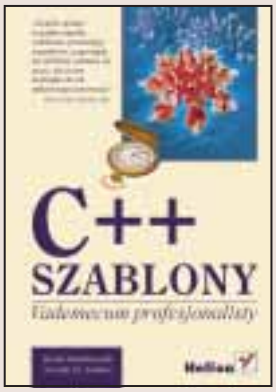
C++. Szablony. Vademecum profesjonalisty



Autorzy: David Vandevoorde, Nicolai M. Josuttis
Tłumaczenie: Przemysław Szeremiota
ISBN: 83-7361-175-4
Tytuł oryginału: C++ Templates The Complete Guide
Format: B5, stron: 474

Choć szablony są częścią definicji języka C++ od ponad dziesięciu lat, wciąż są źródłem nieporozumień, omyłek i kontrowersji. Z drugiej strony, ich popularność jako efektywnych instrumentów tworzenia bardziej eleganckiego, szybszego i „inteligentniejszego" oprogramowania stale rośnie. W rzeczy samej, szablony osiągnęły rangę kamieni węgielnych dla kilku nowych modeli programowania w języku C++.

Książka "C++. Szablony. Vademecum profesjonalisty." zawiera komplet informacji niezbędnych do rozpoczęcia korzystania z szablonów i pełnego wykorzystania ich możliwości, jak również wiedzy pomagającej doświadczonym programistom przekroczyć granicę, za którą programowanie z rzemiosła staje się sztuką. Autorzy założyli, że znasz język C++ i potrafisz programować z wykorzystaniem komponentów biblioteki standardowej. Prezentowane w książce przykłady znakomicie ilustrują abstrakcyjne pojęcia i demonstrują najlepsze praktyki programistyczne.

Poznasz:

- sposoby unikania pułapek towarzyszących stosowaniu szablonów,
- idiomy i techniki związane z szablonami – od technik najprostszych do takich, które poza tą książką nie zostały jeszcze nigdzie dokumentowane,
- sposoby wielokrotnego wykorzystywania tego samego kodu źródłowego bez zmniejszania jego wydajności i bezpieczeństwa,
- sposoby zwiększania efektywności programów w języku C++,
- sposoby tworzenia oprogramowania bardziej elastycznego i łatwiejszego w utrzymaniu.

Niektóre z technik przestawionych w „C++. Szablony. Vademecum profesjonalisty" nie doczekały się jeszcze opracowania w innych publikacjach. Jeśli w programowaniu w C++ chcesz osiągnąć najwyższy poziom, nie obędziesz się bez tej książki.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Rozdział 2.

Szablony funkcji

W rozdziale tym wprowadzimy pojęcie szablonu funkcji. Szablony funkcji to funkcje sparametryzowane tak, aby mogły reprezentować całe rodziny funkcji.

2.1. Szablony funkcji — wstęp

Szablony funkcji definiują kod, który może być realizowany dla różnych typów danych. Innymi słowy, szablon funkcji reprezentuje rodzinę funkcji. Reprezentacja ta przypomina zwykłą funkcję języka C++, pozbawioną tylko ścisłego określenia niektórych elementów. Elementy te są więc sparametryzowane. Najlepiej wytłumaczymy to na prostym przykładzie.

2.1.1. Definiowanie szablonu

Poniższy kod stanowi deklarację szablonu funkcji zwracającej większą z dwu przekazanych do niej wartości:

```
// podstawy/max.hpp

template <typename T>
inline T const& max (T const& a, T const& b)
{
    // jeżeli a < b zwróć b; w przeciwnym przypadku zwróć a
    return a < b ? b : a;
}
```

Szablon ten definiuje rodzinę funkcji, które zwracają większą z dwu przekazanych do nich wartości (przekazanie odbywa się za pośrednictwem parametrów wywołania funkcji: `a` i `b`). Typ parametrów funkcji nie został jawnie określony i występuje w szablonie jako *parametr szablonu* `T`. Przykład pokazuje, że parametry szablonów muszą zostać podane w ramach następującej konstrukcji składniowej:

```
template < lista-parametrów-oddzielanych-przecinkami >
```

W prezentowanym przykładzie lista parametrów zawiera wyrażenie `typename T`. Charakterystyczne jest umieszczenie listy parametrów szablonu pomiędzy znakami mniejszości (<) i większości (>) — utworzone tak nawiasy nazywamy nawiasami ostrymi. Słowo kluczowe `typename` wprowadza do listy parametrów tzw. *parametr typu* (ang. *type*

parameter). Jest to chyba najbardziej znany parametr szablonów języka C++ wykorzystywany często w programach, nie jest jednak jedynym dopuszczalnym parametrem; omówimy je nieco później (patrz rozdział 4.).

