasoby wykorzystywane przez bibliotekę.
- Do konstruktora przekazuje się argumenty funkcji `main()`.
- Wywołanie metody `exec()` uruchamia pętlę obsługi zdarzeń.

Pętla obsługi zdarzeń (*ang. event loop*)

Pętla, w której zdarzenia sygnalizowane przez różne części podsystemu graficznego są przetwarzane i przekazywane odpowiednim widżetom.

Obiekt klasy `QApplication` musi być utworzony przed jakimkolwiek innym obiektem związanym z GUI i może być tylko jeden.

Podstawowe klasy z biblioteki *Qt* (c. d.)

`QObject`

Od tej klasy pochodzi większość klas dostępnych w bibliotece *Qt*. Definiuje ona interfejs umożliwiający przekazywanie informacji o zdarzeniach od jednego komponentu GUI do drugiego (jego konstrukcja wykracza poza standard C++).

Podstawowe klasy z biblioteki *Qt* (c. d.)

`QObject`

Od tej klasy pochodzi większość klas dostępnych w bibliotece *Qt*. Definiuje ona interfejs umożliwiający przekazywanie informacji o zdarzeniach od jednego komponentu GUI do drugiego (jego konstrukcja wykracza poza standard C++).

`QPaintDevice`

Obiekty tej klasy reprezentują prostokątne powierzchnie, na których można rysować (np. prostokątne wycinki ekranu lub piksmapy).

Podstawowe klasy z biblioteki *Qt* (c. d.)

`QObject`

Od tej klasy pochodzi większość klas dostępnych w bibliotece *Qt*. Definiuje ona interfejs umożliwiający przekazywanie informacji o zdarzeniach od jednego komponentu GUI do drugiego (jego konstrukcja wykracza poza standard C++).

`QPaintDevice`

Obiekty tej klasy reprezentują prostokątne powierzchnie, na których można rysować (np. prostokątne wycinki ekranu lub piksmapy).

Z każdą taką powierzchnią jest związana prostokątna siatka współrzędnych (o wartościach całkowitych nieujemnych) i punktem o współrzędnych $[0, 0]$ w lewym górnym rogu. Każdy punkt tej siatki reprezentuje jeden piksel.

Podstawowe klasy z biblioteki *Qt* (c. d.)

`QWidget`

Klasa pochodna od `QObject` i `QPaintDevice`. Od tej klasy pochodzą wszystkie klasy reprezentujące typowe komponenty GUI. Własne komponenty GUI najłatwiej tworzy się poprzez definiowanie własnych klas pochodnych względem `QWidget`.

Podstawowe klasy z biblioteki *Qt* (c. d.)

`QWidget`

Klasa pochodna od `QObject` i `QPaintDevice`. Od tej klasy pochodzą wszystkie klasy reprezentujące typowe komponenty GUI. Własne komponenty GUI najłatwiej tworzy się poprzez definiowanie własnych klas pochodnych względem `QWidget`.

`QPainter`

Obiekty tej klasy służą do przeprowadzania podstawowych operacji graficznych (takich, jak rysowanie podstawowych elementów grafiki) na „powierzchniach” reprezentowanych przez obiekty klasy `QPaintDevice`.

Podstawowe klasy z biblioteki *Qt* (c. d.)

`QWidget`

Klasa pochodna od `QObject` i `QPaintDevice`. Od tej klasy pochodzą wszystkie klasy reprezentujące typowe komponenty GUI. Własne komponenty GUI najłatwiej tworzy się poprzez definiowanie własnych klas pochodnych względem `QWidget`.

`QPainter`

Obiekty tej klasy służą do przeprowadzania podstawowych operacji graficznych (takich, jak rysowanie podstawowych elementów grafiki) na „powierzchniach” reprezentowanych przez obiekty klasy `QPaintDevice`.

W celu narysowania czegoś tworzy się obiekt klasy `QPainter` związany z obiektem klasy `QPaintDevice` reprezentującym powierzchnię, na której ma powstać rysunek.

Najprostszy program wykorzystujący *Qt*

```
#include <QApplication>
#include <QLabel>

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

        QLabel hello("Hello world!");
        hello.resize(100, 30);

        hello.show();
        return app.exec();  // Start pętli obsługi zdarzeń
}
```

Najprostszy program wykorzystujący *Qt*

```
#include <QApplication>
#include <QLabel>

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

        QLabel hello("Hello world!");
        hello.resize(100, 30);

        hello.show();
        return app.exec();  // Start pętli obsługi zdarzeń
}
```

- Argumenty funkcji `main()` są przekazywane do konstruktora obiektu app (inicjalizacja struktur danych *Qt*).
- Obiekt `hello` reprezentuje etykietę (*ang. label*), czyli prostokątny wycinek ekranu z napisem.
- Metody `resize()` i `show()` pochodzą z klasy `QWidget`, która jest klasą bazową dla `QLabel`.

Metody `show()` i `resize()`

`show()`

Powoduje, że widżet reprezentowany przez obiekt, w kontekście którego jest wywoływana, staje się widoczny na ekranie (o ile wszystkie widżety odpowiadające obiektom nadrzędnym w stosunku do niego są widoczne), choć może być „schowany" np. „pod" jakimś oknem.

Metody `show()` i `resize()`

`show()`

Powoduje, że widżet reprezentowany przez obiekt, w kontekście którego jest wywoływana, staje się widoczny na ekranie (o ile wszystkie widżety odpowiadające obiektom nadrzędnym w stosunku do niego są widoczne), choć może być „schowany" np. „pod" jakimś oknem.

Jeżeli widżet nie ma obiektu nadrzędnego, to zostanie wyświetlony w oddzielnym oknie, tzn. wywołanie metody `show()` w jego kontekście spowoduje otworzenie nowego okna, którego zawartość będzie reprezentowana przez ten widżet.

Metody `show()` i `resize()`

`show()`

Powoduje, że widżet reprezentowany przez obiekt, w kontekście którego jest wywoływana, staje się widoczny na ekranie (o ile wszystkie widżety odpowiadające obiektom nadrzędnym w stosunku do niego są widoczne), choć może być „schowany" np. „pod" jakimś oknem.

Jeżeli widżet nie ma obiektu nadrzędnego, to zostanie wyświetlony w oddzielnym oknie, tzn. wywołanie metody `show()` w jego kontekście spowoduje otworzenie nowego okna, którego zawartość będzie reprezentowana przez ten widżet.

`resize()`

Zadaje szerokość i wysokość obszaru ekranu zajmowanego przez widżet (nie na stałe).

Kompilowanie programów używających *Qt*

Interfejs obsługi zdarzeń wykorzystywany przez *Qt* wykracza poza standard C++, więc programy używające *Qt* muszą być kompilowane w specjalny sposób.

Kompilowanie programów używających *Qt*

Interfejs obsługi zdarzeń wykorzystywany przez *Qt* wykracza poza standard C++, więc programy używające *Qt* muszą być kompilowane w specjalny sposób.

1. Tworzymy katalog o nazwie, która ma być nazwą pliku z programem.
2. W tym katalogu tworzymy (przynajmniej) plik o nazwie `main.cpp` zawierający funkcję `main()`. Kod źródłowy programu może być zapisany w wielu plikach (w tym katalogu).
3. W katalogu z plikami źródłowymi wykonujemy polecenia:

```
$ qmake -project
$ qmake
$ make
```

Kompilowanie programów używających *Qt* (c. d.)

`qmake -project`

Tworzy plik z rozszerzeniem `.pro` zawierający informacje m. in. o tym z jakich plików składa się program.

Kompilowanie programów używających *Qt* (c. d.)

