# 1. Wprowadzenie do szablonów funkcji i klas (templates)

**Szablony** *(ang. Templates)* w innych obiektowych językach programowania (Java, C++) nazywane też **Typami generycznymi** *(ang. Generic Types)* lub **Typami uogólnionymi**, to element języka C++ pozwalający na tworzenie kodu niezależnego od typów, algorytmów oraz struktur danych. Szablony są techniką realizacji polimorfizmu na innym poziomie niż za pomocą funkcji wirtualnych i dziedziczenia.

Mechanizm szablonów można rozumieć jako pewną formę makrodefinicji (preprocesora). Szablony ze względu na szerokie rozpowszechnienie się zastosowań biblioteki **STL** *(ang. Standard Template Library)* uważane są za jedną z najważniejszych właściwości języka C++. Dzięki szablonom mamy możliwości **programowania ogólnego** *(ang. generic programming)*. Tworząc szablon funkcji (podobnie jak szablon klasy) tworzymy kod działający na nieopisanych jeszcze typach – funkcja działa na typach ogólnych (które nie istnieją w języku C++), w momencie wywołania funkcji pod te typy ogólne podstawiane są konkretne typy, jak np. **int** czy **char**. Przekazujemy szablonowi typy jako parametry, podczas kompilacji następuje tak zwana **konkretyzacja szablonu** *(ang. template instantiation)*, podczas której kompilator na podstawie typów danych przekazanych wzorcowi generuje docelowy kod do obsługi danego typu.

Przykłady implementacji tej samej funkcji dla różnych typów (bez użycia szablonów):

```
// Wersja 1 int:
void Zamień(int &x, int &y) {
int temp = x;
x = y;
y = temp;
}

// Wersja 2 double:
void Zamień(double &x, double &y) {
double temp = x;
x = y;
y = temp;
}
```

```
// Wersja 3 string:
void Zamień(string &x, string &y) {
string temp = x;
x = y;
y = temp;
}
```

Dzięki wykorzystaniu szablonów programista C++ może skupić się bardziej na algorytmach niż na danych, które są przetwarzane. Użycie szablonów pozwala zredukować ilość nadmiarowego kodu, ponieważ jedną funkcjonalność można zaprogramować dla wielu typów danych. Kompilator w kolejnym kroku precyzuje określony typ oraz dostosowuje odpowiednio wywołania funkcji, metod czy tworzenia obiektów na podstawie klas.

Wadą działanie szablonów w języku C++ jest to, że przypominają one bardzo zaawansowane makra preprocesora. Powoduje to, że kompilator ma zasadnicze trudności z wygenerowaniem prawidłowych, czytelnych komunikatów diagnostycznych w przypadku błędnego użycia poprawnego szablonu.

Szablony w C++ możemy podzielić na:

- **Szablony funkcji**,
- **Szablony klas**.

## 1.1. Szablony funkcji

Szablon funkcji to mechanizm umożliwiający automatyczną generację funkcji. Jest to schemat, według którego postępuje kompilator. Programista dostarcza jedynie ogólny zarys ciała funkcji bez określania konkretnych argumentów wywołania. Na tej podstawie kompilator jest w stanie wytworzyć dowolną ilość funkcji działających identycznie, a różniących się jedynie typem argumentów funkcji. Szablony zezwalają na definiowanie całych rodzin funkcji, które następnie mogą być używane dla różnych typów argumentów.

Definicja szablonu funkcji wygląda następująco:

```
template <typename T> T MojaFunkcja(T a, T b, ...) {
//instrukcje
...};
lub
template <class T> T MojaFunkcja(T &a, T &b, ...) {
//instrukcje
...};
```

Definicja umieszczona bezpośrednio przed definicją funkcji informuje, że funkcja będzie korzystała z fikcyjnego typu o nazwie T. Definicja dotyczy tylko pojedynczej funkcji zdefiniowanej bezpośrednio po frazie „template".

