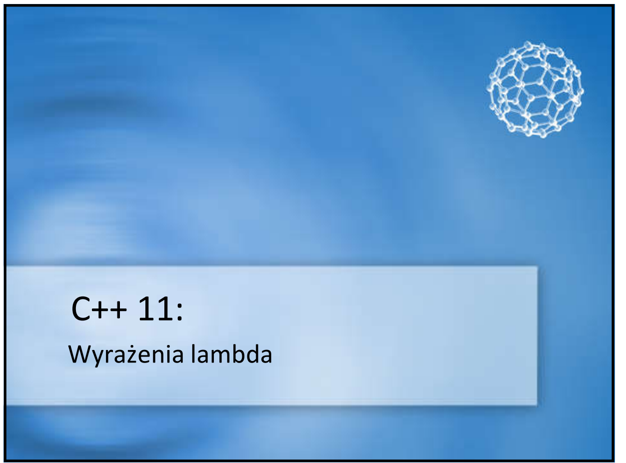


87

C++11: wyrażenia lambda

Obiekty wywoływalne (*callables*):

W C++98 predykaty logiczne i funktory arytmetyczne, przekazywane jako jeden z argumentów przy wywołaniu algorytmu, można było tworzyć na podstawie:

1. funkcji (lub wskaźników do funkcji) o ustalonej, pasującej do potrzeb algorytmu liczbie argumentów,
2. obiektów klas wywoływalnych (z przeciążonym operatorem wywołania obiektu).

W C++11 dochodzą jeszcze **wyrażenia lambda**:

- Wywoływalny kawałek kodu (*inline*, bez własnej nazwy, nie jest to funkcja w rozumieniu takim jak w ANSI C)
- Mają zdefiniowany typ zwracanej wartości, argumenty wywołania i kod wykonywalny, można je definiować wewnątrz innych funkcji.

© UKSW, WMP. SNS, Warszawa 88

88

C++11: wyrażenia lambda

Wyrażenie lambda – budowa:

```
[capture list] (parameter list) -> return type  {
  function body
}
```

gdzie:

`capture list` – lista lokalnych zmiennych zadeklarowanych w otoczeniu

`parameter list` – lista parametrów wywołania

`return type` – typ zwracanych danych

`function body` – wykonywalny kod funkcji

Uwaga: typ zwracanych danych jest deklarowany w miejscu tuż za nagłówkiem z wykorzystaniem strzałki -> (tj. wg notacji *trailing return type*);

© UKSW, WMP. SNS, Warszawa 89

89

C++11: wyrażenia lambda

Wyrażenie lambda – budowa:

```
[capture list] (parameter list) -> return type  {
  function body
}
```

Można pominąć listę parametrów oraz informację o typie zwracanych danych, ale *muszą* wystąpić: lista lokalnych zmiennych oraz kod wykonywalny, np.:

```
auto f = [] { return 42; };
```

`f` – obiekt wywoływalny, który nie przyjmuje żadnych argumentów i zwraca 42.

Wywołanie jest identyczne jak w przypadku funkcji:

```
cout << f() << endl; // wypisuje w oknie konsoli 42
```

Pominięcie informacji o typie zwracanych danych: typ zwracany to `void`, lub jeżeli kod zawiera tylko `return`, to typ jest dedukowany na tej podstawie.

© UKSW, WMP. SNS, Warszawa 90

90

C++11: wyrażenia lambda

Wyrażenie lambda z argumentami:

Przykład: porównanie długości dwóch napisów

```
[](const string &a, const string &b)
    { return a.size() < b.size();}
```

Argumentom nie można przypisywać wartości domyślnych.

Przykład wykorzystania –
wywołanie algorytmu sortującego słowa w kontenerze `words`, z wykorzystaniem predykatu zdefiniowanego w postaci lambdy :

```
stable_sort(words.begin(), words.end(),
                 [](const string &a, const string &b)
                   { return a.size() < b.size();});
```

© UKSW, WMP. SNS, Warszawa 91

91

C++11: wyrażenia lambda

Przechwytywane zmienne – lista:

[ ] lista przechwyconych wartości jest pusta, kod wyrażenia lambda używa tylko tego, co zostało przekazane w argumentach wywołania

[nazwa1, &nazwa2, ..] lista zmiennych lokalnych przechwyconych domyślnie przez wartość; jeżeli przed nazwą stoi symbol &, to następuje przechwycenie przez referencję

© UKSW, WMP. SNS, Warszawa 92

92

C++11: wyrażenia lambda

Wyrażenie lambda z listą lokalnych zmiennych przechwyconych przez wartość:

Przykład: sprawdzenie, czy długość jest większa od wartości przechowywanej w zmiennej **sz**:

```
[sz](const string &a)
      { return a.size() >= sz; };
```

W kodzie wyrażenia lambda można odwoływać się *tylko* do tych zmiennych dostępnych lokalnie w otoczeniu wyrażenia, które zostały zadeklarowane w liście lokalnych zmiennych.

