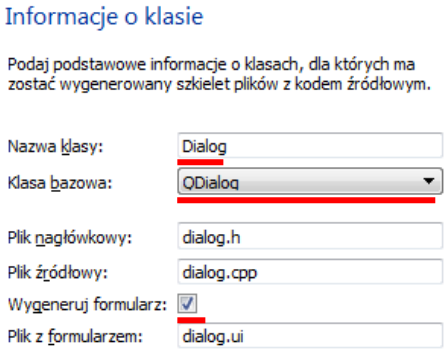
ISSP/C++: tydzień 2

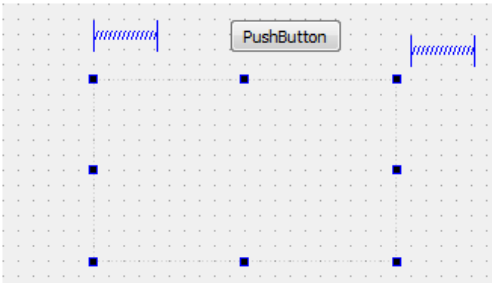
Zadanie 2: napisać prostą animację, np. złudzenia optycznego przedstawionego przez PhysicsGirl: https://www.youtube.com/watch?v=uNOKWoDtbSk

1. Tworzymy nowy projekt (`ctrl-N`). W oknach dialogowych wybieramy:

   - `Aplikacja / Aplikacja Qt Widgets`
   - Nazwa projektu: np. `illusion`
   - W oknie `Informacja o klasie` wybieramy jako klasę bazową `QDialog` (żeby nauczyć się czegoś nowego). Nazwę klasy możemy zostawić domyślną: `Dialog`. Zaznaczany opcję „Wygeneruj formularz”.



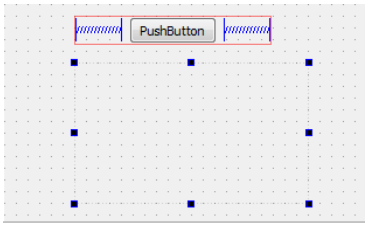
2. W oknie projektanta interfejsu użytkownika umieszczamy tylko dwa widgety: `PushButton` (z grupy `Buttons`) i `Widget` (z grupy `Containers`). Do tego dodajemy dwie „poziome sprężynki”, czyli dwa egzemplarze `Horizontal Spacer` (z grupy `Spacers`).



   Komentarze:

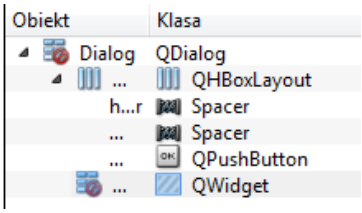
   - `PushButton` to zwyczajny przycisk, widget podobny do `ToolButton` z wykładu 1, jednak mający nieco inne zastosowania. Coś nowego.
   - QWidget to najbardziej podstawowy obiekt graficzny Qt. W Qt wszystko, co komunikuje się z użytkownikiem za pomocą graficznego interfejsu użytkownika (GUI) jest Widgetem, który został następnie „rozszerzony” o nowe właściwości. Na tym Widgecie będziemy rysować.

3. Zaznacz myszką sprężynki oraz przycisk i w znany już sposób rozmieść je w poziomie.

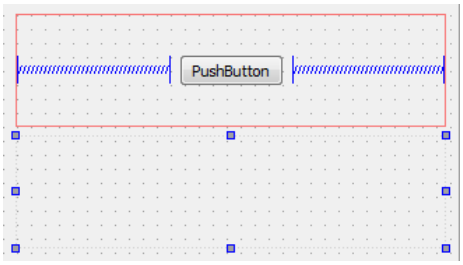


   Komentarz:

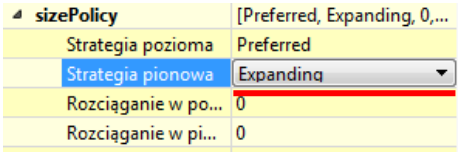
   - Do zaznaczania obiektów można też wykorzystać panel „hierarchia obiektów”, który zwyczajowo znajduje się w prawym górnym rogu stołu montażowego. Jeśli umiesz zaznaczać kilka plików w eksploratorze Windows, dasz radę z tym panelem: klawisze `Ctrl` i `Shift` działają zgodnie z oczekiwaniami.



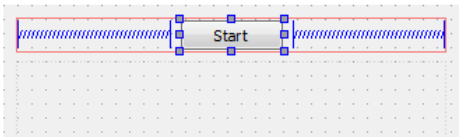
4. Zaznacz okno główne programu (Dialog klasy QDialog) i zastosuj rozmieszczanie w pionie



5. Uzyskany efekt rozczarowuje: `Widget` zajmuje tylko połowę obszaru roboczego, gdyż przycisk rozpycha się na całej górnej połowie okna. Żeby temu zaradzić, zaznacz `Widget` i w oknie Edytora Właściwości (prawy dolny róg) znajdź właściwość `sizePolicy`. Nadaj (wybierz) jej składowej „strategia pionowa" wartość `Expanding`. W ten sposób poinformowałeś obiekt odpowiedzialny za wyrównanie w pionie (z poprzedniego punktu), że Widget prosi o wygospodarowanie jak najwięcej miejsca w pionie.