Nazwą parametru typu jest w naszym przykładzie `T`. Nazwa ta może być dowolnym ciągiem dopuszczalnym z punktu widzenia zasad konstruowania identyfikatorów języka C++, dla parametru typu przyjęło się jednak stosować nazwę `T`. Parametr typu reprezentuje typ umowny precyzowany przez programistę dopiero w miejscu wywołania funkcji. Wywołanie może określać dowolny typ (typ podstawowy, klasę itp.) pod warunkiem, że dla tego typu zdefiniowane są wszystkie operacje wykorzystywane w szablonie funkcji. W prezentowanym przykładzie typ ten musi więc obsługiwać operator relacji „mniejsze niż" (<), ponieważ parametry przekazywane do funkcji są porównywane właśnie za pomocą tego operatora.

Z przyczyn historycznych w określeniu parametru typu dopuszcza się stosowanie w miejsce `typename` słowa kluczowego `class`. Słowo kluczowe `typename` pojawiło się w definicji języka C++ stosunkowo późno; wcześniej jedynym sposobem wprowadzenia do listy parametrów szablonu parametru typu było zastosowanie właśnie słowa kluczowego `class`. Możliwość ta została podtrzymana również w ostatnich wersjach standardu C++. Nasz szablon można więc równie dobrze zdefiniować jako:

```
template <class T>
inline T const& max (T const& a, T const& b)
{
    // jeżeli a < b zwróć b; w przeciwnym przypadku zwróć a
    return a < b ? b : a;
}
```

Definicja ta jest znaczeniowo tożsama z definicją prezentowaną jako pierwszą. Należy przy tym pamiętać, że pomimo zastosowania słowa kluczowego `class` zakres dopuszczalnych typów parametrów szablonu nie jest redukowany do typów definiowanych przez użytkownika (klas). Widać więc, że stosowanie słowa kluczowego `class` może być mylące (`T` może przecież zostać zastąpione w wywołaniu również przez typ podstawowy), dlatego w przypadkach wątpliwych zaleca się stosowanie słowa kluczowego `typename`. W miejsce słowa kluczowego `typename` nie można natomiast z dobrym skutkiem zastosować słowa kluczowego `struct` (choć jest on w pewnym zakresie tożsamy ze słowem kluczowym `class`).

2.1.2. Korzystanie z szablonu

Poniższy program ilustruje sposób wykorzystania zdefiniowanego wcześniej szablonu funkcji `max()`:

```
// podstawy/max.cpp

#include <iostream>
#include <string>
#include "max.hpp"

int main()
{
    int i = 42;
    std::cout << "max(7, i):   " << ::max(7, i) << std::endl;
```

```
    double f1 = 3.4;
    double f2 = -6.7;
    std::cout << "max(f1, f2): " << ::max(f1, f2) << std::endl;

    std::string s1 = "matematyka";
    std::string s2 = "matema";
    std::cout << "max(s1, s2): " << ::max(s1, s2) << std::endl;
}
```

W tym programie funkcja `max()` została wywołana trzykrotnie: dla dwóch argumentów typu `int`, dla dwóch argumentów typu `double` oraz dla dwóch argumentów typu `std::string`. Za każdym razem funkcja zwróciła większą z dwóch przekazanych wartości. Wydruk programu powinien wyglądać następująco:

```
max(7, i):   42
max(f1, f2): 42
max(s1, s2): matematyka
```

Zauważmy, że każde z wywołań funkcji `max()` zostało opatrzone operatorem zasięgu `::`. Dzięki niemu mamy gwarancję, że funkcja `max()` będzie wyszukiwana w globalnej przestrzeni nazw programu. Istnieje bowiem również szablon funkcji `std::max()`, który w pewnych warunkach mógłby zostać wywołany zamiast naszego szablonu `max()`[1].