Przykład wykorzystania – znalezienie w kontenerze **words** pierwszego słowa o długości równej lub większej niż **sz**:

```
auto wc = find_if(words.begin(), words.end(),
                  [sz](const string &a)
                      { return a.size() >= sz; });
```

© UKSW, WMP. SNS, Warszawa 93

93

C++11: wyrażenia lambda

Wyrażenie lambda i zmienne globalne:

Po znalezieniu pierwszego słowa o odpowiedniej długości, możemy następnie dodać kod, który wypisuje wszystkie słowa występujące po znalezionym słowie:

```
for_each(wc, words.end(),
              [](const string &s){cout << s << " ";});
  cout << endl;
```

Uwaga: mimo, że lista zmiennych lokalnych jest pusta, kod korzysta z **cout** (?)

1. Lista lokalnych zmiennych służy do przekazania do kodu wyrażenia wyłącznie zmiennych lokalnych *non-static.*
2. Zmienne zdefiniowane poza funkcją, gdzie wywołana została lambda, ale dostępne wewnątrz tej funkcji, są również dostępne w kodzie lambda.

© UKSW, WMP. SNS, Warszawa 94

94

C++11: wyrażenia lambda

Obiekt lambda:

1. Kiedy przekazujemy wyrażenie lambda do funkcji, tworzymy nowa klasę oraz obiekt tej klasy, którym będzie argument wywołania tej funkcji.
2. Odpowiednio, kiedy używamy **auto**, żeby zdefiniować zmienną zainicjalizowaną wyrażeniem lambda, zmienna ta jest obiektem nowo utworzonej klasy.
3. Klasa utworzona z wyrażenia lambda posiada listę pól odpowiadających przechwyconym zmiennym; są one inicjalizowane, kiedy obiekt jest tworzony.

Mimo, że w sieci znajdują się artykuły, w których stosuje się określenie „lambda function", właściwszym określeniem na lambdę jest „closure".

© UKSW, WMP. SNS, Warszawa 95

95

C++11: wyrażenia lambda

Obiekt lambda:

Przykład:

```
auto wc = find_if(words.begin(), words.end(),
               [sz](const string &a)
                   { return a.size() >= sz; });
```

wygeneruje anonimową klasę, która będzie wyglądała tak, jak klasa **SizeComp** poniżej:

```
class SizeComp {
    SizeComp(size_t n): sz(n) { } // każdy kolejny parametr
    // konstruktora odpowiada jednej przechwytywanej zmiennej;

    // operator wywołania, który zwraca daną takiego samego
    // typu, ma takie same parametry i kod jak wyrażenie lambda
    bool operator()(const string &a) const
        { return a.size() >= sz; }
private:
    size_t sz; // pola odpowiadają przechwytywanym zmiennym
};
```

© UKSW, WMP. SNS, Warszawa 96

96

C++11: wyrażenia lambda

Przechwytywanie zmiennych:

Zmienne lokalne przechwycone przez wyrażenie mogą zostać przechwycone w trybie *przez referencję* bądź *przez wartość.*

W przykładach dotychczas – przez wartość:

```
void fcn1() {
  size_t v1 = 42;  // zmienna lokalna
  auto f = [v1]{ return v1;}; // kopiowanie v1 do obiektu
  v1 = 0;
  auto j = f(); // j==42 ponieważ f przechowuje kopię v1
}
```

© UKSW, WMP. SNS, Warszawa 97

97

C++11: wyrażenia lambda

Zmienne przechwycone przez referencję:

Przykład:

```
void fcn2() {
  size_t v1 = 42;  // zmienna lokalna

  // obiekt f2 przechowuje referencję do v1
  auto f2 = [&v1] { return v1; };

  v1 = 0;
  auto j = f2(); // j==0 ponieważ f2 ma referencję do v1
}
```

