Zaawansowane CPP/Wykład 1: Szablony I

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Szablony funkcji

Funkcje uogólnione

W praktyce programowania często spotykamy się z funkcjami (algorytmami), które można zastosować do szerokiej klasy typów i struktur danych. Typowym przykładem jest funkcja obliczająca maksimum dwu wartości. Ten trywialny, aczkolwiek przydatny kod można zapisać np. w postaci:

```
int max(int a,int b) {
  return (a>b)?a:b;
};
```

PRZYKŁAD 1.1

Funkcja `max` wybiera większy z dwu `int`-ów, ale widać, że kod będzie identyczny dla argumentów dowolnego innego typu pod warunkiem, iż istnieje dla niego operator porównania i konstruktor kopiujący. W językach programowania z silną kontrolą typów, takich jak C, C++ czy Java definiując funkcję musimy jednak podać typy przekazywanych parametrów oraz typ wartości zwracanej. Oznacza to, że dla każdego typu argumentów musimy definiować nową funkcję `max`:

```
int    max(int a, int b)      {return (a>b)?a:b;};
double max(double a,double b) {return (a>b)?a:b;};
string max(string a,string b) {return (a>b)?a:b;};
skorzystaliśmy tu z dostępnej w C++ możliwości przeładowywania funkcji
```

```
main() {
  cout<< max(7,5)<<end;
  cout<< max(3.1415,2.71)<<endl;
  cout<< max("Ania","Basia")<<endl;
}
```

PRZYKŁAD 1.2

Takie powtarzanie kodu, poza oczywistym zwiększeniem nakładu pracy, ma inne niepożądane efekty, związane z trudnością zapewnienia synchronizacji kodu każdej z funkcji. Jeśli np. zauważymy błąd w kodzie, to musimy go poprawić w kilku miejscach. To samo dotyczy optymalizacji kodu. W powyższym przykładzie kod jest wyjątkowo prosty, ale taki sam problem dotyczy np. funkcji sortujących. Rozważmy prosty algorytm sortowania bąbelkowego:

```
inline void swap(double &a,double &b) {
double   tmp=a;a=b;b=tmp;
}
void buble_sort(double *data,int N) {
for(int i=n-1;i>0;--i)
        for(int j=0; j < i;++j)
                if(data[j]>data[j+1])
                        swap(data[j],data[j+1]);
}
```

Powyższa funkcja sortuje tablicę zawierającą wartości typu `double`, ale widać, że znów kod będzie identyczny, jeśli zamiast `double` użyjemy dowolnego innego typu, którego wartości możemy porównywać za pomocą funkcji `operator>()` i dla którego zdefiniowany jest operator przypisania. Co więcej, kod nie zmieni się jeśli zamiast tablicy użyjemy dowolnej innej struktury danych, umożliwiającej indeksowany dostęp do swoich składowych, np. `std::vector` ze Standardowej Biblioteki Szablonów STL. W tym przypadku kod jest już bardziej skomplikowany i kłopoty związane z jego powielaniem będą większe. Przykłady takie można mnożyć, istnieje bowiem wiele takich funkcji czy *algorytmów uogólnionych*. Ich kod może być znacznie bardziej skomplikowany niż w podanych przykładach, a zależność od typu argumentów nie musi ograniczać się do sygnatury, ale występować również we wnętrzu funkcji, jak np. w przypadku zmiennej `tmp` w funkcji `swap`. Powielanie takiego kodu dla różnych typów parametrów może łatwo prowadzić do błędów, utrudnia ich wykrywanie, a konieczność edycji każdego egzemplarza kodu zniechęca do wprowadzania ulepszeń.

Funkcje uogólnione bez szablonów

Jak radzili, a właściwie jak radzą sobie programiści bez możliwości skorzystania z szablonów? Tradycyjne sposoby rozwiązywania tego typu problemów to między innymi makra:

```
#define max(a,b) ( (a>b)?a:b) )
```

lub używanie wskaźników typów ogólnych, takich jak `void *`, jak np. w funkcji `qsort` ze standardowej biblioteki C:

```
void qsort(void *base, size_t nmemb, size_t size,
           int(*compar)(const void *, const void *));
```

Mimo iż użyteczne, żadne z tych rozwiązań nie może zostać uznane za wystarczająco ogólne i bezpieczne.

Można się również pokusić o próbę rozwiązania tego problemu za pomocą mechanizmów programowania obiektowego. W sumie jest to bardziej wyrafinowana odmiana rzutowania na `void *`. Polega na zdefiniowaniu ogólnego typu dla obiektów, które mogą być porównywane:

```
class GreaterThenComparable {
public:
```

```
  virtual bool operator>(const GreaterThenComparable &) const = 0;
};
```

następnie zdefiniowaniu funkcji `max` w postaci:

```
const GreaterThenComparable &
max(const GreaterThenComparable &a,
    const GreaterThenComparable &b) {
      return (a>b)? a:b;
    }
```

(Źródło: max_oop.cpp)

i używaniu jej np. w następujący sposób:

```
class Int:public GreaterThenComparable {
  int val;
public: Int(int i = 0):val(i) {};
  operator int() {return val;};
  virtual bool
  operator>(const GreaterThenComparable &b) const {
    return val > static_cast<const Int&>(b).val;
  }
};

main() {
  Int a(1),b(2);
  Int c;
  c = (const Int &)::max(a,b);
  cout<<(int)c<<endl;
}
```

(Źródło: max_oop.cpp)

Widać więc wyraźnie, że to podejście wymaga sporego nakładu pracy, a więc w szczególności w przypadku tak prostej funkcji jak `max` jest wysoce niepraktyczne. Ogólnie rzecz biorąc ma ono następujące wady:

1. Wymaga dziedziczenia z abstrakcyjnej klasy bazowej `GreaterThenComparable`, czyli może być zastosowane tylko do typów zdefiniowanych przez nas. Inne typy, w tym typy wbudowane, wymagają kopertowania w klasie opakowującej, takiej jak klasa `Int` w powyższym przykładzie.
2. Ponieważ potrzebujemy polimorfizmu funkcja `operator>()` musi być funkcją wirtualną, a więc musi być funkcją składową klasy i nie może być typu `inline`. W przypadku tak prostych funkcji niemożność rozwinięcia ich w miejscu wywołania może prowadzić do dużych narzutów w czasie wykonania.
3. Funkcja `max` zwraca zawsze referencje do `GreaterThenComparable`, więc konieczne jest rzutowanie na typ wynikowy (tu `Int`).

