Zaawansowane CPP/Wykład 3: Szablony II

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wprowadzenie

Mechanizm szablonów jest bardzo użyteczny ale może się okazać, że kod ogólny, który szablon implementuje, nie nadaje się do stosowania w każdym przypadku. W tej sytuacji mamy do dyspozycji dodatkowe własności implementacji szablonów w C++: przeciążanie i specjalizację. W poniższym wykładzie omówię sposób stosowania tych mechanizmów i różnice pomiędzy nimi.

Przeciążanie szablonów funkcji

Przeciążenie szablonu funkcji, podobnie jak przeciążenie zwykłych funkcji, definiuje nam nowy szablon. Możemy za pomocą przeciążenia zdefiniować np. funkcję służącą do znajdywania maksymalnego elementu w tablicy:

```
template<typename T> T max(T *data,size_t n) {
  T _max = data[0];
  for(size_t i=0;i<n;i++)
    if(data[i]>_max) _max=data[i];
  return _max;
}
```

Oba szablony: powyższy i wcześniej zdefiniowany

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

PRZYKŁAD 3.1

mogą ze sobą współistnieć i kompilator automatycznie wybierze poprawną definicję na podstawie argumentów wywołania funkcji. Oczywiście w obu przypadkach zadziała mechanizm automatycznej dedukcji argumentu szablonu.

```
int i,j,k;
double x,t[20];
```

```
k=max(i,j); //wywołanie max(int,int)
x=max(t,k); //wywołanie max<double>(double *,int)
```

(Źródło: max_overload.cpp)

Możemy jednak chcieć nie tyle zdefiniować nową funkcję, ile zmienić kod już istniejącego szablonu, tak aby dla pewnego podzbioru parametrów działał inaczej. Np. działanie funkcji `max` dla dwu wskaźników nie koniecznie jest tym, czego byśmy sobie życzyli. Możemy się spodziewać, że w tej sytuacji funkcja powinna zwrócić wskaźnik do większej wartości, a nie wskaźnik o wyższym adresie. Definiujemy więc nowy przeciążony szablon funkcji `max`:

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

(Źródło: max_overload.cpp)

PRZYKŁAD 3.2

Teraz sytuacja nie jest już jednoznaczna. Kompilator, napotykając wyrażenie

```
int i,j;
max(&i,&j);
```

może dopasować zarówno oryginalny szablon 3.1 z `T = int*` lub szablon 3.2 z `T = int`. I choć wydaje się że, otrzymamy błąd kompilacji, to do głosu dochodzi mechanizm rozstrzygania przeciążenia i kompilator wybierze dopasowanie drugiego szablonu jako “bardziej wyspecjalizowanego”, tzn. do którego pasuje mniejszy zbiór argumentów. Ewidentnie algorytm rozstrzygania przeciążenia szablonów funkcji nie jest prosty, polega on na częściowym porządkowaniu przeciążonych funkcji według stopnia ich specjalizacji. Dokładny opis tego algorytmu można znaleźć w D. Vandervoorde, N. Josuttis *"C++ Szablony. Vademecum profesjonalisty"*, rozdz. 12. Z grubsza rzecz biorąc szablon funkcji `F` jest bardziej wyspecjalizowany niż szablon `G` jeśli każdy zestaw argumentów, który da się dopasować do `F` da sie również dopasować do szablonu `G`, ale nie na odwrót. W naszym przypadku do szablonu 3.2 da się dopasować argumenty typu `(T *,T *)`, które ewidentnie można dopasować również do szablonu 3.1. Na odwrót już nie: `(int,int)` pasuje do 3.1, a do szablonu 3.2 nie.

Specjalizacja szablonów funkcji

Przy dotychczasowych definicjach szablonów `max`

```
template<typename T> T  max(T a, T b);         //(1)
template<typename T> T* max(T *a, T *b);       //(2)
template<typename T> T  max(T *data,size_t n); //(3)
```

będziemy dalej mieli kłopoty z funkcją `max` wywołaną dla argumentów typu `char*`. Takie argumenty zwyczajowo oznaczają napisy. Zgodnie z tym, co napisałem wcześniej, wywołany zostanie dla nich przeciążony szablon (2) i porówna tylko pierwsze litery napisów, co ewidentnie nie jest tym czego się oczekuje.

