PRZESTRZENIE NAZW I STANDARDOWE BIBLIOTEKI

281

C++ - przestrzenie nazw

Czasem w plikach C++ zakładanych automatycznie używana jest *dyrektywa using*:

using namespace std;

Po co?

Ponieważ w C++ wszystko, co programista utworzy i wszystko, co zawarte jest w bibliotekach funkcjonuje w jakiejś *przestrzeni nazw*.

A my najczęściej chcemy używać tego, co należy do przestrzeni nazw **std**

Co to jest przestrzeń nazw?

282

C++ - przestrzenie nazw

- Przestrzenie nazw pozwalają na podział kodu na logiczne jednostki. Stąd: przestrzeń nazw ↔ fragment kodu programu.
- Oczekiwania:
  wewnątrz jednej przestrzeni nazw wszystkie funkcje i klasy powinny logicznie uzupełniać się tworząc całość skierowaną na realizacje określonego celu, np. kompletną implementację złożonego algorytmu, bądź np. jeżeli mamy program łączący się przez internet – kompletną funkcjonalność związaną z tą łącznością:

```
namespace net_connect
{
  int make_connection();
  int test_connection();
  // itd...
}
```


283

C++ - przestrzenie nazw

- Poza tą przestrzenią można odwoływać się do jej funkcji i klas korzystając z prefixu takiego, jak nazwa przestrzeni oraz operatora zakresu **::**
- Przykład:

```
net_connect::make_connection();
```

Czemu ma służyć taki podział kodu?

- Podziałowi składowych programu na grupy, podobne do klas albo struktur.
- Przestrzenie nazw nie tworzą jednak żadnych obiektów, nie potrzebują tworzenia swoich instancji, żeby być używane. Po prostu aby użyć określonej funkcji piszemy tylko nazwę przestrzeni i nazwę tej funkcji.
- Ten podział pozwala na nadawanie tych samych nazw funkcjom należącym do różnych przestrzeni nazw, a więc chroni przed potencjalnymi konfliktami nazw w przypadku łączenia bibliotek.

284

C++ - przestrzenie nazw

Jeżeli nie chcemy za każdym razem pisać nazwy przestrzeni możemy się do niej odwołać przez słowo kluczowe *using*

Jeżeli w bloku kodu napiszemy

```
using namespace nazwa_przestrzeni
```

w obrębie tego bloku możemy odwoływać się do wszystkich funkcji należących do tej przestrzeni bez potrzeby podawania jej nazwy jako prefixu (jeżeli napiszemy tak na początku pliku – dostęp do funkcji obowiązuje w obrębie całego pliku)

Jeżeli chcemy odwoływać się tylko do wybranej funkcji z pewnej przestrzeni nazw możemy napisać:

```
using nazwa_przestrzeni::nazwa_funkcji
```


285

C++ - przestrzenie nazw

```
namespace A {
   const int two = 2;
   void napisz() {
   printf("A\n"); };
}
namespace B {
   void napisz()
   {printf("B\n"); };
}
namespace C {
   const int two = 22;
}
```

```
int main () {
   using A::napisz;
   using namespace C;
   napisz();
   B::napisz();
   printf("%i\n",two);

   return 0;
}
```

// co się pojawi w oknie konsoli?

286

C++ - przestrzenie nazw

A jeżeli chcemy w obrębie całego pliku korzystać ze wszystkich funkcji bibliotecznych, piszemy:

```
using namespace std;
```

ponieważ właśnie te funkcje są zadeklarowane w przestrzeni nazw **std**

C++ - przestrzenie nazw

Przestrzenie nazw to nowy element w C++.

W C też używamy plików nagłówkowych bibliotek standardowych, ale tam wszystkie nazwy w nich zadeklarowane są jednakowo dostępne bez żadnych kwalifikacji – nie ma tam mechanizmu przestrzeni nazw.

Aby zapewnić możliwość kompilowania programów w C przez kompilatory C++ przyjęto umowę:

jeżeli włączymy plik w sposób tradycyjny, podając pełną nazwę pliku nagłówkowego, jest on włączany oraz automatycznie otwierana jest nowa przestrzeń nazw **std**. Jeżeli ten sam plik włączymy pod nową nazwą, to przestrzeń std nie jest automatycznie otwierana.