`qmake -project`

Tworzy plik z rozszerzeniem `.pro` zawierający informacje m. in. o tym z jakich plików składa się program.

`qmake`

Przygotowuje plik `Makefile` dla polecenia `make`, zawierający instrukcje kompilacji programu.

Kompilowanie programów używających *Qt* (c. d.)

`qmake -project`

Tworzy plik z rozszerzeniem `.pro` zawierający informacje m. in. o tym z jakich plików składa się program.

`qmake`

Przygotowuje plik `Makefile` dla polecenia `make`, zawierający instrukcje kompilacji programu.

`make`

Wykonuje instrukcje zapisane w pliku `Makefile`. Poza uruchamianiem kompilatora C++ z odpowiednimi opcjami wykonuje programy przekształcające niestandardowy kod związany *Qt* na standardowy kod w C++.

Przesłanianie metody `paintEvent()`

W celu zaprogramowania tworzenia rysunku z użyciem *Qt* najłatwiej jest utworzyć klasę pochodną w stosunku do `QWidget`, w której zostanie przesłonięta (*ang. override*) metoda `paintEvent()`, np.:

```
class Obrazek : public QWidget {
  public:
    Obrazek(QWidget *parent);
  protected:
    void paintEvent(QPaintEvent *event);
};
```

Przesłanianie metody `paintEvent()`

W celu zaprogramowania tworzenia rysunku z użyciem *Qt* najłatwiej jest utworzyć klasę pochodną w stosunku do `QWidget`, w której zostanie przesłonięta (*ang. override*) metoda `paintEvent()`, np.:

```
class Obrazek : public QWidget {
  public:
    Obrazek(QWidget *parent);
  protected:
    void paintEvent(QPaintEvent *event);
};
```

`void paintEvent(QPaintEvent *event)`

Metoda wirtualna wywoływana wtedy, gdy obszar ekranu (lub piksmapy) przypisany obiektowi, w kontekście którego jest ona wywoływana, wymaga odświeżenia (*ang. refresh*).

Przesłanianie metody `paintEvent()`

W celu zaprogramowania tworzenia rysunku z użyciem *Qt* najłatwiej jest utworzyć klasę pochodną w stosunku do `QWidget`, w której zostanie przesłonięta (*ang. override*) metoda `paintEvent()`, np.:

```
class Obrazek : public QWidget {
  public:
    Obrazek(QWidget *parent);
  protected:
    void paintEvent(QPaintEvent *event);
};
```

`void paintEvent(QPaintEvent *event)`

Metoda wirtualna wywoływana wtedy, gdy obszar ekranu (lub piksmapy) przypisany obiektowi, w kontekście którego jest ona wywoływana, wymaga odświeżenia (*ang. refresh*).

Jej zadaniem jest utworzenie rysunku, który powinien się tam znajdować.

Treść metody `paintEvent()`

Argument metody `paintEvent()` określa rodzaj zdarzenia, które spowodowało konieczność odtworzenia rysunku oraz obszar, który ma być „przemalowany”.

Treść metody `paintEvent()`

Argument metody `paintEvent()` określa rodzaj zdarzenia, które spowodowało konieczność odtworzenia rysunku oraz obszar, który ma być „przemalowany”.

Treść metody `paintEvent()` zwykle zawiera instrukcje rysowania z użyciem obiektu klasy `QPainter`, np.:

```
void Obrazek::paintEvent(QPaintEvent *event)
{
    (void)event;  // Ignoruj argument (blokuje ostrzeżenie kompilatora).

    QPainter painter(this);

    // Narysuj prostokąt o lewym górnym rogu w punkcie (10, 10)
    // oraz szerokości 200 pikseli i wysokości 100 pikseli.
    painter.drawRect(10, 10, 200, 100);
}
```

Treść metody `paintEvent()`

Argument metody `paintEvent()` określa rodzaj zdarzenia, które spowodowało konieczność odtworzenia rysunku oraz obszar, który ma być „przemalowany”.

Treść metody `paintEvent()` zwykle zawiera instrukcje rysowania z użyciem obiektu klasy `QPainter`, np.:

```
void Obrazek::paintEvent(QPaintEvent *event)
{
    (void)event;  // Ignoruj argument (blokuje ostrzeżenie kompilatora).

    QPainter painter(this);

    // Narysuj prostokąt o lewym górnym rogu w punkcie (10, 10)
    // oraz szerokości 200 pikseli i wysokości 100 pikseli.
    painter.drawRect(10, 10, 200, 100);
}
```

Rysowanie polega na wywoływaniu metod dla obiektów klasy `QPainter`.

Konstruktory podklas `QWidget`

Konstruktor klasy pochodnej w stosunku do `QWidget` powinien przyjmować przynajmniej jeden argument będący wskaźnikiem do obiektu klasy `QWidget`, zawierającym adres obiektu nadrzędnego (*ang. parent*).

Konstruktory podklas `QWidget`

Konstruktor klasy pochodnej w stosunku do `QWidget` powinien przyjmować przynajmniej jeden argument będący wskaźnikiem do obiektu klasy `QWidget`, zawierającym adres obiektu nadrzędnego (*ang. parent*).

Wartość tego wskaźnika powinna być używana jako argument wywołania konstruktora z klasy `QWidget`, np.:

```
Obrazek::Obrazek(QWidget *parent) : QWidget(parent)
{
    resize(300, 200);  // Rozmiary zajmowanego obszaru (długość, wysokość).
}
```

Konstruktory podklas QWidget

Konstruktor klasy pochodnej w stosunku do QWidget powinien przyjmować przynajmniej jeden argument będący wskaźnikiem do obiektu klasy QWidget, zawierającym adres obiektu nadrzędnego (*ang. parent*).

Wartość tego wskaźnika powinna być używana jako argument wywołania konstruktora z klasy QWidget, np.:

```
Obrazek::Obrazek(QWidget *parent) : QWidget(parent)
{
    resize(300, 200);  // Rozmiary zajmowanego obszaru (długość, wysokość).
}
```

Wskaźnik ten może mieć domyślną (*ang. default*) wartość NULL, np.:

```
class Obrazek : public QWidget {
  public:
    Obrazek(QWidget *parent = NULL);
    ... // Pola i metody.
};
```

Sygnalizacja zdarzeń przez obiekty

Jeżeli w obrębie widżetu ma miejsce zdarzenie (np. „użytkownik kliknął myszą na grafikę reprezentującą przycisk"), to (w większości przypadków) powinno być ono zasygnalizowane innym obiektom (np. wykonującym działanie, które ma następować po naciśnięciu tego przycisku).

Sygnalizacja zdarzeń przez obiekty

Jeżeli w obrębie widżetu ma miejsce zdarzenie (np. „użytkownik kliknął myszą na grafikę reprezentującą przycisk"), to (w większości przypadków) powinno być ono zasygnalizowane innym obiektom (np. wykonującym działanie, które ma następować po naciśnięciu tego przycisku).

Kod realizujący powiadamianie o zdarzeniu nie powinien zależeć od tego, które obiekty mają być powiadamiane (jeśli by tak było, to ten kod w ogólności nie mógłby być powtórnie wykorzystany).

Sygnalizacja zdarzeń przez obiekty

Jeżeli w obrębie widżetu ma miejsce zdarzenie (np. „użytkownik kliknął myszą na grafikę reprezentującą przycisk"), to (w większości przypadków) powinno być ono zasygnalizowane innym obiektom (np. wykonującym działanie, które ma następować po naciśnięciu tego przycisku).

