

Szablony funkcji i klas

Wykład 9

Szablony

- W językach programowanie takich jak C++ gdzie istnieje ścisła kontrola typów często występuje potrzeba wielokrotnego zdefiniowania takiej samej funkcji, ale pracującej na różnych typach danych
- Rozwiązaniem jest wykorzystanie makrodefinicji znanych z języka C
 - Mechaniczne podstawianie, które może stwarzać problemy
 - **Nie zalecane!!!**
- Dlatego w języku C++ wprowadzono szablony, które rozwiązują większość problemów
 - Mają też swoje wady (o tym później)

Makrodefinicje

- Do generowania „funkcji” wykonujących to samo zadanie na różnych typach danych w języku C można było wykorzystywać makrodefinicje
 - `#define max(a, b) ((a) < (b)) ? (b) : (a)`
- Jednak użycie makrodefinicji może spowodować duże problemy
 - W szczególności kiedy argumentami nie są liczby ani zmienne, ale wyrażenia
 - `max(a++, b++) ;`
 - Ponieważ rozwinięcie `max` daje rezultat
 - `((a++) < (b++)) ? (b++) : (a++)`

Szablony

- Szablony reprezentują funkcje, a nawet typy danych tworzone przez programistów (klasy)
 - Ale same nie są funkcjami ani klasami
- Nie zostają one zaimplementowane dla określonego typu danych, ponieważ zostanie on zdefiniowany później
 - W większości sytuacji parametryzowane są typem, ale nie jest to reguła
- Aby użyć szablonu kompilator lub programista musi określić dla jakiego typu ma on zostać użyty

Szablony klas wykorzystanie - tablica

`std::array`

Defined in header <array>

```
template<
    class T,
    std::size_t N    (since C++11)
> struct array;
```

<https://en.cppreference.com/w/cpp/container/array>

- Prosta tablica statyczna
 - ❑ Alokacja na stosie
 - ❑ Elementy określonego typu (możliwe konwersje)
 - ❑ Znany rozmiar czasie kompilacji
 - ❑ Zamiennik zwykłej tablicy
- Przykład cpp9.0a

Szablony klas wykorzystanie - wektor

std::vector

Defined in header <vector>

```
template<
    class T,
    class Allocator = std::allocator<T>                (1)
> class vector;

namespace pmr {
    template <class T>
        using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;    (2) (since C++17)
}
```

1) std::vector is a sequence container that encapsulates dynamic size arrays.

2) std::pmr::vector is an alias template that uses a [polymorphic allocator](#)

<https://en.cppreference.com/w/cpp/container/vector>

■ Dynamiczna tablica

- ❑ Alokacja pamięci na stacku
- ❑ Elementy określonego typu (możliwe konwersje)
- ❑ Ciągły obszar pamięci
- ❑ Rozmiar rośnie w miarę potrzeb - UWAGA

■ Przykład cpp9.0b

Szablony klas wykorzystanie - string

std::basic_string

Defined in header <string>

```
template<
    class CharT,
    class Traits = std::char_traits<CharT>,           (1)
    class Allocator = std::allocator<CharT>
> class basic_string;

namespace pmr {
    template <class CharT, class Traits = std::char_traits<CharT>>
    using basic_string = std::basic_string< CharT, Traits,
                                           std::polymorphic_allocator<CharT>> (2) (since C++17)
}
```

https://en.cppreference.com/w/cpp/string/basic_string

■ Dynamiczna tablica znaków

- `std::string` to jest `std::basic_string<char>`
- Alokacja pamięci na stercie
- Elementy określonego typu `char`
- Ciągły obszar pamięci

■ Przykład cpp9.0c

Szablony funkcji wykorzystanie - find

std::find

Defined in header <algorithm>

```
template< class InputIt, class T >
```

```
constexpr InputIt find( InputIt first, InputIt last, const T& value );
```

<https://en.cppreference.com/w/cpp/algorithm/find>

- Algorytm do wyszukiwania
 - Obsługuje dowolne typy
 - Znajduje pierwszy element zgodny ze wzorcem
 - Przeszukuje podany zakres - nie musi być cały kontener
 - Są też inne wersje np. find_if
- Przykład cpp9.0d