Przykład:

C: `#include <string.h>` C++: `#include <cstring>`

C++ - przestrzenie nazw

Przyjęto konwencję, że pliki nagłówkowe o tradycyjnej nazwie „**nazwa.h**" są w C++ nazywane „**cnazwa**":

- `assert.h` `cassert`
- `limits.h` `climits`
- `stdio.h` `cstdio`
- `time.h` `ctime`
- `ctype.h` `cctype`
- `stdlib.h` `cstdlib`
- `math.h` `cmath`
- `...` `iostream (!)`
- …

W kodzie programu pisanego w C++ należy używać wersji przygotowanej dla C++.

C++ - przestrzenie nazw

- Cały kod, który został napisany poza jakąkolwiek przestrzenią nazw w rzeczywistości należy do tzw. globalnej przestrzeni nazw
- Jeżeli chcemy tylko zamknąć kod w swojej przestrzeni nazw, możemy napisać:

```
namespace
 {
  class Car
  {
   .... // składowe klasy Car
  }
  // inne składowe „przestrzeni nazw bez nazwy" (unnamed namespace)
 }
```


C++ - przestrzenie nazw

Kiedy używamy przestrzeni nazw bez nazwy, kompilator sam wymyśla dla niej unikalną nazwę

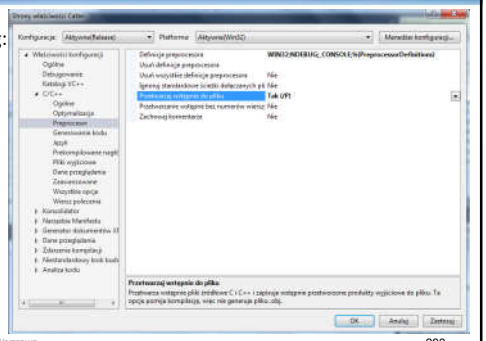
Po co używać takiej przestrzeni nazw?

Jeżeli deklarujemy struktury lub klasy, które będą używane tylko w lokalnym fragmencie kodu, mamy pewność, że ich nazwa nie wejdzie w kolizję z nazwą zadeklarowaną gdziekolwiek w globalnej przestrzeni nazw.

STANDARDOWE DYREKTYWY PREPROCESORA

Dyrektywy Preprocesora

Preprocessing:
Zbiór działań wykonywanych tuż przed kompilacją wskazanego pliku.
Aby zobaczyć wynik działania preprocesora należy (VS2017):



293

Dyrektywy Preprocesora

Standardowe dyrektywy dzielą się na trzy rodzaje:

1. Dołączające plik we wskazanym miejscu:
   `#include`
2. Makra:
   `#define`
3. Definiujące warunkową kompilację:
   `#if, #ifdef, #ifndef, #elif, #else, #endif`

Składnia i zasady stosowania

1. Zawsze na początku linii zaczynają się od #
2. Mogą wystąpić w każdym miejscu w programie
3. Mogą za nimi wystąpić komentarze
4. Zajmują jedną linię, chyba że jawnie wprowadzono znak kontynuacji

294

Dyrektywy Preprocesora

Dołączanie plików

Dołączane pliki powinny być do tego odpowiednio przygotowane. Przyjmuje się, że tak przygotowany przez programistę jest każdy plik z rozszerzeniem „h"

Polecenie:

`#include <filename>`

mówi preprocesorowi, aby zastąpił w/w linię zawartością wskazanego pliku.

Są dwa formaty:

1. **`#include "filename"`** – szuka pliku w bieżącym folderze, a jeżeli nie znajdzie, w folderach wskazanych w systemie (w systemowych zmiennych środowiskowych, np. PATH, INCLUDE, itp.) – *dla plików h użytkownika*
2. **`#include <filename>`** – szuka pliku tylko w folderach wskazanych w systemie (w systemowych zmiennych środowiskowych, np. PATH, INCLUDE, itp.) – *dla plików h bibliotecznych i systemowych*

295

Dyrektywy Preprocesora

Dyrektywa #define

Polecenie:

`#define nazwa` ***`dużo_różnego_tekstu`***

mówi preprocesorowi, aby każde wystąpienie **`nazwa`** zastąpił przez ***`dużo_różnego_tekstu`***.

Drugi argument, tj. ***`dużo_różnego_tekstu`*** jest opcjonalny. Jeżeli go nie ma, identyfikator **`nazwa`** istnieje, ale nie ma wartości. Taki identyfikator też może być przydatny – jako flaga (można sprawdzać, czy jest, albo czy go nie ma)

Polecenie:

`#undef nazwa`

mówi preprocesorowi, aby usunął („*oddefiniował*") **`nazwa`**.