Zazwyczaj szablony nie są kompilowane do postaci jednostek obsługujących typ dowolny. Dla każdego nowego typu, dla którego wywołany zostanie szablon, generowana jest osobna jednostka programowa[2]. Funkcja `max()` zostanie więc skompilowana trzykrotnie, dla każdego z typów, z którym została wywołana. Dla przykładu, pierwsze wywołanie funkcji `max()`:

```
int i = 42;
… max(7, i) …
```

oznacza takie odwołanie do szablonu funkcji, które określa parametr `T` jako `int`. Wywołanie to odpowiada więc wywołaniu następującej funkcji:

```
inline int const& max (int const& a, int const& b)
{
    // jeżeli a < b zwróć b; w przeciwnym przypadku zwróć a
    return a < b ? b : a;
}
```

Proces zastępowania parametrów szablonu konkretnymi typami nazywamy *konkretyzacją* (ang. *instantiation*) szablonu. Jej efektem jest utworzenie *egzemplarza* (ang. *instance*) szablonu. Co prawda pojęcia egzemplarza i konkretyzacji w kontekście programowania obiektowego stosowane są również w odniesieniu do obiektów klas, jednak ze względu na tematykę niniejszej książki terminy te będziemy odnosić zawsze do szablonów (o ile nie zaznaczymy inaczej).

---

[1] Przykładowo, jeżeli typ jednego z parametrów wywołania zostanie zdefiniowany w przestrzeni nazw `std` (jak choćby typ `std::string`), reguły wyszukiwania języka C++ spowodują dopasowanie w miejscu wywołania obu szablonów globalnego `max()` i `std::max()`.

[2] Alternatywa, polegająca na generowaniu przez kompilator jednostek uniwersalnych, jest co prawda do pomyślenia, ale w praktyce stosuje się ją rzadko. Wszystkie reguły języka oparte są na koncepcji zakładającej generowanie osobnych jednostek dla różnych typów szablonów.

Istotne jest, że konkretyzację wyzwala samo zastosowanie szablonu funkcji — programista nie musi jawnie żądać utworzenia egzemplarza szablonu.

Pozostałe wywołania `max()` konkretyzują szablon `max()` dla typów `double` i `std::string` i są wykorzystywane dokładnie tak, jakby specjalnie dla tych wywołań zostały zdefiniowane i zaimplementowane funkcje:

```
const double& max (double const&, double const&);
const std::string& max (std::string const&, std::string const&);
```

Próba konkretyzacji szablonu dla typu, który nie obsługuje wszystkich operacji wykorzystywanych w szablonie, sprowokuje błąd kompilacji. Przykład:

```
std::complex<float> c1, c2;   // typ complex nie obsługuje operatora <
…
max(c1, c2);                  // BŁĄD kompilacji
```

Jak widać, szablony są kompilowane dwukrotnie:

1. Pierwszy raz bez konkretyzacji: kod szablonu jest analizowany pod kątem poprawności składniowej. Na tym etapie wykrywane są błędy składniowe, takie jak brakujące średniki.
2. Drugi raz podczas konkretyzacji: kod szablonu jest weryfikowany pod kątem poprawności wszystkich wywołań. Na tym etapie wykrywane są niepoprawne wywołania, takie jak wywołania funkcji nieobsługujące danego typu.

W praktyce dwukrotna kompilacja jest przyczyną dość istotnego problemu: kiedy szablon funkcji jest wykorzystywany w sposób prowokujący jego konkretyzację, kompilator (w pewnym momencie) musi odwołać się do definicji szablonu. Powoduje to naruszenie uświęconej wieloletnią tradycją języka C separacji pomiędzy etapem kompilacji a etapem konsolidacji kodu funkcji, gdyż normalnie do kompilacji wywołania funkcji wystarczające jest jej uprzednie zadeklarowanie. W rozdziale 6. pokażemy, jak poradzić sobie z tym problemem. Na razie zaś wykorzystamy metodę najprostszą: implementację każdego szablonu w pliku nagłówkowym, za pomocą słowa kluczowego `inline`.