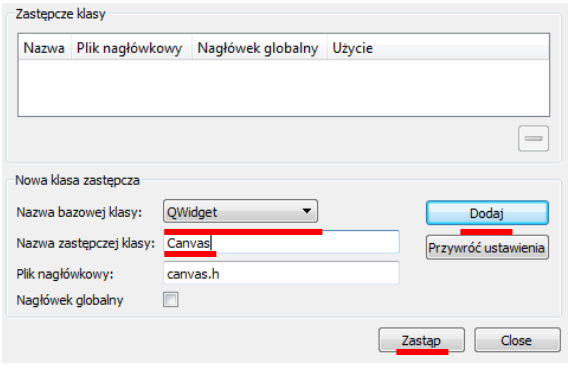


6. Kliknij dwa razy przycisk i zmień wyświetlany przezeń tekst na `&Start`. Co prawda ten napis będziemy zmieniać dynamicznie na `Start/Stop`, ale póki co, polepszamy czytelniość projektu.



7. Mam nadzieję, że nie zapominasz po każdej poprawce skompilować i uruchomić programu?

8. Teraz kluczowy krok. Nasz centralny `widget` jest bezużyteczny, bo nic nie potrafi, a przecież chcemy na nim rysować różne obrazki. Zaznaczamy go i z jego menu kontekstowego wybieramy `Zastąp...` W oknie dialogowym `Zastępcze klasy` wybieramy `QWidget` jako klasę bazową oraz `Canvas` („płótno") jako „nazwę zastępczej klasy". Wciskamy `Dodaj` i na końcu – `Zastąp`.

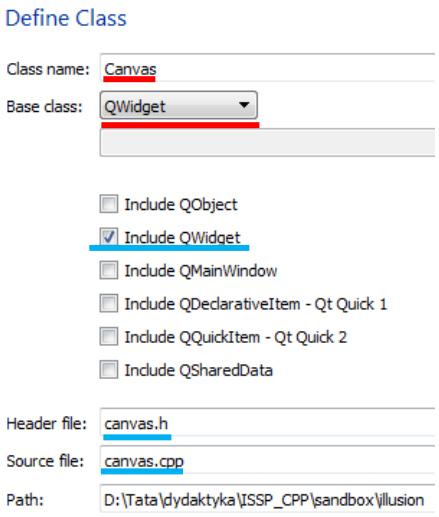


9. Spróbuj skompilować program. Ujrzysz komunikat `błąd: canvas.h: No such file or directory.`

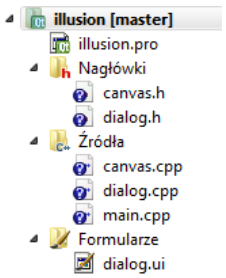
10. Zanim rozwiążemy ten problem, spróbujmy dowiedzieć się o nim czegoś więcej (nie zawsze będziesz miał(a) tak szczegółowy przewodnik jak ten). Kliknij w komunikat dwukrotnie. Program przeniesie Cię do miejsca, w którym kompilator zidentyfikował błąd w programie. Zwróć uwagę na to, że błąd pojawił się w pliku, którego nie edytował(a/e)ś i którego nawet nie ma na liście plików projektu. U mnie ten plik nazywa się `ui_dialog.h` i można go znaleźć w osobnym katalogu (poza projektem). Dzieje się tak dlatego, że plik ten jest generowany automatycznie na podstawie stanu projektu intefejsu użytkownika wyklikanego ma stole montażowym. OK. Możesz zamknąć ten plik (krzyżykiem przy nazwie na pasku nad obszarem edycji).

11. Skąd wziąć brakujący plik `canvas.h`? Wciśnij `Ctrl-N`. Tym razem z grupy `Pliki i klasy` wybierz `C++`, ze środkowego panelu wybierz `C++ Class`, i wciśnij przycisk `Wybierz...`

12. W kolejnym oknie dialogowym wpisz `Canvas` jako nazwę generowanej klasy (`class name`) i koniecznie wybierz klasę bazową (`Base class`) jako `QWidget`. Reszta pól powinna się wypełnić automatycznie. Obowiązuje prosta zasada: to, co tu wpisujemy, musi się zgadzać z tym, co wpisaliśmy w okienku `zastępcze klasy` (punkt 8).

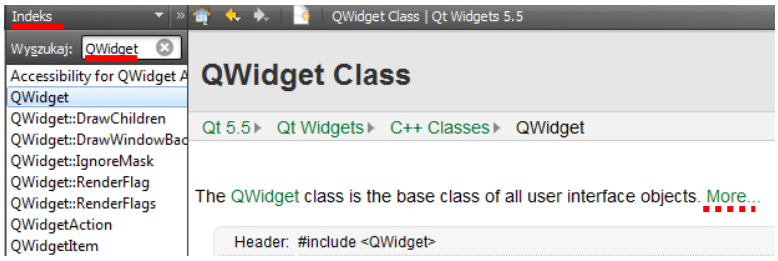


13. Okno projektu powinno teraz zawierać 2 pliki nagłówkowe, 3 źródłowe i jeden plik z formularzem



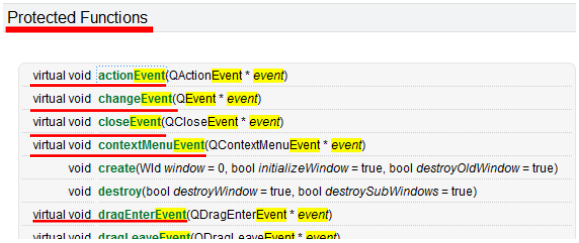
14. Sprawdź, że program znów się kompiluje (choć jeszcze nic nie robi).