Szablony funkcji

Widać, że podejście obiektowe nie nadaje się najlepiej do rozwiązywania tego szczególnego problemu powielania kodu. Dlatego w C++ wprowadzono nowy mechanizm: szablony. Szablony zezwalają na definiowanie całych rodzin funkcji, które następnie mogą być używane dla różnych typów argumentów.

Definicja szablonu funkcji `max`, odpowiadającej definicji 1.1 wygląda następująco:

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

(Źródło: max_template.cpp)

Przyjrzyjmy się jej z bliska. Wyrażenie `template<typename T>` oznacza, że mamy do czynienia z szablonem, który posiada jeden parametr formalny nazwany `T`. Słowo kluczowe `typename` oznacza, że

parametr ten jest typem (nazwą typu). Zamiast słowa `typename` możmy użyć słowa kluczowego `class`. Nazwa tego parametru może być następnie wykorzystywana w definicji funkcji w miejscach, gdzie spodziewamy się nazwy typu. I tak powyższe wyrażenie definiuje funkcję `max`, która przyjmuje dwa argumenty typu `T` i zwraca wartość typu `T`, będącą wartością większego z dwu argumentów. Typ `T` jest na razie niewyspecyfikowany. W tym sensie szablon definiuje całą rodzinę funkcji. Konkretną funkcję z tej rodziny wybieramy poprzez podstawienie za formalny parametr `T` konkretnego typu będącego argumentem szablonu. Takie podstawienie nazywamy konkretyzacją szablonu. Argument szablonu umieszczamy w nawiasach ostrych za nazwą szablonu (w praktyce można uniknąć konieczności jawnej specyfikacji argumentów szablonu, opiszemy to w następnych częściach wykładu):

```
int i,j,k;
k=max<int>(i,j);
```

Takie użycie szablonu spowoduje wygenerowanie identycznej funkcji jak 1.1. W powyższym przypadku za `T` podstawiamy `int`. Oczywiście możemy podstawić za `T` dowolny typ i używając szablonów program 1.2 można zapisać następująco:

```
template<typename T> T max(T a,T b) {return (a>b)?a:b;}
main() {
  cout<<::max<int>(7,5)<<endl;
  cout<<::max<double>(3.1415,2.71)<<endl;
  cout<<::max<string>("Ania","Basia")<<endl;
}
```

(Źródło: max_template.cpp)

W powyższym kodzie użyliśmy konstrukcji `::max(a,b)`. Dwa dwukropki oznaczają, że używamy funkcji `max` zdefiniowanej w ogólnej przestrzeni nazw. Jest to konieczne aby kod się skompilował, ponieważ szablon `max` istnieje już w standardowej przestrzeni nazw `std`. W dalszej części wykładu będziemy te podwójne dwukropki pomijać.

Oczywiście istnieją typy których podstawienie spowoduje błędy kompilacji, np.

```
complex<double> c1,c2;
max<complex<double> >(c1,c2); //brak operatora >
```

(Źródło: max_template.cpp)

lub

```
class X {
private:
  X(const X &){};
};
X a,b;
max<X>(a,b); //prywatny (niewidoczny) konstruktor kopiujący
```

(Źródło: max_template.cpp)

Ogólnie rzecz biorąc, każdy szablon definiuje pewną klasę typów, które mogą zostać podstawione jako jego argumenty.

Dedukcja argumentów szablonu

Użyteczność szablonów funkcji zwiększa istotnie fakt, że argumenty szablonu nie muszą być podawane jawnie. Kompilator może je wydedukować z argumentów funkcji. Tak więc zamiast

```
int i,j,k;
k=max<int>(i,j);
```

możemy napisać

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

i kompilator zauważy, że tylko podstawienie `int`-a za `T` umożliwi dopasowanie sygnatury funkcji do parametrów jej wywołania i automatycznie dokona odpowiedniej konkretyzacji.

Może się zdarzyć, że podamy takie argumenty funkcji, że dopasowanie argumentów wzorca będzie niemożliwe, otrzymamy wtedy błąd kompilacji. Trzeba pamiętać, że mechanizm automatycznego dopasowywania argumentów szablonu powoduje wyłączenie automatycznej konwersji argumentów funkcji. Podanie jawnie argumentów szablonu (w nawiasach ostrych za nazwą szablonu) jednoznacznie określa sygnaturę funkcji, a więc umożliwia automatyczną konwersję typów. Ilustruje to poniższy kod:

```
template<typename T> T max(T a,T b) {return (a>b)?a:b;}
main() {
  cout<<::max(3.14,2)<<endl;
  // błąd: kompilator nie jest w stanie wydedukowac argumentu szablonu, bo typy
  // argumentów (double,int) nie pasują  do (T,T)

  cout<<::max<int>(3.14,2)<<endl;
  // podając argument jawnie wymuszamy sygnaturę int max(int,int), a co za tym
  // idzie automatyczną konwersję argumentu 1 do int-a

  cout<<::max<double>(3.14,2)<<endl;
  // podając argument szablonu jawnie wymuszamy sygnaturę
  // double max(double,double)
  // a co za tym idzie automatyczną konwersję argumentu 2 do double-a

  int i;
  cout<<::max<int *>(&i,i)<<endl;
  //błąd: nie istnieje konwersja z typu int na int*
```