2.2. Dedukcja typu argumentów

Wywołując szablon funkcji (jak `max()`) z pewnymi argumentami, określamy za ich pośrednictwem parametry szablonu. Jeżeli jako parametry `T const&` przekazane zostaną zmienne typu `int`, kompilator języka C++ założy, że typem `T` jest typ `int`. Zauważmy, że taki sposób wnioskowania o typie parametrów wyklucza stosowanie automatycznej konwersji typów w miejscu wywołania szablonu funkcji. Każdy typ `T` musi zostać określony w sposób jawny. Oto przykład:

```
template <typename T>
inline T const& max (T const& a, T const& b);
…
max(4, 7);        // DOBRZE: T to int dla obu argumentów
max(4, 4.2);      // BŁĄD: pierwszy argument jest typu int, drugi typu double
```

Błąd ten można wyeliminować na trzy sposoby:

1. Przez jawne rzutowanie parametrów do wspólnego typu:

```
max(static_cast<double>(4), 4.2);   // DOBRZE
```

2. Przez jawne określenie typu T w wywołaniu szablonu:

```
max<double>(4, 4.2);
```

3. Przez zdefiniowanie szablonu o dwóch różnych parametrach typu.

Szczegółowe omówienie każdej z trzech metod znajduje się w kolejnym podrozdziale.

2.3. Parametry szablonu

Szablony funkcji można zdefiniować przy użyciu dwóch rodzajów parametrów:

1. Parametrów szablonu, deklarowanych wewnątrz nawiasów ostrych przed nazwą szablonu funkcji:

```
template <typename T>              // T jest parametrem szablonu
```

2. Parametrów wywołania, deklarowanych wewnątrz nawiasów zwykłych po nazwie szablonu funkcji:

```
… max (T const& a, T const& b)      // a i b są parametrami wywołania
```

Nie istnieje ograniczenie liczby parametrów szablonu. Niemniej jednak w szablonach funkcji (w przeciwieństwie do szablonów klas) nie można zdefiniować domyślnych argumentów szablonu[3]. Definicja szablonu `max()` taka, aby w wywołaniu przyjmowane były argumenty różnych typów, wyglądałaby następująco:

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
    return a < b ? b : a;
}
…
max(4, 4.2);  // DOBRZE, ale typ pierwszego argumentu determinuje typ wartości zwracanej
```

Metoda polegająca na dopuszczeniu w wywołaniu parametrów różnych typów wydaje się skuteczna, ale skuteczność ta jest pozorna. Problem tkwi w typie wartości zwracanej przez szablon funkcji. Jeżeli wartość ta będzie typu zgodnego z typem jednego z parametrów, argument drugiego typu będzie musiał zostać poddany konwersji do typu pierwszego, niezależnie od intencji programisty. Język C++ nie dysponuje bowiem metodami pozwalającymi na wyznaczenie „lepszego" z dwóch typów (choć można tę możliwość zaimplementować przy użyciu pewnych sztuczek z szablonami, prezentowanymi w punkcie 15.2.4). Dlatego kolejność przekazywania argumentów wywołania `max()` jako maksimum z wartości 42 i 66.66 możemy otrzymać wartość 66.66 (typu `double`) bądź 66

[3] Ograniczenie to jest rezultatem zaszłości związanych z projektowaniem szablonów funkcji. Wydaje się, że nie ma technicznych przeciwwskazań do implementacji takiej możliwości w nowoczesnych kompilatorach języka C++; w przyszłości możliwość ta będzie najprawdopodobniej dostępna (patrz podrozdział 13.3).

(typu `int`). Inny problem to tworzenie lokalnego obiektu tymczasowego podczas konwersji typu drugiego parametru — tak utworzonego obiektu nie można zwrócić przez referencję[4]. W naszym przykładzie oznacza to konieczność modyfikacji definicji szablonu tak, aby zwracał typ `T1` zamiast `T1 const&`.

Typy parametrów wywołania tworzone są na podstawie parametrów szablonu, więc parametry szablonu i parametry wywołania można zazwyczaj ze sobą powiązać. Wiązanie takie nazywamy *dedukcją argumentów szablonu funkcji*. Mechanizm ten pozwala na wywoływanie szablonów funkcji dokładnie tak jak funkcji „zwykłych".

Tym niemniej możliwa jest również jawna konkretyzacja szablonu dla wskazanych typów:

```
template <typename T>
inline T const& max (T const& a, T const& b);
…
max<double>(4, 4.2);                // konkretyzacja jawna typu T do typu double
```

W przypadkach, kiedy nie ma powiązania pomiędzy parametrami wywołania a parametrami szablonu i niemożliwe jest określenie parametrów szablonu, konieczne jest jawne określenie argumentu szablonu w miejscu wywołania. Przykładowo, typ wartości zwracanej przez szablon funkcji można określić, wprowadzając do definicji szablonu trzeci parametr szablonu:

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
```

Niemniej jednak dedukcja typu argumentów nie obejmuje typu wartości zwracanej[5], a `RT` nie występuje na liście typów parametrów wywołania funkcji. Nie można więc wydedukować typu `RT`. W efekcie konieczne jest jawne określanie w miejscu wywołania pełnej listy parametrów szablonu. Przykład:

```
template <typename T1, typename T2, typename RT>
inline RT max (T1 const& a, T2 const& b);
…
max<int, double, double>(4, 4.2);   // DOBRZE, ale niewygodnie
```

Pokazaliśmy jak dotąd przypadki, w których jawnie (w miejscu wywołania) określany był albo każdy, albo żaden z argumentów szablonu. Możliwe jest również jawne określanie jedynie pierwszych kilku argumentów — pozostałe zostaną wydedukowane. W ogólności konieczne jest określenie wszystkich argumentów typów aż do ostatniego argumentu, którego typ nie może zostać określony niejawnie (przez dedukcję). Jeżeli więc zmienimy kolejność na liście parametrów przykładowego szablonu, będziemy mogli wywoływać funkcję szablonu, podając jawnie wyłącznie typ wartości zwracanej:

```
template <typename RT, typename T1, typename T2>
inline RT max (T1 const& a, T2 const& b);
…
max<double>(4, 4.2);       // DOBRZE: typ wartości zwracanej to double
```

[4] Zwracanie wartości przez referencję nie jest dozwolone w przypadku obiektów lokalnych względem funkcji, ponieważ po wyjściu z funkcji obiekt ten przestanie istnieć.

[5] Dedukcję można postrzegać jako część procesu rozstrzygania przeciążenia — proces ten również ignoruje typy wartości zwracanych. Jedynym wyjątkiem jest typ zwracany składowych operatorów konwersji klasy.

W powyższym przykładzie wywołanie `max<double>` określa w sposób jawny typ wartości zwracanej przez funkcję; parametry `T1` i `T2` muszą zaś zostać wydedukowane na podstawie argumentów wywołania (tu: `int` i `double`).

Zauważmy, że kolejne przeróbki szablonu funkcji `max()` nie zmieniają zasadniczo jego funkcjonalności. Już w przypadku wersji jednoparametrowej szablonu można określać typ parametrów przekazywanych w wywołaniu (i typ wartości zwracanej), nawet w przypadku przekazania argumentów dwóch różnych typów. Zaleca się więc maksymalne upraszczanie szablonów — w kolejnych podrozdziałach korzystać będziemy z jednoparametrowego szablonu funkcji `max()`.

Proces dedukcji parametrów został szczegółowo omówiony w rozdziale 11.

2.4. Przeciążanie szablonów funkcji

Podobnie jak to ma miejsce w przypadku zwykłych funkcji, szablony funkcji mogą być przeciążane. Przeciążanie oznacza możliwość korzystania z różnych definicji funkcji, o tych samych nazwach. Kiedy nazwa ta zostanie wykorzystana w wywołaniu, kompilator języka C++ podejmuje decyzję o wyborze funkcji do obsługi wywołania. Reguły rządzące procesem decyzyjnym są dostatecznie skomplikowane i bez szablonów. W bieżącym podrozdziale omówimy przeciążanie wykorzystujące szablony funkcji. Czytelnicy, którym podstawowe reguły rozstrzygania przeciążania bez szablonów nie są znane, powinni zapoznać się z dodatkiem B, gdzie zostały one opisane w dość szczegółowy sposób.

