ISSP/C++: tydzień 4

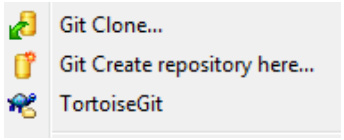
Zadanie: rozbudować program z katalogu https://github.com/zkoza/ISSP_CPP/tree/master/w4_virtual o kolejne figury i podpiąć je pod różne przyciski myszy, przy okazji dostępując iluminacji w zakresie funkcji wirtualnych C++

Cel: zrozumieć funkcje wirtualne i pseudowskaźnik `this`.

A. Zaczynamy od rozwiązania kwestii synchronizacji plików źródłowych. Listy zadań, programy i materiały pomocnicze niniejszego kursu udostępniane są na serwisie GitHub. Trudno znaleźć programistę, który przynajmniej o nim nie słyszał. Aby dostrzec w nim coś więcej niż magazyn plików z interfejsem webowym, uruchom terminal (Linux, Mac) lub zainstaluj program TortoiseGit (Windows). Utwórz katalog, w którym będziesz przechowywać pliki z GitHuba. Przejdź do tego katalogu. Uruchom polecenie (Linux, Mac OS)

```
> git clone https://github.com/zkoza/ISSP_CPP.git
```

Po kilku sekundach w katalogu bieżącym pojawi się katalog `ISSP_CPP` wraz z zawartością. Jest to dużo prostszy sposób aktualizowania repozytorium niż plik `*.zip` pobrany poprzez przeglądarkę WWW. W systemie Windows tę samą operację wykonuje się w programie TortoiseGit z menu kontekstowego (`Git Clone…`) menedżera plików. W kolejnych dniach/tygodniach aktualizację katalogu można wykonać komendą




```
> git pull
```

Jeżeli zajdzie sytuacja, że zmienisz coś w projekcie, po czym zechcesz przywrócić jego oryginalny stan, wystarczy wydać polecenie

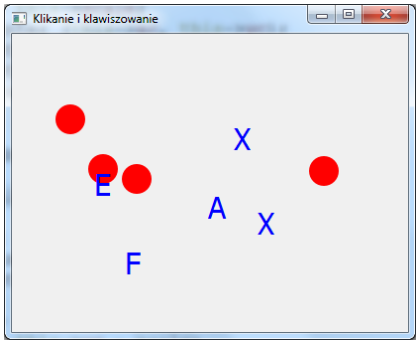
```
> git checkout .
```

Inne pożyteczne i bezpieczne komendy to `git status` oraz `git log`.

Warto jeszcze zauważyć, że GitHub nie jest jedynym (darmowym) serwisem udostępniającym miejsce na repozytoria w formacie `git`. Funkcję tę może pełnić każdy komputer na stałe wpięty do sieci i mający stały adres IP; w szczególności nasz serwer studencki – panoramix.

B. W kolejnym punkcie skoncentrujemy się na zrozumieniu programu udostępnionego w serwisie GitHub, a dopiero później spróbujemy ten program ulepszyć.

1. Załaduj do QtCreatora, skompiluj i uruchom program `w4_virtual`.
2. Przetestuj ten program. W tym celu:
   - klikaj różnymi przyciskami myszy;
   - następnie sprawdź działanie klawisza `backspace`;
   - następnie sprawdź, że działanie prawego klawisza myszy można kontrolować klawiaturą (litery A-Z).



