Języki programowania+, 11.01.2018
Zadanie 11 - ostatnie

Dzisiaj wykorzystamy kolejną bardzo ważną rzecz języka C++ - szablony funkcji i szablony klas.

Wstęp – co to są klasy i funkcje szablonowe w języku C++?

Rozważmy prostą klasę, która przechowuje jakiś typ informacji, na przykład tablicę `int`'ów:

```
class KlasaInt
{
      int *tablica
      int rozmiar;
      public:
      KlasaInt(int Rozmiar);
      void UstawElement(int poz, int wartosc);
      int PobierzElement(int poz);
};

KlasaInt::KlasaInt(int Rozmiar)
{
      rozmiar = Rozmiar;
      tablica = new int[Rozmiar];
}
void KlasaInt::UstawElement(int poz, int wartosc)
{
      tablica[poz]=wartosc;
}
int KlasaInt::PobierzElement(int poz)
{
      return tablica[poz];
}

int main()
{
      KlasaInt kl(10);
      kl.UstawElement(5) = 20;
      cout<<kl.PobierzElement(5)<<endl;


      return 0;
}
```

Co by należało zrobić, gdybyśmy potrzebowali przechować w identyczny sposób dodatkowo inne typy danych. np. `double`, `std::string`, `char*`, etc.? Musielibyśmy za każdym razem, dla każdego oddzielnego typu danych, który jest nam potrzebny, tworzyć nową klasę (np. `KlasaString`, gdzie mielibyśmy `string *tablica` zamiast `int *tablica` itp.). Napisanie sporej ilości tego typu klas zajęłoby nam pewną ilość czasu oraz zwiększyło znacznie ilość plików w projekcie (można sobie wyobrazić duży projekt, gdzie każdy dodaje klasy różniące się właściwie tylko typem przechowywanych danych a służących do tych samych celów – ogrom klas!). Na szczęście język C++ przychodzi nam tu z rozwiązaniem takiego problemu i pozwala tworzyć klasy szablonowe (ang. *template class*), które dostosowują się automatycznie do typu danych używanego w programie głównym. Przykład klasy szablonowej analogicznej do przykładu powyżej jest następujący:

```
template<class T>
class KlasaSzablonowa
{
      T *tablica
      int rozmiar;
      public:
      KlasaSzablonowa(int Rozmiar);
      void UstawElement(int poz, T wartosc);
      T PobierzElement(int poz);
};

template<class T>
KlasaSzablonowa<T>::KlasaSzablonowa(int Rozmiar)
{
      rozmiar = Rozmiar;
      tablica = new T[Rozmiar];
}
```

```
template<class T>
void KlasaInt<T>::UstawElement(int poz, T wartosc)
{
	tablica[poz]=wartosc;
}

template<class T>
T KlasaInt<T>::PobierzElement(int poz)
{
	return tablica[poz];
}

int main()
{
	KlasaSzablonowa<int> klInt(10);
	klInt.UstawElement(5) = 20;
	cout<<klInt.PobierzElement(5)<<endl;

	KlasaSzablonowa<string> klString(10);
	klString.UstawElement(5) = "tekst";
	cout<<klString.PobierzElement(5)<<endl;

	return 0;
}
```

Załóżmy teraz, że w naszej klasie szablonowej mamy również metodę `Dodaj`, która zwraca sumę elementów dwóch obiektów z wybranych pól tablicy `tablica`, czyli:

```
template<class T>
T KlasaSzablonowa<T>::Dodaj(int poz1, int poz2)
{
	T obiekt = tablica[poz1]+tablica[poz2];
	return obiekt;
}
```

Taka metoda zadziała dla prostych typów danych, np. `int`, `double`, `string`, itp. (czyli takich, gdzie mamy zdefiniowany operator + i takie działanie ma sens). Co by się jednak stało, gdyby nasza klasa przechowywała na przykład napisane wcześniej obiekty klasy `Histogram`?(czyli coś bardziej skomplikowanego)? Taka metoda oczywiście nam nie zadziała dla histogramu (nie zdefiniowaliśmy przeciążonego operatora + dla klasy `Histogram` i jego użycie wyrzuciłoby nam błąd). Nic nie szkodzi! Język C++ przynosi nam i tutaj rozwiązanie – można tworzyć specjalizowane metody w klasach szablonowych, oto przykład metody `Dodaj` dla histogramów:

```
template<>
Histogram KlasaSzablonowa<Histogram>::Dodaj(int poz1, int poz2)
{
	Histogram c = tablica[poz1];
	for(int i=0;i<c->IloscBinow())
		c.UstawLiczbeZliczen(i,tablica[poz1]->PobierzLiczbeZliczen(i)
+tablica[poz]->PobierzLiczbeZliczen(i));
	return c;
	//to tylko przykład - tu robimy cokolwiek z obiektami typu Histogram, np.
tworzymy wyjsciowy histogram gdzie w kazdym binie liczba zliczen jest suma z
histogramow tablica[poz1] i tablica[poz2]
}
```

W naszej klasie możemy mieć wiele specjalizowanych metod (również wiele specjalizowanych metod o tej samej nazwie – dla różnych obiektów). Kompilator w trakcie kompilowania programu sam zdecyduje, którą metodę ma użyć (w przypadku wszystkich typów obiektów użyje metody ogólnej `Dodaj` z operatorem +, ale jeśli nasza klasa będzie akurat przechowywać histogramy, to automatycznie użyje metody specjalizowanej dla klasy `Histogram`).