Na szczęście można dokonać specjalizacji tego szablonu dla argumentów typu `char *` i `const char *`:

```
template<> char *max<char *>(char *a,char *a) {
  return (strcmp(a,b)>0)?a:b;
}
template<> const char* max<const char *>(const char *a,const char *a) {
```

```
  return (strcmp(a,b)>0)?a:b;
}
```

Jak zwykle możemy pominąć argumenty szablonu podane w nawiasach ostrych za nazwą szablonu, jeśli mogą być one wydedukowane na podstawie argumentów wywołania i najczęściej spotkamy się z następującym kodem:

```
template<> char *max(char *a,char *a) {
  return (strcmp(a,b)>0)?a:b;
}
template<> const char* max(const char *a,const char *a) {
  return (strcmp(a,b)>0)?a:b;
}
```

(Źródło: max_spec.cpp)

Powyższe specjalizacje są pełne, tzn. określają dokładnie wszystkie argumenty wywołania szablonu. Dlatego lista parametrów szablonu w tych szablonach jest pusta. Tylko takie specjalizacje są dozwolone dla szablonów funkcji. Specjalizacja, w przeciwieństwie do przeciążenia, musi dotyczyć już istniejącego szablonu. Dlatego niedozwolona jest specjalizacja:

```
template<> const char* max<char *>(char *a,const char *a) {
 return (strcmp(a,b)>0)?a:b;}
```

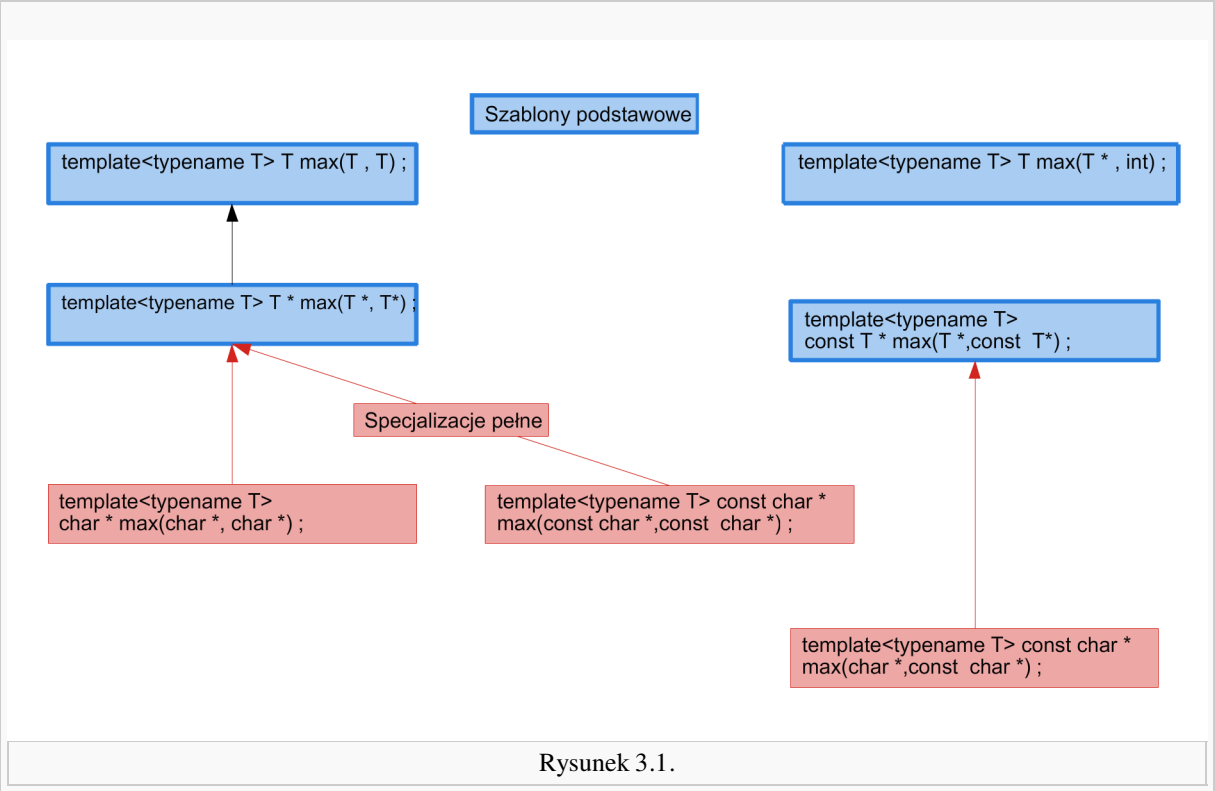
(Źródło: max_spec.cpp)