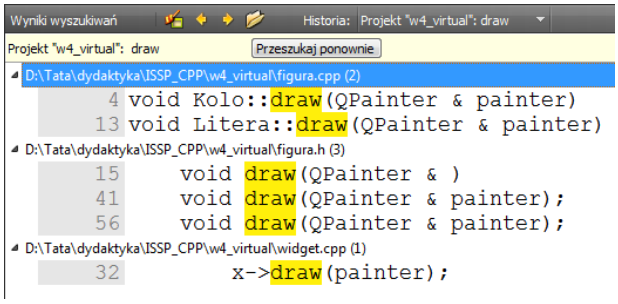
3. Otwórz plik `figura.h`. Zwróć uwagę na to, że:
   - Nazwa tego pliku nagłówkowego ma rozszerzenie h. Trzymaj się tej zasady, pozostawiając nazwy bez rozszerzeń bardzo znanym i dużym bibliotekom, jak Qt lub biblioteka standardowa C++. Wszyscy użytkownicy Twojego kodu będą ci za to wdzięczni.
   - W pliku tym znajdują się deklaracje trzech klas (czyli typów zmiennych użytkownika): `Figura`, `Kolo` i `Litera`. Nie jest to sytuacja typowa, gdyż różne klasy zwykle posiadają odrębne pliki nagłówkowe, jednak jeśli klasy są ze sobą ściśle związane, można je umieścić w tym samym pliku. Żadne z tych rozwiązań nie jest bezwzględnie lepsze, choć zasada „jedna klasa – jeden plik" wprowadza porządek. W naszym przypadku związek pomiędzy klasami polega na tym, że Figura jest klasą bazową klas Kolo i Litera:
     ```
     class Kolo : public Figura
     ...
     class Litera : public Figura
     ```
     Powyższa konstrukcja deklaruje więc, że każde `Kolo` jest przypadkiem specjalnym Figury; podobnie każda `Litera` jest przypadkiem specjalnym Figury.
   - W klasie `Figura` zadeklarowano dwie **funkcje wirtualne**: *`draw`*`(QPainter &)` oraz destruktor ~*Figura*(). Uwaga: większość klas z funkcjami wirtualnymi potrzebuje wirtualnego destruktora, nawet jeśli on nic nie robi. Dlaczego – o tym wkrótce.
     W klasach `Kolo` i `Litera` nie zadeklarowaliśmy funkcji *`draw`*`(QPainter &)` z atrybutem `virtual` – tym niemniej QtCreator wyświetlił jej nazwę pochyłą czcionką, gdyż pomimo braku jawnej deklaracji `virtual`, funkcja ta dziedziczy swoją wirtualność z klasy bazowej. Innymi słowy, jeśli klasa bazowa zadeklaruje swoją funkcję jako wirtualną, to klasa po niej dziedzicząca nie może anulować atrybutu `virtual`. Co więcej, kompilator na pewno sam wygenerował destruktory klas `Kwadrat` i `Litera`, mimo że wcale go o to nie prosiliśmy. Te destruktory są trywialne (pusty kod), ale z całą pewnością są i można je wywołać.
   - Dotychczas widział(a/e)ś wyłącznie przykłady, w których kod funkcji składowych klasy umieszczany był w plikach źródłowych. To jest DOBRA KONWENCJA: każdy sposób porządkowania kodu jest dobry. W moim kodzie kod funkcji umieściłem jednak w pliku nagłówkowym w deklaracji wszystkich funkcji klasy `Figura` i kilku funkcji pozostałych klas. Wniosek: można tak robić. Generalnie kod funkcji klas umieszcza się w ich deklaracjach w plikach nagłówkowych tylko wtedy, gdy jest on krótki i nie zaciemnia samej deklaracji lub gdy fizycznie nie da się go umieścić w pliku źródłowym (o tym później, hasło: szablony). To, że nie można z góry określić, w którym z dwóch plików znajduje się kod funkcji, jest jednym z powodów, dla których programiści używają zaawansowanych środowisk programistycznych, które ułatwiają nawigację w plikach programu.
4. Żeby docenić QtCreatora, otwórz plik `figura.cpp`, umieść kursor nad dowolną zmienną i wciśnij klawisz F2. Ten klawisz działa też zgodnie z oczekiwaniami w stosunku do nazw funkcji i klas, w tym funkcji i klas Qt.
5. Możesz przejrzeć pozostały kod programu, żeby nabrać ogólnego rozeznania, gdzie, co i jak się dzieje.
6. Czas na konkrety. Usuń obie deklaracje `virtual` z klasy `Figura`. Skompiluj i uruchom program. Nic się nie dzieje? Najgorsze jest nawet nie to, ze nic się nie dzieje, tylko że program został skompilowany niepoprawnie: informacja o utracie atrybutu `virtual` nie dotarła do wszystkich części programu i to cud, że on działa (u mnie). Powód: środowisko Qt usiłuje zminimalizować czas kompilacji, ograniczając ją do tych plików, które od ostatniej kompilacji zmie-