(Źródło: max_template.cpp)

Może warto zauważyć, że automatyczna dedukcja parametrów szablonu jest możliwa tylko wtedy, jeśli parametry wywołania funkcji w jakiś sposób zależą od parametrów szablonu. Jeśli tej zależności nie ma, z przyczyn oczywistych dedukcja nie jest możliwa i trzeba parametry podawać jawnie. Wtedy istotna jest kolejność parametrów na liście. Jeżeli parametry, których nie da się wydedukować, umieścimy jako pierwsze, wystarczy, że tylko je podamy jawnie, a kompilator wydedukuje resztę. Ilustruje to poniższy kod:

```
template<typename T,typename U> T convert(U u) {
return (T)u;
};
template<typename U,typename T> T inv_convert(U u) {
return (T)u;
};
fukcje różnią się tylko kolejnością parametrów szablonu

main() {
cout<<convert(33)<<endl;
błąd: kompilator nie jest w stanie wydedukować pierwszego parametru
szablonu,  bo  nie zależy on od parametru wywołania funkcji

cout<<convert<char>(33)<<endl;
w porządku: podajemy jawnie argument T, kompilator sam dedukuje
argument U z typu argumentu wywołania funkcji

cout<<inv_convert<char>('a')<<endl;
```

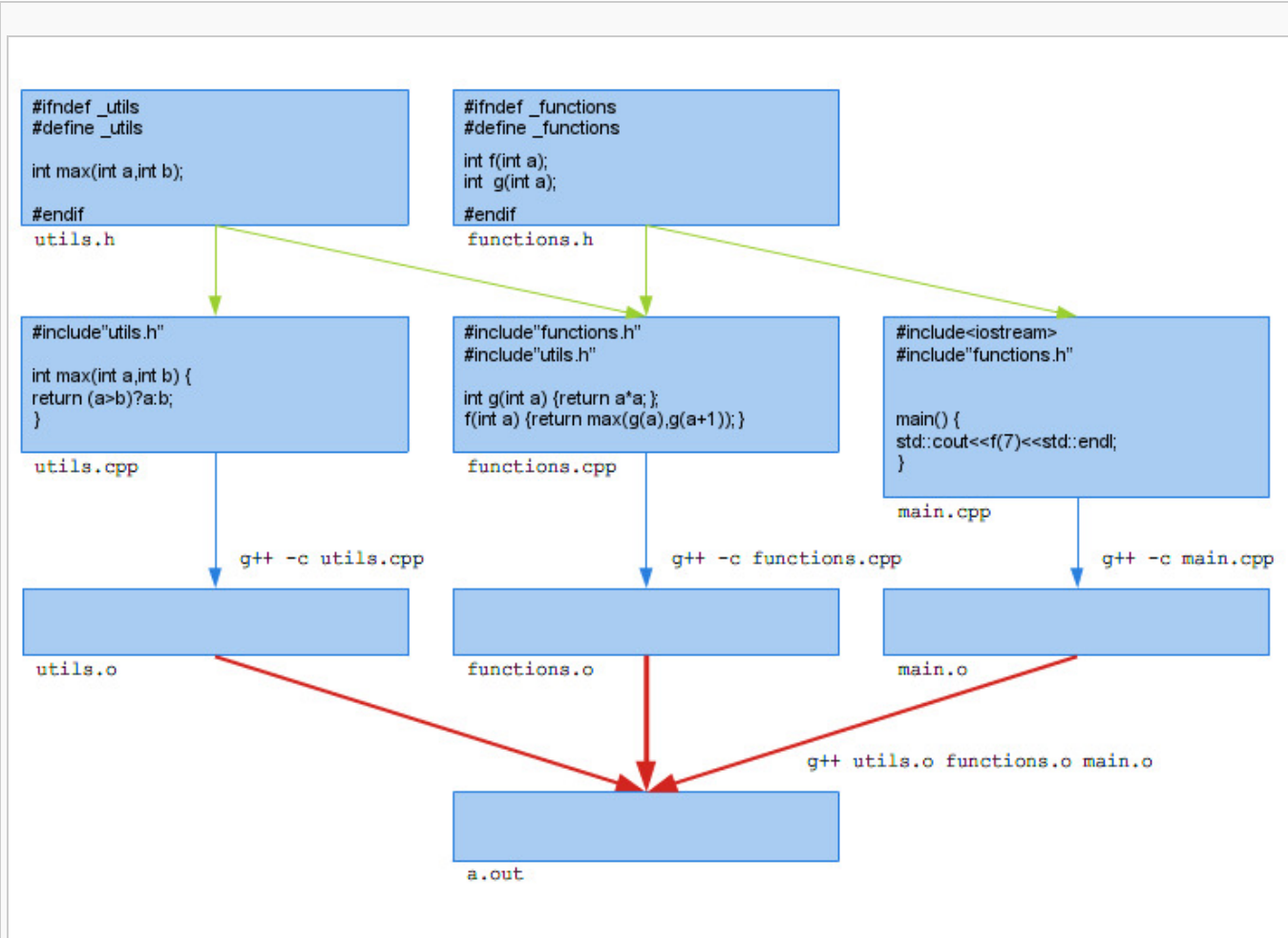
```
błąd: podajemy jawnie argument odpowiadający parametrowi U.
Kompilator nie jest w stanie wydedukować argumentu T, bo nie zależy on od argumentu
wywołania funkcji

cout<<inv_convert<int,char>(33)<<endl;
w porządku: podajemy jawnie oba argumenty szablonu
}
```

(Źródło: convert.cpp)

Używanie szablonów

Z użyciem szablonów wiąże się parę zagadnień niewidocznych w prostych przykładach. W językach C i C++ zwykle rozdzielamy deklarację funkcji od jej definicji i zwyczajowo umieszczamy deklarację w plikach nagłówkowych `*.h`, a definicję w plikach źródłowych `*.c`, `*.cpp` itp. Pliki nagłówkowe są w czasie kompilacji włączane do plików, w których chcemy korzystać z danej funkcji, a pliki źródłowe są pojedynczo kompilowane do plików "obiektowych" `*.o`. Następnie pliki obiektowe są łączone w jeden plik wynikowy (zob. rysunek 1.1). W pliku korzystającym z danej funkcji nie musimy więc znać jej definicji, a tylko deklarację. Na podstawie nazwy funkcji konsolidator powiąże wywołanie funkcji z jej implementacją znajdującą się w innym pliku obiektowym. W ten sposób tylko zmiana deklaracji funkcji wymaga rekompilacji plików, w których z niej korzystamy, a zmiana definicji wymaga jedynie rekompilacji pliku, w którym dana funkcja jest zdefiniowana.