15. Rysowanie zawartości okna roboczego widżetów jest czynnością specyficzną. Programista nie jest w stanie przewidzieć wszystkich sytuacji, w których potrzebne jest odrysowanie zawartości okna. Wystarczy sobie wyobrazić, że okno może być przesuwane, w tym wysuwane „spoza” monitora, minimalizowane, maksymalizowane, albo inne okno chwilowo przesłania część naszego okna. Większość tych sytuacji obsługuje system operacyjny. Z tego względu jest bardzo pożądane, aby system wiedział, czy nasze okno jest aktualne, czy też wymaga odświeżenia. Innymi słowy, prawidłowy sposób rysowania czegokolwiek w oknie wymaga oddania inicjatywy systemowi operacyjnemu. Jeśli zdarzy się, że to program chce zmienić obraz wyświetlany w oknie, powinien poinformować system, że dane okno jest nieaktualne i wymaga odświeżenia. System, w stosownym momencie, reaguje na to w ten sposób, że wywołuje *naszą* funkcję odświeżającą wygląd okienka. Dowcip polega na tym, że ta funkcja nie może należeć do systemu operacyjnego: ona musi być częścią naszego programu. Innymi słowy, do odświeżenia wyglądu okienka system operacyjny wywołuje jedną, ściśle określoną funkcję naszego programu. Takie funkcje, które mogą być wywoływane przez system, nazywają się funkcjami zwrotnymi, (ang. *callback functions* lub po prostu *callbacks*). Jak jednak podsunąć systemowi do wywołania taką funkcję? W Języku C służą do tego specjalne funkcje, rejestrujące inne funkcje jako *callbacks*. W C++ podstawowym mechanizmem są ***funkcje wirtualne***. W szczególności w Qt ten mechanizm polega na użyciu funkcji wirtualnych o ściśle określonych nazwach i argumentach. Te nazwy można odczytać w dokumentacji klasy QWidget. Mają one w nazwie cząstkę „event”.
I w końcu zadanie:



- odszukaj w dokumentacji Qt opis klasy `QWidget` (`Ctrl-F` będzie działać).

- Odszukaj w tym opisie wystąpienia frazy `Event`. Zapewne zlokalizujesz bardzo dużo funkcji w sekcji `Protected functions.`



16. Odszukaj opis funkcji `paintEvent`:

void **QWidget::**paintEvent(**QPaintEvent** ***** ***event*****)**

- Zaznacz tę nazwę w całości, od `void` po okrągły nawias zamykający, i skopiuj do schowka (`Ctrl-C`). Nie kopiuj informacji zawartych w nawiasach kwadratowych po prawej stronie (**`[virtual protected]`**).

- Skopiuj ten wpis do sekcji publicznej deklaracji klasy `Canvas` w pliku `canvas.h`. Na końcu postaw średnik i koniecznie usuń z nazwy przestrzeń nazw `QWidget::`

```
void paintEvent(QPaintEvent * event);
```

(uwaga: hiperpoprawny kod powinien być raczej umieszczony w sekcji `protected:`, ale `public:` też jest dobre)

- Skopiuj swoją deklarację do schowka (`Ctrl-C`), a następnie do pliku źródłowego `canvas.cpp`, np. na jego końcu (`F4`). Usuń średnik, dodaj klamry, koniecznie dodaj przestrzeń nazw `Canvas::`. Usuń też nazwę argumentu funkcji, gdyż nie będziemy go używać, a usuwając nazwę, unikniemy zbędnego ostrzeżenia kompilatora.

```
void Canvas::paintEvent(QPaintEvent *)
{

}
```

Jeśli wszystko jest OK, identyfikator *`paintEvent`* wyświetlany jest pochyloną czcionką, co sygnalizuje tzw. funkcję wirtualną.

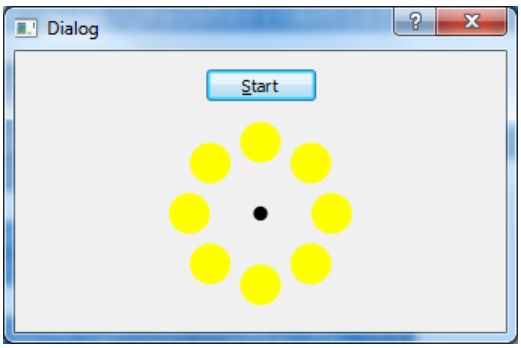
17. W pliku `canvas.cpp` dodaj na początku dyrektywę

```
#include <QPainter>
```

natomiast w ciele funkcji Canvas::*paintEvent* zdefiniuj obiekt `painter`:

```
QPainter painter(this);
```

18. Narysuj na obiekcie Canvas obrazek jak na rysunku obok.
Oto jedno z możliwych rozwiązań tego zadania:



```
void Canvas::paintEvent(QPaintEvent *)
{
   QPainter painter(this);
   painter.setRenderHint(QPainter::Antialiasing);

   QRect r = this->rect();
   int side = std::min(r.width(), r.height());
   painter.translate(0.5*r.width(), 0.5*height());
   painter.scale(side/200.0, side/200.0);

   painter.setBrush(QBrush(QColor("black")));
   painter.setPen(Qt::NoPen);

   painter.drawEllipse(QPoint(0,0), 7, 7);

   painter.setBrush(QBrush(QColor("yellow")));
   const int N = 8;
   for (int i = 0; i < N; i++)
   {
       painter.drawEllipse(QPoint(0,70), 20, 20);
       painter.rotate(360.0/N);
   }
}
```

19. Jak dodać do powyższego kodu animację? Podstawowym mechanizmem animacji w Qt są minutniki. Qt dostarcza trzy podstawowe ich rodzaje: QObject::startTimer(), QTimer oraz QBasicTimer. Poniżej użyjemy QTimer, gdyż pozwoli to zapoznać się z mechanizmem sygnałów i slotów, który jest wizytówką Qt.