296

Dyrektywy Preprocesora

Dyrektywa #define

1. Zastępuje tekst powtarzany w wielu miejscach kodu innym tekstem.
2. Najczęściej używana do definiowania stałych.
3. Stałe zwykle pisane są DRUKOWANYMI literami (*taka konwencja*), aby programistom było łatwiej je rozpoznać.

Przykład: to może powodować komunikat ostrzeżenia (*warning*):

```
#define SOME_VALUE 5
#define SOME_VALUE 6
```

A to pójdzie gładko:

```
#define SOME_VALUE 5
#undef  SOME_VALUE
#define SOME_VALUE 6
```


297

Dyrektywy Preprocesora

Makra

Tutaj nie wolno dać spacji (!)

Ogólna forma definicji makra:

```
#define macro(param1, param2, …) value
```

Przykład:

```
#define SUM(a, b) a + b
...
x = SUM(2, 6); /* zostanie zamienione na: "x = 2 + 6;" */
```

Jeżeli makro jest dłuższe, niż zaplanowana szerokość wiersza, można je łamać:

```
#define MIN(a, b) a < b \
      ? a : b
```


298

Dyrektywy Preprocesora

Makra

Łamanie wierszy:

```
#define INC(A)      \
   if ((A) < 100)   \
      (A)++;        \
   else             \
      printf ("Błąd: przepełnienie.\n");
```


299

Dyrektywy Preprocesora

Makra – przykłady:

```
#define SQUARE(x) ((x) * (x))

s = SQUARE(5);
```

staje się:

```
s = ((5) * (5));

s = SQUARE(a + 1);
```

staje się:

```
s = ((a + 1) * (a + 1));
```


300

Dyrektywy Preprocesora

Makra – przykłady:

```
#define MAX(a, b) \
((a) > (b) ? (a) : (b))

x = 4;
y = 9;
z = MAX(x, y);
```

staje się:

```
z = ((x) > (y) ? (x) : (y));
```


301

Dyrektywy Preprocesora

Makra – przykłady:

Stosowanie makra działa jak polecenie „znajdź i zastąp" w edytorze tekstu:

```
#define SQUARE1(x) x * x
#define SQUARE2(x) (x * x)
#define BAD_SUM(x, y) (x) + (y)
s = SQUARE1(a + 1);
/* s = a + 1 * a + 1; */
s = SQUARE2(a + 1);
/* s = (a + 1 * a + 1); */
s = BAD_SUM(a + 1, b + 2) * 3;
/* s = (a + 1) + (b + 2) * 3; */
```

Dlatego stosowanie nawiasów jest takie ważne (!)

302

Dyrektywy Preprocesora

Makra

Działanie preprocesora – zamiana symboli na wartości:

1. Argumenty wywołanych makr sprawdzane są, czy same nie reprezentują symboli zdefiniowanych za pomocą **`#define`**
2. Argumenty będące zdefiniowanymi symbolami są zastępowane wartościami
3. Zmodyfikowany program jest sprawdzany, czy występują w nim nadal symbole zdefiniowane za pomocą **`#define`**, i jeżeli tak – powrót do pkt 1.

Wyjątek: literały nie są skanowane, np.:

```
#define MAX 10
printf("Wartość MAX wynosi %d.\n", MAX);
```

Wyjście: Wartość MAX wynosi 10.

303

Dyrektywy Preprocesora

Makra to nie funkcje!

Ponieważ:

1. Są podmieniane tekstowo co może dać szybszy kod w przypadku prostych czynności.
2. Mogą być rekurencyjne.
3. Nie obejmuje ich kontrola typów podczas kompilacji.
4. Mogą generować bardzo skomplikowane błędy zachowania programu mimo poprawnej kompilacji.
5. Mogą mieć identyfikatory typów jako argumenty, np.:

```
#define PRINT_SIZE(type) \
printf("%d\n", (int) sizeof(type))
```


304

Dyrektywy Preprocesora

Makra łatwo dają efekty uboczne

Przykład:

```
#define MIN(x, y) ((x) < (y) ? (x) : (y))

m = MIN(++a, --b);
/* m = ((++a) < (--b) ? (++a) : (--b)); */
```

- Co wtedy robić?
  – Nie używać wyrażeń dających efekty uboczne jako argumentów.
- Ale skąd wiadomo, że wywołuję makro?
  – Należy stosować konwencję nazewniczą dotyczącą makr i robi się łatwiej..