niły swoją treść lub włączają (#include) pliki o zmienionej treści. Wirtualność przekracza te granice.

7. Skompiluj program poprawnie: z menu wybierz Budowanie/Przebuduj wszystko. Uruchom i przetestuj program (lewy, prawy klawisz myszki, klawisz backspace na klawiaturze). W oknie „komunikaty aplikacji" QtCreator-a powinieneś zobaczyć komunikaty generowane funkcjami klasy podstawowej Figura:

```
qDebug() << "w funkcji" << __func__; // draw(...)
...
qDebug() << "deleting a Figure";     // destruktor
```

8. Zwróć uwagę na to, że po usunięciu modyfikatorów virtual, program nadal działa, ale uruchamia zupełnie inne funkcje draw. Przedtem uruchamiał egzemplarze a klas pochodnych (Kwadrat, Litera), teraz – z klasy podstawowej (Figura).
9. Żeby zorientować się, o co chodzi, przyciśnij Ctrl-F, wybierz Zaawansowane, w polu Wyszukaj wpisz draw, zaznacz Uwzględniaj wielkość liter oraz Tylko całe słowa, wciśnij Wyszukaj. U mnie pojawia się widok jak obok. Przy pewnej wprawie można od razu zobaczyć, że 5 pierwszych wystąpień słowa draw to deklaracje funkcji (por: void na początku wiersza). Jedyne miejsce, w którym funkcja draw jest wywoływana znajduje się w 32. wierszu pliku widget.cpp. Klikamy dwa razy we wpis, by ujrzeć kontekst tego użycia: plik widget.cpp. wiersze 31-35, funkcja Widget::*paintEvent* odpowiedzialna za wyświetlanie kółek i liter na obszarze roboczym aplikacji,




```
for (auto x : this->figury)
    x->draw(painter);
```

10. Co oznacza powyższy zapis? Z grubsza „na każdym elemencie kontenera this->figury uruchom draw(painter)". Element tego kontenera otrzymuje tymczasową nazwę x. Ponieważ następuje po nim operator ->, x musi być wskaźnikiem, czyli this->figury musi być kontenerem wskaźników. Słowo kluczowe auto w C++11 uzyskało zupełnie nowe znaczenie: „drogi kompilatorze, sam się domyśl, jaki jest typ zmiennej deklarowanej jako auto (tu: x) na podstawie sposobu jego inicjalizacji". Aby dowiedzieć się, co to jest this->figury, najeżdżamy kursorem nad identyfikator figury i przyciskamy F2. QtCreator przeprowadzi nas do następującej deklaracji (pik widget.h, wiersz 28):

```
std::vector<Figura*> figury;
```

11. Oznacza to, że figury jest standardowym (std::) wektorem (vector) wskaźników (*) na obiekty klasy Figura. Wynika z tego, że w powyższej pętli słowo kluczowe auto moglibyśmy zastąpić deklaracją Figura*:

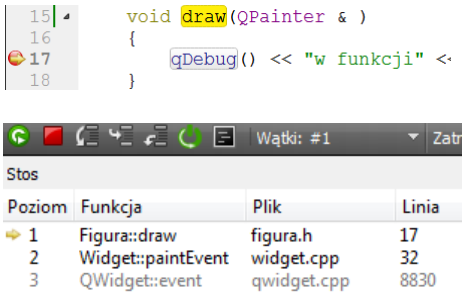
```
for (Figura* x : this->figury)
    x->draw(painter);
```