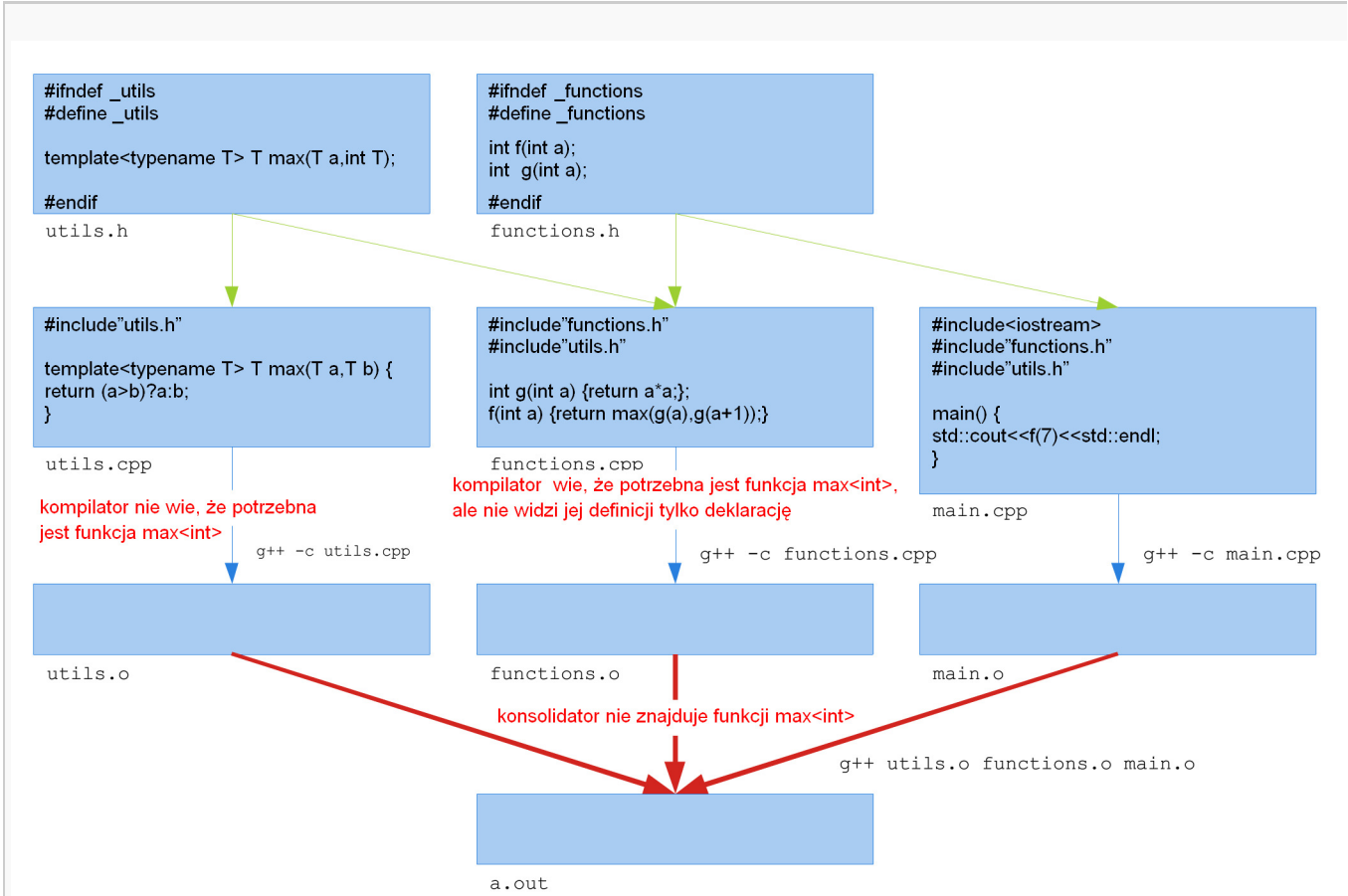
Rysunek 1.1. Przykład organizacji kodu C++ w przypadku użycia zwykłych funkcji.

Taka organizacja umożliwia przestrzeganie "reguły jednej definicji" (one definition rule), wymaganej przez C++. Osobom nieobeznanym z programowaniem w C/C++ zwracam uwagę na konstrukcję

```
#ifndef _nazwa_pliku_
#define _nazwa_pliku_
...
#endif
```

