Stephen Prata

Język C++

Szkoła programowania

Wydanie VI

Najlepsze źródło informacji o C++!

Helion

Tytuł oryginału: C++ Primer Plus, Sixth Edition

Tłumaczenie: Przemysław Szeremiota
na podstawie „Język C++. Szkoła programowania. Wydanie V”
w tłumaczeniu Tomasza Żmijewskiego, Przemysława Szeremioty, Tomasza Walczaka, Przemysława Stecia

ISBN: 978-83-246-4336-3

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/cppri6.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/cppri6
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

8

Funkcje — zagadnienia zaawansowane

W rozdziale zostaną omówione następujące zagadnienia:

- Funkcje *inline*.
- Zmienne referencyjne.
- Przekazywanie funkcji parametrów przez referencję.
- Parametry domyślne.
- Przeciążanie funkcji.
- Szablony funkcji.
- Specjalizacje szablonów funkcji.

W rozdziale 7. omalże mimochodem pojawiło się wiele informacji o funkcjach w C++, ale jeszcze więcej jest przed nami. Funkcje C++ mają sporo nowych właściwości, których nie było w starym dobrym C. Te nowe możliwości to funkcje *inline*, przekazywanie zmiennych przez referencję, wartości domyślne parametrów, przeciążanie funkcji i szablony funkcji. W niniejszym rozdziale bardziej niż dotąd zajmiemy się tymi elementami C++, które nie występują w C, zatem rozdział ten to pierwszy istotny wypad w kierunku dwóch plusów.

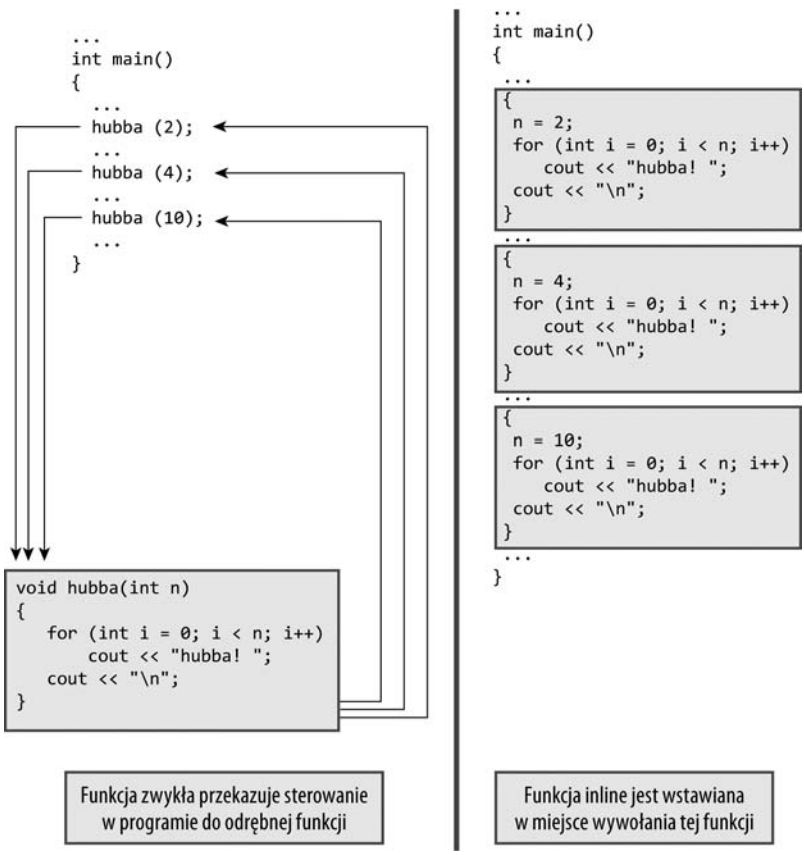
Funkcje inline

Funkcje inline to rozszerzenie C++ stworzone z myślą o przyspieszeniu działania programów. Podstawową różnicą między zwykłymi funkcjami a funkcjami *inline* nie jest sposób ich kodowania, ale metoda włączania ich do programu przez kompilator. Aby zrozumieć różnicę między zwykłymi funkcjami a funkcjami *inline*, musimy, dokładniej niż robiliśmy to dotąd, przyjrzeć się wnętrzu programu. Przystąpmy od razu do rzeczy.

Ostatecznym wynikiem procesu kompilacji jest program wykonywalny, który składa się z zestawu instrukcji języka maszynowego. Kiedy uruchamiamy program, system operacyjny ładuje wszystkie te instrukcje do pamięci komputera, tak że każda instrukcja ma własny adres w pamięci. Następnie komputer kolejno wykonuje instrukcję po instrukcji. Czasami kiedy występuje na przykład pętla albo instrukcja warunkowa, wykonywanie programu przeskakuje wprzód lub wstecz, pod inny adres.