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Dyrektywy wspierające warunkową kompilację:

```
#if
#ifdef
#ifndef
#else
#elif
#endif
```


Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

`#if const-expr1`
Linijki kodu znajdujące się tutaj są dołączane przez procesor do pliku wynikowego tylko jeżeli **`const-expr1`** jest prawdziwe

`#elif const-expr2`
Linijki kodu znajdujące się tutaj są dołączane przez procesor do pliku wynikowego tylko jeżeli **`const-expr1`** jest fałszywe a **`const-expr2`** jest prawdziwe

`#else`
Linijki kodu znajdujące się tutaj są dołączane przez procesor do pliku wynikowego tylko jeżeli **`const-expr1`** jest fałszywe i **`const-expr2`** też jest fałszywe

`#endif`

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Najbardziej użyteczna dyrektywa:

`#ifdef identifier`

1. Cały kod występujący po niej zostanie dołączony przez preprocesor do pliku wynikowego w zależności od tego, czy **`identifier`** istnieje, czy nie.
2. Jej obszar działania ogranicza wystąpienie **`#endif`**. Następne linijki kodu traktowane są już jako konieczne do dołączenia chyba, że kolejne wywołanie dyrektywy to zmieni.

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Przykład użycia
– funkcja drukująca wartość tylko w trybie **DEBUG**:

```
void debug_print (int i) {
#ifdef DEBUG
   printf ("DEBUG PRINT: %d\n", i);
#endif
}
```

Ale w trybie „release" to nadal jest wywołanie funkcji. Czy można tego uniknąć?..

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

Makro wykorzystujące warunkową kompilację:

```
#ifdef DEBUG
#define debug_print(a) \
          printf ("DEBUG PRINT: %d\n", (int)a);
#else
#define debug_print(a)
#endif
```

Dyrektywy Preprocesora

Definiowanie warunkowej kompilacji

```
#ifdef NDEBUG
#define assert(EX)
#else
#define assert(EX) (void)((EX) || (__assert (#EX, __FILE__, __LINE__),0))
#endif

#ifdef __cplusplus
extern "C" {
#endif

extern void __assert (const char *msg, const char *file, int line);

#ifdef __cplusplus
};
#endif
```


C++ - szablony

- Szablony, nazywane też wzorcami, pozwalają na abstrakcyjne definiowanie funkcji i klas w terminach parametrów, którymi są typy danych
- Na podstawie szablonu kompilator sam może wygenerować wiele definicji konkretnych funkcji i klas, kiedy będą potrzebne
- Standardowa biblioteka C++ bazuje na szablonach i dlatego jest nazywana STL: Standard Template Library

C++ - szablony

Szablony funkcji

Zdarza się, że wiele funkcji jest do siebie bardzo podobnych w działaniu – różnią się tylko typem argumentu, który przyjmują

Np. funkcja która z tablicy wybiera element o najmniejszej wartości: algorytm szukania jest wspólny dla większości typów danych.

Jednak jeżeli chcemy znajdować najmniejszą wartość dla tablic z elementami typu: **`int`**, **`double`**, lub **`Integer`** (*zdefiniowana klasa, dla której istnieje przeciążony operator porównania*), to dla każdego z typów należy napisać na nowo całą funkcję

Jeżeli te funkcje będą fragmentem biblioteki, to za każdym razem kiedy będziemy potrzebowali dopisać funkcję dla nowego typu danych w tablicy, będzie trzeba rekompilować bibliotekę i udostępniać jej użytkownikom nowy release

To jest uciążliwe. Lepiej zrobić tę funkcję raz wykorzystując szablony.

C++ - szablony

Kompilator tworzy tyle wersji funkcji wg wskazanego szablonu, ile programista zażąda w swoim programie: będą takie same co do działania, ale będą różniły się co do typów argumentów wywołania, typów zmiennych lokalnych czy typu wartości zwracanej

Szablony można tworzyć nawet dla typów, które w chwili pisania szablonu jeszcze w ogóle nie istniały

Aby poinformować kompilator, że kod, który piszemy reprezentuje szablon, należy użyć słowa kluczowego **`template`**
– kod, który znajduje się poniżej tego słowa jest definicją wzorca funkcji lub klasy

C++ - szablony

Przykład:

```
template <class T1, class T2>
T2& moja_funkcja(T1 arg1, T2* arg2)
{
  … // Tutaj jakiś kod funkcji
    // wykorzystującej typy T1 i T2
  return *arg2;
}
```

