Laboratorium 9 - szablony i kontenery – STL i Qt

Szablony

Szablon (ang. *template*) – element języka C++, umożliwiający programowanie uogólnione – tworzenie kodu niezależnego od typów, algorytmów oraz struktur danych.

Przykład szablonu klasy:

```
template <typename T>
class Type{
    T value;
public:
    T getValue(){
        return value;
    }
    void setValue(T v){
        value = v;
    }
};
```

Powyższy kod tworzy klasę `Type` która posiada pole `value` typu `T` oraz 2 metody które umożliwiają dostęp do tego pola. Typ `T` jest w tym momencie nie znany zostanie podany w miejscu użycia Oczywiście możliwe jest stworzenie szablonu który posiada więcej niż jeden parametr, w takim przypadku podaje się je po przecinku.

Użycie:

```
Type<int> type1;
Type<float> type2;
type1.setValue(10); //argument typu int
type2.setValue(10.1f); //argument typu float
int value1 = type1.getValue();
float value2 = type2.getValue();
```

Podczas kompilacji następuje tak zwana konkretyzacja szablonu (ang. template instantiation), podczas której kompilator na podstawie typów danych przekazanych wzorcowi generuje kod właściwy do obsługi danego typu. W powyższym przypadku zostaną wygenerowane 2 wersje klasy jedna dla `int` druga dla `float`.

Ćwiczenie 1

Ćwiczenie zrealizuj jako program konsolowy.

1. Utwórz klasę `Pies` oraz `Kot` zawierające po jednej metodzie zwracającej łańcuch znaków zwierający rodzaj zwierzęcia.
2. Utwórz szablon klasy `Buda` zawierający wskaźnik do typu będącego parametrem szablonu oraz metody pozwalające na dostęp do niego (get* i set*)
3. W funkcji `main` utwórz kota i psa oraz budę dla kota i budę dla psa, umieść zwierzęta we właściwych miejscach.

Szablony – parametry

Parametrami szablonów mogą być nie tylko nazwy typów, ale również stałe.

Przykład:

```
template <int capacity, float factor>
class Type{
    int array[capacity];
public:
    int getValue(int i){
        return factor * array[i];
    }
};
```

Użycie:

```
Type<3, 0.5f> type;
```

W trakcie kompilacji powyższego przykładu odpowiednie stałe zostaną umieszczone w wygenerowanym kodzie w miejscu capacity i factor. Ponieważ dzieje się to na etapie kompilacji mogą zostać użyte tylko wartości znane w czasie kompilacji.

Ćwiczenie 2

Ćwiczenie zrealizuj jako program konsolowy.

Zmodyfikuj klasę buda z poprzedniego przykładu tak aby było możliwe przetrzymywanie więcej niż jednego zwierzątka.

1. Dodaj parametr szablonu typu `int` oznaczający pojemność budy.
2. Zamień wskaźnik do zwierzątka na tablicę wskaźników
3. Zmodyfikuj metody dostępowe tak aby umożliwiały operowanie na wybranym zwierzęciu (poprzez podanie jego indeksu)

Utwórz jeszcze po jednym zwierzątku, i umieść je w odpowiednich budach.

Kontenery

Kontener (lub inaczej pojemnik, ang. *container*, *collection*) - struktura danych, której zadaniem jest przechowywanie w zorganizowany sposób zbioru danych (obiektów). Kontener zapewnia narzędzia dostępu, w tym dodawanie, usuwanie i wyszukiwanie danej (obiektu) w kontenerze. W zależności od przyjętej organizacji, poszczególne kontenery różnią się wydajnością poszczególnych operacji.

Najprostszym kontenerem, oferowanym przez większość języków jest tablica.

Kontenery STL

Standard Template Library, **STL** – biblioteka C++ zawierająca algorytmy, kontenery oraz inne konstrukcje w formie szablonów, gotowe do użycia w programach.

Jednym z najczęściej stosowanych kontenerów z biblioteki STL jest `vector` – jednowymiarowa dynamiczna tablica. Klasa `vector` jest szablonem którego parametrem jest typ przechowywanych wartości.

Przykład tworzenia `vectora` dla typu `int`:

```
#include <vector>
// ...
vector<int> v1, v2(10); // 10 - początkowy rozmiar tablicy (domyślnie 0)
```

Najważniejsze metody klasy `vector`:

- `size()` - zwraca aktualny rozmiar
- `rezise(int)` – pozwala zmienić aktualny rozmiar
- `clear()` - usuwa wszystkie elementy oraz ustawia rozmiar na 0
- `push_back(T)` – dodaje element na koniec zwiększając jednocześnie rozmiar o 1

Opis pozostałych metod można znaleźć tutaj:

http://www.cplusplus.com/reference/vector/vector/

Dostęp do poszczególnych elementów odbywa się za pomocą operatora [ ] jak w przypadku zwykłych tablic.

Przykład

```
vector<int> v(1); // 10 - początkowy rozmiar tablicy (domyślnie 0)
v[0] = 1;
v.push_back(2);
cout << v[0] << v[1]; // 12
```

Ćwiczenie 3

Ćwiczenie zrealizuj jako program konsolowy.

