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;`

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`
- Przykład cpp_9.22

SFINAE - Substitution Failure Is Not An Error

- Sytuacja dotyczy szablonów 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

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)