Wywołania zwykłych funkcji oznaczają skok programu pod inny adres (adres funkcji), a po wykonaniu całej funkcji skok powrotny. Przyjrzyjmy się typowej realizacji tego procesu nieco dokładniej. Kiedy program dochodzi do instrukcji skoku funkcji, zapamiętuje adres instrukcji znajdującej się za instrukcją skoku, kopiuje parametry funkcji na stos (specjalnie przygotowany w tym celu fragment pamięci), skacze do miejsca w pamięci zawierającego początek kodu funkcji, wykonuje kod funkcji (ewentualnie umieszczając wynik funkcji w rejestrze), a następnie przeskakuje z powrotem do instrukcji, której adres został zapisany wcześniej[1]. Skoki wprzód i wstecz oraz zapamiętywanie lokalizacji skoków zajmują czas — jest to koszt wywoływania funkcji.

Funkcje *inline* stanowią alternatywę dla takiego rozwiązania. W funkcji *inline* kod skompilowany jest włączony bezpośrednio w resztę programu — kompilator zastępuje wywołanie funkcji całym jej kodem. Wobec tego program nie musi przeskakiwać w inne miejsce ani potem wracać, dzięki czemu funkcje takie działają nieco szybciej niż ich zwykłe odpowiedniki, ale ceną za to jest większe zużycie pamięci. Jeśli program w 10 różnych miejscach wywołuje funkcję *inline*, w uzyskanym kodzie pojawi się 10 kopii tej funkcji (rysunek 8.1).



Rysunek 8.1. *Funkcje inline a funkcje zwykłe*

[1] To trochę jak zostawianie na chwilę czytanego tekstu w celu zapoznania się z przypisem — po lekturze tego przypisu z powrotem wracamy w to samo miejsce, w którym skończyliśmy czytanie.

Funkcji *inline* trzeba używać ostrożnie. Jeśli czas potrzebny na wykonanie funkcji jest o wiele dłuższy niż czas obsługi wywołania, to czas zaoszczędzony na braku skoków jest niewielkim ułamkiem całego procesu wykonywania funkcji. Jeśli czas wykonywania kodu jest krótki, to funkcja *inline* pozwala znacznie skrócić czas pracy programu. Jeżeli często udaje nam się zaoszczędzić nawet krótki czas, który stanowi jednak istotny fragment większego procesu, oszczędności mogą być znaczące.

Aby skorzystać z opisywanych tu możliwości, musimy podjąć przynajmniej jedno z dwóch działań:

- Poprzedzić deklarację funkcji słowem kluczowym `inline`.
- Poprzedzić definicję funkcji słowem kluczowym `inline`.

Powszechną praktyką jest pomijanie prototypu i wstawianie pełnej definicji (czyli nagłówka funkcji i całego kodu funkcji) tam, gdzie zwykle jest prototyp.

Kompilator wcale nie musi honorować żądania użytkownika, aby funkcja była *inline*. Kompilator może przyjąć, że funkcja jest zbyt duża lub że wywołuje samą siebie (funkcje *inline* nie powinny i nawet nie mogą być rekurencyjne), albo dany kompilator może mieć w ogóle funkcje *inline* wyłączone lub nawet niezaimplementowane.

Listing 8.1 pokazuje technikę funkcji *inline* w przypadku funkcji `square()`, która podnosi do kwadratu swój parametr. Zauważmy, że cała definicja tej funkcji mieści się w jednym wierszu. Nie jest to wymóg, ale jeśli definicja nie mieści się w jednym albo dwóch wierszach kodu (przy założeniu rozsądnej długości identyfikatorów), funkcja zwykle nie nadaje się na funkcję *inline*.

Listing 8.1. *inline.cpp*

```
// inline.cpp -- użycie funkcji inline
#include <iostream>

// definicja funkcji inline
inline double square(double x) { return x * x; }

int main()
{
    using namespace std;
    double a, b;
    double c = 13.0;

    a = square(5.0);
    b = square(4.5 + 7.5);    // można przekazać wyrażenie
    cout << "a = " << a << ", b = " << b << "\n";
    cout << "c = " << c;
    cout << ", c kwadrat = " << square(c++) << "\n";
    cout << "Teraz c = " << c << "\n";
    return 0;
}
```

Oto wynik uruchomienia programu z listingu 8.1:

```
a = 25, b = 144
c = 13, c kwadrat = 169
Teraz c = 14
```

Widać tu, że funkcjom *inline* parametry przekazuje się przez wartość, tak jak zwykłym funkcjom. Jeśli parametr jest wyrażeniem, na przykład `4.5 + 7.5`, funkcja przekaże wartość tego wyrażenia — w tym przypadku 12. Powoduje to, że funkcje *inline* znacznie przewyższają makra znane z języka C; zobacz wskazówkę „Funkcje *inline* a makra”.

Mimo że nie istnieje osobny prototyp, prototypowanie nadal obowiązuje. Chodzi o to, że w tym przypadku cała definicja funkcji pojawia się przed pierwszym użyciem tej funkcji i to ona służy jako prototyp. Znaczy to, że można użyć funkcji `square()` z parametrem typu `int` albo typu `long`, a program i tak parametry te zrzutuje na typ `double` i dopiero wtedy przekaże funkcji.

