Szablony funkcji i klas

Wykład 9

Szablony

- W językach programowanie takich jak C++ gdzie istnieje ścisła kontrola typów często występuję 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
> struct array;
```
(since C++11)

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>
> class vector;                                                    (1)

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 stercie
  - 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,     (2)  (since C++17)
                                            std::polymorphic_allocator<CharT>>
}
```

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 zamienne 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 jakieś przestrzeni nazw
  - Wszystkie szablony zdefiniowane w standardzie języka znajdują się w przestrzeni nazw `std`
- Zdefiniowanie szablonu zaoszczędza nam programistom pisania, ale wcale nie zmniejsza kodu wygenerowanego przez kompilator
  - Po prostu kompilator generuje funkcje z szablonu dla każdego typu dla jakiego 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 powoduję 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 żądaną 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 bezpośrednie wywołane destruktorów
    - `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 zwykle 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 definicję 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 przeładowaniem nazw
  - Ciągle 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);`

Szablony klas

- Podobnie jak szablony funkcji w języku C++ mamy możliwość definiowania szablonów klas
- Szablon klasy to nic innego jak narzędzie do automatycznego pisania różnych wersji bardzo podobnych klas
  - Szablon klasy to nie sama klasa, ale przepis jak taką klasę stworzyć
- Definiowanie szablonu klasy
  - `template<typename T> class Box {/*…*/};`

Definiowanie szablonu klasy

- Nazwa szablonu klasy musi być unikatowa
  - Nie może być taka jak nazwa innej klasy, szablonu, funkcji typu wyliczeniowego …
  - Nie istnieje przeładowanie klas
- Szablony mogą być definiowane tylko w zakresie globalnym (oczywiście mogą się znajdować w przestrzeniach nazw)
  - Nie można szablonów klas zagnieżdżać
- Klasy szablonowe powstałe z jednego szablonu nie mają nic wspólnego ze sobą (np. dziedziczenie czy przyjaźń)

Parametry szablonu klasy

- W szablonach funkcji kompilator mógł na podstawie argumentów wywołania określić jaką wersję funkcji wygenerować
- Parametry szablonu klasy muszą być podane przy tworzeniu obiektów danego typu klasy
  - Typ parametru(-ów) szablonu klasy jest jakby częścią jego nazwy, ponieważ klasy nie mogą być przeładowane
  - Parametry szablonu umieszcza się w nawiasach <>
    - Np. `box<int> a;`
- Przykład cpp_9.9

Parametry szablonu klasy…

- Parametrów szablonu klas może być więcej niż jeden
  - Parametry umieszczamy na liście (podobnie jak dla funkcji)
    - Np. `template<typename T1, typename T2> class Box{…};`
- Parametrami szablonu klas mogą być
  - Typ
  - Stałe wyrażenia
    - Stała dosłowna typu całkowitego, adresy (obiektu globalnego, funkcji globalnej, składnika statycznego klasy)

Parametry szablonu klasy…

- Parametrem aktualnym szablonu klas nie może być
  - Stała dosłowna będąca łańcuchem
  - Adres elementu tablicy
  - Adres zwykłego niestatycznego składnika klasy
- Jeżeli dwa wyrażenia będące parametrem aktualnym szablonu, mają taką samą wartość to, uznawane są za identyczne
- Przykład cpp_9.10

Funkcje składowe szablonu klas

- Definiowanie funkcji w ciele szablonu
- Definiowanie na zewnątrz szablonu klasy
  - Taką funkcję składową definiujemy w podobny sposób do szablonu funkcji
  - `template<typename T> typ_zwaracany nazwa_sz_klasy<T>::nazwa_funkcji(args) {…}`
  - `<T>` - używane jest do rozróżnienia między rożnymi funkcjami składowymi dla różnych wersji szablonu klasy
- Przykład cpp_9.11