    - W pliku `dialog.h` w sekcji prywatnej klasy `Dialog` dodaj deklarację wskaźnika na minutnik:

      QTimer * timer;

    - Przejdź do pliku źródłowego (`F4`) i w konstruktorze klasy Dialog (`Dialog::Dialog`) dodaj instrukcję inicjalizującą ten wskaźnik:

      ```
      Dialog::Dialog(QWidget *parent) :
        QDialog(parent),
        ui(new Ui::Dialog)
        {
          ui->setupUi(this);
          timer = new QTimer(this);
        }
      ```

      Uwaga: inicjalizacji tego wskaźnika można też dokonać w preambule konstruktora, między dwukropkiem a klamrą otwierającą ciało funkcji (`{`), natomiast w C++11 można ją umieścić bezpośrednio w pliku nagłówkowym, po znaku `=`.

20. Następnym krokiem jest włączenie minutnika. Chyba najlepszym miejscem jest kod obsługujący przycisk (`pushButton`).

    - Otwieramy okno projektanta interfejsu użytkownika i z menu kontekstowego przycisku wybieramy `Przejdź do slotu…`

    - Wpisujemy kod jak poniżej:

```
void Dialog::on_pushButton_clicked()
{
    myTimerEnabled = !myTimerEnabled;
    if (myTimerEnabled)
    {
       timer->start(333);
       ui->pushButton->setText("&Stop");
    }
    else
    {
       timer->stop();
       ui->pushButton->setText("&Start");
    }
}
```

- Zastanawiamy się, co ten kod robi ☺
- Zauważamy, że program przestał się kompilować, gdyż użyliśmy niezdefiniowanej zmiennej `myTimerEnabled`.
- Kopiujemy nazwę zmiennej (`myTimerEnabled`) do schowka, przechodzimy do pliku nagłówkowego `dialog.h` (`F4`) i w sekcji prywatnej klasy Dialog umieszczamy deklarację zmiennej logicznej `myTimerEnabled`

  ```
  bool myTimerEnabled;
  ```

- To nie koniec procesu dodawania zmiennej: **zawsze**, gdy do programu dodajesz zmienną, **musisz** od razu pomyśleć o tym, jaka będzie jej wartość początkowa. W C++11 możesz inicjalizację zadeklarować wprost w pliku nagłówkowym:

  ```
  bool myTimerEnabled = false;
  ```

  W starszych wersjach języka inicjalizację zawsze umieszcza się w konstruktorze (aczkolwiek w programach okienkowych, ze względu na skomplikowane interakcje między okienkami, czasami lepszym pomysłem jest dokonanie ostatecznej inicjalizacji obiektów graficznego interfejsu użytkownika dopiero podczas obsługi pierwszego zdarzenia `resizeEvent`). Ja zastosowałem drugą z powyższych metod, umieszczając kod inicjalizacyjny w preambule konstruktora, po przecinku:

  ```
  Dialog::Dialog(QWidget *parent) :
      QDialog(parent),
      ui(new Ui::Dialog),     // <- przecinek!
      myTimerEnabled(false)  // <- a tu już bez przecinka
  {
      ui->setupUi(this);
      timer = new QTimer(this);
  }
  ```

  preambuła

- Kompilujemy program. Sprawdzamy, że przynajmniej tekst wyświetlany na przycisku zmienia się zgodnie z planem.

21. W naszym programie natrafiamy na kolejną barierę: w jaki sposób odczytać „alarm” z minutnika? Zastosowany minutnik (klasy QTimer) jest najbardziej uniwersalnym minutnikiem Qt i jego obsługa wymaga poznania kolejnej „magii”. Obiekty klasy QTimer wysyłają komunikaty o alarmie wyłącznie do Qt. Zadaniem programisty jest poinformowanie Qt, by za każdym razem, gdy obiekt `timer` zgłosi mu komunikat „alarm”, Qt zareagowało, uruchamiając odpowiednią funkcję w naszym programie. Jak

się do tego zabrać? Przede wszystkim musimy wiedzieć, że obiekty klasy QTimer komunikują się z Qt za pomocą sygnałów. W tej chwili trzeba wiedzieć tylko tyle, że sygnały to koncepcja charakterystyczna dla Qt a nie C++ i że są to stosunkowo proste funkcje, których jedyną rolą jest zasygnalizowanie, że stan obiektu wysyłającego sygnał w określony sposób zmienił swój stan. Programista może poprosić Qt, by w reakcji na sygnał S pochodzący z konkretnego obiektu X, Qt uruchomił w wybranym obiekcie Y określoną funkcję, zwaną w żargonie Qt „slotem" (Y może być tożsame X, ale zwykle X i Y są różne). Jest to piękny i bardzo funkcjonalny mechanizm pozwalający całkowicie odseparować od siebie implementację różnych modułów („klas") Qt oraz różnych modułów („klas") użytkownika. Innymi słowy, programista odpowiedzialny za implementację przycisków (`QPushButton`) nie musi wiedzieć, kto ich będzie używał. Wystarczy, że w klasie `QPushButton` zaimplementuje sygnał `clicked()`, by każdy użytkownik Qt mógł zaimplementować sposób reakcji na to zdarzenie. Do związania sygnału ze slotem służy funkcja `connect`. Warto ją umieścić w początkowej części programu, np. w konstruktorze klasy Dialog:

```
Dialog::Dialog(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Dialog),
    myTimerEnabled(false)
{
    ui->setupUi(this);
    timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), ui->widget, SLOT(update()));
}
```

Jak widać, powyższa funkcja `connect` łączy sygnał `timeout()` („alarm!") minutnika `timer` ze slotem `update()` naszego widgetu mającego wyświetlać animację. W ten sposób zapewniamy, że co 333 milisekund do obiektu `widget` trafiać będzie zdarzenie `PaintEvent` generowane slotem `update()`. Z dokumentacji funkcji `update()` dowiadujemy się bowiem, iż jej wywołanie oznacza invalidację (= uznanie z nieważny) obrazu wyświetlanego w okienku danego obiektu oraz prośbę o odświeżenie jego stanu. Prośba ta kierowana jest do obiektu nieco okrężną drogą – za pośrednictwem okienkowego systemu operacyjnego, który też musi wiedzieć, które okienka są ważne, a które nie (pomyśl, jaki problem sprawiają systemowi częściowo przezroczyste okienka).