Szablony funkcji wykorzystanie - sort

`std::sort`

```
template< class RandomIt >  
constexpr void sort( RandomIt first, RandomIt last );  
template< class RandomIt, class Compare >  
void sort( RandomIt first, RandomIt last, Compare comp );
```

<https://en.cppreference.com/w/cpp/algorithm/sort>

- Algorytm do sortowania
 - Obsługuje dowolne typy
 - Domyślnie sortuje używając operatora <
 - W wersji drugiej potrafi użyć obiektu funkcyjnego służącego jako narzędzie do porównywania
- Przykład cpp9.0e

Szablony funkcji wykorzystanie - for_each

std::for_each

```
template< class InputIt, class UnaryFunction >  
UnaryFunction for_each( InputIt first, InputIt last, UnaryFunction f );
```

https://en.cppreference.com/w/cpp/algorithm/for_each

- Algorytm do wykonywania operacji na elementach
 - Stosowany zamiennie z zakresową pętlą for
 - Działa w trybie tylko do odczytu albo modyfikowania
 - Przyjmuje jako argumenty zakres oraz funkcję/funktor
- Przykład cpp9.0f

Szablony i STL - inne (podstawowe) ciekawostki

- <https://en.cppreference.com/w/cpp/utility/pair>
- <https://en.cppreference.com/w/cpp/utility/tuple>
- https://en.cppreference.com/w/cpp/memory/unique_ptr
- https://en.cppreference.com/w/cpp/memory/shared_ptr
- <https://en.cppreference.com/w/cpp/utility/functional/less>
- <https://en.cppreference.com/w/cpp/utility/functional/bind>
- <https://en.cppreference.com/w/cpp/utility/functional/ref>
- https://en.cppreference.com/w/cpp/utility/initializer_list

STL ogólne linki

- <https://en.cppreference.com/w/cpp/language/templates>
- https://en.cppreference.com/w/cpp/standard_library
- <https://en.cppreference.com/w/cpp/container>
- <https://en.cppreference.com/w/cpp/algorithm>
- <https://en.cppreference.com/w/cpp/iterator>
- <https://en.cppreference.com/w/cpp/numeric>
- <https://en.cppreference.com/w/cpp/memory>

Definiowanie szablonu funkcji

- Do definiowania szablonów używane jest słowo kluczowe **template**
 - `template<class Typ> Typ max(Typ a, Typ b)`
`{ return (a < b) ? b : a; }`
 - Do określania typu w starszej notacji służyło słowo **class**
 - `template<typename Typ> Typ min(Typ a, Typ b)`
`{ return (a < b) ? a : b; }`
 - Nowa specyfikacja wprowadza słowo kluczowe **typename** do określania typu
- W tym przykładzie parametrem szablonu jest **Typ**, który może zostać zamieniony na dowolny typ rzeczywisty (wbudowany lub zdefiniowany przez programistę)
 - Najczęściej używa się do nazwania typu symbolu **T**

Definiowanie szablonu funkcji...

- Szablon musi zostać zdefiniowany w takim miejscu, żeby znalazł się w zakresie globalnym
 - Innymi słowy musi być zdefiniowany poza innymi funkcjami lub klasami, a najlepiej w jakiejś przestrzeni nazw
 - Wszystkie szablony zdefiniowane w standardzie języka znajdują się w przestrzeni nazw `std`
- Zdefiniowanie szablonu zaoszczędza nam programistom pisanie, ale wcale nie zmniejsza kodu wygenerowanego przez kompilator
 - Po prostu kompilator generuje funkcje z szablonu dla każdego typu dla którego jest ona potrzebna
- Szablony funkcji jest mechanizmem umożliwiającym definiowanie funkcji identycznych w działaniu, ale różniących się tylko typem argumentów
- Przykład `cpp_9.1`