Uwaga: kiedy przechwytujemy obiekt przez referencję, musimy być pewni, że będzie nadal istniał w momencie działania kodu wyrażenia lambda

© UKSW, WMP. SNS, Warszawa 98

98

C++11: wyrażenia lambda

Zmienne przechwycone przez referencję:

Przykład, kiedy przechwycenie przez referencję jest konieczne:

```
void funwords(vector<string> &words,
              vector<string>::size_type sz,
              ostream &os = cout, char c = ' ')
{
  // tutaj kod, który np. porządkuje słowa
  // wg określonego kryterium
  …
  // a teraz kod który je wypisuje do strumienia
  // podanego w argumencie wywołania funkcji 'funwords'
  for_each(words.begin(), words.end(),
    [&os, c](const string &s) { os << s << c; });
}
```


C++11: wyrażenia lambda

Zmienne przechwycone przez referencję - podsumowanie:

- Zmienne typów bazowych – `int`, `double` – najprościej przechwytywać w obiekcie lambda w trybie „przez wartość".
- Kiedy przechwytujemy w trybie „przez referencję", np. iteratory, lub wskaźniki, należy zapewnić istnienie wskazywanych struktur danych, kiedy obiekt lambda będzie wykonywany, oraz zapewnić, że zawartość będzie taka, jakiej oczekiwaliśmy.
- Funkcja, w której powstał obiekt lambda, może go na zakończenie zwrócić, ale – zmienne lokalne funkcji nie mogą być w nim przechowane przez referencję.

C++11: wyrażenia lambda

Zmienne przechwycone *implicite*:

[=] przechwycenie przez wartość wszystkich zmiennych lokalnych z otoczenia

[&] przechwycenie przez referencję wszystkich zmiennych lokalnych z otoczenia

[=, lista referencji] przechwycenie przez referencję zmiennych wymienionych w liście referencji, natomiast przez wartość wszystkich pozostałych zmiennych lokalnych z otoczenia

[&, lista zmiennych] przechwycenie przez wartość zmiennych wymienionych w liście zmiennych, natomiast przez referencję wszystkich pozostałych zmiennych lokalnych z otoczenia

C++11: wyrażenia lambda

Zmienne przechwycone *implicite*:

Przykład: sprawdzenie, czy długość jest większa od wartości przechowywanej w zmiennej `sz`:

```
wc = find_if(words.begin(), words.end(),
             [=](const string &s)
               { return s.size() >= sz; });
```


C++11: wyrażenia lambda

Zmienne przechwycone *implicite* i *explicite*:

Przykład na mieszane przechwytywanie *implicite* i *explicite* (domyślne i jawne):

```
char c = ' ';
for_each(words.begin(), words.end(),
     [=, &os](const string &s) { os << s << c; });
```

Dla mieszanego sposobu przechwytywania pierwszym elementem w liście przechwytywanych zmiennych musi być **&** lub **=**. Ten symbol ustala domyślny sposób przechwytywania, który będzie użyty wobec zmiennych niewymienionych po przecinku.

Zmienne wymienione po przecinku musza być przechwytywane w sposób przeciwny do domyślnego (reprezentują wyjątki od „reguły" podanej przed przecinkiem).

C++11: wyrażenia lambda

Modyfikacja przechwyconych zmiennych:

Domyślnie w kodzie lambdy nie wolno modyfikować zmiennych przechwyconych „przez wartość".

Jeżeli potrzebna jest modyfikacja, stosujemy słowo kluczowe `mutable`

Przykład:

```
size_t v1 = 42; // v1 - zmienna lokalna
auto f3 = [v1] () mutable { return ++v1; };
v1 = 0;
auto j = f3(); // j == 43
```

C++11: wyrażenia lambda

Modyfikacja przechwyconych zmiennych:

O tym, czy zmienna przechwycona sposobem „przez referencje" może być zmieniana, decyduje wyłącznie to, czy referencja dotyczy zmiennej typu **const**, czy nie.