Uwaga!

W przypadku klas szablonowych wszystko (zarówno definicję klasy jak i metody klasy) piszemy tylko w pliku **.h** (czyli najczęściej dla jednej klasy jeden plik .h) – nie możemy pisać definicji metod w pliku **.cpp**!

W języku C++ oprócz klas szablonowych możemy również tworzyć funkcje szablonowe. Np. załóżmy, że mamy kilka klas, które mają taką samą metodę (dajmy na to `int GetRozmiar()`). Chcielibyśmy stworzyć funkcję, która porównuje rozmiar dwóch obiektów dowolnej z tych klas i zwraca większą wartość. Nic trudnego, możemy dostawić więcej typów danych:

```
template<class T1, class T2>
int CompareRozmiar(T1 obiekt1, T2 obiekt2)
{
```

```
	if(obiekt1.GetRozmiar()>obiekt2.GetRozmiar())
		return obiekt1.GetRozmiar();
	else
		return obiekt2.GetRozmiar();
}
```

Oczywiście możemy dodawać więcej typów klas w nagłówku `template`, możemy je też dowolnie nazywać (np. `template<class A, class B, class C, class D>` itp.).

Zadanie do wykonania

Część I

W naszym zadaniu wykorzystamy napisaną wcześniej klasę `Histogram` tworzącą listę. Klasę tą napisaliśmy tak, by mogła przechowywać jedynie dane typu `int`. Teraz natomiast, zamiast pisać dwie lub więcej klas, chcemy nasz histogram uogólnić, by mógł przyjmował elementy dowolnego typu (np. wynikiem dzielenia dwóch histogramów przez siebie może być histogram z niecałkowitą liczbą zliczeń w binach). Jak to zrobić? Musimy go przerobić właśnie na klasę szablonową. Na koniec, aby przećwiczyć dziedziczenie i abstrakcyjne typy danych (ATD), należy również stworzyć abstrakcyjną klasę bazową `THistogram` dla klasy szablonowej `Histogram`.

2 p.

Część II

Chcemy rozszerzyć jej działanie napisanej na wcześniejszych zajęciach klasy `List` na elementy dowolnego typu.

Zasada działania naszej klasy z punktu widzenia programu będzie praktycznie identyczna jak listy ze standardowej biblioteki szablonów STL, np.:

```
List<double> *myList = new List<double>(); //tworzy liste przechowujaca typ double za pomoca naszej klasy List
list<double> *myListSTL = new list<double>(); //tworzy liste przechowujaca typ double za pomoca klasy z biblioteki STL
```

Do wykonania:

- przerobienie klasy `List` tak, by stała się klasą szablonową. Stworzenie w programie listy przy użyciu naszej klasy oraz listy z biblioteki STL na zmiennych typu `double`, dodanie do list 3 elementów, wypisanie obu list. Przy tworzeniu klasy `List` stworzyliśmy pomocniczą klasę `Element` – chcemy, aby klasa `Element` była zadeklarowana wewnątrz klasy `List` (klasa w klasie).

2 p.

Uwaga!

Należy posiadać poprawnie działającą wersję klasy `List` oraz poprawnie działającą klasę z histogramem – `Histogram`.

Klasy szablonowe piszemy **tylko w plikach .h** – nie tworzymy pliku **.cpp** (ponieważ klasa szablonowa jest kompilowana dopiero przy jej użyciu w trakcie wykonywania programu).

Używamy pliku `Makefile` w najbardziej zaawansowanej postaci.

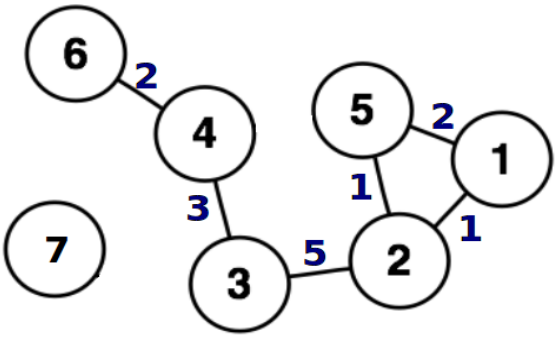
Przydatne linki:

http://cpp0x.pl/kursy/Kurs-C++/Szablony-klas/317

http://4programmers.net/C/Szablony_klas

Część III – do skończenia i przećwiczenia w domu 1 p.
Wyszukiwanie ścieżki w grafie. Macierz sąsiedztwa. Algorytm Djikstry. Czytanie pseudokodu.