Funkcje *inline* a makra

Funkcje *inline* to cecha specyficzna dla języka C++, natomiast język C korzystał z dyrektywy preprocesora `#define`, aby tworzyć *makra* stanowiące pewne przybliżenie kodu *inline*. Oto przykład makra podnoszącego liczbę do kwadratu:

```
#define SQUARE(X) X*X
```

Jednak definicja taka nie działa przez przekazywanie parametrów, ale przez podstawianie tekstu, gdzie `X` jest etykietą symboliczną i stanowi jedynie namiastkę parametru:

```
a = SQUARE(5.0); zostanie zastąpione przez a = 5.0*5.0;
b = SQUARE(4.5 + 7.5); zostanie zastąpione przez b = 4.5 + 7.5*4.5 + 7.5;
d = SQUARE(c++); zostanie zastąpione przez d = c++*c++;
```

Jedynie pierwszy przykład z powyższych zadziała prawidłowo. Sytuację można nieco poprawić, dodając nawiasy:

```
#define SQUARE(X) ((X)*(X))
```

jednak nadal pozostaje problem związany z tym, że makra nie są przekazywane przez wartość. Nawet dla ostatniej definicji wywołanie `SQUARE(c++)` spowoduje dwukrotną inkrementację `c`, podczas gdy w przypadku analogicznej funkcji *inline* `square()` z listingu 8.1 wyznaczana jest wartość `c`; wartość ta jest przekazywana do funkcji, a na koniec jest raz inkrementowana.

Celem nie jest tu pokazywanie, jak pisze się makra w C. Chodzi raczej o to, że jeśli trzeba w formie makra zapisać jakieś działanie mające charakter funkcji, to lepiej użyć funkcji *inline*.

Zmienne referencyjne

Język C++ dodaje do języka nowy typ złożony — zmienną referencyjną. *Referencja* działa jak alias, czyli inna nazwa zmiennej zdefiniowanej już wcześniej. Jeśli na przykład `twain` będzie referencją do zmiennej `clemens`, do tej zmiennej będziemy mogli równoprawnie odwoływać się tak przez nazwę `clemens`, jak i przez `twain`. Ale po co nam to? Chodzi o pomoc dla tych programistów, którzy wstydzą się nazw, jakie wcześniej nadali swoim zmiennym? Może i tak, ale najważniejszym zastosowaniem zmiennych referencyjnych jest używanie ich jako parametrów funkcji. Jeśli parametr jest referencją, funkcja działa na danych oryginalnych, a nie na kopii zmiennej. Referencje to wygodna alternatywa dla wskaźników przy przetwarzaniu dużych struktur za pomocą funkcji; są one ogromnie ważne przy projektowaniu klas. Zanim zobaczymy, jak używać referencji w funkcjach, przeanalizujmy podstawowe cechy referencji. Pamiętać trzeba, że naszym celem jest omówienie działania referencji, a nie przekonanie się, jak są one używane najczęściej.

Tworzenie zmiennej referencyjnej

Jak pamiętamy, w C i C++ symbol `&` pozwala pozyskać adres zmiennej. W C++ symbolowi `&` przypisano dodatkowe znaczenie, zaprzęgając go do deklarowania referencji. Aby na przykład `rodents` było alternatywną nazwą zmiennej `rats`, można zapisać:

```
int rats;
int & rodents = rats;      // rodents staje się aliasem dla rats
```

W tym kontekście & nie jest operatorem adresu, ale jest częścią identyfikatora typu. Tak jak `char *` oznacza wskaźnik typu char, tak `int &` sugeruje referencję do wartości int. Deklaracja referencji pozwala wymiennie używać nazw rats i rodents — obie nazwy odnoszą się do tej samej wartości, położonej w tym samym miejscu w pamięci. Na listingu 8.2 widać, że tak jest naprawdę.

Listing 8.2. *firstref.cpp*

```
// firstref.cpp -- definiowanie i użycie referencji
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats;    // rodents to referencja

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

// w niektórych implementacjach C++ wymagane jest rzutowanie
// typu adresów na typ unsigned
    cout << "adres rats = " << &rats;
    cout << ", adres rodents = " << &rodents << endl;
    return 0;
}
```

Zwróćmy uwagę na operator & w instrukcji:

```
int & rodents = rats;
```

Nie jest on tu operatorem adresu, ale służy do zadeklarowania rodents jako zmiennej typu `int &`, czyli jako referencji do wartości int. Jednak symbol & w wyrażeniu:

```
cout << ", adres rodents = " << &rodents << endl;
```

to już operator adresu, a `&rodents` to adres zmiennej, do której odnosi się rodents. Oto wynik działania programu z listingu 8.2:

```
rats = 101, rodents = 101
rats = 102, rodents = 102
adres rats = 0x0065fd48, adres rodents = 0x0065fd48
```

Jak widać, rats i rodents mają tę samą wartość i ten sam adres (w różnych systemach sposób wyświetlania adresu jest różny). Zwiększenie wartości zmiennej rodents o jeden wpływa na obie zmienne. Ściślej rzecz biorąc, wykonanie instrukcji rodents++ zwiększa o jeden wartość jednej zmiennej, która ma dwie nazwy. Pamiętajmy, że choć w przykładzie pokazano modelowe działanie referencji, to w praktyce zwykle używa się ich inaczej — do przekazywania parametrów funkcji, szczególnie kiedy parametry te są strukturami lub obiektami.