PRZYKŁAD 3.3

ponieważ argumenty są typu `char *` i `const char *`, i jako takie nie pasują do żadnego z istniejących szablonów (1-3). Musimy więc zdefiniować kolejne przeciążenie:

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

i dopiero wtedy kompilacja kodu 3.3 jest możliwa. Sytuację podsumowuje rysunek 3.1.



Rysunek 3.1.

Jawne podstawienie argumentów szablonu w miejsce parametru może prowadzić, w przypadku istnienia szablonów przeciążonych, do powstanie szeregu przeciążonych funkcji. Wtedy obowiązują "zwykłe" reguły rozstrzygania przeciążenia, np. wyrażenie

```
max<int>(0,0);
```

spowoduje "wygenerowanie" trzech funkcji:

```
int    max(int,int);
int   *max(int  *,int*);
int    max(int  *,int);
```

(Źródło: max_over_explicit.cpp)

Ponieważ zero lepiej pasuje do `int`-a niż do wskaźnika na `int`, wybrana zostanie pierwsza z powyższych funkcji.

Funkcje zwykłe a szablony

Obok szablonów mogą istnieć zwykłe funkcje o tej samej nazwie. Algorytm rozstrzygający przeciążenie preferuje dopasowanie zwykłych funkcji nad szablonami, więc jeśli zdefiniujemy sobie funkcję

```
int max(int i, int j);
```

to kompilator dokona następujących podstawień:

```
max(0,1); //zwykla funkcja int max(int,int)
max(0,1.0); //zwykla funkcja int max(int,int) z rzutowaniem double na int
max(1.0,1.0); //szablon max<double>(double, double)
```

(Źródło: max_func.cpp)

Z pozoru specjalizacje pełne opisane w poprzedniej części zachowują się jak zwykłe funkcje i moglibyśmy napisać:

```
char *max(char *a,char *a) {
  return (strcmp(a,b)>0)?a:b;}
```

zamiast

```
template<> char *max<char *>(char *a,char *a) {
  return (strcmp(a,b)>0)?a:b;}
```

Jest tak jednak tylko, jeśli możliwa jest dedukcja argumentów szablonu. W przypadku szablonu

```
template<typename T,typename U> T convert(U u) {
  return static_cast<T>(u);
};
```

możemy zdefiniować np. specjalizacje:

```
template<> int    convert<int,double>(double u) {...};
template<> double convert<double,double>(double u) {...};
```

i używać ich podając jawnie pierwszy, niededukowalny argument szablonu:

```
convert<int>(3.14);
convert<double>(2.71);
```

natomiast zdefiniowanie dwóch funkcji o tej samej nazwie i argumentach wywołania, różniących się tylko zwracanym typem, nie jest możliwe.

Nieudane podstawienie nie jest błędem

Jawne podstawienie wszystkich argumentów szablonu funkcji generuje nam jedną lub więcej funkcji "zwykłych". Może się jednak zdażyć, że niektóre podstawienia generują niepoprawny kod:

```
template<typename T> typename T::value t(T x) {
cerr<<"t1"<<endl;
};
```

Wywołanie

```
  t<int>(0);
```

(Źródło: sfinae.cpp)

prowadzi do `int::value` i jest nieprawidłowe. Spowoduje to błąd kompilacji, ale tylko wtedy, jeśli nie będzie innych przeciążonych szablonów funkcji `t`. Jeśli dodamy przeciążenie

```
template<typename T> void t(T x ) {cerr<<"t2"<<endl;};
```

(Źródło: sfinae.cpp)

to wyrażenie `t<int>(0)` zostanie do niego dopasowane. Innymi słowy, algorytm dopasowania przeciążenia pomija błędne podstawienia, nie generując błędów kompilacji.