Kod realizujący powiadamianie o zdarzeniu nie powinien zależeć od tego, które obiekty mają być powiadamiane (jeśli by tak było, to ten kod w ogólności nie mógłby być powtórnie wykorzystany).

Kod implementujący reakcje na zdarzenia nie powinien zależeć od tego, z jakiego obiektu sygnalizowane jest zdarzenie (z tej samej przyczyny).

Sloty

Slot

Metoda implementująca reakcje na zdarzenia określonego rodzaju. Może także być wywoływana „synchronicznie” (jak każda inna metoda).

Sloty

Slot

Metoda implementująca reakcje na zdarzenia określonego rodzaju. Może także być wywoływana „synchronicznie” (jak każda inna metoda).

Slot implementuje się tak, jak każdą inną metodę, lecz musi on być specjalnie zadeklarowany w definicji klasy, do której należy, np.:

```
class Obrazek : public QWidget {
    Q_OBJECT

  private:
    int n;

  protected:
    void paintEvent(QPaintEvent *event);

    ...  // Inne pola i metody.

  public slots:
    void zmiana(int lw);  // Ta metoda jest slotem.
};
```

Sygnały

Sygnał (*ang. signal*)

Deklaracja możliwości (*ang. ability*) informowania o zdarzeniach w określony sposób.

Sygnały

Sygnał (*ang. signal*)

Deklaracja możliwości (*ang. ability*) informowania o zdarzeniach w określony sposób.

Sygnały definiuje się tak, jak nagłówki metod nie zwracających wyniku („typu" `void`), ale nie są one metodami, np.:

```
class Input : public QWidget {
    Q_OBJECT

  private:
    QLineEdit *edit;

    ...  // Inne pola i metody.

  signals:
    void newValue(int);  // Sygnał.
};
```

Łączenie slotów z sygnałami

Reakcje obiektów na zdarzenia sygnalizowane przez inne obiekty (lub za pośrednictwem innych obiektów) programuje się poprzez tworzenie połączeń (*ang. connection*) sygnałów ze slotami.

Łączenie slotów z sygnałami

Reakcje obiektów na zdarzenia sygnalizowane przez inne obiekty (lub za pośrednictwem innych obiektów) programuje się poprzez tworzenie połączeń (*ang. connection*) sygnałów ze slotami.

`QObject::connect()`

Metoda statyczna w klasie `QObject` (tzn. metoda w klasie `QObject`, która może być wywoływana bez kontekstu), służąca do tworzenia połączeń sygnałów ze slotami.

Łączenie slotów z sygnałami

Reakcje obiektów na zdarzenia sygnalizowane przez inne obiekty (lub za pośrednictwem innych obiektów) programuje się poprzez tworzenie połączeń (*ang. connection*) sygnałów ze slotami.

`QObject::connect()`

Metoda statyczna w klasie `QObject` (tzn. metoda w klasie `QObject`, która może być wywoływana bez kontekstu), służąca do tworzenia połączeń sygnałów ze slotami.

Utworzenie połączenia między sygnału `newValue(int)` z obiektu o adresie we wskaźniku `pole` ze slotem `zmiana(int)` z obiektem o adresie we wkaźniku `rysunek`:

```
QObject::connect(pole, SIGNAL(newValue(int)), rysunek, SLOT(zmiana(int)));
```

Emitowanie sygnałów

Sygnalizacja zdarzenia polega na wyemitowaniu określonego sygnału przez obiekt (sygnał ten musi być wcześniej zdefiniowany).

Emitowanie sygnałów

Sygnalizacja zdarzenia polega na wyemitowaniu określonego sygnału przez obiekt (sygnał ten musi być wcześniej zdefiniowany).

`emit`

Pseudoinstrukcja służąca do emitowania sygnałów, np.:

```
void Input::getValue(void)
{
    int n;
    bool ok;

    n = edit->text().toInt(&ok);  // Przekształć zawartość pola tekstowego na int.
    if (ok)  // ok ma wartość true, jeżeli przekształcenie było udane.
        emit newValue(n);  // Wyemituj sygnał newValue(n).
}
```

Emitowanie sygnałów

Sygnalizacja zdarzenia polega na wyemitowaniu określonego sygnału przez obiekt (sygnał ten musi być wcześniej zdefiniowany).

`emit`

Pseudoinstrukcja służąca do emitowania sygnałów, np.:

```
void Input::getValue(void)
{
    int n;
    bool ok;

    n = edit->text().toInt(&ok);  // Przekształć zawartość pola tekstowego na int.
    if (ok)  // ok ma wartość true, jeżeli przekształcenie było udane.
        emit newValue(n);  // Wyemituj sygnał newValue(n).
}
```

Wszystkie sloty połączone z sygnałem `newValue()` z bieżącego obiektu zostaną wykonane z argumentem `n` (kolejność nie ustalona).

Reguły łączenia sygnałów ze slotami

W celu połączenia sygnału ze slotem sygnatura (*ang. signature*) sygnału musi odpowiadać sygnaturze slotu.

Reguły łączenia sygnałów ze slotami

W celu połączenia sygnału ze slotem sygnatura (*ang. signature*) sygnału musi odpowiadać sygnaturze slotu.

To znaczy, że typy danych argumentów metody zadeklarowanej jako slot muszą odpowiadać typom danych w pseudonagłówku użytym do zdefiniowania sygnału.

Reguły łączenia sygnałów ze slotami

W celu połączenia sygnału ze slotem sygnatura (*ang. signature*) sygnału musi odpowiadać sygnaturze slotu.

To znaczy, że typy danych argumentów metody zadeklarowanej jako slot muszą odpowiadać typom danych w pseudonagłówku użytym do zdefiniowania sygnału.

```
class Input : public QWidget {
    Q_OBJECT

  private:
    QLineEdit *edit;
    ...  // Inne pola i metody.

  signals:
    void newValue(int);
    // Ten sygnał można połączyć ze slotem
    // o jednym argumencie typu int.
};
```

```
class Obrazek : public QWidget {
    Q_OBJECT

    ...  // Pola i metody.

  public slots:
    // Ten slot można połączyć z newValue(int).
    void zmiana(int lw);
    // Tego slotu nie można połączyć z newValue(int).
    void zmianaPozycji(int x, int y);
};
```

Reguły łączenia sygnałów ze slotami (c. d.)

Metoda zadeklarowana jako slot może mieć mniej argumentów, niż jest „miejsc" na argumenty w pseudonagłówku użytym do zdefiniowania sygnału, który ma być z nią połączony.

Reguły łączenia sygnałów ze slotami (c. d.)

Metoda zadeklarowana jako slot może mieć mniej argumentów, niż jest „miejsc" na argumenty w pseudonagłówku użytym do zdefiniowania sygnału, który ma być z nią połączony.

Jeżeli tak jest, to typy danych argumentów tej metody (slotu) muszą dokładnie odpowiadać typom danych w pseudonagłówku użytym do zdefiniowania sygnału, znajdującym się na tych samych pozycjach.

Reguły łączenia sygnałów ze slotami (c. d.)

Metoda zadeklarowana jako slot może mieć mniej argumentów, niż jest „miejsc" na argumenty w pseudonagłówku użytym do zdefiniowania sygnału, który ma być z nią połączony.

Jeżeli tak jest, to typy danych argumentów tej metody (slotu) muszą dokładnie odpowiadać typom danych w pseudonagłówku użytym do zdefiniowania sygnału, znajdującym się na tych samych pozycjach.

Wartości używane przy emitowaniu sygnału, którym nie odpowiadają żadne argumenty slotu połączonego z tym sygnałem, są odrzucane podczas wywoływania tego slotu.

Reguły łączenia sygnałów ze slotami (c. d.)