Przykład:

```
size_t v1 = 42;  // zmienna lokalna, która nie jest const

// 'v1' jest referencją na zmienną lokalną nie-const
auto f2 = [&v1] { return ++v1; }; // modyfikuje 'v1'

v1 = 0;
auto j = f2(); // j == 1
```

© UKSW, WMP. SNS, Warszawa

C++11: wyrażenia lambda

Specyfikowanie typu zwracanych danych:

Kiedy w kodzie wyrażenia lambda występuje tylko instrukcja **return** – nie ma potrzeby specyfikowania typu zwracanych danych, np. :

```
transform(vi.begin(), vi.end(), vi.begin(),
    [](int i) { return i < 0 ? -i : i; });
```

Kiedy w kodzie wyrażenia występuje jakiekolwiek inne wyrażenie niż **return**, domyślnie wyrażenie lambda zwraca **void**.

```
transform(vi.begin(), vi.end(), vi.begin(),
    [](int i) { if (i<0) return -i; else return i; });
```

Pojawia się problem, bo miała być zwrócona wartość absolutna, a nie ma nic..

© UKSW, WMP. SNS, Warszawa

C++11: wyrażenia lambda

Specyfikowanie typu zwracanych danych:

Aby jawnie podać zwracany typ danych należy *typ zwracanych danych zadeklarować* w miejscu tuż za nagłówkiem z wykorzystaniem strzałki -> (tj. wg notacji *trailing return type*);

Przykład:

```
transform(vi.begin(), vi.end(), vi.begin(),
          [](int i) -> int
          { if (i < 0) return -i; else return i; });
```

© UKSW, WMP. SNS, Warszawa

C++ 11:

Szablony ze zmienną liczbą parametrów

C++11: szablony ze zmienną liczbą parametrów

Argumenty funkcji - rekurencja

```
template<typename T>
T adder(T v) {
  return v;
} // the "stop-recursion-case"

template<typename T, typename... Args>
T adder(T first, Args... args) {
  return first + adder(args...);
}
```

typename... Args – paczka parametrów szablonu (*a template parameter pack*)

Args... args – paczka argumentów funkcji (*a function parameter pack*)

```
long sum =
    adder(1, 2, 3, 8, 7);

string s1 = "xx", s2 = "aa",
       s3 = "bb", s4 = "yy";
string ssum =
    adder(s1, s2, s3, s4);
```

© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

Obliczanie wartości wyrażeń logicznych:

```
template<typename T>
bool pair_comparer(T a, T b) {
  return a == b;
} // the "stop-recursion-case"

template<typename T, typename... Args>
bool pair_comparer(T a, T b, Args... args) {
  return a == b && pair_comparer(args...);
}
…
pair_comparer(1.5, 1.5, 2, 2, 6, 6);
```

© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

Obliczanie wartości wyrażeń logicznych – w razie nieparzystej liczby argumentów..

```
template<typename T>
bool pair_comparer(T a) {
  return false;
} // the "stop-recursion-case"

…
pair_comparer(1.5, 1.5, 2, 2, 6, 6, 7);
```

© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

```
// Variadic template declaration
template<typename... Args> class Test{};

// Specialization 1
template<typename T>
class Test<T> {
public:
  T Data;
};

// Specialization 2
template<typename T1, typename T2>
class Test<T1, T2> {
public:
  T1 Left;
  T2 Right;
};
```

```
void Foo()
{
  Test<int> data;
  data.Data = 24;

  Test<int, int> twovalues;
  twovalues.Left = 12;
  twovalues.Right = 15;
}
```

© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

- W szablonach klas paczka parametrów może być zadeklarowana wyłącznie jako ostatnia w liście parametrów
- W szablonach funkcji może być zadeklarowana wcześniej, tj. mogą po niej wystąpić parametry pod warunkiem, że dla tych parametrów uda się wydedukować typy na podstawie argumentów wywołania, bądź gdy zostały zadeklarowane dla nich wartości domyślne

© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

Obliczanie liczby parametrów

```
template<typename... Args>
class Test {
public:
  size_t GetTCount() { return sizeof...(Args); }
};
Test<int> data;
size_t args = data.GetTCount(); // 1

Test<int, int, char*> data2;
args = data2.GetTCount(); // 3

Test<int, float> data3;
args = data3.GetTCount(); // 2
```

© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

```
template<typename... Args> class Test;

// the "stop-recursion-case"
template<> class Test<> {};

// Specialization for at least 1 argument
template<typename T1, typename... TRest>
class Test<T1, TRest...> : public Test<TRest...> {
public:
  T1 Data;

  // This will return the base type
  Test<TRest...>& Rest() {
    return *this;
  }
};
```