Specjalizacje szablonów klas

Podobnie jak dla szablonów funkcji również dla szablonów klas istnieje możliwość podania różnych implementacji dla różnych zestawów argumentów szablonu. W przeciwieństwie jednak do szablonów funkcji, szablony klas nie mogą być przeciążane, a jedynie specjalizowane. Oznacza to, że w programie może istnieć tylko jeden szablon podstawowy o danej nazwie. Szablon podstawowy to szablon, w którego definicji nie występują nawiasy ostre po nazwie szablonu. Wszystkie szablony prezentowane do tej pory były podstawowe. Z tej reguły wynika, że trzy zdefiniowane do tej pory szablony stosu

```
template<typename T> Stack {...};
template<typename T,int N = 100> Stack {...}; //błąd szablon Stack już istnieje
template<typename T,template<typename X> C> Stack {
  C<T> _rep;
} //błąd szablon Stack już istnieje
```

nie mogą istnieć razem! Oczywiście w przypadku zastosowania domyślnych parametrów szablonu pierwsza definicja jest niepotrzebna, ale również bardziej pożyteczny trzeci szablon jest niedozwolony.

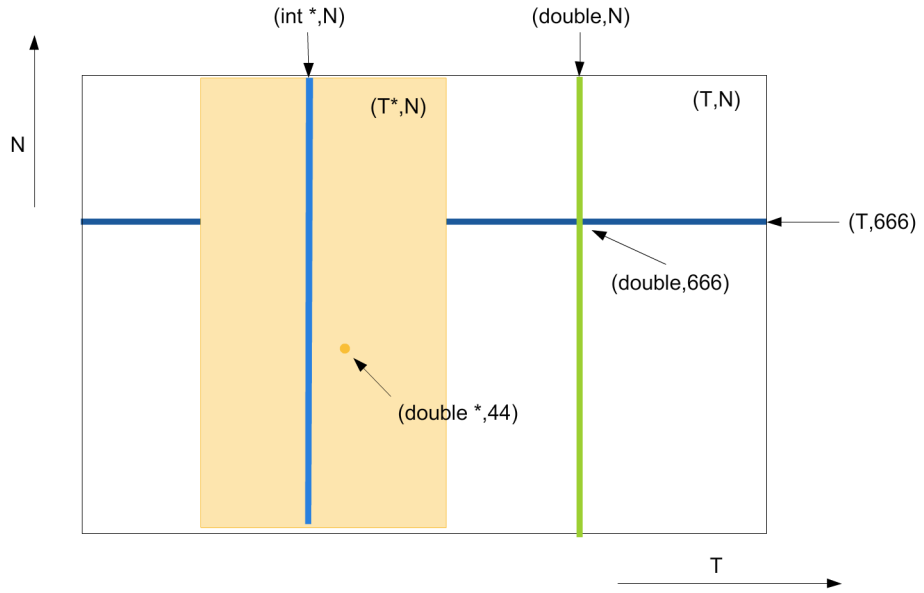
To ograniczenie można po części obejść, dokonując specjalizacji częściowej, która jest dozwolona tylko dla szablonów klas i daje możliwość specjalizacji szablonu dla pewnego podzbioru jego argumentów, a nie dla pojedynczego zestawu, jak specjalizacja pełna. Oczywiście specjalizacja pełna też jest możliwa. Rozważmy następujący przykład, definiując szablon podstawowy:

```
template<typename T,int N = 100> class Stack {};
```

możemy dokonać następujących specjalizacji:

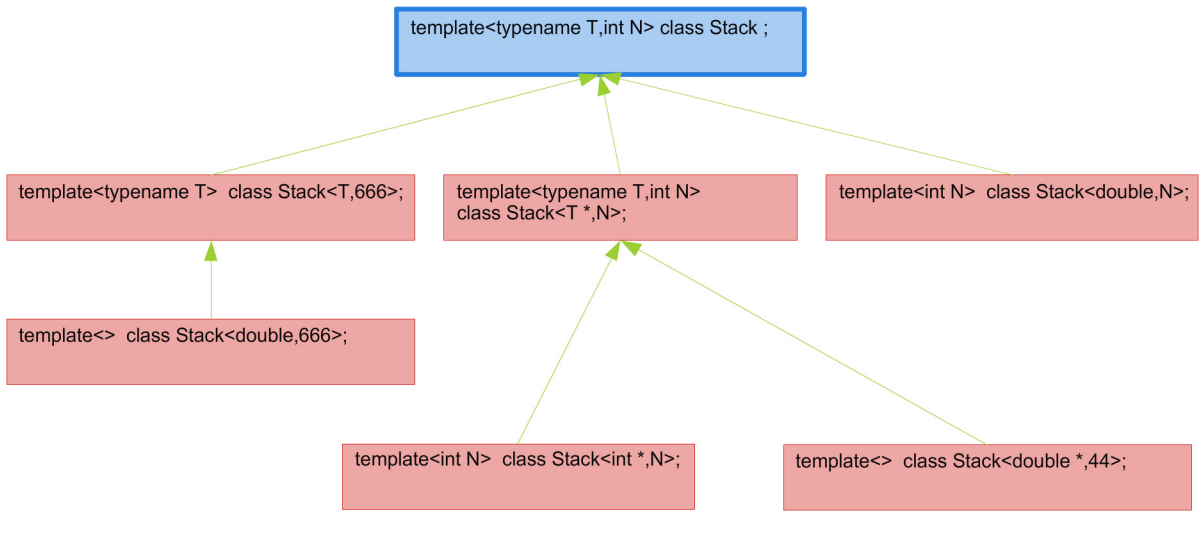
```
template<typename T>        class Stack<T,666>     {};
template<typename T,int N>  class Stack<T*,N>      {};
template<int N>             class Stack<double ,N> {};
template<int N>             class Stack<int *,N>   {};
template<>                  class Stack<double,666>{};
template<>                  class Stack<double *,44> {};
```

(Źródło: stack_spec.cpp)



Rysunek 3.2. Symboliczne przedstawienie zbiorów argumentów dla różnych specjalizacji szablonu `Stack<T,N>`.

Każda z tych specjalizacji definiuje pewien podzbiór parametrów szablonu podstawowego (zob. rysunek 3.2). Jeśli któryś z podzbiorów zawiera się w drugim, to mówimy, że jedna specjalizacja jest bardziej wyspecjalizowana od drugiej. Hierarchia specjalizacji dla powyższego przykładu pokazana jest na rysunek 3.3. Jeżeli jakiś zestaw parametrów należy do dwóch (lub więcej) podzbiorów, które się przecinaja, ale żeden nie zawiera się w drugim, to dla tych parametrów kompilator nie bedzie w stanie wybrać specjalizacji.



Rysunek 3.3. Uporządkownie specjalizacji szablonu `Stack<T,N>`.

Oczywiście ten przykład jest bardzo sztuczny i trudno sobie wyobrazić powód tworzenia takich specjalizacji. Rozważmy bardziej realistyczny przypadek: deklarujemy szablon podstawowy, ale bez podawania jego definicji; będziemy korzystać jedynie z jego specjalizacji:

```
template<typename T,int N = 100, typename R = T*> class Stack;
```