Referencje są zwykle czymś niepojętym dla ekspertów języka C, którzy uczą się C++, gdyż są dziwnie podobne do wskaźników, ale jednocześnie jest w nich coś innego. Można na przykład utworzyć referencję i wskaźnik odwołujące się do zmiennej rats:

```
int rats = 101;
int & rodents = rats;    // rodents to referencja
int * prats = &rats;     // prats to wskaźnik
```

Dalej można używać wymiennie wyrażeń `rodents`, `*prats` i `rats` oraz `&rodents`, `prats` i `&rats`. Z tego punktu widzenia referencje wyglądają jak nieco inaczej zapisane wskaźniki, w których operator dereferencji * jest dany niejawnie. I tak naprawdę tym mniej więcej są referencje. Jednakże poza samym zapisem istnieją i inne różnice. Po pierwsze, referencję trzeba zainicjalizować w chwili jej deklarowania — nie można referencji zadeklarować, a wartości przypisać jej później, co jest normalnym postępowaniem w przypadku wskaźników:

```
int rat;
int & rodent;
rodent = rat;      // nie, tak nie wolno
```

Uwaga

Deklarując zmienną referencyjną, należy ją zainicjalizować.

Referencja to coś w rodzaju stałego wskaźnika — trzeba ją zainicjalizować w chwili jej tworzenia, ale kiedy już raz powiążemy ją ze zmienną, to to powiązanie zostanie na zawsze. Wobec tego zapis:

```
int & rodents = rats;
```

można w zasadzie traktować jako zamienny z zapisem:

```
int * const pr = &rats;
```

W tym przypadku referencja `rodents` odgrywa taką samą rolę jak wyrażenie `*pr`.

Na listingu 8.3 pokazano, co się stanie, jeśli spróbujemy zmienić powiązanie referencji ze zmiennej `rats` na zmienną `bunnies`.

Listing 8.3. *secref.cpp*

```
// secref.cpp -- definiowanie i stosowanie referencji
#include <iostream>
int main()
{
    using namespace std;
    int rats = 101;
    int & rodents = rats;    // rodents to referencja

    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    cout << "adres rats = " << &rats;
    cout << ", adres rodents = " << &rodents << endl;

    int bunnies = 50;
    rodents = bunnies;       // czy można zmienić referencję?
    cout << "bunnies = " << bunnies;
    cout << ", rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    cout << "adres bunnies = " << &bunnies;
    cout << ", adres rodents = " << &rodents << endl;
    return 0;
}
```

Oto wyniki działania programu z listingu 8.3:

```
rats = 101, rodents = 101
adres rats = 0x0065fd44, adres rodents = 0x0065fd44
bunnies = 50, rats = 50, rodents = 50
adres bunnies = 0x0065fd48, adres rodents = 0x0065fd4
```

Początkowo `rodents` odnosi się do `rats`, ale dalej program próbuje zmienić przypisanie:

```
rodents = bunnies;
```

Przez chwilę wygląda na to, że przypisanie się powiodło, gdyż wartość `rodents` zmieniła się ze 101 na 50. Jednak dokładniejsze przyjrzenie się pokazuje, że zmienna `rats` także zmieniła swoją wartość na 50, a `rats` i `rodents` mają ten sam adres, inny od adresu `bunnies`. Zmienna `rodents` to alias dla `rats`, więc przypisanie tak naprawdę znaczy tylko tyle:

```
rats = bunnies;
```

Zatem zmiennej `rats` przypisujemy wartość zmiennej `bunnies`. Krótko mówiąc, referencję można ustawić w deklaracji, inicjalizując ją, ale nie można jej później niczego przypisać.

Załóżmy, że wypróbowaliśmy następujący kod:

```
int rats = 101;
int * pt = &rats;
int & rodents = *pt;
int bunnies = 50;
pt = &bunnies;
```

Zainicjalizowanie `rodents` wartością `*pt` powoduje, że `rodents` odnosi się do `rats`. W efekcie zmiana `pt` tak, aby wskazywała `bunnies`, nie zmieni w niczym tego, że `rodents` odnosi się do `rats`.