12. Jeżeli w klasie podstawowej Figura nie ma funkcji wirtualnych, kompilator w instrukcji x->draw(painter); wygeneruje wywołanie funkcji x->Figura::draw(painter), gdyż

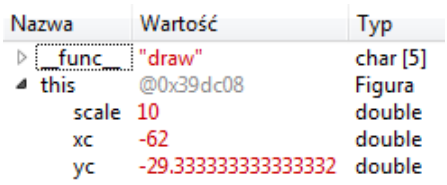
nie ma żadnych podstaw by przypuszczać, że programiście może chodzić o coś innego. Dlatego bez funkcji wirtualnych widzimy komunikaty generowane funkcjami klasy podstawowej `Figura`:

```
qDebug() << "w funkcji" << __func__; // draw(...)
...
qDebug() << "deleting a Figure";     // destruktor
```

13. Upewnij się, że nie masz funkcji wirtualnych. Ustaw pułapki debugera w funkcjach `draw` klas `Figura`, `Kwadrat`, `Litera`. Uruchom debuger. Kliknij lewy klawisz mysz. Debuger zatrzyma się w funkcji `Figura::draw`, w żaden sposób nie wywołasz zaś funkcji `Kolo::draw` lub `Litera::draw`. W momencie zatrzymania programu na pułapce, stos programu będzie wyglądał jak na rysunku obok. Oznacza to, że bieżąca funkcja, `Figura::draw`, została wywołana bezpośrednio z funkcji `Widget::paintEvent`. A nie mówiłem?



14. Istnieje jeszcze jeden sposób, by zorientować się w sytuacji. W prawym górnym rogu QtCreatora powinno znajdować się okienko „Zmienne lokalne i wyrażenia”. (jeśli się nie wyświetla, można je włączyć z menu `Okno/Widoki`). Rozwiń zawartość wskazywaną przez pseudowskaźnik `this`. Powinieneś zobaczyć, że wg debugera bieżący obiekt jest typu `Figura` (kolumna Typ) i składa się z 3 składowych: `scale`, `xc` i `yc`. Nie jest to zgodne z tym, co zapisano w programie w pliku `widget.cpp`:




```
case Qt::LeftButton:
{
    Kolo * kolo = new Kolo(x, y, 10);
    this->figury.push_back(kolo);
    break;
}
case Qt::RightButton:
{
    Litera * kwadrat = new Litera(x, y, 10, this->jaki_znak());
    this->figury.push_back(kwadrat);
    break;
}
```

Jak widać, w kontenerze `figury` umieszczane są (funkcją `push_back`) wyłącznie obiekty klasy `Koło` lub `Litera`. Obiekty tej drugiej klasy posiadają nie tylko składowe `scale`, `xc` oraz `yc`, ale i `litera`. Wniosek jest taki, że w pętli

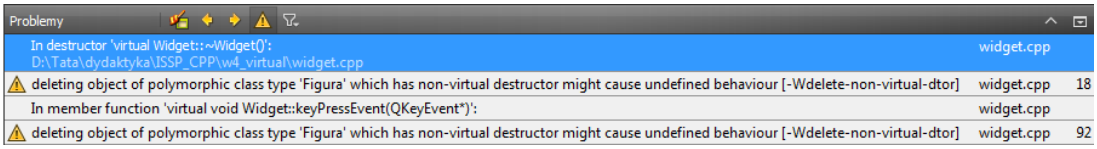
```
for (Figura* x : this->figury)
    x->draw(painter);
```

kompilator nie ma pełnej informacji o rzeczywistym typie obiektu wskazywanego przez `x`. Nie jest to jeszcze błąd. Subtelny błąd czai się w destruktorze klasy `Widget`:

```
Widget::~Widget()
{
    delete ui;
    for (auto x : this->figury)
        delete x;
}
```