Przykłady implementacji funkcji dla różnych typów z użyciem szablonów:

```
// Listing 1
template <typename T> T Sprawdz(T a, T b) {
return (a>b)?a:b;
};
// Listing 2
template <typename T> T Minimum(T a, T b, T c) {
if (a <= b && a <= c) return(a);
if (b <= a && b <= c) return(b);
return(c);
};
// Listing 3
template <class T> void Zamień(T &x, T &y) {
T temp = x;
x = y;
y = temp;};
```

Wyrażenie template <typename T> lub template <class T> oznacza, że mamy do czynienia z szablonem, który posiada jeden parametr formalny nazwany T. Słowo kluczowe typename/class oznacza, że parametr ten jest typem (nazwą typu) lub klasy. Nazwa tego parametru może być następnie wykorzystywana w definicji funkcji w miejscach, gdzie spodziewamy się nazwy typu/klasy. I tak powyższe wyrażenie (*Listing 1*) definiuje funkcję, która przyjmuje dwa argumenty typu T i zwraca wartość typu T, będącą wartością większego z dwu argumentów. Typ T jest na razie niewyspecyfikowany. W tym sensie szablon definiuje całą rodzinę funkcji. Konkretną funkcję z tej rodziny wybieramy poprzez podstawienie za formalny parametr T konkretnego typu będącego argumentem szablonu. Takie podstawienie nazywamy konkretyzacją szablonu. Argument szablonu umieszczamy w nawiasach <> za nazwą szablonu.

## 1.2. Szablony klas

Na podobnej zasadzie co szablony funkcji można także definiować szablony klas (inne określenia: wzorce klas, klasy ogólne). Szablony klas są podobne do makr, tyle że wykonywane są przez kompilator, a nie przez preprocesor. Cechują je pewne własności,

których jednak nie ma preprocesor, np. można tworzyć rekurencyjne wywołania. Podobnie jak w przypadku szablonów funkcji, szablon klasy definiuje nam w rzeczywistości całą rodzinę klas.

Definicja szablonu klasy wygląda następująco:

```
template <class T> class MojaKlasa {
//definicja klasy };
```

Definicja umieszczona bezpośrednio przed definicją klasy informuje, że klasa będzie korzystała z fikcyjnego typu o nazwie T. Definicja dotyczy tylko pojedynczej klasy zdefiniowanej bezpośrednio po frazie „template".

Przykłady implementacji i użycia klas z użyciem szablonów:

```
// Listing 4
template <class T> class c_klasa {
public:
T zmienna;
c_klasa(T l) { zmienna = l; }
void wyswietl() { cout << zmienna << endl; }
};

int main() {
c_klasa<int> calkowita(2);
calkowita.wyswietl();
c_klasa<char> znak('b');
znak.wyswietl();
}

// Listing 5
template <class T> class c_klasa {
public:
T zmienna;
c_klasa(T l);
void wyswietl();
};

template <class T>
c_klasa<T>::c_klasa(T l) : zmienna(l) {}
template <class T>
void c_klasa<T>::wyswietl() { cout << zmienna << endl; }
```

```
int main() {
c_klasa<int> calkowita(2);
calkowita.wyswietl();
c_klasa<char> znak('b');
znak.wyswietl();
}
```

## 2. Wyjątki (exceptions)

**Wyjątki** (ang. Exceptions) to mechanizm obsługi sytuacji wyjątkowych (zazwyczaj błędów). W trakcie działania programu zawsze może dojść do powstania sytuacji nietypowej, trudnej do uniknięcia. Można jednak przewidywać, iż taka sytuacja wystąpi i się na nią odpowiednio przygotować. Tak jak Java, C# i wiele innych języków obiektowych, język C++ umożliwia obsługę wyjątków, czyli sytuacji, gdy powstaje w czasie wykonania programu błąd uniemożliwiający dalszy jego przebieg (na przykład dzielenie przez zero, wywołanie metody za pomocą pustego wskaźnika, błąd odczytu/zapisu strumienia). Normalnie, powstanie takiej sytuacji powoduje przerwanie programu, zwykle z mało czytelnym komunikatem o naturze błędu. Obsługa wyjątków pozwala na zdefiniowanie przez programistę akcji, które należy podjąć po powstaniu sytuacji wyjątkowej (w szczególności programista może świadomie podjąć decyzję o ich całkowitym zignorowaniu).

W języku C++ wprowadzono mechanizm pozwalający programiście na reagowanie na sytuacje błędne czy nietypowe. Gdy taka sytuacja wystąpi, programista może wygenerować wyjątek. Wyjątek to **obiekt** pewnej klasy. Wygenerowanie wyjątku polega na przekazaniu obiektu opisującego wyjątek z fragmentu kodu, w którym wystąpił problem, do fragmentu, w którym przewidziano jego obsługę. Wygenerowanie (zgłoszenie) wyjątku powoduje przerwanie wykonywania sprawiającego problemy kodu i przejście do obsługi sytuacji problematycznej. Obsługa ta może znajdować się w innym miejscu kodu. Wyjątek jest obiektem, jego klasa określa typ sytuacji wyjątkowej. Obiekt może w sobie posiadać pola oraz funkcje składowe, pozwalające na sprecyzowanie informacji o zaistniałej sytuacji wyjątkowej.