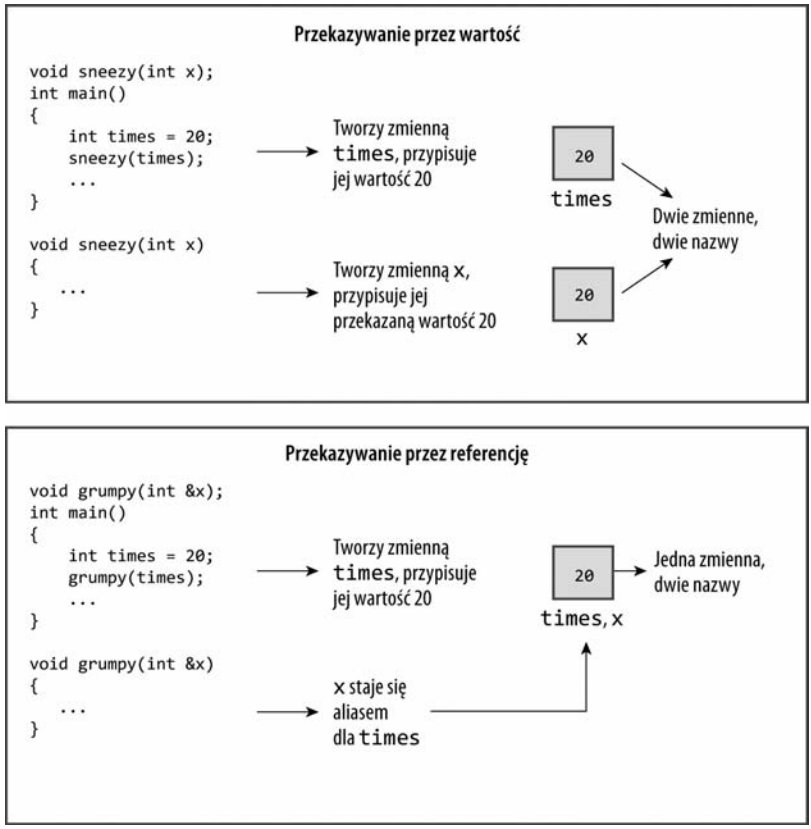
Referencje jako parametry funkcji

Najczęściej referencje służą jako parametry funkcji, powodując, że nazwa zmiennej widoczna w funkcji jest aliasem zmiennej programu wywołującego. Taka metoda przekazywania parametrów nazywana jest *przekazywaniem przez referencję*. Przekazywanie przez referencję pozwala wywołanej funkcji korzystać ze zmiennych funkcji wywołującej. Stanowi to zmianę w stosunku do C, gdzie dane są przekazywane wyłącznie przez wartość. Przekazywanie wartości powoduje, że wywołana funkcja działa na kopiach parametrów funkcji wywołującej (rysunek 8.2). Oczywiście C umożliwia przezwyciężenie tego ograniczenia przez użycie wskaźników.

Porównajmy zasady użycia referencji i wskaźników, wykorzystując do tego typowy problem — zamianę wartości dwóch zmiennych. Funkcja zamieniająca te wartości musi być w stanie zmienić wartości programu wywołującego. Oznacza to, że nie możemy przyjąć klasycznego przekazywania przez wartość, gdyż zamienialibyśmy wartości kopii zmiennych. Jeśli przekażemy referencję, funkcja będzie działała na oryginalnych danych. Można też przekazać wskaźniki. Na listingu 8.4 pokazano wszystkie trzy metody, z niedziałającym w tym przypadku przekazaniem przez wartość; dzięki temu można porównać wszystkie opcje.

Listing 8.4. *swaps.cpp*

```
// swaps.cpp -- użycie referencji i wskaźników do zamiany wartości
#include <iostream>
void swapr(int & a, int & b);    // a, b to aliasy wartości int
void swapp(int * p, int * q);    // p, q to adresy wartości int
```



Rysunek 8.2. *Przekazywanie przez wartość i przez referencję*

```
void swapv(int a, int b);         // a, b to nowe zmienne
int main()
{
    using namespace std;
    int wallet1 = 300;
    int wallet2 = 350;

    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Zamiana wartości za pomocą referencji:\n";
    swapr(wallet1, wallet2);    // przekaż zmienne
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Zamiana wartości za pomocą wskaźników:\n";
    swapp(&wallet1, &wallet2); // przekaż adresy zmiennych
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;

    cout << "Próba zamiany przy przekazywaniu przez wartość:\n";
    swapv(wallet1, wallet2);    // przekazanie wartości zmiennych
    cout << "wallet1 = $" << wallet1;
    cout << " wallet2 = $" << wallet2 << endl;
    return 0;
```

```
}

void swapr(int & a, int & b)      // użycie referencji
{
    int temp;

    temp = a;         // użycie a, b jako wartości zmiennych
    a = b;
    b = temp;
}

void swapp(int * p, int * q)      // użycie wskaźników
{
    int temp;

    temp = *p;        // użycie *p, *q jako wartości zmiennych
    *p = *q;
    *q = temp;
}

void swapv(int a, int b)          // próba użycia wartości
{
    int temp;

    temp = a;         // użycie a, b jako wartości zmiennych
    a = b;
    b = temp;
}
```

Oto wyniki działania programu z listingu 8.4:

```
wallet1 = $300 wallet2 = $350                    <-- wartości oryginalne
Zamiana wartości za pomocą referencji:
wallet1 = $350 wallet2 = $300                    <-- wartości po zamianie
Zamiana wartości za pomocą wskaźników:
wallet1 = $300 wallet2 = $350                    <-- ponownie po zamianie
Próba zamiany przy przekazywaniu przez wartość:
wallet1 = $300 wallet2 = $350                    <-- zamiana nieudana
```

Zgodnie z oczekiwaniami użycie wskaźników i referencji zadziałało prawidłowo, natomiast przekazanie przez wartość nie przyniosło żądanego efektu.