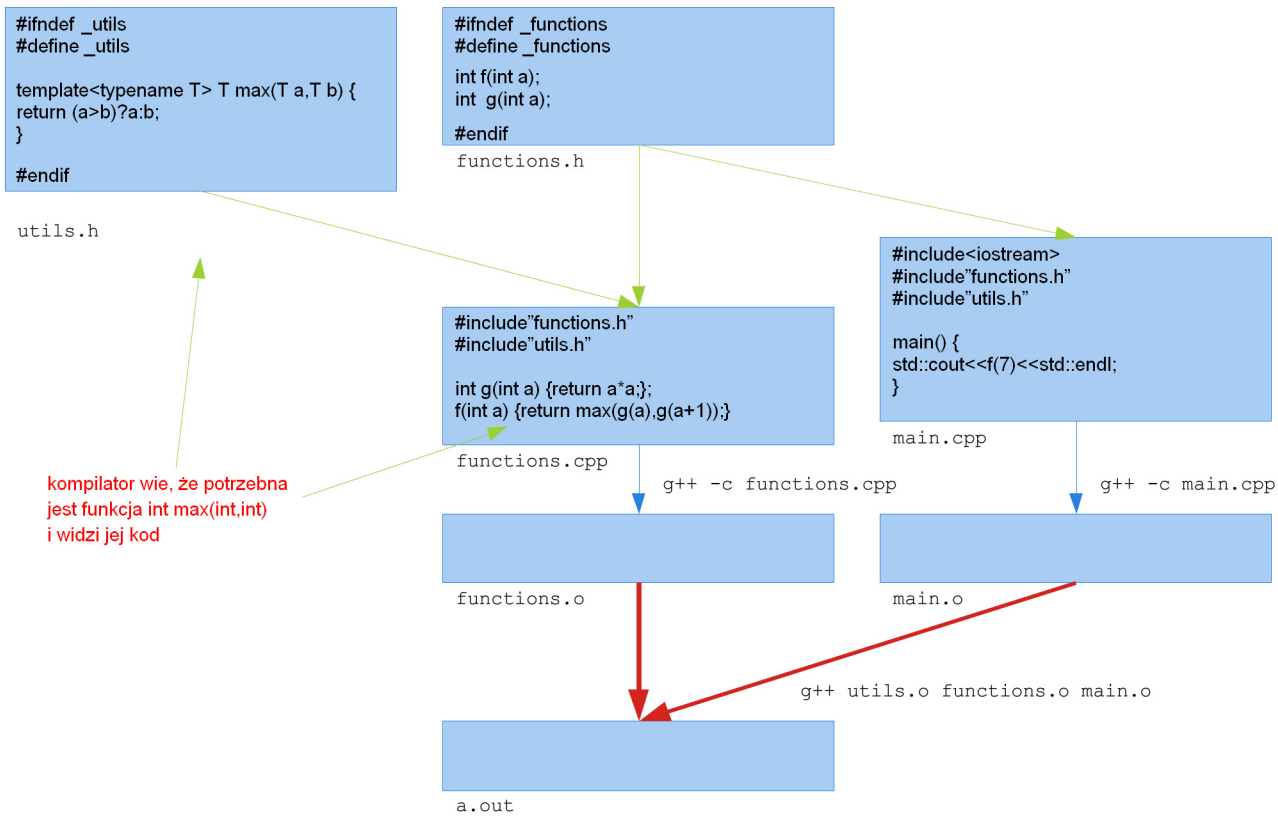
uniemożliwiającą podwójne włączenie tego pliku do jednej jednostki translacyjnej.

Podobne podejście do kompilacje szablonów się nie powiedzie (zob. rysunek 1.2). Powodem jest fakt, że w trakcie kompilacji pliku `utils.cpp` kompilator nie wie jeszcze, że potrzebna będzie funkcja `max<int>`, wobec czego nie generuje kodu żadnej funkcji, a jedynie sprawdza poprawność gramatyczną szablonu. Z kolei podczas kompilacji pliku `main.cpp` kompilator już wie, że ma skonkretyzować szablon dla `T = int`, ale nie ma dostępu do kodu szablonu.



Rysunek 1.2. Przykład błędnej organizacji kodu w przypadku użycia szablonów.

Istnieją różne rozwiązania tego problemu. Najprościej chyba jest zauważyć, że opisane zachowanie jest analogiczne do zachowania podczas kompilacji funkcji rozwijanych w miejscu wywołania (`inline`), których definicja również musi być dostępna w czasie kompilacji. Podobnie więc jak w tym przypadku możemy zamieścić wszystkie deklaracje i definicje szablonów w pliknu nagłówkowym, włączanym do plików, w ktorych z tych szablonów korzystamy (zob. rysunek 1.3). Podobnie jak w przypadku funkcji `inline` reguła jednej definicji zezwala na powtarzanie definicji/deklaracji szablonów w różnych jednostkach translacyjnych, pod warunkiem, że są one identyczne. Stąd konieczność umieszczania ich w plikach nagłówkowych.