Następnie definiujemy dwie specjalizacje. Pierwszą dla stosów opartych o zwykłe tablice:

```
template<typename T,int N> class Stack<T,N,T*> {
  T _rep[N];
  unsigned int _top;
public:
  Stack():_top(0){};
  void push(T e) {_rep[_top++]=e;}
  T pop() {return _rep[--_top];}
};
```

i drugą opartą o kontenery STL:

```
template<typename T,int N,template<typename E> class Sequence>
  class Stack<T,N,Sequence<T> > {
  Sequence<T> _rep;
public:
  void push(T e) {_rep.push_back(e);};
  T pop() {T top = _rep.top();_rep.pop_back();return top;}
  bool is_empty() const {return _rep.empty();}
};
```

(Źródło: Stack_2.cpp)

Korzystając z tych specjalizacji możemy pisać następujący kod.

```
main() {
  Stack<int,100,int *>            s_table;
  Stack<int,100>                  s_default ;
  Stack<int,0,std::vector<int> >  s_container;
}
```

(Źródło: Stack_2.cpp)

W każdym przypadku kompilator wybierze implementację odpowiednią dla podanych parametrów.

Szablony a dziedziczenie

Szablony klas mogą oczywiście dziedziczyć z innych klas. Deklaracja

```
template<typename T> Stack: public Container {
  ...
};
```

oznacza, że każda instancja danego szablonu `Stack<T>` dziedziczy z klasy `Container`. Ponieważ konkretna instancja szablonu jest klasą, to dowolna klasa czy szablon może dziedziczyć z instancji szablonu:

```
class special Stack_int : public Stack<int> {...}
```

Definiując specjalizację szablonu klasy możemy dziedziczyć z innych specjalizacji tej samej klasy; nie może to jednak prowadzić do rekurencji. Jeśli napiszemy:

```
template<typename T,int N> Stack {...};
template<typename T>
Stack<T*,N>: private Stack<void *,N> {...};
```

(Źródło: stack_void.cpp)

to kompilator odmówi skompilowania tego kodu z powodu rekurencyjnej definicji specjalizacji szablonu `Stack`. Wszystko będzie w porządku jeśli dodamy specjalizację dla typu `void *`:

```
template<int N> class Stack<void *,N> {...}
```

(Źródło: stack_void.cpp)

Dlaczego mielibyśmy jednak dziedziczyć implementację klasy `void*`? Powodem jest unikanie powielania kodu. Ponieważ każda konkretyzacja (instancja) szablonu jest osobną klasa, to dla każdej generowany jest pełny kod potrzebnych funkcji. Jeśli te funkcje są proste, to nie jest to kłopot. W praktyce implementacja stosu musi zwykle uwzględniać dynamiczne zarządzanie pamięcią i może być dużo bardziej skomplikowana, a zatem generowany kod będzie odpowiednio większy. Ogólnie jest to nie do uniknięcia, ale ponieważ wszystkie wskaźniki mają ten sam rozmiar i można je rzutować na `void *` to możemy wykorzystać implementację `Stack<void *>` do implementacji pozostałych typów wskaźnikowych:

```
template<typename T,size_t N>
Stack<T*,N>: private Stack<void *,N> {
public:
  T* pop() {
    return static_cast<T*>(Stack<void *>::pop());
  };
  void push(T *e) {
    Stack<void *>::push(e);
  }
  bool is_empty() {return Stack<void *>::is_empty();}
};
```

(Źródło: stack_void.cpp)

Korzystamy tu z automatycznej konwersji `T*` na `void *`. W ten sposób, np. kod funkcji `Stack<int *>::push(int *)` będzie zawierał tylko parę instrukcji opakowujących wywołanie kodu funkcji `Stack<void *>::push(void *)`. Proszę zwrócić uwagę na zastosowanie dziedziczenia prywatnego.

Zależne klasy bazowe

Szablon klasy może również dziedziczyć z innego szablonu klasy, którego argumenty bedą zależały od jego parametrów:

```
template <typename T> class Base<T> {...};
template<typename S>
class Derived: public Base<S> {};
```

Przy tych definicjach klasa `Derived<double>` dziedziczy z klasy `Base<double>`. Taką klasę nazywamy zależną klasą bazową i jest to bardzo częsta konstrukcja w programowaniu uogólnionym.