Uwagi o programie

Najpierw zwróćmy uwagę na sposób wywołania poszczególnych funkcji na listingu 8.4:

```
swapr(wallet1, wallet2);    // przekaż zmienne
swapp(&wallet1, &wallet2); // przekaż adresy zmiennych
swapv(wallet1, wallet2);    // przekazanie wartości zmiennych
```

Przekazanie przez referencję (`swapr(wallet1, wallet2)`) oraz przekazanie przez wartość (`swapv(wallet1, wallet2)`) wyglądają identycznie. Jedyny sposób zauważenia, że w przypadku `swapr()` mamy do czynienia z referencjami, to spojrzenie na prototyp funkcji i jej definicję. Z drugiej strony obecność operatora adresu od razu ujawnia nam, gdzie mamy do czynienia z przekazywaniem przez wskaźnik (`swapp(&wallet1, &wallet2)`). Przypomnijmy, że deklaracja typu `int *p` oznacza, że `p` jest wskaźnikiem danej typu `int`, zatem parametr odpowiadający `p` powinien być adresem, na przykład `&wallet1`.

Teraz porównajmy kod funkcji `swapr()` (przekazanie przez referencję) i `swapv()` (przekazanie przez wartość). Jedyna zewnętrzna różnica leży w sposobie deklaracji parametrów funkcji:

```
void swapr(int & a, int & b)
void swapv(int a, int b)
```

Wewnętrzna różnica to oczywiście to, że zmienne `a` i `b` są aliasami `wallet1` i `wallet2`, wobec czego zamiana wartości `a` i `b` powoduje też zamianę wartości `wallet1` i `wallet2`. Jednak w funkcji `swapv()` zmienne `a` i `b` to nowe zmienne będące kopiami `wallet1` i `wallet2`, wobec czego zamiana wartości `a` i `b` nie wpływa na `wallet1` i `wallet2`.

Na koniec porównajmy funkcję `swapr()` (przekazywanie przez referencję) ze `swapp()` (przekazywanie przez wskaźnik). Pierwsza różnica to inna deklaracja funkcji:

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

Druga różnica polega na tym, że wersja wskaźnikowa wymaga użycia operatora dereferencji w przypadku korzystania z `p` i `q`.

Wcześniej mówiliśmy, że zmienne referencyjne należy inicjalizować w chwili ich definiowania. Wywołanie funkcji oznacza zainicjalizowanie parametrów funkcji wartościami argumentów wywołania funkcji. Więc parametry funkcji będące referencjami należy traktować jako zainicjalizowane parametrami przekazanymi w wywołaniu funkcji. Zatem wywołanie funkcji:

```
swapr(wallet1, wallet2);
```

powoduje zainicjalizowanie parametrów argumentami wywołania: `a` jest inicjalizowane `wallet1`, a `b` — `wallet2`.

Właściwości referencji

Użycie parametrów referencyjnych wiąże się z pewnymi specjalnymi warunkami, o których trzeba wiedzieć. Spójrzmy najpierw na listing 8.5. Używane są na nim dwie funkcje podnoszące parametr do sześcianu. Jedna pobiera parametr typu `double`, druga referencję `double`. Kod realizujący potęgowanie jest celowo nieco zagmatwany, gdyż chodzi o dokładne pokazanie pewnej rzeczy.

Listing 8.5. *cubes.cpp*

```
// cubes.cpp -- parametry zwykłe i referencyjne
#include <iostream>
double cube(double a);
double refcube(double &ra);
int main ()
{
    using namespace std;
    double x = 3.0;

    cout << cube(x);
    cout << " = sześcian " << x << endl;
    cout << refcube(x);
    cout << " = sześcian " << x << endl;
    return 0;
}

double cube(double a)
{
    a *= a * a;
    return a;
}
```

```
double refcube(double &ra)
{
    ra *= ra * ra;
    return ra;
}
```

Oto wynik działania programu z listingu 8.5:

```
27 = sześcian 3
27 = sześcian 27
```

Zwróćmy uwagę, że funkcja `refcube()` modyfikuje wartość zmiennej `x` z funkcji `main()`, a funkcja `cube()` nie; dlatego właśnie przekazywanie przez wartość jest standardem. Zmienna `a` jest inicjalizowana wartością `x`, ale zmiana `a` nie wpływa już na `x`. Funkcja `refcube()` korzysta z referencji, więc zmiany wartości `ra` automatycznie przenoszą się na `x`. Jeśli funkcja ma używać przekazanych jej wartości, ale ich nie modyfikować, a korzystamy z referencji, należy skorzystać z referencji stałej. Oto przykład, jak powinno to wyglądać w prototypie i nagłówku funkcji:

```
double refcube(const double &ra);
```

Jeśli zastosujemy taki zapis, przy próbie zmiany wartości `ra` kompilator zgłosi błąd.