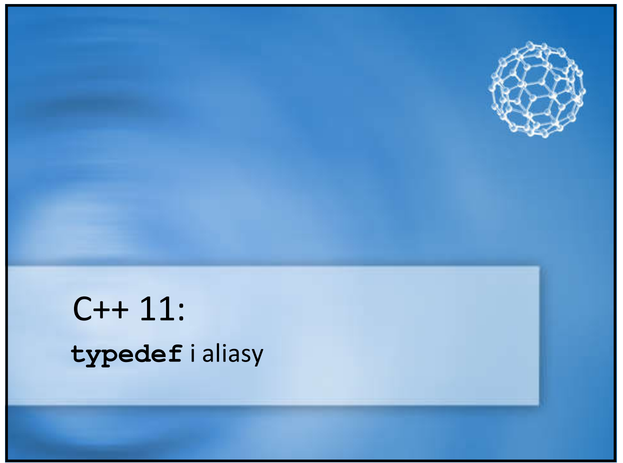
© UKSW, WMP. SNS, Warszawa

C++11: szablony ze zmienną liczbą parametrów

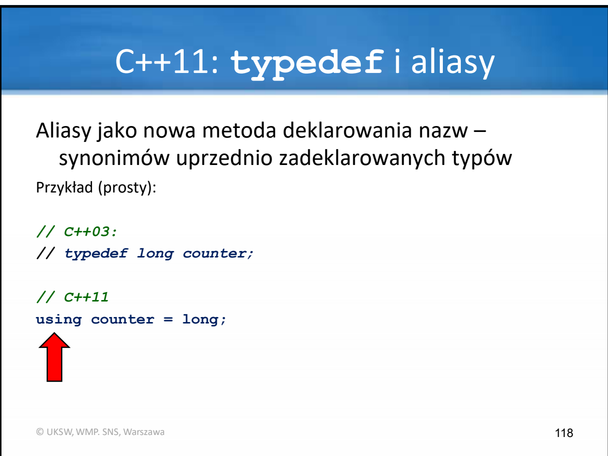
```
void Foo() {
  Test<int> data;
  data.Data = 24;

  Test<int, int> twovalues;
  twovalues.Data = 10;
  // Rest() returns Test<int>
  twovalues.Rest().Data = 11;

  Test<int, int, char*> threevalues;
  threevalues.Data = 1;
  // Rest() returns Test<int, int>
  threevalues.Rest().Data = 2;
  // Rest().Rest() returns Test<char*>
  threevalues.Rest().Rest().Data = "test data";
}
```

To jest przykład bardzo akademicki..

© UKSW, WMP. SNS, Warszawa



117



118

C++11: `typedef` i aliasy

Aliasy jako nowa metoda deklarowania nazw – synonimów uprzednio zadeklarowanych typów

Przykład drugi (też prosty):

```
// C++11
using fmtfl = std::ios_base::fmtflags;
// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;

fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) |
                   std::cout.showbase | std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

© UKSW, WMP. SNS, Warszawa

119

C++11: `typedef` i aliasy

Aliasy jako nowa metoda deklarowania nazw – synonimów uprzednio zadeklarowanych typów

Przykład trzeci – wskaźniki do funkcji (tak właściwie też prosty):

```
// C++11
using func = void(*)(int);

// C++03 equivalent:
// typedef void (*func)(int);

// func can be assigned to a function pointer value
void actual_function(int arg) { /* some code */ }
func fptr = &actual_function;
```

© UKSW, WMP. SNS, Warszawa

120

C++11: `typedef` i aliasy

Aliasy jako nowa metoda deklarowania nazw – synonimów uprzednio zadeklarowanych typów

Przykład czwarty – szablony aliasów:

```
template<typename T>
using ptr = T*;