W nawiasach kątowych lista parametrów formalnych wzorca, każdy poprzedzony słowem **class**, lub **typename** (**typename** wprowadzono później)

Parametry zwyczajowo nazywa się z wykorzystaniem T, np. T1, T2

Następnie występuje definicja wzorca wykorzystująca T1 i T2 tam, gdzie powinna występować nazwa konkretnego typu

C++ - szablony

Na podstawie typów argumentów kompilator ma zgadnąć, który szablon i jak należy wykorzystać:

```
template <class T1, class T2>
T2& moja_funkcja(T1 arg1, T2* arg2) {
    return *arg2;
}
int main(int argc, char *argv[])
{
    int a = 9;
    double b = 10.2, c = 3.14;
    moja_funkcja(a, &b);  // konkretyzacja wzorca na
                          // double& moja_funkcja(int arg1, double* arg2)
    moja_funkcja(a, &c);  // tu nie ma konkretyzacji - funkcja już jest.
    moja_funkcja(b, &a);  // konkretyzacja wzorca na
                          //int& moja_funkcja(double arg1, int* arg2)
```


C++ - szablony

Słowo kluczowe **class** w linii:

```
template <class T1, class T2>
```

nie oznacza, że T1 i T2 mogą być tylko klasami (jak widać na przykładzie); mogą to być dowolne typy (wbudowane lub definiowane przez użytkownika)

Dla czytelności w późniejszych wersjach kompilatorów zastąpiono to słowo słowem **typename**

```
template < typename T1, typename T2>
```

C++ - szablony

Przeciążanie szablonów funkcji:

```
template <typename T>
T wiekszy(const T& k1, const T& k2) {
                  return k1 < k2 ? k2 : k1 ;
}
double wiekszy(const double& d1, const double& d2)
                  return d1 < d2 ? d2 : d1 ;
}
int main(int argc, char *argv[])
{
    int a, b=0, c=1;
    a = wiekszy(b,c);  // która wersja funkcji zostanie użyta?
                       // Istnieje standardowa konwersja int na double, ale
                       // istnieje też szablon wg którego można utworzyć
                       // właściwą funkcję
    double x, y=0.3, z=2.14;
    x = wiekszy(y,z);  // która wersja funkcji zostanie użyta?
```


C++ - szablony

Wskazywanie wartości parametru szablonu

```
class Integer {
   int i;
public:
   Integer(int ii): i(ii) {}
   bool operator<(
        const Integer& rv) const
   {
        return i < rv.i;
   }
   bool operator>(
        const Integer& rv) const
   {
        return i > rv.i;
   }
    …
};
```

```
template <typename T>
T wiekszy(const T& k1, const T& k2) {
    return k1 < k2 ? k2 : k1 ;
}

int main(int argc, char *argv[])
{
    Integer I1(0), I2(1), I3(2);
    I1 = wiekszy<Integer>(I2, I3);
```

Jeżeli nie ma jednoznacznej interpretacji, można za nazwą szablonu w nawiasach kątowych podać listę typów, które należy w tym miejscu zastosować

C++ - szablony

Informacja o typie zmiennej

Testując szablony czasem niezbędne jest sprawdzenie, jakiego typu są zmienne, których szablon używa (czy np. kompilator sam podjął decyzję o konwersji zmiennej na podobny typ, etc)

Do sprawdzenia służy operator **typeid**

Operator zwraca obiekt typu **const std::type_info**

Klasa ta ma pożyteczną metodę **name()**

C++ - szablony

Chciałoby się napisać:

```
std::type_info typ = typeid(a);
printf("%s\n",typ.name());
```

Jednak autorzy biblioteki sprytnie zastrzegli konstruktor kopiujący oraz operator przypisania jako **private** uniemożliwiając de facto tworzenie w programie własnych obiektów tego typu z obiektów zwracanych przez operator `typeid`:

```
private:
   /// Assigning type_info is not supported.  Made private.
   type_info& operator=(const type_info&);
   type_info(const type_info&);
```

Dlatego aby wywołać metodę **name** można wyłącznie napisać tak:

```
printf("%s\n",typeid(k1).name());
```