Kiedy produkowane są klasy z szablonu

- Oczywiście jeśli definiujemy obiekt klasy
  - `box<int> a;`
- Również przy definiowaniu wskaźnika mogącego pokazywać na obiekt klasy szablonowej
  - `box<int> *a;`
  - Jest to potrzebne chociażby w sytuacji kiedy wielkość obiektu ma znaczenie
- Podobnie przy deklaracji funkcji, która jako argument przyjmuje klasę szablonową
  - `void fun(int a, box<int> b);`
- Jeżeli klasa szablonowa używana jest jako klasa podstawowa
  - `class boxA: public box<int> {…};`

Szablon funkcji z argumentem będącym szablonem klasy

- Dlaczego takiej funkcji nie zrobić w postaci funkcji składowej?
  - Nie wszystkie funkcje mogą być funkcjami składowym np. funkcje operatorowe takie jak <<
- W takiej sytuacji definiujemy sobie szablon funkcji, który jako argument przyjmuje obiekt klasy, ale powstały na podstawie typ szablonu funkcji
  - ```
    template<typename T>
    ostream& operator<<(ostream &o, klasa<T>& K);
    ```
- Przykład cpp_9.12
    ```

Obiekt klasy szablonowej będący składnikiem innej klasy szablonowej

- Podobna sytuacja do poprzedniej, parametr tym razem szablonu klasy zostanie wykorzystany do stworzenia odpowiedniego składnika klasy
  - ```
    template<typename T> class K1
    {...};

    template<typename T> class K2
    {
            K1<int> a;
            K1<T> b;
    };
    ```
- Przykład cpp_9.13
    ```

Zagnieżdżanie definicji w szablonie klas

- Szablon klas może być definiowany tylko w obszarze globalnym
  - Nie da się stworzyć szablonu klasy wewnątrz innego szablonu klasy, a nawet wewnątrz innej klasy
  - Nic nie stoi jednak na przeszkodzie, aby zdefiniować zwykłą klasę wewnątrz szablonu klasy
    - Składowe i metody zagnieżdżonej klasy mogą być definiowane na podstawie parametrów szablonu
- Przykład cpp_9.14

Składniki statyczne w szablonie klas

- Każdy składnik statyczny danego typ klasy jest wspólny dla wszystkich obiektów tej klasy
- Poszczególne klasy powstające z tego samego szablonu nie łączy nic, czyli każdy rodzaj klasy ma swój własny zestaw składników statycznych
  - Obiekt statyczny może być określonego typu
    - `static int a;`
  - Może też być typu zależnego od parametru szablonu
    - `static K<T>* ptr;`
- Składniki statyczne definiujemy w zakresie globalnym (lub lepiej w jakiejś przestrzeni nazw)
  - `template<typename T> int K<typ>::a;`
  - `template<typename T> K<T>* K<typ>::a;`
- Przykład cpp_9.15

Typedef

- Instrukcja `typedef` umożliwia tworzenie synonimów dla znanych typów danych
- `template<typename T, unsigned short a, double (*ptr)(double, double) class K{...};`
- Deklaracja obiekty takiej klasy może mieć postać
  - ```
    K<std::string, 10, fun> a;
    typedef K<std::string, 10, fun> Kstr10Fun;
    Kstr10Fun b;
    ```
- Inny przykład
  - ```
    typedef box<box<std::string> > bbstr;
    bbstr a;
    ```
- `using` - standard C++11
  - Działa dobrze z szablonami, definiuje się tak jak szablon

Specjalizacja, a szablony klas

- Podobnie jak przy szablonach funkcji możemy tworzyć specjalizowane wersje klasy szablonowej
  - `template<typename T> class K {...};`
  - `template<> class K<char*> {...};`
  - `template<> class K<std::string> {...};`
- Przy nazwie klasy specjalizowanej powinna być umieszczona instrukcja `template<>`
- Kompilator widząc w nawiasach parametr aktualny nie przystępuje do produkcji klasy, ale korzysta z tego co programista zaimplementował
- Możliwa jest również częściowa specjalizacja
  - `template<typename T> class K<T &> {...};`
- Przykład cpp_9.16, cpp_9.16a