Metoda zadeklarowana jako slot może mieć mniej argumentów, niż jest „miejsc" na argumenty w pseudonagłówku użytym do zdefiniowania sygnału, który ma być z nią połączony.

Jeżeli tak jest, to typy danych argumentów tej metody (slotu) muszą dokładnie odpowiadać typom danych w pseudonagłówku użytym do zdefiniowania sygnału, znajdującym się na tych samych pozycjach.

Wartości używane przy emitowaniu sygnału, którym nie odpowiadają żadne argumenty slotu połączonego z tym sygnałem, są odrzucane podczas wywoływania tego slotu.

Na przykład sygnał zdefiniowany jako `void newValue(int);` można łączyć ze slotami bez argumentów.

Predefiniowane sygnały i sloty

Wiele klas dostępnych w bibliotece *Qt* zawiera definicje sygnałów i slotów gotowych do wykorzystania.

Predefiniowane sygnały i sloty

Wiele klas dostępnych w bibliotece *Qt* zawiera definicje sygnałów i slotów gotowych do wykorzystania.

Przykład

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

    QPushButton koniec("Koniec");
    koniec.resize(100, 50);
    koniec.show();

    QObject::connect(&koniec, SIGNAL(clicked()), &app, SLOT(quit()));

    return app.exec();
}
```

Predefiniowane sygnały i sloty

Wiele klas dostępnych w bibliotece *Qt* zawiera definicje sygnałów i slotów gotowych do wykorzystania.

Przykład

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

    QPushButton koniec("Koniec");
    koniec.resize(100, 50);
    koniec.show();

    QObject::connect(&koniec, SIGNAL(clicked()), &app, SLOT(quit()));

    return app.exec();
}
```

Sygnał `clicked()` z obiektu `koniec` łączymy ze slotem `quit()` z obiektu `app`, co powoduje zakończenie działania programu po naciśnięciu przycisku.

Łączenie sygnałów z innymi sygnałami

Metoda `QObject::connect()` może być użyta do połączenia sygnału bezpośrednio z innym sygnałem.

Łączenie sygnałów z innymi sygnałami

Metoda `QObject::connect()` może być użyta do połączenia sygnału bezpośrednio z innym sygnałem.

Przykład

Utworzenie połączenia sygnału `valueChanged(int)` z obiektu pod adresem slider z sygnałem `valueChanged(int)` z obiektu pod adresem this.

```
QObject::connect(slider, SIGNAL(valueChanged(int)), this, SIGNAL(valueChanged(int)));
```

Łączenie sygnałów z innymi sygnałami

Metoda `QObject::connect()` może być użyta do połączenia sygnału bezpośrednio z innym sygnałem.

Przykład

Utworzenie połączenia sygnału `valueChanged(int)` z obiektu pod adresem slider z sygnałem `valueChanged(int)` z obiektu pod adresem this.

```
QObject::connect(slider, SIGNAL(valueChanged(int)), this, SIGNAL(valueChanged(int)));
```

Wyemitowanie sygnału `valueChanged(int)` przez obiekt pod adresem slider będzie automatycznie powodować wyemitowanie sygnału `valueChanged(int)` przez obiekt pod adresem this.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Metody w klasach są specjalnym rodzajem funkcji.

Rola funkcji (procedur) w programach

Funkcje (procedury) służą do tego, aby kod, który jest (lub może być) wielokrotnie powtarzany w programie, można było zapisać tylko raz, a później odwoływać się do niego w określony sposób (z pomocą nazwy funkcji).

Na ogół wygodnie jest używać funkcji (procedur) do logicznej separacji części kodu przeznaczonych do różnych zadań.

Metody w klasach są specjalnym rodzajem funkcji.

Aby funkcje mogły spełniać swoje zadania, muszą operować na określonych danych, które trzeba przekazywać do kodu zapisanego w postaci funkcji.

Przykład – dodawanie macierzy 2x2

```
class Matrix22 {
  public:
    double w1k1, w1k2, w2k1, w2k2;
    Matrix22(void): w1k1(0), w1k2(0), w2k1(0), w2k2(0) {}
    Matrix22(double r): w1k1(r), w1k2(0), w2k1(0), w2k2(r) {}
    Matrix22(double a, double b, double c, double d):
             w1k1(a), w1k2(b), w2k1(c), w2k2(d) {}
    double det(void) const { return w1k1*w2k2 - w1k2*w2k1; }
    Matrix22 operator +(Mattix22 m)
    {
        return Matrix22(w1k1 + m.w1k1, w1k2 + m.w1k2,
                        w2k1 + m.w2k1, w2k2 + m.w2k2);
    }
};

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    cout << C.w1k1 << " " << C.w1k2 << " " << C.w2k1 << " " << C.w2k2 << endl;
    A = C + B;
    cout << A.w1k1 << " " << A.w1k2 << " " << A.w2k1 << " " << A.w2k2 << endl;
    B = A + C;
    cout << B.w1k1 << " " << B.w1k2 << " " << B.w2k1 << " " << B.w2k2 << endl;
    return 0;
}
```

Przekazywanie argumentu przez wartość

Aby uniknąć wielokrotnego powtarzania tego samego kodu wprowadzamy funkcję `drukuj()`:

```
void drukuj(Matrix22 M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    drukuj(C);
    A = C + B;
    drukuj(A);
    B = A + C;
    drukuj(B);
    return 0;
}
```

Przekazywanie argumentu przez wartość

Aby uniknąć wielokrotnego powtarzania tego samego kodu wprowadzamy funkcję `drukuj()`:

```
void drukuj(Matrix22 M)
{
    cout << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}

int main()
{
    Matrix22 A(1), B(2), C;

    C = A + B;
    drukuj(C);
    A = C + B;
    drukuj(A);
    B = A + C;
    drukuj(B);
    return 0;
}
```

Przy każdym wywołaniu `drukuj()` tworzona jest kopia obiektu przekazywanego jako argument (używana w funkcji pod nazwą `M`).

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Koszty te nie byłyby ponoszone, gdyby można było „powiedzieć” funkcji, że ma posługiwać się tym samym obiektem, który jest przekazywany jako argument (bez kopiowania go).

Wady przekazywania argumentów funkcji przez wartość

Każdym wywołanie funkcji `drukuj()` wymaga

1. zarezerwowania pamięci na 4 zmienne typu `double`,
2. przeprowadzenia operacji kopiowania 4 wartości typu `double` (poświęcamy czas).

Ponosimy zatem koszty związane z użyciem funkcji.

Koszty te nie byłyby ponoszone, gdyby można było „powiedzieć" funkcji, że ma posługiwać się tym samym obiektem, który jest przekazywany jako argument (bez kopiowania go).

Do tego celu służą referencje (*ang. reference*).

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy funkcja będzie używać tego samego obiektu, który został przekazany jako argument (bez kopiowania go), ale pod inną nazwą.

Przekazywanie argumentu przez referencję

Aby uniknąć kopiowania obiektu przekazywanego jako argument do funkcji można użyć referencji:

```
void drukuj(Matrix22& M)
{
    cout << M.w1k1 << " " << M.w1k2 << " "
         << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy funkcja będzie używać tego samego obiektu, który został przekazany jako argument (bez kopiowania go), ale pod inną nazwą.

Wówczas modyfikacje tego obiektu przez funkcję zostaną zachowane i będą „widoczne" po zakończeniu wykonywania jej.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Ma to sens jedynie w przypadku obiektów, o których wiadomo, że będą istniały po zakończeniu wykonywania funkcji zwracających referencje do nich (tzn. nie wolno zwracać referencji do obiektów będących zmiennymi lokalnymi w funkcji zwracającej referencję do obiektu).

Zwracanie referencji przez funkcje

Referencje mogą być zwracane przez funkcje (lub metody) jako wyniki.

Coś takiego oznacza, że funkcja przekazuje kontrolę nad obiektem, którym posługiwała się, do funkcji wywołującej ją.

Ma to sens jedynie w przypadku obiektów, o których wiadomo, że będą istniały po zakończeniu wykonywania funkcji zwracających referencje do nich (tzn. nie wolno zwracać referencji do obiektów będących zmiennymi lokalnymi w funkcji zwracającej referencję do obiektu).