Z zależnymi klasami bazowymi wiąże się jednak pewna zasada, związana z wyszukiwaniem nazw, która może być sporym zaskoczeniem. Rozważmy następujący przykład:

```
template<typename T>  class Base {
public:
  Base():basefield(0){};
```

```
  int basefield;
};
template<typename T> class DD :public Base<T> {
public:
  void f() {std::cerr<<basefield<<std::endl;}
};
```

(Źródło: base.cpp)

Ten kod się nie skompiluje przy pomocy kompilatora C++ zgodnego ze standardem. Np. nie skompiluje go kompilator g++-3.4, a g++-3.3 tak. Powód tego faktu jest następujący: nazwa `basefield`, występująca w klasie `DD` jest nazwą niezależną (od parametru szablonu). Klasa bazowa, w której ta nazwa jest zdefiniowana jest klasą bazową zależną (od parametru szablonu). Według standardu kompilator nie wyszukuje nazw niezależnych w zależnych klasach bazowych. Kompilator g++-3.4 jest bliżej stadardu niż g++-3.3 i stąd to całe zamieszanie. Aby kod się skompilował należy uczynić tę nazwę zależną, np. poprzez kwalifikowanie jej nazwą klasy:

```
template<typename T> class DD :public Base<T> {
public:
  void f() {std::cerr<<DD::basefield<<std::endl;}
};
```

(Źródło: base.cpp)

lub przez

```
template<typename T> class DD :public Base<T> {
public:
  void f() {std::cerr<<this->basefield<<std::endl;}
};
```

(Źródło: base.cpp)

CRTP

Dziedziczenie szablonów można też wykorzystać do przydatnej "sztuczki", zwanej po angielsku *"couriously reccuring template pattern"* (autorem tego idiomu jest James O. Coplien). Rozważmy następujący problem: chcemy zaimplementować mechanizm automatycznego liczenia ilości obiektów danej klasy. To standardowe zadanie na zastosowanie konstruktorów, destruktorów i statycznych składowych klasy:

```
class Countable {
protected:
  static size_t _counter;
public:
  Countable() {++_counter;}
  Countable(const Countable &) {++_counter;}
  virtual ~Countable {--_counter}
  static size_t counter()  {return _counter;}
};
size_t Countable::_counter = 0;
```

Oczywiście wpisywanie tego kodu do każdej klasy, której obiekty chcemy zliczać jest nużące i łamie zasadę niepowielania kodu. Postaramy się więc wykorzystać kod klasy `Countable`, dziedzicząc go w innych klasach:

```
class MyClass1 : public Countable {
  ...
};
class MyClass2 : public Countable {
```

```
  ...
};
```

Niestety ponieważ obie klasy `MyClass1` i `MyClass2` dziedziczą z tej samej klasy, dziedziczą również ten sam wspólny licznik. Tak więc zliczaniu podlegać będą obiekty obu klas wspólnie. W rozwiązaniu pomogą nam szablony. Wystarczy uczynić klasę `Countable` szablonem

```
template<typename T> class Countable {
protected:
  static size_t _counter;
public:
  Countable() {++_counter;}
  Countable(const Countable &) {++_counter;}
  virtual ~Countable() {--_counter}
  static size_t counter()  {return _counter;}
};
template<typename T> size_t Countable<T>::_counter = 0;
```

(Źródło: countable.cpp)

i używać go w następujący sposób:

```
class MyClass1 : public Countable<MyClass1> {
  ...
};
class MyClass2 : public Countable<MyClass2> {
  ...
};
```

(Źródło: countable.cpp)

Ponieważ każda konkretyzacja szablonu jest osobną klasą, klasy `MyClass1` i `MyClass2` dziedziczą z różnych klas bazowych i będą posiadać różne liczniki, ale ciągle wspólne w ramach każdej klasy. Parametryzowanie klasy bazowej typem klasy dziedziczącej gwarantuje jej unikatowość.

Źródło: "http://wazniak.mimuw.edu.pl/index.php?title=Zaawansowane_CPP/Wyk%C5%82ad_3:_Szablony_II"

- Tę stronę ostatnio zmodyfikowano o 13:35, 22 lis 2006;