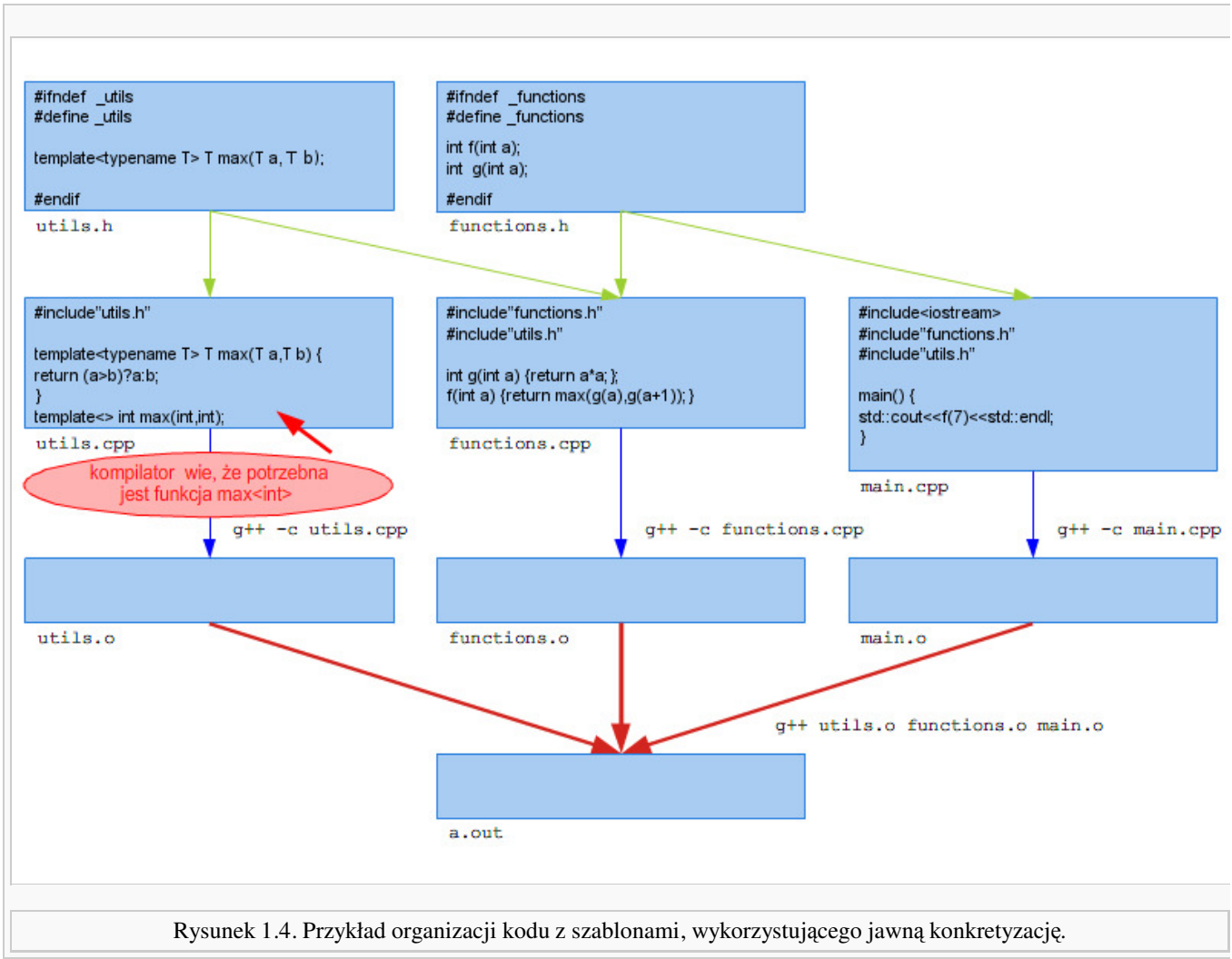
Rysunek 1.3. Przykład organizacji kodu z szablonami, wykorzystującego strategię włączania.

Ten sposób organizacji pracy z szablonami, nazywany modelem włączenia, jest najbardziej uniwersalny. Jego główną wadą jestkonieczność rekompilacji całego kodu korzystającego z szablonów przy każdej zmianie definicji szablonu. Również jeśli zmienimy coś w pliku, w którym korzystamy z szablonu, to musimy rekompilować cały kod szablonu włączony do tego pliku, nawet jeśli nie uległ on zmianie. Jeśli się uwzględni fakt, że kompilacja szablonu jest bardziej skomplikowana od kompilacji "zwykłego" kodu, to duży projekt intensywnie korzystający z szablonów może wymagać bardzo długich czasów kompilacji.

Możemy też w jakiś sposób dać znać kompilatorowi, że podczas kompilacji pliku `utils.cpp` powinien wygenerować kod dla funkcji `max<int>`. Można to zrobić dodając jawne żądanie konkretyzacji szablonu (zob. rysunek 1.4):

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

Używając konkretyzacji jawnej musimy pamiętać o dokonaniu konkretyzacji każdej używanej funkcji, tak że to podejście nie skaluje się zbyt dobrze. Ponadto w przypadku szablonów klas (omawianych w następnym module) konkretyzacja jawna pociąga za sobą konkretyzację wszystkich metod danej klasy, a konkretyzacja “na żądanie” - jedynie tych używanych w programie.



Rysunek 1.4. Przykład organizacji kodu z szablonami, wykorzystującego jawną konkretyzację.

Pozatypowe parametry szablonów

Poza parametrami określającymi typ, takimi jak parametr `T` w dotychczasowych przykładach, szablony funkcji mogą przyjmować również parametry innego rodzaju. Obecnie mogą to być inne szablony, co omówię w następnym podrozdziale lub parametry określające nie typ, ale wartości. Jak na razie (w obecnym standardzie) te wartości nie mogą być dowolne, ale muszą mieć jeden z poniższych typów:

1. typ całkowitoliczbowy bądź typ wyliczeniowy
2. typ wskaźnikowy
3. typ referencyjny.

Takie parametry określające wartość nazywamy parametrami pozatypowymi. W praktyce z parametrów pozatypowych najczęściej używa się parametrów typu całkowitoliczbowego. Np.

```
template<size_t N,typename T> T dot_product(T *a,T *b) {
        T total=0.0;
        for(size_t i=0;i<N;++i)
                total += a[i]*b[i] ;

return total;
};
```

(Źródło: dot_product.cpp)

Po raz drugi zwracam uwagę na kolejność parametrów szablonu na liście parametrów. Dzięki temu, że niededukowalny parametr `N` jest na pierwszym miejscu wystarczy podać jawnie tylko jego, drugi parametr typu `T` zostanie sam automatycznie wydedukowany na podstawie przekazanych argumentów wywołania funkcji:

```
main() {
double x[3],y[3];
dot_product<3>(x,y);
}
```

(Źródło: dot_product.cpp)

Parametry pozatypowe są zresztą "ciężko dedukowalne". Właściwie jedynym sposobem na przekazania wartości stałej poprzez typ argumentu wywołania jest skorzystanie z parametrów będących szablonami klas (zob. następny podrozdział).