22. Na pewno chcielibyśmy sprawdzić, czy sygnały minutnika docierają do naszego widżetu. Moglibyśmy ustawić pułapkę w funkcji
`void Canvas::paintEvent(QPaintEvent *)`
ale debugowanie *tej konkretnej funkcji* debugerem to zwykle zły pomysł (jak z niej wyjść?). Zamiast tego włączamy do pliku `canvas.cpp` plik nagłówkowy

    ```
    #include <QDebug>
    ```

    i na początku funkcji `Canvas::paintEvent` wpisujemy instrukcję wyświetlającą jakiś napis za pośrednictwem obiektu **`qDebug()`**.

    ```
    void Canvas::paintEvent(QPaintEvent *)
    {
        qDebug() << "hello!";
        ...
    ```

    Uruchamiamy program. Sprawdzamy, że kolejne kliknięcia przycisku `pushButton` włączają i wyłączają automatyczne wywoływanie funkcji Canvas::*paintEvent* co 333 ms.

23. Skoro minutnik działa, możemy już wyświetlić animację. Potrzebujemy jeszcze tylko jednej informacji: którego żółtego kółka w danej chwili nie powinniśmy wyświetlać? Tę informację można umieścić w jednej zmiennej typu `int`. Wygodnie będzie, by zmienną tę widziały wszystkie funkcje klasy `Canvas`. Dlatego w sekcji prywatnej klasy `Canvas` (plik `canvas.h`) dodajemy deklarację nowej zmiennej:

```
class Canvas : public QWidget
{
    Q_OBJECT
public:
    explicit Canvas(QWidget *parent = 0);
    void paintEvent(QPaintEvent * event);

signals:

public slots:

private:
    int currentIndex;
};
```

24. Nie możemy zapomnieć o jej inicjalizacji (F4/plik `canvas.cpp`):

```
Canvas::Canvas(QWidget *parent) : QWidget(parent)
{
    currentIndex = 0;
}
```

25. Na początku funkcji `Canvas::paintEvent` wpisujemy kod zmieniający cyklicznie wartość zmiennej **`currentIndex`** (na zasadzie karuzeli) oraz na końcu kod wykorzystujący aktualną wartość tej zmiennej do pominięcia jednego z żółtych kółek:

```
void Canvas::paintEvent(QPaintEvent *)
{
    qDebug() << "hello!";

 currentIndex++;
 if (currentIndex == 8)
     currentIndex = 0;
 ...
 for (int i = 0; i < N; i++)
 {
      if (currentIndex != i)
          painter.drawEllipse(QPoint(0,70), 20, 20);
     painter.rotate(360.0/N);
 }
```

26. Kompilujemy i testujemy program. Powinien działać. Ale jeśli wszechstronnie go przetestujemy, odkryjemy jego słabość: animację można w nim włączyć nie tylko przyciskiem, ale i zmianą rozmiaru okienka albo cyklicznym przesuwaniem go na wierzch i na spód hierarchii okienek. Sprawdź to! Przyczyna jest prosta: stan programu uzależniliśmy od kodu odpowiedzialnego za jego wyświetlanie. To nie jest dobry pomysł.

27. Spróbujmy wyeliminować powyższy feler. Sygnał `timeout()` minutnika powinien trafiać do wyspecjalizowanego slotu i dopiero tam powinno się zlecać programowi odświeżenie zawartości okna. W tym celu:

    - deklarujemy nowy slot (np. o nazwie `nextFrame`) w klasie `Canvas`:

```
class Canvas : public QWidget
{
    Q_OBJECT
public:
    explicit Canvas(QWidget *parent = 0);
    void paintEvent(QPaintEvent * event);

signals:

public slots:
    void nextFrame();

private:
    int currentIndex;
};
```

Uwaga. Slot w Qt to po prostu funkcja umieszczona w sekcji `public slots:` danej klasy.

- W pliku `canvas.cpp` umieszczamy definicję tej funkcji:

```
void Canvas::nextFrame()
{
    currentIndex++;
    if (currentIndex == 8)
        currentIndex = 0;

    this->update();
}
```

- Z funkcji `Canvas::paintEvent` usuwamy kod modyfikujący zmienną `currentIndex` (bo przenieśliśmy go powyżej do funkcji `Canvas::nextFrame`).

- W pliku `dialog.cpp` zmieniamy wartość slotu podłączonego do minutnika:

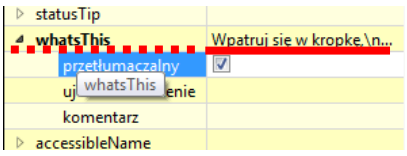
```
connect(timer, SIGNAL(timeout()), ui->widget, SLOT(nextFrame()));
```

- Kompilujemy i testujemy program. Animacja powinna działać prawidłowo i nie zależeć od zmian rozmiaru okienka.

28. Dokonujemy ostatecznej kosmetyki.

- W znany już sobie sposób zmieniamy tytuł okienka.

- Przechodzimy do projektanta graficznego interfejsu użytkownika (GUI) i nadajemy wybranym elementom interfejsu rozsądne wartości składowej `whatsThis`.



- Poniższym kodem, umieszczonym w konstruktorze klasy `Dialog`, można ustalić rozsądną początkową wielkość okna programu:

```
#include <QDesktopWidget>
...
QDesktopWidget desktop;
int desktopHeight = desktop.geometry().height();
int desktopWidth  = desktop.geometry().width();
this->resize(desktopWidth/2, desktopHeight/2);
```

- Z ciała funkcji `Canvas::paintEvent(QPaintEvent *)` można już usunąć instrukcję diagnostyczną

```
qDebug() << "hello!";
```

29. (*) Zadanie dla ambitnych, którzy chcą nieco lepiej zrozumieć, jak działają programy okienkowe.