// nazwa 'ptr<T>' jest teraz aliasem wskaźnika do T
ptr<int> ptr_int; // (!)
```

`typedef` nie działa z szablonami, więc nie istniał odpowiednik dla wcześniejszych wersji składni C++.

© UKSW, WMP. SNS, Warszawa

121

C++11: `typedef` i aliasy

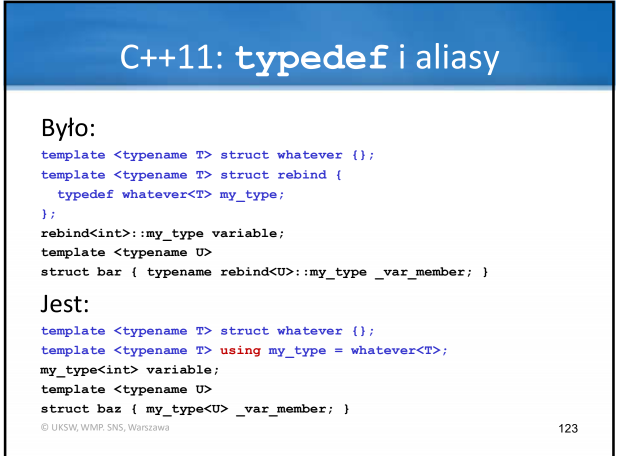
Aliasy jako nowa metoda deklarowania nazw – synonimów uprzednio zadeklarowanych typów

Przykład czwarty – szablony aliasów:

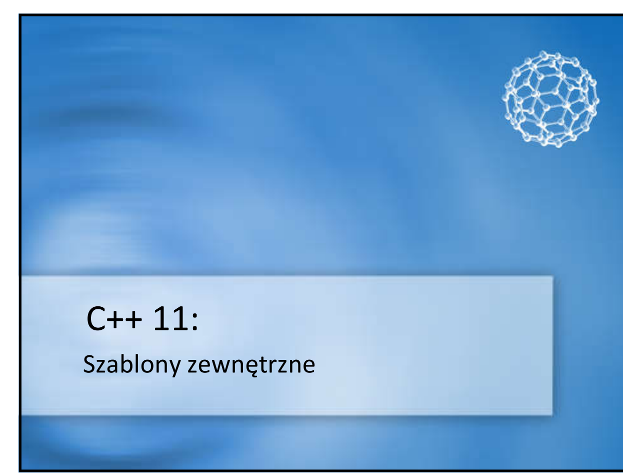
```
template<typename T>
using ref = T&;
template<typename T>
void f(ref<T> x) {
  x = 10;
}
int main() {
  int a;
  f(a);
  return a; /* 10 */
}
```

© UKSW, WMP. SNS, Warszawa

122



123



124

C++11: szablony zewnętrzne

Zdarza się, że szablon np. funkcji jest wykorzystywany w wielu plikach kodu źródłowego. Dodatkowo przyjmijmy, że w każdym z plików będzie potrzebna konkretyzacja tego szablonu dla tego samego typu (listy typów).

Wtedy kompilator, tworząc postaci pośrednie kodu, będzie kompilował do każdego z plików `obj` identyczną wersję skonkretyzowanej funkcji.

Jednak na etapie linkowania zostawiona będzie tylko jedna kopia skonkretyzowanej funkcji, a wszystkie pozostałe zostaną odrzucone.

Jeżeli funkcja jest duża i złożona, warto pomyśleć o redukcji czasu kompilacji i rozmiarów plików `obj` .

© UKSW, WMP. SNS, Warszawa
125

125

C++11: szablony zewnętrzne

Możliwe jest zmuszenie kompilatora do pominięcia konkretyzacji szablonu funkcji, jeżeli mamy pewność, że w innym pliku na pewno zostanie ona skonkretyzowana. Przykład:

```
// header.h
template<typename T>
void
  ReallyBigFunction()
{
   // Body
}
```

```
// source1.cpp
#include "header.h"
void something1() {
    ReallyBigFunction<int>();
}
```

```
// source2.cpp
#include "header.h"
void something2() {
    ReallyBigFunction<int>();
}
```

© UKSW, WMP. SNS, Warszawa
126

126

C++11: szablony zewnętrzne

Po kompilacji takiego projektu powstaną pliki:

```
source1.obj
 void something1()
 void ReallyBigFunction<int>() // kompilowane po raz pierwszy

source2.obj
 void something2()
 void ReallyBigFunction<int>() // kompilowane po raz drugi
```

Na etapie linkowania jedna `void ReallyBigFunction<int>()` zostanie odrzucona.

© UKSW, WMP. SNS, Warszawa
127

127

C++11: szablony zewnętrzne

Rozwiązanie:

```
// source2.cpp

#include "header.h"
extern template ReallyBigFunction<int>();

void something2()
{
    ReallyBigFunction<int>();
}
```

© UKSW, WMP. SNS, Warszawa
128

128