Ilustracją przeciążania szablonu funkcji może być następujący, prosty program:

```
// podstawy/max2.cpp

// maksimum z dwóch wartości typu int
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}

// maksimum z dwóch wartości dowolnego typu
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// maksimum z trzech wartości dowolnego typu
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max(max(a, b), c);
}

int main()
{
    ::max(7, 42, 68);      // wywołanie szablonu z przekazaniem trzech argumentów
    ::max(7.0, 42.0);      // wywołanie szablonu max<double> (dedukcja typu argumentów)
```

```
    ::max('a', 'b');       // wywołanie szablonu max<double> (dedukcja typu argumentów)
    ::max(7, 42);          // wywołanie funkcji zwykłej dla dwóch argumentów int
    ::max<>(7, 42);        // wywołanie szablonu max<int> (dedukcja typu argumentów)
    ::max<double>(7, 42);  // wywołanie szablonu max<double> (bez dedukcji typu)
    ::max('a', 42.7);      // wywołanie funkcji zwykłej dla dwóch argumentów int
}
```

Jak widać, „zwykłe" funkcje mogą współistnieć z szablonami funkcji o tych samych nazwach i konkretyzowanych dla tych samych co owe zwykłe funkcje typów. Jeżeli wszystkie pozostałe czynniki są identyczne, proces rozstrzygający przeciążenia preferuje funkcje „zwykłe" przed szablonami. Sytuację tę ilustruje czwarte z kolei wywołanie:

```
max(7, 42);         // oba argumenty odpowiadają funkcji zwykłej
```

Jeżeli na podstawie szablonu można wygenerować lepsze dopasowanie, wybierany jest oczywiście szablon. Przypadek taki miał miejsce w wywołaniach drugim i trzecim:

```
max(7.0, 42.0);     // wywołanie szablonu max<double> (dedukcja typu argumentów)
max('a', 'b');      // wywołanie szablonu max<char> (dedukcja typu argumentów)
```

Możliwe jest również jawne wskazanie pustej listy argumentów szablonu. W ten sposób programista sygnalizuje konieczność wykorzystania szablonu, przy czym jego parametry powinny zostać wydedukowane na podstawie argumentów wywołania:

```
max<>(7, 42);       // wywołanie szablonu max<int> (dedukcja typu argumentów)
```

Ponieważ w przypadku szablonów funkcji nie jest wykonywana automatyczna konwersja typu parametrów wywołania, ale konwersja taka jest przeprowadzana dla zwykłych funkcji, ostatnie wywołanie dopasowane zostanie do funkcji zwykłej (parametry `'a'` i `42.7` zostaną poddane konwersji do typu `int`):

```
max('a', 42.7);     // tylko zwykła funkcja dopuszcza różne typy argumentów wywołania
```

Bardziej użytecznym przykładem będzie przeciążenie szablonu zwracającego maksimum tak, aby obsługiwał on wskaźniki i ciągi znakowe języka C:

```
// podstawy/max3.cpp

#include <iostream>
#include <cstring>
#include <string>

// maksimum z dwóch wartości dowolnego typu
template <typename T>
inline T const& max (T cost& a, T const& b)
{
    return a < b ? b : a;
}

// maksimum z dwóch wskaźników
template <typename T>
inline T* const& max (T* cost& a, T* const& b)
{
    return *a < *b ? b : a;
}
```

```
// maksimum z dwóch ciągów języka C
inline char const* const& max (char const* const& a, char const* const& b)
{
    return std::strcmp(a, b) < 0 ? b : a;
}

int main()
{
    int a = 7;
    int b = 42;
    ::max(a, b);        // maksimum dwóch wartości typu int

    std::string s = "hej";
    std::string t = "ty";
    ::max(s, t);        // maksimum dwóch wartości typu std::string

    int* p1 = &b;
    int* p2 = &a;
    ::max(p1, p2);      // maksimum dwóch wskaźników

    char const* s1 = "David";
    char const* s2 = "Nico";
    ::max(s1, s2);      // maksimum dwóch ciągów języka C
}
```