- Zadeklaruj w klasie `Dialog` funkcję wirtualną

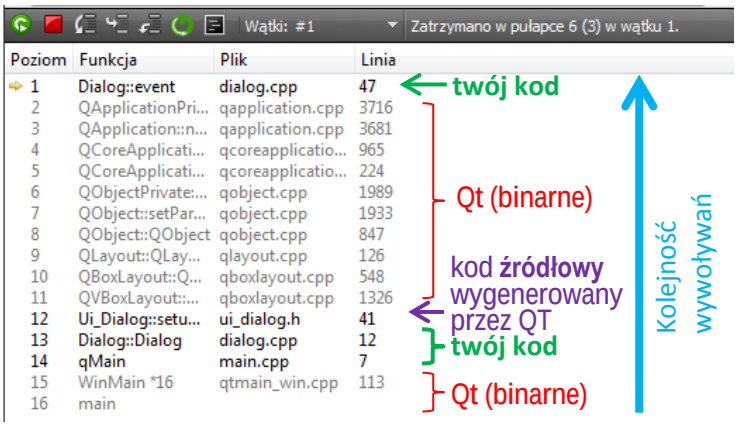
```
bool  event(QEvent * event)
```

- Oto moja implementacja tej funkcji:

```
bool Dialog::event(QEvent * event)
{
    static int counter = 0;
    counter++;
    qDebug() << counter << event->type();
    return QDialog::event(event);
}
```

- Uruchom program. Sprawdź, jak reaguje on (w okienku diagnostycznym) na różne akcje zewnętrzne, np. ruch myszki, klikniecie myszką (lewy, praw klawisz), przekręcenie rolki myszy, zmianę rozmiaru okna, naciśnięcie klawisza na klawiaturze itp.

30. (**) Bardzo ambitne zadanie. Uruchom swój program pod debugerem, zakładając pułpki w trzech funkcjach:
`Dialog::event (QEvent *)`
`Canvas::nextFrame()`
`Dialog::on_pushButton_clicked()`



Zaobserwuj w okienku Stos, w jakiej kolejności wywoływano które funkcje, nim program osiągnął daną pułapkę. Zwróć uwagę, że Qt wywołuje Twój kod, który wywołuje kod Qt, który z kolei znów wywołuje Twój kod. W szczególności Twoja funkcja `main` widoczna jest jako `qMain`. Twój program jest więc swego rodzaju wtyczką do Qt, przy czym istnieje kilka poziomów i mechanizmów wywoływania tych „wtyczek". Jednym z nich są funkcje wirtualne (funkcja `Dialog::event`).

Komentarze

Powyższe zadanie, wraz z poprzednim, wymagają kilku słów komentarza.

1. Typowy, funkcjonalny program w C++ składa się z wielu plików i korzysta z zewnętrznych bibliotek (dostępnych w postaci kodu źródłowego lub jako gotowych plików binarnych). Wymaga to opanowania zasad posługiwania się bibliotekami, w tym sposobów posługiwania się narzędziami do sprawnej kompilacji programów składających się z wielu plików i korzystających z zewnętrznych bibliotek. Qt posiada i wykorzystuje w tle własny mechanizm – qmake. Inne popularne narzędzia tego typu to m.in. make, cmake i automake.

2. Oba programy, przeglądarka WWW i animacja złudzenia optycznego, to przykłady **programów obiektowych**. Qt jest z kolei przykładem **biblioteki obiektowej**. Programowanie obiektowe polega na definiowaniu **klas** (czyli nowych typów danych) w oparciu o inne klasy, zwykle dostępne w postaci zewnętrznych bibliotek. Biblioteki te dostarczają podstawowych, uniwersalnych funkcjonalności, które w swoich programach dostosowujemy do własnych potrzeb.

   Podstawowym sposobem definiowania nowych klas w oparciu o klasy już istniejące jest **dziedziczenie**. Jeśli klasa X dziedziczy po klasie Y, to oznacza to, że każdy obiekt klasy X posiada co najmniej wszystkie cechy obiektu klasy Y i być może jakieś cechy dodatkowe. Każdy obiekt klasy pochodnej X może więc z powodzeniem udawać obiekt klasy bazowej Y. Mówi się nawet, że X i Y są w relacji „X jest Y”. O tym, czy jakaś klasa została wyprowadzona z innej decyduje początek jej deklaracji. Por. deklarację klasy `Dialog` w pliku nagłówkowym `dialog.h`:

   ```
   class Dialog : public QDialog
   ```

   która deklaruję klasę `Dialog` jako klasę dziedziczącą (publicznie) po klasie `QDialog`. Dzięki temu każdego obiektu klasy `Dialog` można używać, jakby jednocześnie był obiektem klasy `QDialog`. W szczególności Qt widzi każdy obiekt klasy `Dialog` jako obiekt klasy podstawowej (`QDialog`), a ponieważ ma bezpośredni dostęp do definicji tej klasy bazowej (`QDialog`), może wchodzić z naszym obiektem w interakcje, nie znając pełnej definicji klasy pochodnej (`Dialog`), czyli rzeczywistego typu tego obiektu. Jest to niesamowita cecha, umożliwiająca pełną modularyzację programów. Klasa bazowa (`QDialog`) służy Qt jako **interfejs** do naszych obiektów. Ten interfejs zawiera informację częściową o naszym obiekcie (posiada on co najmniej wszystkie cechy obiektów klasy `QDialog`), która wystarcza Qt do wykonania użytecznych operacji na naszym obiekcie (np. do wyświetlenia jego *rzeczywistego* stanu na ekranie).