Używając pozatypowych parametrów szablonów musimy pamiętać, że odpowiadające im argumenty muszą być stałymi wyrażeniami czasu kompilacji. Stąd jeżeli używamy typów wskaźnikowych, muszą to być wskaźniki do obiektów łączonych zewnętrznie, a nie lokalnych. Ponieważ jednak jeszcze ani razu nie używałem pozatypowych parametrów szablonów innych niż typy całkowite, to nie będę podawał żadnych przykładów takich paremtrów na tym wykładzie.

Szablony parametrów szablonu

Jak już wspomniałem w poprzednim podrozdziale, parametrami szablonu funkcji mogą być również szablony klas (zob. następny podrozdział). Szablony parametrów szablonu umożliwiają przekazanie nazwy szablonu jako argumentu szablonu funkcji. Więcej o nich napiszę w drugiej części wykładu. Tutaj tylko pokażę jako ciekawostkę w jaki sposób można dedukować wartości pozatypowych argumentów szablonu.

```
template< template<int N> class  C,int K>
taka definicja oznacza, że parametr C określa szablon klasy
posiadający jeden parametr typu int. Parametr N służy tylko
do definicji szablonu C i nie może być użyty nigdzie indziej
void f(C<K>){
  cout<<K<<endl;
};

template<int N> struct SomeClass {};

main() {
  SomeClass<1>  c1;
  SomeClass<2>  c2;

  f(c1); C=SomeClass K=1
  f(c2); C=SomeClass K=2
}
```

(Źródło: deduce_N.cpp)

Szablony metod

Jak na razie definiowaliśmy szablony zwykłych funkcji. C++ umożliwia również definiowanie szablonów metod klasy np.:

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

```
  m.max(1,2);
}
```

(Źródło: max_method.cpp)

Szablonów metod składowych dotyczą takie same reguły jak szablonów funkcji.

Szablony klas

Typy uogólnione

Uwagi na początku poprzedniego rozdziału odnoszą się w tej samej mierze do klas, jak i do funkcji. I tutaj mamy do czynienia z kodem, który w niezmienionej postaci musimy powielać dla różnych typów. Sztandarowym przykładem takiego kodu są różnego rodzaju kontenery (pojemniki), czyli obiekty służące do przechowywania innych obiektów. Jest oczywiste, że kod kontenera jest w dużej mierze niezależny od typu obiektów w nim przechowywanych. Jako przykład weźmy sobie stos liczb całkowitych. Możliwa definicja klasy stos może wyglądać następująco, choć nie polecam jej jako wzoru do naśladowania w prawdziwych aplikacjach:

```
class Stack {
private:
  int rep[N];
  size_t top;
public:
  static const size_t N=100;
  Stack():_top(0) {};
  void push(int val) {_rep[_top++]=val;}
  int pop() {return rep[--top];}
  bool is_empty {return (top==0);}
}
```

Ewidentnie ten kod będzie identyczny dla stosu obiektów dowolnego innego typu, pod warunkiem, że typ ten posiada zdefiniowany `operator=()` i konstruktor kopiujący.

W celu zaimplementowania kontenerów bez pomocy szablonów możemy probować podobnych sztuczek jak te opisane w poprzednim rozdziale. W językach takich jak Java czy Smalltalk, które posiadają uniwersalną klasę `Object`, z której są dziedziczone wszystkie inne klasy, a nie posiadają (Java już posiada) szablonów, uniwersalne kontenery są implementowane właśnie poprzez rzutowanie na ten ogólny typ. W przypadku C++ nawet to rozwiązanie nie jest praktyczne, bo C++ nie posiada pojedynczej hierarchii klas.

Szablony klas

Rozwiązaniem są znów szablony, tym razem szablony klas. Podobnie jak w przypadku szablonów funkcji, szablon klasy definiuje nam w rzeczywistości całą rodzinę klas. Szablon klasy `Stack` możemy zapisać następująco:

```
template<typename T> class Stack {
public:
  static const size_t N=100;
private:
  T _rep[N];
  size_t _top;<br>
public:
  Stack():_top(0) {};
  void push(T val) {_rep[_top++]=val;}
  T pop() {return _rep[--_top];}
  bool is_empty {return (_top==0);}
 };
```

(Źródło: stack.cpp)

Tak zdefiniowanego szablonu możemy używać podając jawnie jego argumenty.

```
Stack<string> st ;
st.push("ania");
st.push("asia");
st.push("basia");
while(!st.is_empty() ){
  cout<<st.pop()<<endl;
}
```

(Źródło: stack.cpp)

Dla szablonów klas nie ma możliwości automatycznej dedukcji argumentów szablonu, ponieważ klasy nie posiadają argumentów wywołania, które mogłyby do tej dedukcji posłużyć. Jest natomiast możliwość podania argumentów domyślnych, np.

```
template<typename T = int> Stack {
  ...
}
```

(Źródło: stack.cpp)

Wtedy możemy korzystać ze stosu bez podawania argumentów szablonu i wyrażenie

```
Stack s;
```

będzie równoważne wyrażeniu:

```
Stack<int> s;
```

Dla domyślnych argmentów szablonów klas obowiązują te same reguły, co dla domyślnych argumentów wywołania funkcji.

Należy pamiętać, że każda konkretyzacja szablonu klasy dla różniących się zestawów argumentów jest osobną klasą:

```
Stack<int> si;
Stack<double> sd;
sd=si; //błąd: to są obiekty różnych klas a nie zdefiniowano przypisania
```

(Źródło: stack.cpp)

Okazuje się, że próba zdefiniowania operatora przypisania, który np. przypisywałby do siebie stosy różnych typów, nie jest łatwa, ponieważ dwa takie stosy nie widzą swoich reprezentacji.