Wywołanie funkcji szablonowej

- Zdefiniowanie szablonu nie powoduje powstania żadnej funkcji szablonowej
 - Funkcje szablonowe zostaną zdefiniowane w momencie kiedy będą potrzebne
 - W miejscu w programie, gdzie wywołujemy funkcję
 - Lub gdzie pytamy o adres funkcji
- Skąd wiadomo jaka funkcja szablonowa jest potrzebna
 - Po prostu kompilator patrzy na typ(-y) argumentów wywołania i produkuje żadaną funkcję
 - Typ zwracany jak zwykle nie ma znaczenia
 - Programista deklaruje, że chce użyć szablonu do stworzenia funkcji odpowiedniego typu
- Przykład `cpp_9.2`

Funkcja szablonowa dla dowolnego typu

- Jeśli mam szablon to czy można zbudować na jego podstawie funkcje dla każdego typu danych?
 - To zależy, ale w ogólności nie
 - Nie można wygenerować funkcji szablonowej dla typu, dla którego ta funkcja byłaby błędna
- Programista jest odpowiedzialny za sens ciała szablonu w stosunku do konkretnego typu danych
 - Np. wywołanie operatora `<` dla typu zdefiniowanego przez użytkownika wymaga jego wcześniejszej implementacji
 - Jawne wywoływanie operatorów sprawia problemy dla wbudowanych typów danych
 - Odwołanie do składnika klasy znacząco uszczupla możliwości wykorzystywania szablonu
 - ...
- Przykład `cpp_9.3`

Szablony dla typów wbudowanych

- W celu bardziej uniwersalnego podejścia do pisania szablonów wprowadzono modyfikacje w stosunku do wbudowanych typów danych
 - Dopuszczenie wywołania konstruktorów
 - `int obiekt(value);`
 - Dopuszczenie inicjalizacji w postaci
 - `int obiekt = int(value);`
 - Dopuszczono bezpośrednio wywołane destruktory
 - `obiekt.int::~~int();`
- Gdyby te oczywiste zapisy nie były tolerowane przez kompilator to, na ogół trzeba by było pisać osobne wersje szablonów dla typów wbudowanych
- Przykład `cpp_9.4`

Wiele parametrów szablonu

- Szablon oczywiście może mieć więcej niż jeden parametr
- Każdy unikatowy typ użyty w wywołaniu funkcji, musi się znaleźć na liście parametrów szablonu
 - `template <typename T1, typename T2>`
`fun(T1 a, T2 b) {...}`
 - `template <typename T1, typename T2>`
`fun(T1 a, T1 b, T2 c, T2 d) {...}`
- Szablon funkcji może przyjmować także zwykłe znane od razu typy danych jako argumenty
 - `template <typename T1, typename T2>`
`fun(T1 a, T2 b, int c, float d) {...}`

Szczególne przypadki szablonów

- Jeden szablon może być szczególnym przypadkiem drugiego
 - `template<typename Typ> Typ max(Typ a, Typ b)`
`{ return (a < b) ? b : a; }`
 - `template<typename T1, typename T2> T1 max(T1 a, T2 b)`
`{ return (a < b) ? b : a; }`
- Oba szablony mogą istnieć niezależnie od siebie
 - Może jednak pojawić się konflikt, ponieważ w wywołaniu `max(1, 2)`, kompilator może wykorzystać obie wersje do wyprodukowanie funkcji
 - Nie ma przeciwwskazań, żeby `T1` było tym samym co `T2`
 - Na ogół jednak kompilator przy wywołaniu np. `max(1, 2)`, skorzysta z szablonu z jednym typem, nie generując błędu
- Tworząc szablony o tej samej nazwie należy uważać czy nie są one szczególnymi przypadkami siebie nawzajem
- Przykład `cpp_9.5`

Typy pochodne

- W ciele szablonu możemy posługiwać się zarówno jego argumentem do tworzenia zmiennych automatycznych (np. `Typ A;`) jak i typów pochodnych takich jak wskaźniki, referencje czy tablice
- Definiowanie typów pochodnych odbywa się w taki sam sposób jak w normalnych funkcjach
 - `T* a; //wskaźnik do zmiennej typu T`
 - `T& a = b; //referencja do zmiennej typu T`
 - `T a[10]; //tablica elementów typu T`
- Przykład - szablon `swap(a, b);`