Specjalizacja dla typów wskaźnikowych

- Można napisać specjalizację częściową dla typu **`T*`**
  - Imputujemy praktycznie w standardowy sposób tylko tak aby poprawnie działało ze wskaźnikami
    - `template<typename T> class K {...};`
    - `template<typename T> class K<T *> {...};`
  - Ewentualnie definiujemy pełną specjalizację dla typu **`void*`** i wykorzystujemy ją potem w przypadku **`T*`**
    - `template<> class K<void *> {...};`
  - Po co takie zabiegi?
- Przykład cpp_9.16b

Specjalizacja, a szablony klas

- Definicja specjalizowanej wersji klasy szablonowej może wystąpić dopiero po samej definicji szablonu, dla którego jest dedykowana
  - Kompilator sprawdza czy zdefiniowana przez nas wersja specjalizowana klasy faktycznie mogłaby powstać z szablonu klas
- Definicja specjalizowanej wersji klasy szablonowej nie musi występować bezpośrednio po szablonie klasy, do którego przynależy
- Specjalizowana wersja klasy nie musi mieć takich samych składników jak szablon

Specjalizowana funkcja składowa

- Nie zawsze jest sens od razu definiować specjalną klasę szablonową
- Czasami wystarczy tylko zdefiniować specjalną funkcję składową, która w odpowiedni sposób obsłuży jakiś „nietypowy" typ
- Definiowanie specjalizowanej funkcji składowej zasadniczo niczym się nie różni od definiowania specjalizowanej „zwykłej" funkcji szablonowej
- Przykład cpp_9.17

Przyjaźń i szablony klas

- Szablony klas podobnie jak zwykłe klasy mogą posiadać przyjaciół
- W przypadku szablonów klas możemy mieć do czynienia z następującymi przypadkami
  - Jeden przyjaciel dla wszystkich klas powstałych z danego szablonu
  - Każda klasa wyprodukowana z szablonu ma swojego przyjaciela
- Oczywiście przyjaciółmi mogą być funkcje i inne klasy

Przyjaźń i szablony klas

- Jednego wspólnego przyjaciela dla wszystkich klas powstających z szablonu, określa się w sposób niczym się nie różniący od deklaracji przyjaźni w „zwykłych" klasach
- Zadeklarowanie przyjaźni różnej dla każdej wersji klasy polega na uzależnieniu tej deklaracji od parametry szablonu
  - `friend void fun(K<T> obj);`
  - `friend class Klasa<T>;`

Każda klasa szablonowa posiada swojego przyjaciela (funkcję)

- Mogą wystąpić problemy jeżeli szablon klasy jest uzależniony nie tylko od typu, ale także np. od stałej, a chcemy mieć funkcję szablonową inną dla każdej wersji szablonu klasy
  - Wtedy jedynym rozwiązaniem jest zdefiniowanie funkcji szablonowej w zakresie leksykalnym klasy, czyli całą funkcję należy umieścić w ciele szablonu klasy
- Przykład cpp_9.18

Domyślnie typy w szablonach

- Parametry mogą też mieć wartości domyślne
  - Bardzo podobnie jak domyślne wartości przy argumentach wywołania funkcji
- Parametry domyślne mogą być
  - Typami
  - Wartościami
  - Szablonami
- Parametry domyślne mogą być podawane przy deklaracji
  - I przy definicji jeśli jest ona napisana od razu
  - Nie mogą się znaleźć przy definicji jeśli jest ona odroczona
- Przykład
  - `template<typename T1, typename T2 = int> class A;`
  - `template<class T, class Allocator = std::allocator<T>> class vector;`

Wytyczne dla szablonów klas (CTAD)