```
ostream& drukuj(ostream& out, Matrix22& M)
{
    out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
    return out;  // Przekazanie kontroli nad (obiektem reprezentowanym przez) out z powrotem.
}
```

Zwracanie referencji przez funkcje (c. d.)

Zapis funkcji `drukuj()` z użyciem referencji pozwala na wykonanie następującej operacji bez kopiowania obiektów (najpierw drukowane są elementy macierzowe `A`, później – `B`, a na końcu – `C`):

```
drukuj(drukuj(drukuj(cout, A), B), C);
```

Zwracanie referencji przez funkcje (c. d.)

Zapis funkcji `drukuj()` z użyciem referencji pozwala na wykonanie następującej operacji bez kopiowania obiektów (najpierw drukowane są elementy macierzowe `A`, później – `B`, a na końcu – `C`):

```
drukuj(drukuj(drukuj(cout, A), B), C);
```

Można zapisać `drukuj()` w postaci operatora <<:

```
ostream& operator <<(ostream& out, Matrix22& M)
{
    return out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}
```

Zwracanie referencji przez funkcje (c. d.)

Zapis funkcji `drukuj()` z użyciem referencji pozwala na wykonanie następującej operacji bez kopiowania obiektów (najpierw drukowane są elementy macierzowe `A`, później – `B`, a na końcu – `C`):

```
drukuj(drukuj(drukuj(cout, A), B), C);
```

Można zapisać `drukuj()` w postaci operatora <<:

```
ostream& operator <<(ostream& out, Matrix22& M)
{
    return out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}
```

Wtedy drukowanie elementów macierzowych `A`, `B` i `C` (kolejno) można zapisać jako:

```
cout << A << B << C;
```

Kiedy referencje sprawiają problemy

Funkcja `operator <<()` z poprzedniego slajdu nie nadaje się do przeprowadzenia następującej operacji:

```
cout << (A + B);
```

Kiedy referencje sprawiają problemy

Funkcja `operator <<()` z poprzedniego slajdu nie nadaje się do przeprowadzenia następującej operacji:

```
cout << (A + B);
```

Jest tak dlatego, że metoda `operator +()` z klasy `Matrix22` zwraca wynik przez wartość i wymaga skopiowania go do jakiegoś istniejącego obiektu (tzn. obiekt tworzony podczas wykonywania instrukcji `return` w tej metodzie istnieje tylko do czasu skopiowania go do innego obiektu).

Kiedy referencje sprawiają problemy

Funkcja `operator <<()` z poprzedniego slajdu nie nadaje się do przeprowadzenia następującej operacji:

```
cout << (A + B);
```

Jest tak dlatego, że metoda `operator +()` z klasy `Matrix22` zwraca wynik przez wartość i wymaga skopiowania go do jakiegoś istniejącego obiektu (tzn. obiekt tworzony podczas wykonywania instrukcji `return` w tej metodzie istnieje tylko do czasu skopiowania go do innego obiektu).

Rozwiązanie tego problemu wymaga przekazywania `M` do funkcji `operator <<()` przez wartość:

```
ostream& operator <<(ostream& out, Matrix22 M)
{
    return out << M.w1k1 << " " << M.w1k2 << " " << M.w2k1 << " " << M.w2k2 << endl;
}
```

Do czego są potrzebne wskaźniki (w C++)

W C++ wskaźniki (tzn. zmienne zawierające adresy innych zmiennych, np. obiektów) są *głównie* potrzebne do operowania zmiennymi tworzonymi na żądanie (dynamicznie) z pomocą `new`.

Do czego są potrzebne wskaźniki (w C++)

W C++ wskaźniki (tzn. zmienne zawierające adresy innych zmiennych, np. obiektów) są *głównie* potrzebne do operowania zmiennymi tworzonymi na żądanie (dynamicznie) z pomocą `new`.

Zmienne tworzone na żądanie (z pomocą `new`) są „anonimowe” (tzn. nie mają nazw), więc ich położenie w pamięci jest znane tylko dzięki adresom przechowywanym we wskaźnikach.

Do czego są potrzebne wskaźniki (w C++)

W C++ wskaźniki (tzn. zmienne zawierające adresy innych zmiennych, np. obiektów) są *głównie* potrzebne do operowania zmiennymi tworzonymi na żądanie (dynamicznie) z pomocą `new`.

Zmienne tworzone na żądanie (z pomocą `new`) są „anonimowe” (tzn. nie mają nazw), więc ich położenie w pamięci jest znane tylko dzięki adresom przechowywanym we wskaźnikach.

Przypomnienie

Wynik zwracany przez `new` jest adresem zmiennej (lub tablicy) utworzonej na żądanie (dynamicznie) lub ma wartość `NULL` (czyli 0), jeżeli zmienna nie mogła być utworzona.

Wskazywane zmienne

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` oznacza referencję do zmiennej, której adres jest wartością `wsk`. Nazywana się ją zmienną wskazywaną przez `wsk` lub zmienną pod adresem `wsk`.

Wskazywane zmienne

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` oznacza referencję do zmiennej, której adres jest wartością `wsk`. Nazywana się ją zmienną wskazywaną przez `wsk` lub zmienną pod adresem `wsk`.

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` może zastępować nazwę zmiennej we wszystkich sytuacjach, w których może ona być użyta, a w szczególności:

- po lewej stronie operatorów przypisania i modyfikacji,
- przy przekazywaniu argumentu do funkcji przez referencję,
- przy zwracaniu referencji przez funkcję.

Wskazywane zmienne

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` oznacza referencję do zmiennej, której adres jest wartością `wsk`. Nazywana się ją zmienną wskazywaną przez `wsk` lub zmienną pod adresem `wsk`.

Jeżeli `wsk` jest wskaźnikiem, to `*wsk` może zastępować nazwę zmiennej we wszystkich sytuacjach, w których może ona być użyta, a w szczególności:

- po lewej stronie operatorów przypisania i modyfikacji,
- przy przekazywaniu argumentu do funkcji przez referencję,
- przy zwracaniu referencji przez funkcję.

Operator obliczania adresu &

Dla zmiennej o nazwie `var` symbol `&var` oznacza adres tej zmiennej. Zatem `*(&var)` oznacza to samo, co `var`.

Wskaźnikowe typy danych

Wszystkie wskaźniki mają jednakowe rozmiary, ale przypisuje się im typy danych określające typy danych dla wskazywanych zmiennych.

Wskaźnikowe typy danych

Wszystkie wskaźniki mają jednakowe rozmiary, ale przypisuje się im typy danych określające typy danych dla wskazywanych zmiennych.

Nazwa wskaźnikowego typu danych składa się z nazwy typu danych dla wskazywanych zmiennych i symblu * (rozdzielnych znakami przerwy), np.:

`(int *)` – wskazywane zmienne są typu `int`.

`(double *)` – wskazywane zmienne są typu `double`.

`(Matrix22 *)` – wskazywane zmienne są obiektami klasy `Matrix22`.

Wskaźnikowe typy danych

Wszystkie wskaźniki mają jednakowe rozmiary, ale przypisuje się im typy danych określające typy danych dla wskazywanych zmiennych.

Nazwa wskaźnikowego typu danych składa się z nazwy typu danych dla wskazywanych zmiennych i symblu * (rozdzielnych znakami przerwy), np.:

`(int *)` – wskazywane zmienne są typu `int`.

`(double *)` – wskazywane zmienne są typu `double`.

`(Matrix22 *)` – wskazywane zmienne są obiektami klasy `Matrix22`.

W deklaracjach wskaźników nazwy odpowiadających im typów danych występują bez nawiasów, np.:

```
int *wsk;  // Wskaźnik zawierający adresy zmiennych typu int.
Matrix22 *ptr;  // Wskaźnik zawierający adresy obiektów klasy Matrix22.
```

Wskaźnikowe typy danych i wskazywane zmienne

Dzięki wskaźnikowym typom danych wiadomo, co oznacza `*wsk` dla danego wskaźnika `wsk`.

Wskaźnikowe typy danych i wskazywane zmienne

Dzięki wskaźnikowym typom danych wiadomo, co oznacza `*wsk` dla danego wskaźnika `wsk`.

`*wsk`

Zmienna typu określonego przez typ danych wskaźnika `wsk` znajdująca się pod adresem będącym wartością `wsk`.

Wskaźnikowe typy danych i wskazywane zmienne

Dzięki wskaźnikowym typom danych wiadomo, co oznacza `*wsk` dla danego wskaźnika `wsk`.

`*wsk`

Zmienna typu określonego przez typ danych wskaźnika `wsk` znajdująca się pod adresem będącym wartością `wsk`.

Operator ->

Jeżeli zmienna wskazywana przez wskaźnik `wsk` jest obiektem (tzn. jej typ danych jest klasą), to operator -> pozwala na odwoływanie się do pól i wywoływanie metod w kontekście tej zmiennej, np.:

```
Matrix22 *ptr;  // Wskaźnik zawierający adresy obiektów klasy Matrix22.

ptr = new Matrix22(1);
ptr->w1k2 = -1;  // Odwołanie do pola w1k2 obiektu wskazywanego przez ptr.
cout << ptr->det();  // Wywołanie metody det() dla obiektu wskazywanego przez ptr.
```

Stałe wskaźnikowe i czyste adresy

Stała wskaźnikowa

Adres miejsca w pamięci skojarzony z typem danych zmiennej znajdującej się pod tym adresem.

Stałe wskaźnikowe i czyste adresy

Stała wskaźnikowa

Adres miejsca w pamięci skojarzony z typem danych zmiennej znajdującej się pod tym adresem.

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest stałą wskaźnikową.

Stałe wskaźnikowe i czyste adresy

Stała wskaźnikowa

Adres miejsca w pamięci skojarzony z typem danych zmiennej znajdującej się pod tym adresem.

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest stałą wskaźnikową.

Czysty adres

Adres miejsca w pamięci nie skojarzony z typem danych zmiennej.

Stałe wskaźnikowe i czyste adresy

Stała wskaźnikowa

Adres miejsca w pamięci skojarzony z typem danych zmiennej znajdującej się pod tym adresem.

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest stałą wskaźnikową.

Czysty adres

Adres miejsca w pamięci nie skojarzony z typem danych zmiennej.

`(void *)`

Wskaźnikowy typ danych reprezentujący czyste adresy. Dla wskaźników tego typu operacje `*wsk` i `wsk->` nie mają sensu.

Stałe wskaźnikowe i czyste adresy

Stała wskaźnikowa

Adres miejsca w pamięci skojarzony z typem danych zmiennej znajdującej się pod tym adresem.

Dla dowolnej zmiennej `var` wynik wyrażenia `&var` jest stałą wskaźnikową.

Czysty adres

Adres miejsca w pamięci nie skojarzony z typem danych zmiennej.

`(void *)`

Wskaźnikowy typ danych reprezentujący czyste adresy. Dla wskaźników tego typu operacje `*wsk` i `wsk->` nie mają sensu.

Czyste adresy można przekształcać w stałe wskaźnikowe i odwrotnie.

Wskaźniki i funkcje

Wskaźniki mogą być argumentami funkcji, np.:

```
void drukuj(Matrix22 *M)
{
    cout << M->w1k1 << " " << M->w1k2 << " " << M->w2k1 << " " << M->w2k2 << endl;
}
```

Wskaźniki i funkcje

Wskaźniki mogą być argumentami funkcji, np.:

```
void drukuj(Matrix22 *M)
{
    cout << M->w1k1 << " " << M->w1k2 << " " << M->w2k1 << " " << M->w2k2 << endl;
}
```

W ten sposób unikamy kopiowania zmiennych przy przekazywaniu ich do funkcji (oszczędność czasu i miejca, podobnie jak przy posługiwaniu się referencjami).

Wskaźniki i funkcje

Wskaźniki mogą być argumentami funkcji, np.:

```
void drukuj(Matrix22 *M)
{
    cout << M->w1k1 << " " << M->w1k2 << " " << M->w2k1 << " " << M->w2k2 << endl;
}
```

W ten sposób unikamy kopiowania zmiennych przy przekazywaniu ich do funkcji (oszczędność czasu i miejca, podobnie jak przy posługiwaniu się referencjami).

Jednak w tym celu musimy znać adres zmiennej, którą funkcja ma posługiwać się lub obliczyć go, np.:

```
Matrix22 *wsk;

wsk = new Matrix22(1);
drukuj(wsk);  // Adres zmiennej jest wartością wsk.
```

```
Matrix22 M(1);

drukuj(&M);  // Jawnie obliczamy adres M.
```

Zgodność typów danych dla wskaźników

Kompilator C++ ostrzega w przypadkach, w których wartość wskaźnikowa jest przypisywana wskaźnikowi o niewłaściwym typie danych, np.:

```
double *wsk;

wsk = double[100];
drukuj(wsk);  // UWAGA! Oczekiwanym typem danych jest (Matrix22 *)!
```

Zgodność typów danych dla wskaźników

Kompilator C++ ostrzega w przypadkach, w których wartość wskaźnikowa jest przypisywana wskaźnikowi o niewłaściwym typie danych, np.:

```
double *wsk;

wsk = double[100];
drukuj(wsk);  // UWAGA! Oczekiwanym typem danych jest (Matrix22 *)!
```

Można zastosować jawną konwersję typów danych w celu wskazania kompilatorowi, że wiemy co robimy, np.:

```
drukuj((Matrix22 *)wsk);
```

Zgodność typów danych dla wskaźników

Kompilator C++ ostrzega w przypadkach, w których wartość wskaźnikowa jest przypisywana wskaźnikowi o niewłaściwym typie danych, np.:

```
double *wsk;

wsk = double[100];
drukuj(wsk);  // UWAGA! Oczekiwanym typem danych jest (Matrix22 *)!
```

Można zastosować jawną konwersję typów danych w celu wskazania kompilatorowi, że wiemy co robimy, np.:

```
drukuj((Matrix22 *)wsk);
```

Jawna konwersja wskaźnikowych typów danych nie jest potrzebna, jeżeli docelowy wskaźnik jest typu `(void *)` lub przypisujemy czysty adres wskaźnikowi dowolnego typu.

Wskaźniki i referencje

Wskaźniki są zmiennymi, a referencje nie.

Wskaźniki i referencje

Wskaźniki są zmiennymi, a referencje nie.

Użycie referencji oznacza przekazanie kontroli nad obiektem, natomiast wskaźnik określa położenie obiektu w pamięci.

Wskaźniki i referencje

Wskaźniki są zmiennymi, a referencje nie.

Użycie referencji oznacza przekazanie kontroli nad obiektem, natomiast wskaźnik określa położenie obiektu w pamięci.

Dlatego wskaźniki mogą być polami obiektów (tzn. można stworzyć klasę, w której część pól będzie wskaźnikami), a referencje – nie.

Wskaźniki i referencje

Wskaźniki są zmiennymi, a referencje nie.

Użycie referencji oznacza przekazanie kontroli nad obiektem, natomiast wskaźnik określa położenie obiektu w pamięci.

Dlatego wskaźniki mogą być polami obiektów (tzn. można stworzyć klasę, w której część pól będzie wskaźnikami), a referencje – nie.

Dlatego w obiektach klasy `QWidget` przechowuje się wskaźnik do obiektu nadrzędnego (jego adres), a nie referencję do niego (referencje nie mogą być przechowywane!).

Wskaźniki i referencje

Wskaźniki są zmiennymi, a referencje nie.

Użycie referencji oznacza przekazanie kontroli nad obiektem, natomiast wskaźnik określa położenie obiektu w pamięci.

Dlatego wskaźniki mogą być polami obiektów (tzn. można stworzyć klasę, w której część pól będzie wskaźnikami), a referencje – nie.

Dlatego w obiektach klasy `QWidget` przechowuje się wskaźnik do obiektu nadrzędnego (jego adres), a nie referencję do niego (referencje nie mogą być przechowywane!).

Dlatego konstruktor `QWidget()` przyjmuje argument typu `(QWidget *)`.

Klasa QLayout

Widżety powinny być rozmieszczane w obrębie dostępnej im przestrzeni w konsystenty sposób niezależnie od okoliczności (np. wtedy, gdy użytkownik zmieni rozmiary okna).

Klasa QLayout

Widżety powinny być rozmieszczane w obrębie dostępnej im przestrzeni w konsystenty sposób niezależnie od okoliczności (np. wtedy, gdy użytkownik zmieni rozmiary okna).

W tym celu stosuje się obiekty opisujące rozmieszczenie widżetów w danym pojemniku (*ang. container*).

Klasa `QLayout`

Widżety powinny być rozmieszczane w obrębie dostępnej im przestrzeni w konsystenty sposób niezależnie od okoliczności (np. wtedy, gdy użytkownik zmieni rozmiary okna).

W tym celu stosuje się obiekty opisujące rozmieszczenie widżetów w danym pojemniku (*ang. container*).

W bibliotece *Qt* jest pewna liczba klas służących do definiowania takich obiektów. Są one klasami pochodnymi w stosunku do klasy `QLayout`.

Klasa `QLayout`

Widżety powinny być rozmieszczane w obrębie dostępnej im przestrzeni w konsystenty sposób niezależnie od okoliczności (np. wtedy, gdy użytkownik zmieni rozmiary okna).

W tym celu stosuje się obiekty opisujące rozmieszczenie widżetów w danym pojemniku (*ang. container*).

W bibliotece *Qt* jest pewna liczba klas służących do definiowania takich obiektów. Są one klasami pochodnymi w stosunku do klasy `QLayout`.

Klasa `QLayout` jest klasą pochodną w stosunku to `QObject` oraz `QLayoutItem`. Definiuje ona pola i metody wykorzystywane przez wszystkie klasy służące do sterowania rozmieszczaniem widżetów.

Klasy pochodne w stosunku do QLayout

QHBoxLayout

Widżety są rozmieszczane w sposób horyzontalny, w porządku od lewej do prawej (jeśli używany w systemie język tego wymaga, widżety mogą być rozmieszczane w porządku od prawej do lewej).

Klasy pochodne w stosunku do QLayout

QHBoxLayout

Widżety są rozmieszczane w sposób horyzontalny, w porządku od lewej do prawej (jeśli używany w systemie język tego wymaga, widżety mogą być rozmieszczane w porządku od prawej do lewej).

QVBoxLayout

Widżety są umieszczane w pionowej kolumnie, w porządku od góry do dołu.

Klasy pochodne w stosunku do `QLayout`

`QHBoxLayout`

Widżety są rozmieszczane w sposób horyzontalny, w porządku od lewej do prawej (jeśli używany w systemie język tego wymaga, widżety mogą być rozmieszczane w porządku od prawej do lewej).

`QVBoxLayout`

Widżety są umieszczane w pionowej kolumnie, w porządku od góry do dołu.

`QGridLayout`

Widżety są rozmieszczane z wykorzystniem prostokątnej siatki (pojedynczy widżet może zajmować wiele „oczek").

Klasy pochodne w stosunku do `QLayout` (c. d.)

`QFormLayout`

Widżety są rozmieszczane w strukturze formularza o dwóch kolumnach (tzn. w „systemie” etykieta-pole).

Klasy pochodne w stosunku do `QLayout` (c. d.)

`QFormLayout`

Widżety są rozmieszczane w strukturze formularza o dwóch kolumnach (tzn. w „systemie” etykieta-pole).

`addWidget()`

Metoda, definiowana przez większość wymienionych klas, służąca do kojarzenia widżetów z pojemnikiem (poprzez obiekt rozmieszczający je).

Klasy pochodne w stosunku do QLayout (c. d.)

QFormLayout

Widżety są rozmieszczane w strukturze formularza o dwóch kolumnach (tzn. w „systemie" etykieta-pole).

addWidget()

Metoda, definiowana przez większość wymienionych klas, służąca do kojarzenia widżetów z pojemnikiem (poprzez obiekt rozmieszczający je).

Dla obiektów klas QHBoxLayout i QVBoxLayout jedynym argumentem addWidget() jest wskaźnik do obiektu klasy QWidget, reprezentującego komponent do rozmieszczenia, a kolejność komponentów jest określana przez kolejność wywołań addWidget(), np.:

```
QVBoxLayout *layout = new QVBoxLayout;
layout->addWidget(button1);  // Ten komponent będzie znajdował się na samej górze.
layout->addWidget(button2);  // Ten komponent będzie drugi od góry.
```

Rozmieszczanie widżetów z użyciem `QGridLayout`

Metoda `addWidget()` definiowana przez klasę `QGridLayout` wymaga dwóch dodatkowych argumentów, będących współrzędnymi komponentu na siatce (współrzędne 0, 0 reprezentują lewy górny róg), np.:

```
QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0);
layout->addWidget(button4, 1, 1);
```

Rozmieszczanie widżetów z użyciem `QGridLayout`

Metoda `addWidget()` definiowana przez klasę `QGridLayout` wymaga dwóch dodatkowych argumentów, będących współrzędnymi komponentu na siatce (współrzędne 0, 0 reprezentują lewy górny róg), np.:

```
QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0);
layout->addWidget(button4, 1, 1);
```

Dla komponentów zajmujących wiele oczek siatki istnieje wariant `addWidget()` o 5 argumentach. Ostatnie dwa argumenty tej metody określają zasięg (*ang. span*) komponentu (w poziomie i w pionie).