Szablony funkcji, a przydomki `inline`, `static`, `extern`

- Szablony funkcji można również wykorzystać do produkowanie funkcji typu `inline`, `static` i `extern`
- Należy pamiętać, że to funkcja ma być np. `inline`, a nie sam szablon
 - `template<typename Typ> inline Typ max(Typ a, Typ b); //OK`
 - `inline template<typename Typ> Typ max(Typ a, Typ b); //źle`
 - Podobnie jest z przydomkami `static` i `extern`
- Co znaczy, że funkcja (zwykła) jest `static`?

Funkcje „specjalizowane”

- Czasami funkcja wygenerowana przez szablon może być nieodpowiednia
 - np. `max(char*, char*)`
- Istnieje wtedy możliwość zdefiniowanie normalnej funkcji, która będzie pracować w odpowiedni sposób na takich danych
- W takiej sytuacji kompilator wykorzysta tą specjalizowaną wersję funkcji, dopiero jeżeli takowej nie znajdzie to skorzysta z szablonu
 - Dopasowanie musi być dokładne tzn. `char* != const char*`
- Przykład `cpp_9.6`

Dopasowywanie argumentów

- Dopasowanie dokładne
 - Kompilator szuka funkcji o odpowiedniej nazwie z dokładnie takimi samymi argumentami
- Poszukiwanie szablonu, z którego można wyprodukować funkcję o argumentach takiego samego typu jak wywołanie
 - Dopasowanie wszystkich argumentów musi być idealne (bez konwersji standardowych)
- Kontynuacja poszukiwania wśród funkcji (nie szablonów)
 - Konwersje standardowe
 - Konwersje zdefiniowane przez programistę

Jeden szablon w wielu plikach

- Ponieważ szablony na ogół umieszczamy w plikach nagłówkowych to może się zdarzyć, że powstaną w osobnych modułach programu takie same funkcje
 - Taka sytuacja nastąpi, jeżeli w jednym pliku powstanie funkcja np. `int max(int, int)` w wyniku jej wywołania i w innym też
 - Wtedy aby program został poprawnie skonsolidowany („zlinkowany”) to linker musi być „inteligentny” tzn. usunąć nadmiarowe definicje takich samych funkcji
- Przykład `cpp_9.7` i `cpp_9.8`

Częściowa „specjalizacja”

- Faktycznie w przypadku funkcji nie jest to częściowa specjalizacja
- Cały czas mamy do czynienia z przetładowaniem nazw
 - Ciągłe te same reguły
 - Jakie?
- `template<class T> void f(T a);`
- `template<class T> void f(T* a);`
- `template<class T> void f(const T* a);`
- Przykład `cpp_9.8a`

C++11 i C++14 auto i decltype, a zwracany typ w szablonach funkcji

- W przypadku szablonów rozwiązuje problem wyznaczenia typu zwracanego
 - Typ deklarowany jak `auto`
 - Z informacją dla kompilatora w jaki sposób typ zwracany ma zostać wyznaczony
 - Wymagane w przypadku C++11, opcjonalne w przypadku C++14 (działa automatyczna dedukcja typów)
 - ```
template<typename T, typename U>
auto add(T t, U u) -> decltype(t + u)
{ return t + u; }
```
- Przykład `cpp_9.8b`

# Szablony funkcji uwagi

- Szablon funkcji nie powinien pracować na zmiennych globalnych
- Dwa (lub więcej) szablony o takiej samej nazwie mogą istnieć - jest to po prostu przeładowanie nazw
  - Nie powinny generować funkcji o takich samych argumentach
- Możemy tworzyć funkcję z szablonu i od razu deklarować jakiego typu ma ona być (kompilator nie będzie wtedy decydował na podstawie parametrów wywołania)
  - `a = max<int>(a, b);`
  - `swap<double>(f, g);`