- Wersja standardu C++ >= c++17
- Przydatne do klas, gdzie na podstawie inicjalizacji da się określić typu szablonu
- Stosowane tylko gdy nie zostanie podany żaden typ parametru szablonu
  - Podanie części jest błędem
- Możemy dodawać własne wytyczne co do określania typów
- Dla typów opakowujących domyślnie wytyczne nie opakowują dodatkowo

```
std::tuple t1{1};  //std::tuple<int>
std::tuple t2{t1}; //std::tuple<int>
// a nie std::tuple<std::tuple<int>>
std::vector v1{1, 2};   // std::vector<int>
std::vector v2{v1};     // std::vector<int>
// a nie std::vector<std::vector<int>>
std::vector v3{v1, v2}; // std::vector<std::vector<int>>
```

- Class template argument deduction

Dziedziczenie i szablony klas

- Skoro zwykłe klasy mogą być dziedziczone to klasy powstałe z szablonów również
- Dostępne przypadki
  - Zwykła klasa odziedzicza klasę szablonową
  - Szablon klas odziedzicza zwykłą klasę (może też być klasa szablonowa)
  - Szablon klas odziedzicza inny szablon klas
  - Specjalizowana klasa szablonowa odziedzicza zwykłą klasę (może również być to klasa szablonowa)

Zwykła klasa odziedzicza klasę szablonową

- Klasa szablonowa to po prostu zwykła klasa, która już powstała z szablonu
- Czyli tak naprawdę jest to przypadek normalnego dziedziczenia
- ```
  template <typename T> class Box
  {public: T box;};
  class BoxFloatOpis : public
  Box<float>
  {...};
  ```
  ```

Szablon klas ze zwykłą klasą podstawową

- Takie rozwiązanie może być przydatne w sytuacji kiedy szablon klas ma zawierać skomplikowaną funkcję, która działa niezależnie od typu(-ów) parametru szablonu
  - Wtedy zdefiniowanie takiej funkcji w klasie podstawowej powoduje, że przy kompilacji funkcja ta znajdzie się w pamięci tylko raz
  - W przypadku umieszczanie definicji tej funkcji w szablonie zostanie ona powielona wiele razy (tyle ile będzie różnych klas powstałych z szablonu)
    - Każda klasa powstała z szablonu ma swój zestaw funkcji składowych, nawet jeżeli są one takie same
- Przykład cpp_9.19

Szablon klas odziedziczony przez inny szablon klas

- Szablon pochodny może mieć taki sam lub nawet inny zestaw parametrów w stosunku do szablonu podstawowego
- ```
  template <typename T> Box {...};
  template <typename T> BoxOpis : public
  Box<T> {...};
  ```
- ```
  template <typename T1, typename T2>
  BetterBox : public Box<T2> {...};
  ```
- Przykład cpp_9.20

Specjalizowana klasa szablonowa odziedzicza zwykłą klasę

- Sytuacja to odnosi się do dziedziczenie zwykłej klasy jak i klasy szablonowej
- Przypadek ten niewiele różni się od zwykłego dziedziczenia
- Uwaga
  - Specjalizowana klasa szablonowa może dziedziczyć inną klasę nawet jeśli sam szablon nie dziedziczy niczego
  - Jedynie co nas obowiązuje to nazwa klasy

Szablonowe funkcje składowe w klasach

- Szablonowe metody w przypadku klas nie mogą być deklarowane jako **`virtual`**
  - Wynika to z założenia że implementacja funkcji wirtualnych powinna być możliwie prosta
  - Dlatego vtable – wpisy na temat funkcji wirtualnych zakładają jej stały rozmiar
  - Natomiast liczba instancji szablonu – czyli funkcji wirtualnych w tym przypadku nie byłaby znana, aż do linkowania całego programu
    - Powoduje to za duży koszt z punktu widzenia czasu kompilacji i złożoności wygenerowanych programów
- Jednak nic nie stoi na przeszkodzie aby zwykła klasa z funkcjami wirtualnymi stała się jako całość szablonem
- Przykład cpp_9.21

Szablonowe parametry szablonów

- Mechanizm bardzo przydatny w sytuacji kiedy parametry szablonów wykazuję zależności między sobą
  - Pierwszy parametr jest np. typem przechowywanych (używanych) obiektów
  - Drugi parametr jest „pochodnym" w stosunku do pierwszego
    - Ale niekoniecznie w sensie dziedziczenia
    - Np. kontener do przechowywania elementów lub allokator do zarządzania pamięcią
    - `template <typename T, template <typename ElemType, typename AllocType> class Cont = std::deque> class stack`
  - https://en.cppreference.com/w/cpp/language/template_parameters
- Przykład cpp_9.22

SFINAE - Substitution Failure Is Not An Error

- Sytuacja dotyczy szablonów funkcji dla których analizowany kod po podstawieniu pasujących argumentów staje się błędny
  - ```
    template<typename Iter>
    typename Iter::value_type mean(Iter b, Iter e);
    ```
    - Dla iteratorów nie problem
    - Ale np. dla `int` problem bo nie ma `int::value_type`
    - Sama w sobie nieudana próba podstawienie nie jest błędem
  - W szczególności ważne gdyż może istnieć
  - `template <typename T> T mean(T* ,T*);`
    - W tym momencie implementacja dla `int` staje się poprawna
  - https://en.cppreference.com/w/cpp/language/sfinae
    ```