Tak naprawdę, jeśli chodzi o napisanie funkcji podobnej do pokazanej tutaj (to znaczy z użyciem prostego typu liczbowego), zamiast z cokolwiek egzotycznego przekazywania przez referencję należy skorzystać ze zwykłego przekazania przez wartość. Parametry referencyjne stają się przydatne w przypadku większych porcji danych, na przykład struktur i klas.

Funkcje przekazujące parametry przez wartość, jak `cube()` z listingu 8.5, mogą używać wielu rodzajów parametrów wywołania. Na przykład poprawne są wszystkie poniższe wywołania:

```
double z = cube(x + 2.0);    // wyznaczenie x + 2.0, przekazanie wartości
z = cube(8.0);               // przekazanie wartości 8.0
int k = 10;
z = cube(k);                 // konwersja wartości k na typ double, przekazanie
double yo[3] = { 2.2, 3.3, 4.4};
z = cube(yo[2]);             // przekazanie wartości 4.4
```

Załóżmy, że spróbowalibyśmy przekazać podobne parametry do funkcji korzystającej z parametrów referencyjnych. Mogłoby się wydawać, że przekazywanie przez referencję jest bardziej ograniczone; w końcu `ra` jest alternatywną nazwą zmiennej, więc parametr wywołania powinien być zmienną. Zapis typu:

```
double z = refcube(x + 3.0);      // nie powinno się skompilować
```

nie wygląda zbyt dobrze, bo wyrażenie `x + 3.0` nie jest zmienną. Na przykład nie można przypisać takiemu wyrażeniu wartości:

```
x + 3.0 = 5.0;          // kompletne nieporozumienie
```

A co się stanie, jeśli spróbujemy użyć wywołania `refcube(x + 3.0)`? We współczesnym C++ jest to błąd i większość kompilatorów go zgłosi. A niektóre starsze wygenerują ostrzeżenie typu:

```
Ostrzeżenie: W wywołaniu refcube(double &) jako parametru 'ra' użyto wartości tymczasowej.
```

Powodem tak umiarkowanego protestu jest to, że dawniej C++ pozwalało przekazywać zmiennym referencyjnym wyrażenia. Czasami nadal jest to możliwe. Skoro `x + 3.0` nie jest zmienną typu `double`, program utworzy tymczasową zmienną bez nazwy i zainicjalizuje ją wyrażeniem `x + 3.0`. Parametr `ra` stanie się referencją tej zmiennej tymczasowej. Przyjrzyjmy się teraz zmiennym tymczasowym bliżej i sprawdźmy, kiedy są tworzone, a kiedy nie.

Zmienne tymczasowe, parametry referencyjne i const

Język C++ może wygenerować zmienną tymczasową, jeśli parametr wywołania nie pasuje do parametru referencyjnego. Obecnie C++ dopuszcza to tylko w przypadku parametrów referencyjnych z modyfikatorem `const`, ale i to nie zawsze. Przyjrzyjmy się teraz sytuacjom, w których C++ generuje zmienne tymczasowe, i zastanówmy się, czemu ograniczenie do referencji stałych jest słuszne.

Po pierwsze, kiedy jest tworzona zmienna tymczasowa? Jeśli parametr referencyjny jest stały, kompilator generuje zmienną tymczasową w dwojakich sytuacjach:

- kiedy parametr wywołania ma prawidłowy typ, ale nie jest *l-wartością*,
- kiedy parametr wywołania ma niewłaściwy typ, ale typ ten może być przekonwertowany na właściwy.

Czym jest *l-wartość*? Otóż l-wartości są to takie obiekty danych, do których można się odwołać. Przykładami są zmienna, element tablicy, pole struktury, referencja czy wskaźnik, do którego zastosowano dereferencję. Nie są l-wartościami literały (poza ciągami w cudzysłowach, które są wewnętrznie reprezentowane jako adresy pamięci) ani złożone wyrażenia. Pojęcie *l-wartości* w C pierwotnie odnoszono do tych jednostek programu, które mogą wystąpić po lewej stronie operatora przypisania, ale później ta definicja uległa modyfikacji w wyniku wprowadzenia słowa kluczowego `const`. Obecnie l-wartością nazwiemy zarówno zwyczajne zmienne modyfikowalne, jak i zmienne deklarowane ze słowem `const`, ponieważ do obu można się odwołać przez adres. Z tym że zwyczajna zmienna może zostać określona jako *l-wartość modyfikowalna*, a zmienna deklarowana ze słowem `const` jako *l-wartość niemodyfikowalna*.

