

Szablony

- Szablony są mechanizmem ponownego wykorzystania kodu (reuse)
- W przypadku funkcji – ponownie wykorzystany jest algorytm
- W przypadku klas – ponownie wykorzystane są wszystkie składowe
- Deklaracja szablonu specyfikuje zbiór parametryzowanych klas lub funkcji

Szablony funkcji

Deklaracja szablonu ma postać:

```
template < [typelist] [, [arglist]] > declaration
```

Lista parametrów szablonu jest oddzielona przecinkami (lista typów, identyfikatorów klas lub nazw typów) oraz ewentualnie wartościami, które mają być wykorzystane w treści szablonu

Przykład szablonu funkcji:

```
template <typename T>  
T mmin( T a, T b )  
{ return ( a < b ) ? a : b; }
```

- Użycie szablonu wymaga jego konkretyzacji (ustalenia wartości dla wszystkich generycznych parametrów)
- Konkretyzacja może odbywać się:
 - Niejawnie (na podstawie typów parametrów aktualnych)
 - Jawnie (przez podanie typów parametrów aktualnych)
- Niejawna konkretyzacja funkcji ma miejsce w czasie jej wywołania; wartości parametrów generycznych są ustalane na podstawie parametrów aktualnych

Przykład niejawnej konkretyzacji szablonu funkcji min:

```
int a = 5, b = 10;           double a = 5.0, b = 10.0;
cout << mmin(a,b);         cout << mmin(a,b);
```

Jawna konkretyzacja wymaga, aby, a po nazwie funkcji, w nawiasach <>, zdefiniować wartości aktualne dla parametrów generycznych

Przykład jawnej konkretyzacji szablonu funkcji:

```
char x='a'; int y = 66;
cout << mmin<char>(x,y); // wymagana konwersja z int do char
```

Specjalizacja szablonu

- Szablony można specjalizować, zmieniając ich działanie w zależności od typu parametrów, na którym działają

```
template <typename T>  
void Demo( T a ) {} // definicja funkcji szablonej
```

```
template<>  
void Demo[<typ>](typ a) {} // specjalizacja szablonu dla  
//konkretnego typu
```

Przykłady różnych specjalizacji szablonów:

```
template <typename T>  
void f(T t)  
{ cout << "Wersja ogolna \n"; }
```

```
template <>  
void f<char>(char t) // specjalizacja funkcji f dla  
{ cout << "Wersja char \n"; } // typu char
```

```
template <>  
void f(double t){} // specjalizacja funkcji f dla typu  
// double
```

```
template <typename T, typename I>  
void Demo( T a, I b ) { cout << "Any \n"; }
```

```
template <typename I>  
void Demo<>(double a, I b) { cout << "Double \n"; }
```

Parametrem szablonu może być argument, który nie jest typem. Argumenty, które nie opisują typów (non-type) podlegają następującym **ograniczeniom**:

- Typ argumentu jest typem całkowitym, wyliczeniowym, referencją lub wskaźnikiem
- W kodzie szablonu nie można modyfikować wartości argumentu, ani pobierać jego adresu
- Przy konkretyzacji szablonu wartość parametru musi być stałą

Przykład szablonu funkcji z parametrem, który nie jest typem:

```
template<class T, int zakres>  
int szukaj(T tablica[], T co){}
```

```
const int ile = sizeof(tab1) / sizeof(int);  
cout << szukaj<int, ile>(tab1,2) << endl;
```

Zastosowanie szablonów funkcji dla własnych typów (klas) może wymagać przeciążenia odpowiednich operatorów:

```
template <class T>
T min(T a, T b)
{
    return( a < b ) ? a : b;
}
```

```
class X
{
public:
    int a, b;

    X(int a=0):a(a){};
    int operator<( X &other) { return a < other.a; };
};
```

```
X a,b,c;
c = min(a,b);
```

Kolejność dopasowywania wywołań funkcji szablonowych przez kompilator (mechanizm rozstrzygnięcia przeciążeń):