Wyjątki w C++ możemy podzielić na:

- **Wyjątki własne**,
- **Wyjątki standardowe**.

## 2.1. Blok try...catch

Do obsługi wyjątków wykorzystuje się blok **try...catch**. Definicja bloku **try...catch** wygląda następująco:

```
try {
     // "Wyrzucenie" wyjątku:
     if(coś poszło nie tak)
          throw wyjątek;
     // KOD
}
catch (typ nazwa) {
     // Obsługa wyjątku
}
```

- **throw** – jest operatorem, po nim następuje wyrażenie, które określi rodzaj wyjątku, mogą być różne typy/klasy wyjątków,
- **catch** – może pojawić się tylko po **try** lub po innym **catch**, w nawiasie określamy typ wyjątku i zmienną/obiekt, która przekazuje informacje o wyjątku.

Jeżeli do obsługi błędu wystarczy sam fakt jego wykrycia, to podajemy typ nie podając zmiennej/obiektu. Blok po catch nazywamy *procedurą obsługi wyjątku.* Procedura obsługi wyjątku może znajdować się w tej samej funkcji, w której następuje zgłoszenie wyjątku.

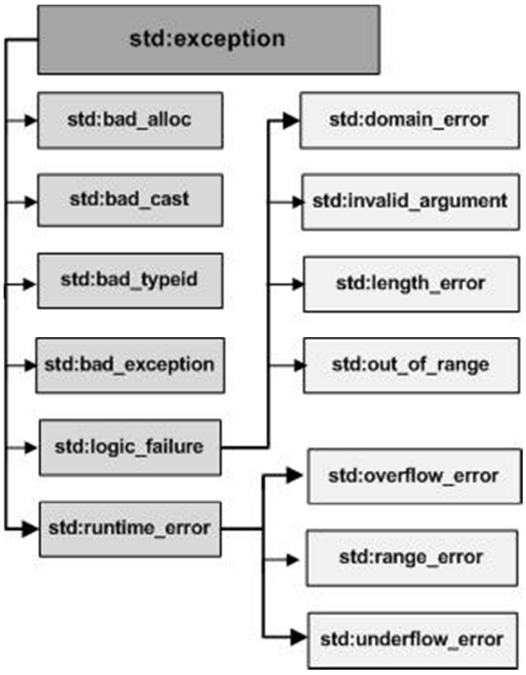
Wyjątki wysłane w zakresie instrukcji **try**, ale nieprzechwycone przez odpowiedni blok **catch** zostają odebrane przez funkcję obsługi z parametrem *„wielokropek”* **catch(...)**. Taki blok łapie wszystkie błędy (pozostałe niepasujące do innych bloków catch błędy).

```
try {
     // KOD
}
catch (int ex) {
  cout << "wyjątek int";
}
catch (float ex) {
  cout << "wyjątek float";
}
catch (...) {
  cout << "wyjątek domyślny";
}
```

Przykłady implementacji i użycia bloku try...catch:

```
// Listing 1
try {
     int n;
     cout << "Podaj liczbę: ";
     cin >> n;

     // Czy ujemna?
     if (n < 0)     throw 1;
     // Czy zero?
     if (n == 0)    throw 2;
     // Czy za duża?
     if (n > 100000) throw 3;

     // Wszystko OK:
     cout << "Kwadrat liczby = " << n*n << endl;
}
catch (int x) {
     cout << "Wyjątek nr " << x << endl;
}

// Listing 2
int FunkcjaWczytująca() {
     try {
          int A, B;
          cout << "Podaj dwie liczby: ";
          cin >> A >> B;

          if (A < 0)     throw "Liczba A ujemna!";
          if (B < 0)     throw "Liczba B ujemna!";
          if (B == 0)    throw "Dzielenie przez zero!";

          int wynik = A / B;

          if (wynik == 0)     throw "Wynik zerowy!";
          cout << "WYNIK = " << wynik << endl;
     }
     catch (char *s) {
          cout << "WYJĄTEK: " << s << endl;
     }
}
```

W języku C++ można zdefiniować własne klasy wyjątków (tworząc własne klasy lub dziedzicząc po standardowej klasie **exception**), albo do opisu wyjątku wykorzystać już istniejące typy/klasy standardowe. W bibliotekach klas korzystających z wyjątków zazwyczaj korzysta się ze specjalnych klas do określania rodzaju wyjątków. Stosowanie klas jako typów wyjątków jest wygodne, możemy tworzyć klasy o nazwach opisujących typy wyjątków. Klasa standardowa **std::exception** to klasa bazowa wyjątków wykorzystywanych przez wiele klas standardowych. Najważniejsza metoda klasy to metoda **what()**, która zwraca tekstowy opis wyjątku.