C++ - szablony

Porada programistyczna

Na etapie testowania programu można w szablonie funkcji i w funkcji dodać instrukcje ujawniające typ danych, które są przetwarzane:

```
template <typename T>
T wiekszy(const T& k1, const T& k2) {
                printf("Szablon dla: %s\n",typeid(k1).name());
                return k1 < k2 ? k2 : k1 ;
}
double wiekszy(const double& d1, const double& d2) {
                printf("Dla double: %s\n",typeid(d1).name());
                return d1 < d2 ? d2 : d1 ;
}
```


część 2:

SZABLONY KLAS

C++ - szablony

Szablony klas

Na tej samej zasadzie co szablony funkcji można tworzyć szablony klas

Składnia jest analogiczna:

```
template <typename T, typename M>
class Klasa {
  // tu używamy typów T i M
  …
};
```

Aby utworzyć obiekt nie możemy napisać:

```
Klasa K;
```

bo nie wiadomo, jakie typy przypisać parametrom M i T, a kompilator nie ma szansy domyślić się. Wskazanie wartości parametrów jest konieczne.

C++ - szablony

Szablony klas

Deklaracja obiektów:

```
template <typename T, typename M>
class Klasa {
  …       // tu używamy typów T i M
};

Klasa<double, int> K;
Klasa<int, Integer> I;
```


C++ - szablony

Szablony klas

Jeżeli chcemy, żeby metody tej klasy zostały definiowane poza szablonem klasy, to musimy tam prawidłowo napisać nazwę szablonu:

```
Klasa::Klasa();            // konstruktor domyślny – Źle!

template <typename T, typename M>
Klasa<T, M>::Klasa();      // kontruktor domyślny – Dobrze!
```

C++ - szablony

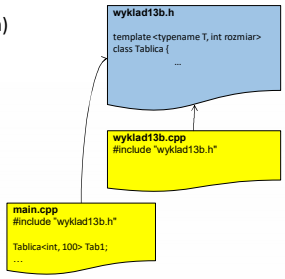
Parametry szablonu klasy mogą być też wartościami określonego typu, np. :

```
template <typename T, int rozmiar >
class Klasa {
  // tu używamy typu T i zmiennej rozmiar
  …
};
Klasa<int, 100> Tab;
Klasa<double, 2000> *Tab2;
Klasa<double, 3000> *Tab3;
```

`Tab`, `Tab2` i `Tab3` są obiektami różnych typów, dlatego kompilator nie zaakceptuje takich instrukcji jak przepisanie wartości między wskaźnikami, itp.

C++ - szablony

Projekt składający się z pliku main.cpp i biblioteki (para plików: .cpp i .h)



C++ - szablony

Kompilacja szablonów

Zawsze kiedy ukonkretniany jest szablon klasy, kod definicji takiej klasy dla danej specjalizacji jest generowany wraz z metodami składowymi wywoływanymi w programie – i tylko z tymi *rzeczywiście* wywoływanymi metodami składowymi.

Jest to unikanie nadmiarowego kodu.

To daje możliwość elastycznego projektowania szablonów, wykorzystującego różne właściwości różnych konkretnych typów.

C++ - szablony

```
class Pierwsza {
public:
  void f() {}
};

class Druga {
public:
  void g() {}
};
```

```
template<typename T>
class Trzecia {
  T t;
public:
  void a() { t.f(); }
  void b() { t.g(); }
};

int main() {

  Trzecia<Pierwsza> TP;
  TP.a(); // nie tworzy Trzecia<Pierwsza>::b()

  Trzecia<Druga> TD;
  TD.b(); // nie tworzy Trzecia<Druga>::a()
}
```


C++ - szablony

Szablony vs. pliki nagłówkowe

Zasada: deklaracje i kompletne definicje szablonów funkcji i klas umieszczamy w pliku nagłówkowym (i tylko tam).

Uzasadnienie:

Skoro szablony są wykorzystywane do generowania kodu tylko gdy w kodzie wystąpi wykorzystanie szablonu, to:

1. Umieszczenie kodu metod i funkcji w pliku cpp sprawi, że będzie on kompilowany oddzielnie – nie będzie w tym pliku instrukcji wykorzystujących szablon, które należą do innych funkcji i metod, a więc kompilator nie będzie wiedział jakich konkretyzacji dokonać
2. Tam, gdzie będą próby wykorzystania szablonu, kompilator będzie miał same nagłówki klas i funkcji, ale bez definicji, więc nie będzie potrafił skonkretyzować żądanych wersji szablonu

Efekt uboczny: kod szablonów jest jawny dla każdego użytkownika naszej biblioteki (pliki nagłówkowe są dostarczane zawsze w postaci źródłowej) ☹