   Programowanie obiektowe koncentruje się na **obiektach**, czyli po prostu zbiorach danych. **Obiekt = zbiór danych**. Klasy definiują typy obiektów, czyli zawierają informację o strukturze obiektów (z czego się one składają, czyli jakie zawierają w sobie dane) oraz informacje o dopuszczalnych sposobach interakcji innych części programu z danymi przechowywanymi w obiektach danej klasy. Z zasady projektanci klas blokują innym częściom programu bezpośredni dostęp do danych obiektu, umieszczając deklaracje danych w sekcji `private:` klasy. Dlatego w powyższych programach wszystkie nowe składowe klas umieszczaliśmy w sekcji prywatnej (por. dodanie zmiennej **currentIndex** w punkcie 23 powyżej). Odczyt stanu obiektu oraz jego modyfikacja możliwa jest wyłącznie poprzez funkcje, których definicje umieszczone są

wewnątrz klasy (w sekcjach `public` i – czasami – `protected`), co daje autorowi klasy możliwość wzięcia pełnej odpowiedzialności za jej działanie i interakcję z otoczeniem. Taki sposób programowania nazywa się **hermetyzacją danych**, a elementami języka ją umożliwiającymi są sekcje `private:` i częściowo `protected:`, wszechobecne w klasach Qt. Hermetyzacja danych jest kolejnym elementem języka ułatwiającym modularyzację programu: chodzi o to, by praca programisty A w jak najmniejszym stopniu wpływała na to, co ze swoim kodem może zrobić programista B (pamiętajmy, że typowy program w C++ to miliony wierszy kodu rozwijanego przez duże grupy programistów, których skład i kompetencje zmieniają się w czasie)

Prawdopodobnie najbardziej kluczową rolę w programu obiektowym pełnią jednak funkcje wirtualne. Dziedziczenie i hermetyzacja to w gruncie rzeczy tylko mechanizmy umożliwiające efektywną implementację wirtualności. Chodzi o to, że jeżeli funkcje wirtualne zdefiniujemy w naszej klasie pochodnej, to w pełni zastąpią one funkcje klasy bazowej bez żadnej ingerencji ze strony twórców biblioteki, z której korzystamy. Innymi słowy, dzięki funkcjom wirtualnym możemy bibliotece podsuwać obiekty naszych klas wyprowadzonych przez dziedziczenie z klas biblioteki. Biblioteka będzie wywoływać na tych obiektach „swoje" funkcje (np. *paintEvent*). Jednak w rzeczywistości podczas wywoływania tych funkcji zostaną podmienione ich adresy (automatycznie!), w wyniku czego wywołane zostaną nasze funkcje a nie ich podstawowe wersje biblioteczne. Innymi słowy, jeżeli do biblioteki przekażemy obiekt klasy, w której przedefiniowano kilka funkcji wirtualnych, to w ten sposób podrzucimy bibliotece nie tylko obiekt traktowany jako zbiór danych, ale także adresy funkcji, które na tych danych należy uruchomić. Jest to kolejny, niesamowicie efektywny sposób modularyzacji programów. W szczególności nie musimy przedefiniowywać wszystkich funkcji wirtualnych – jeżeli tego nie zrobimy, a biblioteka uruchomi na naszym obiekcie taką funkcję, w sposób automatyczny wybrana zostanie funkcja wirtualna zdefiniowana w bibliotece. Dzięki temu w naszych programach w sposób domyślny obsługujemy takie zdarzenia, jak naciśnięcie klawiszy na klawiaturze, ruch kółka myszki itp. Taki sposób wywoływania funkcji wirtualnych w C++ bywa też nazywany **polimorfizmem** (bo w systemie istnieje wiele różnych funkcji o tej samej nazwie).

Zapamiętaj:

Programowanie obiektowe (w C++) = <u>dziedziczenie</u> + <u>hermetyzacja danych</u> + <u>funkcje wirtualne</u>.

Cała reszta (np. wywoływanie funkcji „po kropce", pseudowskaźnik `this` itp.) to tylko narzędzia składni języka, mające umożliwić sprawną realizację powyższego paradygmatu programowania.

Zapamiętaj:

Obiekt to dane, zwane polami (*fields*), atrybutami (*attributes*) lub właściwościami (***properties***), **wraz z operacjami**, jakie można na nich wykonać, zwanymi funkcjami (*functions*), funkcjami składowymi (*member functions* lub *members*) lub metodami (***methods***). Dane definiują bieżący **stan obiektu**, a jego metody – **interfejs**, za pomocą którego można odczytywać lub modyfikować ten stan.

3. Obiektowość doskonale nadaje się do programowania aplikacji z graficznym interfejsem użytkownika, gdyż takie aplikacje są zwykle przykładami programów sterowanych

zdarzeniami (*event-driven applications*): aplikacja oczekuje na zewnętrzne zdarzenia (np. kliknięcie myszy) i stosownie na nie reaguje, zmieniając swój wewnętrzny stan (np. wyświetlając kolejną stronę w przeglądarce WWW). Zupełnie inaczej zachowują się np. programy symulacyjne (w nauce), w których można wyróżnić dobrze zdefiniowany początek i koniec programu. Takie programy również mogą korzystać z bibliotek obiektowych, gdyż ułatwia to zapanowanie nad złożonością oprogramowania. Same programy sterujące symulacjami pisane są jednak zwykle nie obiektowo, lecz proceduralnie, gdyż nie ma potrzeby stosowania obiektowości w tym miejscu.