Przykłady implementacji i użycia własnej klasy wyjątku:

```
// Listing 3
#include <iostream>
#include <exception>

using namespace std;

class MojWyjatek: public exception {
  virtual const char* what() const throw() {
    return "Moj wyjatek";
  }
};
```

```
int main () {
  MojWyjatek myex;

  try  {
    throw myex;
  }
  catch (exception& e) {
    cout << e.what() << endl;
  }
  return 0;
}
```

Przykłady implementacji i użycia klasy standardowej wyjątku:

```
// Listing 4
#include <exception>
using namespace std;


try {
     int* myarray = new int[1000];
}
catch (exception& e) {
     cout << "Wyjątek: " << e.what() << endl;
}
```

## 3. Zadania do wykonania

1. Napisać program, w którym zostaną zaimplementowane trzy funkcje sortujące (algorytmem bąbelkowym, przez wstawianie oraz *quicksort*) z wykorzystaniem szablonów funkcji. Należy zaprezentować ich zastosowanie dla różnych rodzajów danych wejściowych. Funkcje powinny przyjmować jako parametr tablice do posortowania oraz liczbę elementów w tablicy.
2. Napisać program, w którym zostanie zaimplementowana klasa operująca na **stosie** z wykorzystaniem szablonów klas. Należy zaprezentować działanie tak stworzonego stosu dla różnych rodzajów danych wejściowych.
3. Napisać program, w którym zostanie zaimplementowana klasa operująca na **liście** z wykorzystaniem szablonów klas. Należy zaprezentować działanie tak stworzonej listy dla różnych rodzajów danych wejściowych.

4. Napisać program, który będzie modyfikacją zadania nr 9 (instrukcja nr 6) implementującego słownik angielsko-polski przy pomocy kontenera map. Należało wówczas napisać program, który umożliwi wykonywanie takich operacji jak: umieszczanie słowa angielskiego wraz z odpowiednim tłumaczeniem w słowniku, wyszukanie słowa angielskiego i wyświetlenie odpowiednika polskiego, wyświetlenie zawartości słownika na ekranie, przy użyciu prostego menu. Należy tak zmodyfikować program, aby używane były w nim szablony funkcji/klas. Należy zaprezentować działanie tak stworzonego słownika dla różnych rodzajów danych wejściowych.

5. Napisać program wyliczający pierwiastki równania kwadratowego w postaci:

$$ax^2 + bx + c = 0,$$

Program powinien pytać użytkownika o podanie trzech parametrów **a**, **b** oraz **c**. W programie należy wykorzystać obsługę wyjątków zabezpieczającą program przed nietypowymi sytuacjami jak: *dzielnie przez zero*, *pierwiastek z liczby ujemnej* itp. itd.

6. Napisać program, w którym zostaną zaimplementowane **własne** klasy wyjątków. Klasy te należy wykorzystać do obsługi wyjątków w programie wyliczającym następujące wzór matematyczny:

$$Wynik = \frac{\sqrt{A}}{B}$$

Należy zaprezentować działanie tak stworzonych klas dla różnych rodzajów danych wejściowych.

7. Napisać program, który będzie modyfikacją zadania nr 7 (instrukcja nr 6) implementującego stos przy pomocy kontenera list. Należało wówczas napisać program, który umożliwi wykonywanie takich operacji jak: umieszczanie elementu na stosie, zdejmowanie elementu ze stosu, wyświetlanie zawartości stosu, przy użyciu prostego menu. Należy zabezpieczyć ten program tak, aby obsługiwał wyjątki.

8. Napisać program, który będzie modyfikacją zadania nr 8 (instrukcja nr 6) implementującego listę uczniów w klasie przy pomocy kontenera set. Należało wówczas napisać program, który umożliwi wykonywanie takich operacji jak: umieszczanie ucznia na liście, usuwanie ucznia z listy, wyświetlanie listy uczniów, przy użyciu prostego menu. Należy zabezpieczyć ten program tak, aby obsługiwał wyjątki.