- Tworzona jest lista funkcji kandydujących o tej samej nazwie
- Spośród funkcji kandydujących wybierane są te, które mają
 - odpowiednią listę parametrów – istnieje niejawna sekwencja konwersji, zawierająca przypadek dokładnego dopasowania każdego typu parametru aktualnego do typu odpowiedniego parametru formalnego
 - Spośród różnych funkcji, wybierana jest taka, której argumenty są najlepiej dopasowane (zwykle funkcje mają pierwszeństwo przed wzorcami, konwersja przez promocję, np. `char` lub `short` w `int`, `float` w `double`, konwersja standardowa, np. `int` w `char`, `long` w `double`, konwersja zdefiniowana przez użytkownika)
- Jeżeli uda się znaleźć szablon - generowana jest odpowiednia wersja funkcji
- Jeżeli szablonu nie uda się znaleźć – zgłaszany jest błąd kompilacji

Szablony klas

Szablon klasy służy do tworzenia rodziny klas, które operują na pewnym typie (typach) - parametrach szablonu

```
template <typename T, int i>
class TempClass
{
public:
    TempClass(void);
    ~TempClass(void);
    int MemberSet( T a, int b );
private:
    T Tarray[i];
    int arraysize;
};
```

- Szablon klasy ma 2 parametry, 1-szy typ T, 2-gi wartość typu int
- Wartością aktualną T może być dowolny typ
- Wartością aktualną drugiego parametru może być stała int

Definicja funkcji klasy szablonowej:

```
template<deklaracja_parametrów_szablonu>  
typ_wyniku nazwa_klasy<param_szablonu>::nazwa_skladowej ([param])  
{ ... }
```

Przykład definicji klasy szablonowej:

```
template <class T, int i>  
int TempClass<T, i>::MemberSet(T a, int b)  
{  
    if( b >= 0 && b < arraysize )  
        return ( Tarray[b] = a );  
    else  
        return -1;  
}
```

Konkretyzacja klasy szablonowej może być:

- **niejawna** (przy deklarowaniu obiektów z szablonu klasy),
- **jawna** (tworzenie konkretnego obiektu klasy bez natychmiastowego wykorzystania, np. w bibliotekach)

Konkretyzacja niejawna klasy szablonowej ma postać (konkretyzacja niejawna):

```
klasa_szablonowa<parametry_aktualne> obiekt_klasy_konkretnej;
```

Przykład konkretyzacji klasy szablonowej (konkretyzacja niejawna):

```
TempClass<int, 10> klasa1; // parametry aktualne szablonu: int, 10  
TempClass<char, 5> klasa2; // parametry aktualne szablonu: char, 5
```

Konkretyzacja klasy szablonowej ma postać (konkretyzacja jawna):

```
template class klasa_szablonowa<parametry_aktualne>;
```

Przykład konkretyzacji klasy szablonowej (konkretyzacja jawna):

```
template class TempClass<char, 3>; // parametry aktualne szablonu: char, 3  
template class Klasa<int>; // parametry aktualne szablonu: int
```

- Taka deklaracja musi być umieszczona w tej samej przestrzeni nazw, co definicja szablonu;
- Klasa jest generowana zawsze, nawet, gdy nie tworzymy obiektów tej klasy
- Zostaną wygenerowane klasy:
 - `TempClass<char, 3>...`
 - `Klasa<int>`

Szablon klasy może definiować domyślne parametry typu:

```
template <class T2 = typ> deklaracja
```

```
template <typename T = char>
class Kwadrat
{
    void rysuj( T znak ){}
};
```

Możliwe konkretyzacje takiego szablonu:

- `Kwadrat<int> klasa1(4);`
 - `klasa1.rysuj(4);`
- `Kwadrat<> klasa2(5);` `// wykorzystanie typu domyślnego`
 - `klasa2.rysuj('*');`

Szablony są często używane:

- Do tworzenia klas obsługujących kolekcje (np. stos), które manipulują na danych dowolnych typów
- Jako klasy bazowe, jak i typy obiektów składowych

Zalety szablonów:

- łatwe do napisania
- łatwe do zrozumienia
- type-safe - kompilator wykonuje wszelkie niezbędne sprawdzenia parametrów

Szablony znacząco redukują rozmiar kodu źródłowego i zwiększają jego elastyczność nie zmniejszając bezpieczeństwa związanego ze sprawdzaniem typów