Pobieraj od użytkownika słowa i wkładaj je do `vectora` dopóki użytkownik nie wpisze „exit”. Następnie wyświetl wszystkie wpisane słowa.

Kontenery STL – cd.

Inne popularne kontenery znajdujące się w bibliotece STL:

- queue – kolejka
- dequeue – double-ended queue – kolejka dwukierunkowa
- map – tablica asocjacyjna

Kontenery Qt

Biblioteka Qt posiada własne implementacje kontenerów. Jedną z takich klas jest `QList<T>` oraz dziedzicząca po niej klasa `QStringList`. Ciekawostką jest to że dla klasy `QStringList` został przeciążony operator << dzięki czemu można dodawać do niej elementy jak do strumienia.

Przykład:

```
QStringList list;
list << "aaa" << "bbb";
```

Ćwiczenie 4

Ćwiczenie zrealizuj jako aplikację Qt Widgets.

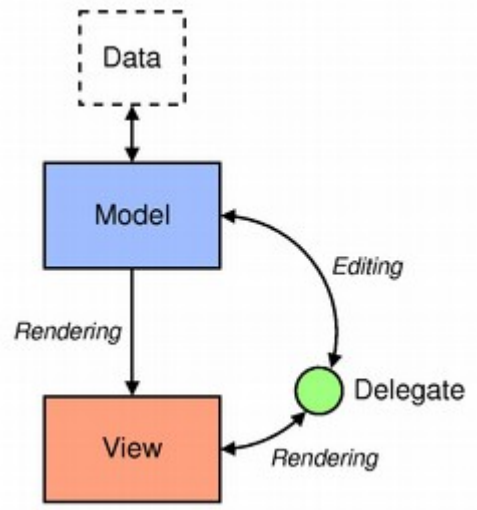
1. W designerze dodaj do okna kontrolkę List Widget tak aby zawsze zajmowała całe okno nadaj jej nazwę `lista`
2. W klasie `MainWindow` utwórz metodę przyjmującą referencję do `QStringList`, która doda do listy 4 dowolne ciągi znaków.
3. W konstruktorze utwórz instancję klasy `QStringList`, korzystając z utworzonej metody dodaj elementy do listy.
4. Wykorzystując metodę addItems z kontroli List Widget wyświetl elementy listy.

Program przetestuj.

Model - Based list

Przedstawiona powyżej kontrolka `QListWidget` należy do grupy kontrolek Item - Based.

W Qt 4 wprowadzono nowy rodzaj kontrolek używających architektury model/view (Model - Based), aby ułatwić zarządzanie relacjami pomiędzy danymi a sposobem ich prezentacji. Separacja modelu od sposobu prezentacji daje programiście możliwość dostosowania widoku do własnych potrzeb i wielokrotne używanie go z różnymi rożnymi źródłami danych.



Architektura model/view

Model komunikuje się ze źródłem danych, oraz dostarcza interfejs dla innych komponentów architektury. Sposób komunikacji ze źródłem danych zależy od jego typu oraz sposobu implementacji modelu

Widok korzystając z modelu uzyskuje referencje (*model indexes*) do elementów źródła danych. Poprzez dostarczenie referencji widok może pobrać elementy danych ze źródła.

W standardowych widokach delegat odpowiada za renderowanie elementów. Kiedy element jest edytowany delegat informuje o tym model.

Klasy model/view można przydzielić do trzech grup widoki, modele oraz delegaty. Każdy z tych komponentów jest zdefiniowany poprzez abstrakcyjną klasę która dostarcza wspólny interfejs, w niektórych przypadkach także domyślną implementację.

Modele, widoki i delegaty komunikują się między sobą używając sygnałów i slotów

- sygnały z modelu informuje widok o zmianach danych
- sygnały z widoku dostarczają informacji o interakcji użytkownika z pokazywanymi elementami
- sygnały z delegatów używane są podczas edycji aby poinformować widok i model o stanie edytora

Modele

Bazową klasą dla wszystkich modeli jest klasa `QAbstractItemModel`. Jednakże implementując model dla listy warto skorzystać z rozszerzającej ją `QAbstractListModel`, która domyślną implementację dla niektórych funkcji.

Istnieją też gotowe implementacje modeli takie jak klasa `QStringListModel` która świetnie się sprawdzi jeśli naszym źródłem danych jest `QStringList`.

Widoki

Biblioteka Qt zawiera implementację różnych rodzajów widoków np.:

- `QListView`
- `QTableView`
- `QTreeView`

Wszystkie one dziedziczą po klasie `QAbstractItemView` która zawiera metodę `setModel` pozwalającą na ustawienie modelu.

Więcej informacji na temat architektury model/view tutaj:

http://doc.qt.io/qt-4.8/model-view-programming.html

Ćwiczenie 5

1. Usuń kontrolkę `QListWidget`, a w jej miejsce wstaw `QListView`
2. Utwórz `QStringListModel` na podstawie listy utworzonej w poprzednim zadaniu.
3. Ustaw model w widoku