Pozatypowe parametry szablonów klas

Zestaw możliwych parametrów szablonów klas jest taki sam jak dla szablonów funkcji. Podobnie najczęściej wykorzystywane są wyrażenia całkowitoliczbowe. W naszym przykładzie ze stosem możemy ich użyć do przekazania rozmiaru stosu:

```
template<typename T = int , size_t N = 100> class Stack {
private:
 T rep[N];
```

```
 size_t top;
public:
 Stack():_top(0) {};

 void push(T val) {_rep[_top++]=val;}
 T pop()          {return rep[--top];}
 bool is_empty    {return (top==0);}
}
```

(Źródło: stack_N.cpp)

Podkreślam jeszcze raz, że `Stack<int,100>` i `Stack<int,101>` to dwie różne klasy.

Szablony parametrów szablonu

Stos jest nie tyle strukturą danych, ile sposobem dostępu do nich. Stos realizuje regułę LIFO czyli Last In First Out. W tym sensie nie jest istotne w jaki sposób dane są na stosie przechowywane. Może to być tablica, jak w powyższych przykładach, ale może to być praktycznie dowolny inny kontener. Np. w Standardowej Bibliotece Szablonów C++ stos jest zaimplementowany jako adapter do któregoś z istniejących już kontenerów. Ponieważ kontenery STL są szablonami, szablon adaptera mógłby wyglądać następująco:

```
template<typename T,
         template<typename X > class Sequence=std::deque >
class Stack {
  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();}
};
```

Konkretyzując stos możemy wybrać kontener, w którym będą przechowywane jego elementy:

```
Stack<double,std::vector> sv;
```

Można zamiast szablonu użyć zwykłego parametru typu:

```
template<typename T,typename C > class stos {
  C rep;
  public:
    ...
}
```

(Źródło: stack_adapter.cpp)

i używać go w następujący sposób:

```
stos<double,std::vector<double> > sv;
```

W przypadku użycia szablonu jako parametru szablonu zapewniamy konsystencję pomiędzy typem `T` i kontenerem `C`, uniemożliwiając pomyłkę podstawienia niepasujących parametrów:

```
stos<double,std::vector<int> > sv; <i>błąd: niezgodność typow</i>
```

Uczciwość nakazuje jednak w tym miejscu stwierdzić, że właśnie takie rozwiązanie jest zastosowane w STL-u. Ma ono tę zaletę, że możemy adaptować na stos dowolny kontener, niekoniecznie będący szablonem.

Na koniec jeszcze jedna uwaga: szablony kontenerów z STL posiadają po dwa parametry typów, z tym, że drugi posiada wartość domyślną (standard dopuszcza dowolną ilość argumentów w implemetacji kontenerów STL jak długo będą one posiadały wartości domyślne). Autorzy D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"* ostrzegają, że w tej sytuacji kompilator może nie zaakceptować wyrażenia:

```
stos<double,std::vector> sv;
```

ponieważ ignoruje fakt istnienia wartości domyślnej dla drugiego parametru szablonu `std::vector`. Mamy wtedy niezgodność pomiędzy przekazanym argumentem szablonu

```
template<typename T>
std::vector<T,typename A = std::allocator<T> >;
```

oraz deklaracją paremetru `Sequence` jako:

```
template<typename X > class Sequence ;
```

która zakłada tylko jeden parametr szablonu. Można wtedy zmienić deklarację szablonu `stos` i podać domyślny argument dla szablony w liście parametrów:

```
template<typename T,template<typename X ,typename A =
std::allocator<X> > class C > class stos {...}
```

W praktyce używane przeze mnie kompilatory (g++ wersja >= 3.3) nie wymagały takiej konstrukcji. Przyznaję, że nie udało mi się doczytać czy jest to cecha kompilatora g++, czy nowego standardu C++ (autorzy D. Vandervoorde, N. Josuttis *"C++ Szablony, Vademecum profesjonalisty"* opierali się na poprzednim wydaniu standardu).

Konkretyzacja na żądanie

Jak już wspomniałem wcześniej, konkretyzacja szablonów może odbywać się "na żądanie". W takim przypadku kompilator będzie konkretyzował tylko funkcje napotkane w kodzie. I tak, jeśli np. nie użyjemy w naszym kodzie funckji `Stack<int>::pop()`, to nie zostanie ona wygenerowana. Można z tego skorzystać i konkretyzować klasy typami, które nie spełniają wszystkich ograniczeń nałożonych na parametry szablonu. Wszystko bedzię w porządku jak długo nie będziemy używać funkcji łamiących te ograniczenia. Np. załóżmy, że do szablonu `Stack` dodajemy możliwość jego sortowania (wiem, to nie jest zgodne z duchem programowania obiektowego, stos nie posiada operacji sortowania, puryści mogą zastąpić ten przykład kontenerem `list`):

```
template<typename T,int N> void Stack<T,N>::sort() {
bubble_sort(_rep,N);
};
```

Możemy teraz np. używać

```
Stack<std::complex<double> > sc;
sc.push( std::complex<double>(0,1));
sc.pop();
```

ale nie

```
sc.sort();
```

(Źródło: stack_sort.cpp)

Natomiast konkretyzacja jawna

```
template Stack<std::complex<double> >;
```

(Źródło: stack_sort.cpp)

nie powiedzie się, bo kompilator będzie się starał skonkretyzować wszystkie składowe klasy `Stack`, w tym metodę `sort()`.

Typy stowarzyszone

W klasach poza metodami i polami możemy definiować również typy, które będziemy nazywali stowarzyszonymi z daną klasą. Jest to szczególnie przydatne w przypadku szablonów. Rozważmy następujący przykład:

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

Możemy teraz używać tej definicji w innych szablonach

```
template<typename S> void f(S s) {
  typename S::value_type total;
    słowo typename jest wymagane, inaczej kompilator założy, że
    S::value_type odnosi się do statycznej składowej klasy
  while(!s.is_empty() ) {
    total+=s.pop();
  }
return total;
}
```

(Źródło: stack_N.cpp)

Bez takich możliwości musielibyśmy przekazać typ elementów stosu w osobnym argumencie. Mechanizm typów stowarzyszonych jest bardzo czesto używany w uogólnionym kodzie.