W teorii grafów **macierz sąsiedztwa** grafu *G* jest kwadratową macierzą w której a_{ij} oznacza wagę krawędzi pomiędzy wierzchołkami *i* i *j*. Dla przykładu – graf z 7 wierzchołkami:



$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 & 0 & 0 \\ 1 & 0 & 5 & 0 & 1 & 0 & 0 \\ 0 & 5 & 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 & 0 & 2 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

1. Należy stworzyć klasę graf używając dwuwymiarowej tablicy (macierz sąsiedztwa) do przechowywania informacji o jej krawędziach. Klasa powinna mieć następujące metody:

`Graf(int max)` – konstruktor ; `max` – maksymalna ilość wierzchołków w danym grafie. W przypadku osiągnięcia tej liczby dodanie nowego wierzchołka nie będzie możliwe.

`void init()` - zeruje wszystkie pola macierzy sąsiedztwa

`void add()` - dodanie wierzchołka

`void remove(int r)` - usunięcie wierzchołka o numerze r

`void display()` - wyświetlenie grafu (macierzy sąsiedztwa)

`void add_edge(int n, int m, int w=1)` - dodanie krawędzi pomiędzy wierzchołkami o numerach m i n (oraz wagą w)

`void remove_edge()` - usunięcie krawędzi (dwa numery wierzchołków powinny być wprowadzane z klawiatury po wywołaniu metody)

oraz destruktor.

W funkcji main stworzyć graf odpowiadający podanemu powyżej w przykładzie.

Macierz sąsiedztwa powinna być zaimplementowana jako dwuwymiarowa tablica z dynamicznie alokowaną pamięcią.

2. Należy zaimplementować algorytm Djikstry do wyszukiwania najkrótszej ścieżki w grafie jako metodę klasy graf na podstawie podanego na następnej stronie pesudokodu.
`void dijkstra(int source, int target)` - wyszukiwanie najkrótszej ścieżki od source do target

Oraz przetestować go dla stworzonego wcześniej grafu, oraz ścieżek: (1,3), (6,7), (6, 5).

3. Rozszerzyć klasę graf w ten sposób, by maksymalna liczba wierzchołków w grafie zmieniała się, jeśli będziemy chcieli dodać kolejny wierzchołek, przekraczający tę liczbę (odpowiednie zwalnianie i alokacja pamięci!).

```
function Dijkstra(Graph, source, target):
      //deklarujemy wszystkie tablice

      for each vertex v in Graph: // Inicjalizacje, dla każdego wierzchołka.
        distance[v] := infinity ; // Odległość od source do v (nieznana, zakładamy nieskończoność)
        previous[v] := -1 ;       // Tablica zapisująca wierzchołki na najkrótszej ścieżce
        VISITED[v] := 0;            // Żaden z wierzchołków nie był odwiedzony. Np. jeśli
      end for ;                          wierzchołek „i” odwiedzony: VISITED[i]=1.

      distance[source] := 0 ;     //Odległość od punktu poczatkowego = 0
      if source = target             // Jeśli source jest równe target - ten sam wierzchołek
            print "The same vertex"
            exit                     //Zakończyć działanie programu

      int tmp := infinity;  //nieskończoność to np. 999999999
      int visited_nodes := 0;       //licznik odwiedzonych wierzchołków
      int current = source; //current = obecny. Startujemy od wierzchołka początkowego source

      while visited_nodes < Graph size  //póki nie sprawdzimy wszystkich wierzchołków w grafie
```

```
            for each node x of Graph
                    if node x was not visited and there exist edge between x and current
                            if distance[current] + edge weight between current and x < distance[x]
                                    distance[x] := distance[current] + distance between current and
x
                                    previous[x] := current
            end for

            VISITED[current] := 1                //Wierzchołek current odwiedzono
            visited_nodes := visited_nodes + 1   //zwiększyć licznik odwiedzonych wierzchołków

            for each vertex v in Graph           //znajdujemy nowy wierzchołek current
                    if vertex v was not visited and distance[v] < tmp
                            tmp := distance[v]
                            current := v
            end for

            tmp := infinity

            if current = target    //doszliśmy do punktu końcowego!
                    print "Path found"  //wypisać na ekran „ścieżka zaleziona” i wypisać ścieżkę
                    int u := target;
                    while previous[u] != -1
                            print u
                            u := previous[u]
                    print source
      end while ;
      print "Path not found"              //ścieżka pomiędzy source i target nie istnieje

  end Djikstra
```

Słowniczek:
edge – krawędź (linia łącząca dwa wierzchołki), ma wagę (**weight**)
vertex – wierzchołek, charakteryzowany przez numer wierzchołka
source – punkt startowy
target – punkt końcowy