Uwagi

- Niemożliwe sytuacje
  - Zwykła klasa chce odziedziczyć szablon
  - Specjalizowana klasa szablonowa chce odziedziczyć szablon
- Inne aspekty
  - Dziedziczenie szablonów może odbywać się tylko i wyłącznie do innych szablonów
  - Klasa może odziedziczyć tylko inna klasę
  - Uwaga przy referencji jako parametrze aktualnym szablonu
  - Przy konsolidacji takie same problemy jak przy szablonach funkcji
- Static polymorphism
  - Przykład cpp_9.23
- Type traits
  - Przykład cpp_9.24
  - Dokumentacja do standardu >=C++11
- Metaprogramowanie
  - Przykład cpp_9.25

Rady (za B. Stroustrup, Język C++)

- Używaj szablonów do wyrażania algorytmów, które można stosować do argumentów rożnego typu
- Używaj szablonów do wyrażania kontenerów
- Przed przystąpieniem do definiowania szablonu zaprojektuj i przetestuj wersję nieszablonową. Dopiero później uogólnij otrzymaną konstrukcję przy użyciu parametrów
- Szablony są bezpieczne ze względu na typy, ale kontrola typów w ich przypadku odbywa się za późno
- Projektując szablon, dokładnie przeanalizuj koncepcje (wymagania) dotyczące jego argumentów

Rady (za B. Stroustrup, Język C++)

- Używaj szablonów funkcji w celu dedukcji typów argumentów szablonu klasy
- Przeciążaj szablony funkcji w celu uzyskania jednakowej semantyki dla rożnych typów argumentów
- Korzystaj z zasady nieudanej próby podstawienia argumentu w celu dostarczenia odpowiedniego zbioru funkcji w programie
- Szablony nie są kompilowane oddzielnie. Definicje szablonów dołączaj w każdej jednostce translacji, w której są potrzebne
- Jako łącznika z kodem, w którym nie można używać szablonów, używaj zwykłych funkcji
- Duże szablony i szablony mające skomplikowane powiązania kontekstowe kompiluj rozdzielnie
- Używaj aliasow szablonów w celu uproszczenia notacji i ukrycia szczegółów implementacyjnych (C++11)

Dodatek
std::any – uproszczony przykład

- Reprezentacja dowolnego typu
  - Np. w wektorze
- Przykład cpp_9.26