```
QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0, 1, 2);
layout->addWidget(button4, 2, 0);
layout->addWidget(button5, 2, 1);
```

Rozmieszczanie widżetów z użyciem `QFormLayout`

Dla klasy `QFormLayout` preferowanym sposobem kojarzenia widżetu z pojemnikiem jest użycie (jednego z wariantów) metody `addRow()`, np.:

```
QFormLayout *formLayout = new QFormLayout;
formLayout->addRow(tr("&Name:"), nameLineEdit);
formLayout->addRow(tr("&Email:"), emailLineEdit);
formLayout->addRow(tr("&Age:"), ageSpinBox);
```

Rozmieszczanie widżetów z użyciem `QFormLayout`

Dla klasy `QFormLayout` preferowanym sposobem kojarzenia widżetu z pojemnikiem jest użycie (jednego z wariantów) metody `addRow()`, np.:

```
QFormLayout *formLayout = new QFormLayout;
formLayout->addRow(tr("&Name:"), nameLineEdit);
formLayout->addRow(tr("&Email:"), emailLineEdit);
formLayout->addRow(tr("&Age:"), ageSpinBox);
```

tr()

Jeżeli to możliwe, tłumaczy ciąg znaków na bieżący język interfejsu użytkownika i zwraca wynik.

Rozmieszczanie widżetów z użyciem `QFormLayout`

Dla klasy `QFormLayout` preferowanym sposobem kojarzenia widżetu z pojemnikiem jest użycie (jednego z wariantów) metody `addRow()`, np.:

```
QFormLayout *formLayout = new QFormLayout;
formLayout->addRow(tr("&Name:"), nameLineEdit);
formLayout->addRow(tr("&Email:"), emailLineEdit);
formLayout->addRow(tr("&Age:"), ageSpinBox);
```

tr()

Jeżeli to możliwe, tłumaczy ciąg znaków na bieżący język interfejsu użytkownika i zwraca wynik.

Znak & wewnątrz napisu

Powoduje, że wciśnięcie kombinacji klawisza Alt z klawiszem reprezentującym literę, przed którą stoi, jest równoważne kliknięciu myszą na dany wiersz (lub np. przycisk itp.).

Łączenie pojemnika z zestawem widżetów

Obiekt rozmieszczający widżety wraz z zestawem widżetów związanych z nim musi być skojarzony z pojemnikiem.

Łączenie pojemnika z zestawem widżetów

Obiekt rozmieszczający widżety wraz z zestawem widżetów związanych z nim musi być skojarzony z pojemnikiem.

Pojemnik jest obiektem klasy `QWidget` (lub klasy pochodnej w stosunku do niej), reprezentującym prostokątny wycinek ekranu.

Łączenie pojemnika z zestawem widżetów

Obiekt rozmieszczający widżety wraz z zestawem widżetów związanych z nim musi być skojarzony z pojemnikiem.

Pojemnik jest obiektem klasy `QWidget` (lub klasy pochodnej w stosunku do niej), reprezentującym prostokątny wycinek ekranu.

Zestaw widżetów umieszcza się w pojemniku z pomocą metody `setLayout()`, np.:

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");

QHBoxLayout *layout = new QHBoxLayout;
layout->addWidget(button1);
layout->addWidget(button2);

window->setLayout(layout);
```

Łączenie zestawów widżetów

Zestaw widżetów (tzn. obiekt rozmieszczający wraz z powiązanymi z nim widżetami) może być skojarzony z innym obiektem rozmieszczającym widżety podobnie, jak pojemnik (obiekt klasy `QWidget`).

Łączenie zestawów widżetów

Zestaw widżetów (tzn. obiekt rozmieszczający wraz z powiązanymi z nim widżetami) może być skojarzony z innym obiektem rozmieszczającym widżety podobnie, jak pojemnik (obiekt klasy `QWidget`).

Zestaw widżetów łączy się z nadrzędnym obiektem rozmieszczającym widżety z pomocą metody `addLayout()`, np.:

```
QHBoxLayout *sublayout = new QHBoxLayout;
sublayout->addWidget(button1);
sublayout->addWidget(button2);

QVBoxLayout *Layout = new QVBoxLayout;
layout->addLayout(sublayout);
layout->addWidget(button3);
```

Łączenie zestawów widżetów

Zestaw widżetów (tzn. obiekt rozmieszczający wraz z powiązanymi z nim widżetami) może być skojarzony z innym obiektem rozmieszczającym widżety podobnie, jak pojemnik (obiekt klasy `QWidget`).

Zestaw widżetów łączy się z nadrzędnym obiektem rozmieszczającym widżety z pomocą metody `addLayout()`, np.:

```
QHBoxLayout *sublayout = new QHBoxLayout;
sublayout->addWidget(button1);
sublayout->addWidget(button2);

QVBoxLayout *Layout = new QVBoxLayout;
layout->addLayout(sublayout);
layout->addWidget(button3);
```

Dla *wszystkich* widżetów skojarzonych z danym obiektem klasy `QWidget` (pojemnikiem) poprzez obiekty rozmieszczające (bezpośrednio lub za pośrednictwem innych takich obiektów) jest on obiektem nadrzędnym.

`sizeHint()` i `sizePolicy()`

`sizeHint()`

Metoda zwracająca obiekt klasy `QSize` określający preferowane rozmiary widżetu. Domyślnie zwraca obiekt oznaczający brak preferencji.

`sizeHint()` i `sizePolicy()`

`sizeHint()`

Metoda zwracająca obiekt klasy `QSize` określający preferowane rozmiary widżetu. Domyślnie zwraca obiekt oznaczający brak preferencji.

`sizePolicy()`

Metoda zwracająca obiekt klasy `QSizePolicy` określający skłonność danego widżetu do zmiany swojej geometrii w odpowiedzi na zmiany geometrii pojemnika (można określić oddzielnie dla kierunków pionowego i poziomego). Domyślnie zwraca obiekt oznaczający, że rozmiary zwracane przez `sizeHint()` są optymalne, ale widżet może być ściśnięty i dalej pozostanie użyteczny, natomiast rozciąganie go nie przyniesie korzyści.

Dostosowywanie rozmieszczania komponentów

`setSpacing()`, `setVerticalSpacing()`, `setHorizontalSpacing()`

Określają przestrzenie między komponentami w obszarze pojemnika odpowiadającym danemu obiektowi rozmieszczającemu.

Dostosowywanie rozmieszczania komponentów

`setSpacing()`, `setVerticalSpacing()`, `setHorizontalSpacing()`

Określają przestrzenie między komponentami w obszarze pojemnika odpowiadającym danemu obiektowi rozmieszczającemu.

`setStretch(n, s)`

Dla obiektów rozmieszczających klasy `QHBoxLayout` lub `QVBoxLayout` określa współczynnik rozciągnięcia (*ang. stretch factor*) s, odpowiadający widżetowi na pozycji n (w przypadku zwiększenia rozmiarów pojemnika widżet na pozycji n jest „rozciągany” proporcjonalnie do s).

Dostosowywanie rozmieszczania komponentów

`setSpacing()`, `setVerticalSpacing()`, `setHorizontalSpacing()`

Określają przestrzenie między komponentami w obszarze pojemnika odpowiadającym danemu obiektowi rozmieszczającemu.

`setStretch(n, s)`

Dla obiektów rozmieszczających klasy `QHBoxLayout` lub `QVBoxLayout` określa współczynnik rozciągnięcia (*ang. stretch factor*) s, odpowiadający widżetowi na pozycji n (w przypadku zwiększenia rozmiarów pojemnika widżet na pozycji n jest „rozciągany” proporcjonalnie do s).

`setColumnStretch()`, `setRowStretch()`

Dla obiektów rozmieszczających klasy `QGridLayout` określają współczynniki rozciągnięcia dla danej kolumny lub danego wiersza siatki, odpowiednio (licząc od zera).

Literatura

 J. Blanchette, M. Summerfield, *C++ GUI Programming with Qt 4 (2nd Edition)* (Prentice Hall, Westford, 2008).

 *Qt 4.5 Reference Documentation*, `http://doc.trolltech.com/4.5/index.html`