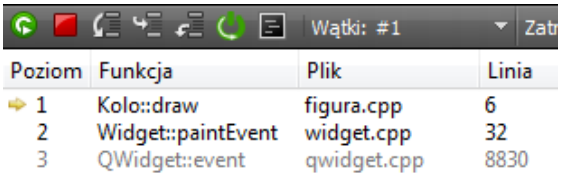
Tutaj, jak łatwo sprawdzić, kompilator wygeneruje operator `delete` dla obiektów klasy podstawowej (`Figura*`), co nie odpowiada klasie użytej podczas tworzenia obiektów (operator new) jako obiektów klas pochodnych, co może być źródłem subtelnych błędów (ale to dość zaawansowany temat).

15. Przywróć w deklaracji funkcji `Figura::draw` atrybut `virtual`. Jedyny prawidłowy sposób skompilowania programu po tej zmianie to wybranie z menu opcji `Budowanie/Przebuduj wszystko`. Kompilator powinien odpowiedzieć ostrzeżeniem:



Tłumacząc na polski, kompilator ostrzega, że jeżeli w klasie choć jedna funkcja jest wirtualna, jej destruktor też powinien być wirtualny. Dodaj więc atrybut `virtual` do destruktora i przebuduj program (`Budowanie/Przebuduj wszystko`). Sprawdź, że znów działa i wyświetla kółka oraz litery.

16. Ponownie ustaw pułapki debugera w funkcjach `draw` klas `Figura`, `Kwadrat`, `Litera`. Pamiętaj, że pułapki możesz ustawiać tylko w kodzie funkcji (a nie w jej deklaracji). Uruchom debuger. Kliknij lewy klawisz mys. Tym razem Debuger zatrzyma się w funkcji klasy pochodnej `Kolo::draw` (bez funkcji wirtualnych była to funkcja `Figura::draw`). Co ciekawe, w żaden sposób jako użytkownik programu nie wywołasz już funkcji `Figura::draw`. W momencie zatrzymania programu na pułapce, stos programu powinien wyglądać jak na rysunku obok. Oznacza to, że bieżąca funkcja, `Kolo::draw`, została wywołana bezpośrednio z funkcji `Widget::paintEvent`, mimo że kompilator nie wiedział, czy pętla




```
for (Figura* x : this->figury)
   x->draw(painter);
```

przetwarza obiekty klasy `Kolo`, `Litera` czy może nawet `Figura` i w jakiej kolejności. Mimo że kompilator nie ma pełnej informacji o rzeczywistym typie obiektów wskazywanych przez wskaźnik `x`, w jakiś magiczny sposób potrafi wygenerować kod odnoszący się do rzeczywistego typu obiektu. Oznacza to, że to obiekt musi zawierać informację o swoim rzeczywistym typie – tylko jeśli posiada funkcje wirtualne! – i że kompilator musi takie funkcje wywoływać w specjalny sposób, uwzględniający tę informację.

17. Spójrz na zawartość obiektu wskazywanego przez `this` po zatrzymaniu programu w funkcji `Kolo::draw`. Debuger podaje, że `this` wskazuje na obiekt klasy `Kolo` (pierwszy wiersz, kolumna Typ), który dziedziczy z klasy `Figura` (drugi wiersz). Jak każda `Figura`, obiekt ten zawiera składowe `scale`, `xc` i `yc`. Jednak wśród składowych obiektu wskazywanego przez `this` pojawia się też nowa składowa: dodatkowe pole oznaczone jako `[vptr]`. Pola tego nie było w programie pozbawionym funkcji wirtualnych. Na podstawie jego wartości, zapisanej jako liczba szesnastkowa (`0x...`), domyślamy się, że `[vptr]` jest wskaźnikiem (adresem zmiennej), a rozwinięte pola oznaczone jako `[0]`, `[1]` i `[2]` informują nas, że `vptr` jest adresem początku tablicy o trzech elementach. Kompilator dodaje ten wskaźnik niejawnie, poza kontrolą programisty, do każdego obiektu klasy zawierającej choć jedną funkcję wirtualną; wskazywana przezeń tablica jest unikatowa dla każdej klasy zawierającej funkcje wirtualne. Wartość wskaźnika `vptr` ustalana jest w momencie konstrukcji obiektu i pozwala zidentyfikować rzeczywisty typ obiektu. Kolejne elementy tablicy to adresy funkcji wirtualnych przypisanych dodanego obiektu (na podstawie jego klasy). W ten sposób wywołanie funkcji `draw` w pętli

