

Wykład 8

Programowanie generyczne

dr Marcin Denkowski

Lublin, 2019

AGENDA

1. Programowanie generyczne
2. Szablony funkcji
3. Szablony klas

PARADYGMATY PROGRAMOWANIA

- Sposób patrzenia na przepływ sterowania i wykonanie programu
- Główny podział
 1. Programowanie imperatywne (jak wykonywać)
 2. Programowanie deklaratywne (co wykonywać)
(logiczne, funkcyjne, np. *prolog*, *haskel*, *lisp*, *ocaml*, *erlang*)

PARADYGMATY PROGRAMOWANIA

- Paradygmat imperatywny
 1. Programowanie strukturalne
 2. Programowanie proceduralne
 3. Programowanie obiektowe
 4. Programowanie generyczne

PROGRAMOWANIE GENERYCZNE

- Programowanie uogólnione (*generic programming*)
 - algorytmy, które działają w kategorii meta-typu
 - meta-typy są konkretyzowane podczas tworzenia instancji tych algorytmów
 - w języku C++ ta technika nosi nazwę szablonów (*templates, parametric polymorphism, wzorce*)

SZABLON FUNKCJI

1. Szablon funkcji – ogólny opis funkcji operująca na typach ogólnych, pod które podstawiane są typy konkretne

```
template<class T>  
T max(T a, T b)  
{  
    if ( a > b) return a;  
    return b;  
}
```

SZABLON FUNKCJI

1. Szablon funkcji – ogólny opis funkcji operująca na typach ogólnych, pod które podstawiane są typy konkretne

```
template<class T>
T max(T a, T b)
{
    if ( a > b) return a;
    return b;
}
```

1. Szablon funkcji – nie jest funkcją, na jego bazie kompilator generuje funkcje w razie potrzeby

```
int main()
{
    int a = 5, b=4;
    int c = max<int>(a,b);
}
```

```
int max(int a, int b)
{
    if ( a > b) return a;
    return b;
}
```

INSTANCJE SZABLONU

- Jawna instancja szablonu

```
template int max(int a, int b);
```

- Niejawna instancja funkcji poprzez domniemanie typu przy wywołaniu (dla typu float)

```
max(0.1f, 0.5f);
```

- Jawna instancja funkcji przy wywołaniu (dla typu Element)

```
Element e1, e2;  
e = max<Element>(e1, e2);
```


INSTANCJE SZABLONU

```
template<class T> max(T a, T b)
{
    if ( a > b) return a;
    return b;
}
```

int max<int>(int a, int b);

int a, b;
int w = max<int>(a,b);

float max<float>(float a, float b);

float a, b;
float w = max<float>(a,b);

Kwadrat max<Kwadrat>(Kwadrat a, Kwadrat b);

Kwadrat a, b;
Kwadrat w = max<Kwadrat>(a,b);

SPECJALIZACJE JAWNE

- Specjalizacja jest jawną instancją danego szablonu funkcji

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

- Specjalizacja (przykrywa szablon)

```
template<> Vector max<Vector>(Vector a, Vector b) {  
    return (len(a) > len(b) ? a : b);  
}
```

- Nieszablon (przykrywa szablon i specjalizację)

```
max(Element a, Element b ) {  
    return ( len(a) > len(b) ? a : b);  
}
```

DOPASOWYWANIE FUNKCJI

Mamy wywołanie:

`funct('A');`

Która funkcja/szablon
zostanie dopasowana?



Oraz szereg deklaracji:

- 1) **void** *funct*(**int**);
- 2) **float** *funct*(**float**, **float** = 3);
- 3) **void** *funct*(**char**);
- 4) **char*** *funct*(**const char ***);
- 5) **char** *funct*(**const char&**);
- 6) **template<class T> void** *funct*(**const T&**);
- 7) **template<class T> void** *funct*(**T***);

DOPASOWYWANIE FUNKCJI

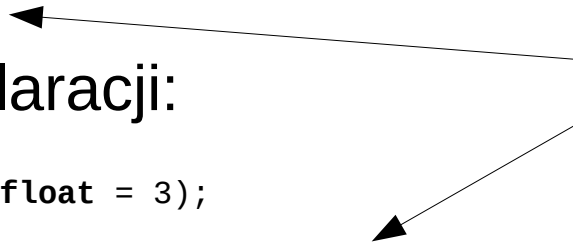
Mamy wywołanie:

`funct('A');`

Oraz szereg deklaracji:

- 1) `void funct(int);`
- 2) `float funct(float, float = 3);`
- 3) `void funct(char);`
- 4) `char* funct(const char *);`
- 5) `char funct(const char&);`
- 6) `template<class T> void funct(const T&);`
- 7) `template<class T> void funct(T*);`

Która funkcja/szablon zostanie dopasowana?



