

Szablony funkcji i klas

Wykład 9

Szablony

- W języku programowania 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

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 funkcję, 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

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 jakimś 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 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 definicje takich samych funkcji
- Przykład `cpp_9.7` i `cpp_9.8`

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 szablone 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 przetadowane
 - 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 dostówna będąca łańcuchem
 - Adres elementu tablicy
 - Adres zwykłego niestatycznego składnika klasy
 - Stała dostówna, gdy szablon oczekuje obiektu
- 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 różnymi 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 szablonej
 - `box<int> *a;`
 - Jest to potrzebne chociażby w sytuacji kiedy wielkość obiektu ma znaczenie
- Podobnie przy deklaracji funkcji, która jako argument przyjmuje klasę szablونową
 - `void fun(int a, box<int> b);`
- Jeżeli klasa szablونowa 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 typu 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 obiektu 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;`

Specjalizacja, a szablony klas

- Podobnie jak przy szablonach funkcji możemy tworzyć specjalizowane wersje klasy szablonej
 - `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`

Specjalizacja, a szablony klas

- Definicja specjalizowanej wersji klasy szablonej 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 szablonej 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`

Dziedziczenie i szablony klas

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

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



# 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 inną klasę
  - Uwaga przy referencji jako parametrze aktualnym szablonu
  - Przy konsolidacji takie same problemy jak przy szablonach funkcji
- Polimorfizm statyczny
  - Przykład `cpp_9.21` i `cpp_9.22`