Nazwa	Wartość	Typ
▷ center	(1.5000000006, 6.87025845266e-308)	QPointF
▷ painter	@0x28b384	QPainter
r	1.809527667808319e-307	double
◢ this	@0x3fdc08	Kolo
◢ [Figura]	@0x3fdc08	Figura
◢ [vptr]	0x4076f0 <vtable for Kolo+8>	
[0]	0x401f28 <Kolo::draw(QPainter&)>	
[1]	0x403988 <Kolo::~Kolo()>	
[2]	0x403968 <Kolo::~Kolo()>	
scale	10	double
xc	-45.333333333333336	double
yc	-30	double

```
for (Figura* x : this->figury)
    x->draw(painter);
```

przebiega następująco: (1) odczytaj wartość `vptr` z obiektu wskazywanego przez x; (2) przejdź do tablicy wskaźników wskazywanych przez `vptr`; (3) w pierwszym polu tej tablicy znajdziesz adres tej funkcji `draw`, którą należy wywołać na obiekcie wskazywanym przez `x`.

18. Sprawdź, jak powyższe uwagi odnoszą się do obiektów klasy `Litera`. Usuń wszystkie pułapki z wyjątkiem tej w funkcji `Litera::draw`. Uruchom program pod kontrola debugera. Przyciśnij prawy przycisk myszy, który generuje wyświetlenie się obiektu klasy `Litera`. W okienku „zmienne lokalne i wyrażenia” ujrzysz wpis jak na rysunku obok. Jak widać, bieżący obiekt jest obiektem klasy `Litera` (zgadza się!) i jako taki posiada wszelkie cechy obiekty klasy `Figura` (na tym polega dziedziczenie!), a do tego składową `litera` (o domyślnej wartości `'X'`). Ponadto adres zapisany w `vtable` różni się od adresu z poprzedniego punktu. Tym razem w wyrażeniu `x->draw(painter)` użyta zostanie funkcja `Litera::draw`, czyli funkcja odpowiadająca rzeczywistej klasie obiektu. Funkcja ta swobodnie może korzystać m.in. ze składowej `litera`, której istnienia w momencie kompilowania wyrażenia `x->draw(painter)` nie można było się domyślić.

◢ this	@0x13f6dc08	Litera
◢ [Figura]	@0x13f6dc08	Figura
▷ [vptr]	0x407640 <vtable for Litera+8>	
scale	10	double
xc	23.333333333333332	double
yc	-31.333333333333332	double
litera	88 'X'	char

Wątki: #1

Poziom	Funkcja	Plik	Linia
1	Litera::draw	figura.cpp	15
2	Widget::paintEvent	widget.cpp	32
3	QWidget::event	qwidget.cpp	8830