4. Gdzie w tym wszystkim znajduje się Qt i jak Qt ma się do C++? Z punktu widzenia Qt C++ jest metajęzykiem programowania, na bazie którego Qt stworzyło potężną bibliotekę oraz środowisko programistyczne. Programy w Qt pisane są w „C++ z rozszerzeniami Qt". Tych rozszerzeń jest naprawdę niewiele: sygnały, sloty, kilka makr, np. Q_OBJECT. Większe zmiany dotyczą środowiska: przed właściwą kompilacją, Qt przegląda pliki nagłówkowe (deklaracje klas) i na tej podstawie generuje dodatkowe pliki źródłowe, ukryte przed użytkownikiem. Analogicznie specjalne programy generują kod na podstawie definicji graficznego interfejsu użytkownika wyklikanego w QtCreatorze oraz pliki z kodem obsługującym zasoby. Dopiero po utworzeniu tych plików pomocniczych następuje właściwa kompilacja programu kompilatorem C++.

5. Jak się mają sygnały i sloty do funkcji wirtualnych?

   Sygnały to funkcje klas, które informują Qt o zmianie stanu obiektu. Źródłem sygnałów są więc obiekty. Otrzymując sygnał, Qt uzyskuje informację o tym, jaki to sygnał, z którego obiektu i z jakimi argumentami został on wysłany (sygnał to przecież zwyczajna funkcja, którą można wywołać z argumentami). Obiekt, wysyłający sygnał, jest jak radiostacja: nie wie, kto ten sygnał odbierze. Pośrednikiem między nadawcą i odbiorcą jest Qt.

   W przeciwieństwie do sygnałów, inicjatorem wywołania **funkcji wirtualnych** na danym obiekcie są inne obiekty (co prawda obiekt może wywołać własną funkcję wirtualną, ale nie bardzo wiadomo, po co). Funkcje wirtualne pełnią więc rolę funkcji wywoływanych zwrotnie (*call back functions*) przez „cudzy" kod wtedy, gdy ten cudzy kod uważa, że nasz obiekt może wymagać modyfikacji lub gdy musi potwierdzić swój stan. Funkcje wirtualne mogą zmienić stan obiektu (w pewnym sensie robi to np. *paintEvent*), ale nie muszą w obiekcie niczego zmieniać. W tym drugim przypadku obiekt może przynajmniej potwierdzić obiektowi, który od niego wymagał działania, że odebrał komunikat i go przetworzył. Na przykład nasze okno może potwierdzić, że odebrało komunikat o kliknięciu myszki, dzięki czemu system operacyjny może uznać, że kliknięcie zostało przetworzone przez właściwe okno i można je usunąć z kolejki nieprzetworzonych komunikatów. Warto w tym kontekście przyjrzeć się ostatniemu wierszowi implementacji metody event:

```
bool Dialog::event(QEvent * event)
{
    static int counter = 0;
    counter++;
    qDebug() <<  this->objectName() << counter << event->type();
    return QDialog::event(event);
}
```

   Idea jest prosta. Qt informuje mnie (funkcją wirtualną `event`), że zewnętrzny świat uznał, że mój obiekt może być zainteresowany jakimś zdarzeniem. Moja implementacja wyświetla tylko informację diagnostyczną i w ostatnim wierszu uruchamia standardową

implementację tego zdarzenia z klasy bazowej (tu: `QDialog`). Dzięki temu typowe zdarzenia, jak przyciśnięcia klawiszy myszki czy klawiatury, zostaną obsłużone w sposób standardowy, bez ryzyka zawieszenia programu (sprawdź, co się stanie, gdy usunie się ten fragment kodu!).

Dowolny obiekt w programie może poprosić, by Qt poinformowało go o wysłaniu określonego sygnału przez dowolny inny obiekt, wywołując na tym obiekcie określoną funkcję. Te funkcje w żargonie Qt nazywają się **slotami**. Przypominają one funkcje wirtualne tym, że są wywoływane z zewnątrz, w tym przypadku – przez Qt. Mechanizm sygnałów i slotów jest jednak znacznie bardziej elastyczny niż funkcje wirtualne, gdyż ani twórca klasy obiektu wysyłającego sygnał, ani klasy odbierającego sygnał w slocie, ani twórcy Qt nie muszą wiedzieć niczego konkretnego o cudzych modułach uczestniczących w transakcji sygnał/slot. W tym sensie separacja modułu zgłaszającego sygnał i modułu uruchamiającego slot jest zupełna. Ceną za to jest nieco większy koszt wywołania slotu w porównaniu do wywołania funkcji wirtualnej (10-100 razy w stosunku do biblioteki libsigc++, http://libsigc.sourceforge.net/benchmark.shtml). Ten koszt jest całkowicie akceptowalny w programach okienkowych. Qt może w ciągu sekundy przetworzyć grube tysiące transakcji sygnał/slot.

Reasumując, sygnały, sloty i funkcje wirtualne służą separacji modułów programu. Funkcje wirtualne umożliwiają obiektowi OA klasy A wywołać prawidłowe funkcje klasy B na obiekcie OB na podstawie niepełnych informacji o tym obiekcie (obiekt A nie potrzebuje pełnej informacji o klasie B obiektu OB). Jest to mechanizm wbudowany w jądro języka C++. Sygnały i sloty to z kolei mechanizm zaimplementowany w Qt w celu jak najpełniejszego odseparowania od siebie różnych modułów programu. Obiekt wysyłający sygnał nie musi wiedzieć niczego o obiekcie sygnał odbierającym; innymi słowy, obiekt wysyłający sygnał może uruchomić funkcję-slot w dowolnym innym obiekcie. Znacznie ułatwia to stosowanie metodologii programowania sterowanego zdarzeniami (*event-driven programming*) charakterystycznego dla aplikacji z graficznym interfejsem użytkownika.