Dopasowanie do najlepszych:

1. Dopasowanie nazwy do funkcji i szablonów
2. Dopasowanie ilości parametrów
3. Dokładne dopasowania, w tym zwykłe funkcje przed szablonami
4. Konwersje będące promocjami typów
5. Konwersje standardowe
6. Konwersje użytkownika

SZABLONY KLAS

- Definicja szablonu klasy

```
template<class Type>           //lub template<typename Type>
class Stack {
    Type items;
public:
    Stack();
    void push(Type t) { items[...] = t; }
    Type pop();
};
```

```
template<class Type> Stack<Type>::Stack() {...}
template<class Type> Type Stack<Type>::pop() {...}
```

INSTANCJA SZABLONU KLASY

- Instancja szablonu klasy

- podczas deklaracji obiektu:

```
Stack<int> numbers;           //stos przechowujacy inty  
Stack<string> napisy;        //stos przechowujacy stringi  
Stack< Stack<float> > stosy; //stos przechowujacy stosy z float
```

- konkretyzacja jawna

```
template class Stack<float>;
```

- konkretyzacja niejawna

```
class Stack<float> stos;
```

- Nie ma niejawnego dopasowania typu szablonu
- Każda instancja szablonu jest odrębną (niekompatybilną) klasą

ARGUMENTY NIEBĘDĄCE TYPAMI

- Klasa tablica

```
template<class T, int N> class Array {
```

```
    T items[N];
```

```
public:
```

```
    Array() {}
```

```
    T& operator[](int i);
```

```
};
```

```
template<class T, int N> T& Array<T, N>::operator[](int i) {
```

```
    if(i<0 || i>=n) throw out_of_range();
```

```
    return items[i];
```

```
}
```

```
...
```

```
Array<Card, 12> cards;
```

parametr
niebędący typem
szablonu



DOMYŚLNE PARAMETRY SZABLONU

- Określenie wartości domyślnej parametru szablonu

```
template<class T1, class T2 = int> class Pair {...};
```

```
Pair<double, double> m1;    //T1 jest double, T2 jest double
```

```
Pair<double> m1;             //T1 jest double, T2 jest int
```

```
template<class T, int N=10> class Array {...};
```

- Wartości domyślne można stosować tylko w przypadku szablonów klas.
- Wartości domyślne parametrów nie będących typami można stosować również do szablonów funkcji

SPECJALIZACJA CZĘŚCIOWA

- Podobnie jak dla funkcji, można stworzyć specjalizację szablonu klasy
- Możliwe jest również utworzenie specjalizacji częściowej

//szablon ogolny

```
template<class T1, class T2> class Pair {...};
```

//specjalizacja pelna

```
template<> class Pair<float, int> {...};
```

//specjalizacja na drugim parametrze

```
template<class T1> class Pair<T1, int> {...};
```

INNE

- Szablon klasy z parametrem będącym szablonem

```
template< template<typename T> class T1,  
          class T2,  
          int z>  
class Trophy { T1<int> elem; ...};  
  
template< class T> class Stack {...};  
  
Trophy<Stack, float, 10> tr;
```

- Instrukcja **typedef**:

```
typedef Stack<int> StackInt;
```

SZABLONY A WIELOPLIKOWOŚĆ

- Szablon funkcji/klasz musi być znany w całości podczas generacji instancji
- Co znaczy, że umieszczamy go w pliku nagłówkowym .h
- Można część definicji umieścić w pliku .cpp ale muszą być też jawne konkretyzacje tego szablonu
- Pociąga to za sobą ograniczenie stosowania tylko tych konkretyzacji