19. Spróbujmy podsumować użycie w niniejszym programie funkcji wirtualnych:
    - W klasie `Widget` utworzyliśmy kontener `figury` przechowujący wszystkie obiekty, jakie mają się wyświetlać na ekranie. Kontener ten dostępny jest dla wszystkich funkcji klasy, w tym dla `paintEvent` (rysowanie zawartości widżetu), `mousePressEvent` (dodawanie i usuwanie elementów graficznych wyświetlanych w programie) oraz `keyPressEvent` (modyfikacja wyświetlanej litery).
    - Ponieważ zawartość kontenera nie jest jednolita (litery, różne kształty geometryczne, w przyszłości mapy bitowe, etc.), definiujemy **hierarchię klas** z wirtualną metodą `draw` (i wirtualnym destruktorem).
    - Klasa bazowa hierarchii, `Figura`, służy jako wspólny **interfejs** dla wszystkich użytecznych składników programu. Nie mamy jednak zamiaru używać bezpośrednio obiektów tej klasy! (por. zad. D.3 poniżej).
    - Użyteczne klasy wyprowadzamy przez dziedziczenie z klasy `Figura`, tu są to klasy `Litera` i `Kolo`, i podpinamy obiekty tych klas pod różne przyciski myszy.
    - Żeby to działało, kontener `figury` nie może przechowywać obiektów: standardowo w takiej sytuacji używa się wskaźników (kontener wskaźników). Inne rozwiązania trudno pogodzić z wirtualnością, natomiast wskaźniki idealnie z nią harmonizują, gdyż **wskaźnikowi na klasę podstawową można przypisać wskaźnik na dowolną klasę pochodną**.
    - Dzięki takiemu projektowi programu, implementacje kodu rysującego okno aplikacji

      ```
      for (auto x : this->figury)
          x->draw(painter);
      ```

      czy też metody usuwania klawiszem `backspace` ostatniego elementu graficznego

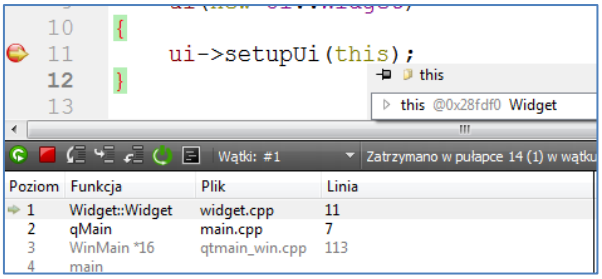
      ```
      Figura * fig = this->figury.back();
      delete fig;
      figury.pop_back();
      ```

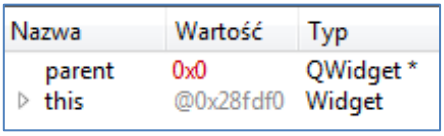
      są wręcz trywialne. Kod, który je realizuje, nie musi wiedzieć, co dokładnie wykonuje funkcja `draw` czy też destruktor wywoływany operatorem `delete`. Ten kod nie musi nawet wiedzieć, na jakich rzeczywistych obiektach pracuje! Co więcej, może być skompilowany w innym miejscu i dużo wcześniej, zanim ktoś wymyśli klasy `Litera`, `Kolo` czy inne klasy pochodne klasy `Figura`. Dokładnie na tej zasadzie działa Qt, które potrafi ze swojego kodu binarnego wywołać nasze funkcje `paintEvent`, `mousePressEvent` czy `keyPressEvent`. Jeśli to rozumiesz i potrafisz stosować w praktyce, można uznać, że rozumiesz i potrafisz stosować paradygmat programowania obiektowego.

C. Tematem biezącego tygodnia jest nie tylko polimorfizm (czyli funkcje wirtualne) C++, ale też pseudowakaźnik `this`.
    1. Znajdź w programie dowolny pseudowskaźnik `this`, po którym występuje operator wyłuskania (`->`). Usuń go wraz z operatorem. Sprawdź, że program się kompiluje i działa dokładnie tak samo. Możesz usunąć wszystkie wystąpienia `this`, po których występuje operator wyłuskania (`->`), efekt będzie identyczny: program tego nawet nie zauważy.
    2. Istnieją sytuacje, gdy psudowskaźnik `this` wraz z operatorem `->` jest niezbędny, jednak w bardzo zaawansowanych kontekstach. Ja używam go tu wyłącznie w celu poprawienia czytelności kodu. Normalnie się go nie używa, jeśli nie jest to konieczne.