We wszystkich implementacjach przeciążonych argumenty były przekazywane przez referencje, bowiem przeciążając szablony funkcji, warto ograniczać zmiany w stosunku do funkcji zwykłych. Zmiany te powinny sprowadzać się do zmiany liczby parametrów lub do jawnego określania parametrów szablonu. W przeciwnym przypadku należy przygotować się na niespodzianki. Przykładowo, po przeciążeniu szablonu funkcji `max()` (zakładającego przekazywanie argumentów przez referencję) dla dwóch ciągów języka C przekazywanych przez wartość, nie można korzystać z trójargumentowej wersji `max()` do wskazywania maksimum z trzech ciągów języka C:

```
// podstawy/max3a.cpp

#include <iostream>
#include <cstring>
#include <string>

// maksimum z dwóch wartości dowolnego typu (przekazanie przez referencję)
template <typename T>
inline T const& max (T cost& a, T const& b)
{
    return a < b ? b : a;
}

// maksimum z dwóch ciągów języka C (przekazanie przez wartość)
inline char const* max (char const* a, char const* b)
{
    return std::strcmp(a, b) < 0 ? b : a;
}

// maksimum z trzech wartości dowolnego typu (przekazanie przez referencję)
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
```

```
{
    return max(max(a, b), c);     // błąd, jeżeli max(a, b) korzysta z przekazania
                                  // przez wartość
}

int main()
{
    ::max(7, 42, 68);             // DOBRZE

    const char* s1 = "Frederic";
    const char* s2 = "Anica";
    const char* s3 = "Lucas";
    ::max(s1, s2, s3);            // BŁĄD
}
```

Błąd pojawia się w miejscu wywołania `max()` dla trzech ciągów języka C:

```
return max(max(a, b), c);
```

Błąd spowodowany jest utworzeniem przez wywołanie `max(a, b)` nowego, tymczasowego lokalnego względem wywołania, obiektu, który nie może zostać zwrócony na zewnątrz przez referencję.

To tylko jeden z możliwych przykładów kodu, który będzie się zachowywał niezgodnie z oczekiwaniem programisty w wyniku zastosowania reguł rozstrzygania przeciążania. Dla przykładu, dla poprawności działania programu znaczący może okazać się fakt widoczności bądź niewidoczności wszystkich wersji przeciążonych w miejscu wywołania funkcji. W rzeczy samej, definiowanie trójargumentowej wersji szablonu `max()` poza zakresem widoczności deklaracji zwykłej funkcji dwuargumentowej dla typu `int` może spowodować preferowanie, zamiast funkcji zwykłej, szablonu:

```
// podstawy/max4.cpp

// maksimum z dwóch wartości dowolnego typu
template <typename T>
inline T const& max (T const& a, T const& b)
{
    return a < b ? b : a;
}

// maksimum z trzech wartości dowolnego typu
template <typename T>
inline T const& max (T const& a, T const& b, T const& c)
{
    return max(max(a, b), c);     // wykorzystana tu zostanie konkretyzacja szablonu
                                  // dla typu int, ponieważ funkcja zwykła
zadeklarowana
                                  // została zbyt późno
}

// maksimum z dwóch wartości typu int
inline int const& max (int const& a, int const& b)
{
    return a < b ? b : a;
}
```

Szersze omówienie tego zagadnienia zamieściliśmy w rozdziale 12.; na razie należałoby pamiętać o konieczności zdefiniowania wszystkich wersji przeciążanych funkcji i szablonów przed pierwszym wywołaniem.

2.5. Podsumowanie

- Szablony funkcji definiują rodzinę funkcji dla różnych parametrów szablonu.
- Przekazując w wywołaniu argumenty szablonu, konkretyzujemy szablon funkcji dla wskazanych typów argumentów.
- Możliwe jest jawne kwalifikowanie parametrów szablonu.
- Możliwe jest przeciążanie szablonów funkcji.
- Przeciążając szablony funkcji, należy ograniczać zmiany do jawnego określania parametrów szablonu.
- Zawsze warto sprawdzić, czy w miejscu wywołania znane są wszystkie wersje przeciążonego szablonu funkcji.