3. Ale co to jest ten `this`? Przywróć wszystkie wystąpienia `this->`. Ustaw (jedyną) pułapkę w konstruktorze klasy `Widget` na instrukcji

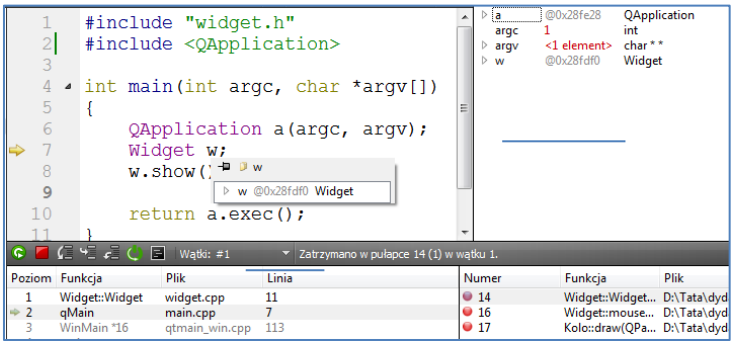
```
ui->setupUi(this);
```





Uruchom debuger. Po dojściu do pułapki możesz najechać kursorem na `this` i po chwili odczytać jego wartość, jak na rysunku obok. U mnie jest to `@0x28fdf0` wyświetlane szarymi literami, co oznacza, że jest to zapis uproszczony przez QtCreatora. W tym przypadku oznacza to, że this wskazuje na adres 0x28fdf0. Tę samą wartość można odczytać w okienku „Zmienne lokalne i wyrażenia", jak na rysunku obok.

4. Kliknij dawa razy w okenku Stos na drugą pozycję stosu (qMain w pliku mani.cpp, linia 7). QtCreator wyświetli kod funkcji main, jak na rysunku obok. W znany już sposób odczytujemy adres obiektu `w`: 0x28fdf0. Wniosek: w konstruktorze klasy `Widget` wartością `this` jest adres obiektu w zdefiniowanego w 7. Wierszu funkcji `main`. Jest to obiekt główny naszej aplikacji, jedyny klasy Widget i to do niego Qt wysyła wszelkie komunikaty, uruchamiając finkcje wirtualne funkcje `paintEvent`, `mousePressEvent` oraz `keyPressEvent`. **Pseudowskaźnik `this` jest po prostu <u>adresem obiektu</u>, na rzecz którego wywołano daną <u>funkcję składową klasy</u>.**



5. W analogiczny sposób jak w poprzednim punkcie, acz bardziej samodzielnie, sprawdź, że wartością `this` w instrukcji

```
double r = this->scale;
```

(plik `figura.cpp`, funkcja `void Kolo::draw(QPainter & painter)`) jest adres zmiennej x z instrukcji

```
x->draw(painter);
```

(plik `widget.cpp`, funkcja `void Widget::paintEvent(QPaintEvent * )`).

D. Czas na pracę twórczą i samodzielną.

1. Rozbuduj (samodzielnie!) omawiany tu program o kolejny rodzaj figur i podpnij je pod środkowy przycisk myszy. Te figury to mogą być kwadraty (łatwe), trójkąty (nieco trudniejsze) lub obrazki (chyba najambitniejsze). Uwaga: nową klasę umieść w nowym pliku!
2. Kto powiedział, że czego nie było na wykładzie, to nie obowiązuje? Wyszukaj informacje o modyfikatorze `override` (C++11) i twórczo zastosuj je w swoim programie. Przekonaj prowadzącego ćwiczenia, że z grubsza rozumiesz, dlaczego wprowadzono to udogodnienie (i że w ogóle jest to udogodnienie).
3. Wyszukaj informacje o czystych funkcjach wirtualnych i klasach abstrakcyjnych (*pure virtual functions*; *abstract classes*) i zamień funkcję `draw` klasy podstawowej `Figura` na czystą funkcję wirtualną. Spróbuj przekonać prowadzącego ćwiczenia, że z grubsza rozumiesz ideę, jaka przyświeca temu mechanizmowi.