Wróćmy do przykładu; załóżmy, że zmienimy definicję `refcube()` tak, aby jako parametr pobierała stałą referencję:

```
double refcube(const double &ra)
{
    return ra * ra * ra;
}
```

Teraz weźmy następujący kod:

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = {2.0, 5.0, 10.0, 12.0};
double c1 = refcube(side);            // ra to side
double c2 = refcube(lens[2]);         // ra to lens[2]
double c3 = refcube(rd);              // ra to rd to side
double c4 = refcube(*pd);             // ra to *pd to side
double c5 = refcube(edge);            // ra to zmienna tymczasowa
double c6 = refcube(7.0);             // ra to zmienna tymczasowa
double c7 = refcube(side + 10.0);     // ra to zmienna tymczasowa
```

Parametry `side`, `lens[2]`, `rd` i `*pd` to obiekty danych typu `double` mające nazwy, więc można utworzyć do nich referencje i zbędne są jakiekolwiek zmienne tymczasowe (przypomnijmy, że element tablicy zachowuje się jak zmienna odpowiedniego typu). Jednak już `edge` jest wprawdzie zmienną, ale niewłaściwego typu. Referencja `double` nie może dotyczyć zmiennej `long`. Parametry `7.0` i `side + 10.0` są z kolei właściwego typu, ale nie są nazwanymi obiektami danych. W każdym z tych przypadków kompilator wygeneruje tymczasową zmienną anonimową i uczyni `ra` referencją do niej. Te zmienne tymczasowe istnieją tak długo, jak długo działa funkcja, ale później kompilator może je usunąć z pamięci.

Czemu zatem działa to dobrze w przypadku referencji stałych, a inaczej nie? Przypomnijmy funkcję `swapr()` z listingu 8.4:

```
void swapr(int & a, int & b)      // użycie referencji
{
    int temp;

    temp = a;          // użycie a, b jako wartości zmiennych
    a = b;
    b = temp;
}
```

Co by się stało, gdybyśmy w starym, bardziej liberalnym C++ wykonali poniższy kod?

```
long a = 3, b = 5;
swapr(a, b);
```

Mamy do czynienia z niezgodnością typów, więc kompilator utworzyłby dwie tymczasowe zmienne `int`, zainicjalizował je wartościami 3 i 5, a następnie zamienił wartości obu tych zmiennych tymczasowych, pozostawiając `a` i `b` niezmienione.

Jeśli zadaniem funkcji mającej parametry referencyjne jest zmiana przekazanych wartości, tworzenie zmiennych tymczasowych to uniemożliwia. Rozwiązaniem jest uniemożliwienie tworzenia zmiennych tymczasowych w takich sytuacjach i to właśnie teraz jest zapisane w standardzie C++. Jednak niektóre kompilatory w takiej sytuacji nadal generują ostrzeżenie, a nie błąd, należy więc zachować ostrożność.

Teraz zastanówmy się nad funkcją `refcube()`. Jej zadaniem jest po prostu użycie przekazanych wartości, a nie ich modyfikowanie. Wobec tego wykorzystanie zmiennych tymczasowych w niczym nie przeszkadza, a funkcja może obsłużyć szerszy wachlarz typów parametrów. Jeśli zatem z deklaracji wynika, że referencja jest stała, C++ generuje w razie potrzeby zmienne tymczasowe. Funkcje C++ z parametrami `const`, którym przekazano parametry wywołania innych typów, zachowują się podobnie jak w przypadku przekazywania przez wartość, co oznacza niemożność zmiany oryginalnych danych i użycie tymczasowej zmiennej na wartość.

Uwaga

Jeśli parametr wywołania funkcji nie jest l-wartością lub nie pasuje co do typu do odpowiedniego stałego parametru referencyjnego, C++ tworzy zmienną anonimową właściwego typu i przypisuje jej wartość przekazanego parametru. Od tej chwili parametr odnosi się do takiej zmiennej anonimowej.

Kiedy to tylko możliwe, należy używać `const`.

Istnieją trzy ważkie powody deklarowania parametrów referencyjnych jako stałych:

- Użycie `const` chroni nas przed błędami programistycznymi powodującymi niezamierzoną zmianę wartości.
- Użycie `const` pozwala funkcji przetwarzać wartości stałe i niestałe, a pominięcie w prototypie `const` oznacza, że funkcja może przetwarzać tylko dane niebędące stałymi.
- Użycie referencji stałej pozwala funkcji generować tymczasowe zmienne i ich używać.

Parametry referencyjne należy deklarować jako stałe zawsze, kiedy tylko jest to możliwe.

W C++11 wprowadzono drugi rodzaj referencji: *referencję do r-wartości*, która (nomen omen) może odnosić się do r-wartości. Deklaruje się ją za pomocą symbolu `&&`:

```
double && rref = std::sqrt(36.00);    // niedozwolone dla &
double j = 15.0;
double && jref = 2.0* j + 18.5;       // niedozwolone dla &
std::cout << rref << '\n';      // wypisanie 6.0
std::cout << jref << '\n';      // wypisanie 48.5
```

Referencja r-wartościowa została wprowadzona przede wszystkim dla umożliwienia tworzenia efektywniejszych implementacji pewnych operacji w bibliotekach. W rozdziale 18. będziemy omawiać sposób stosowania referencji r-wartości w implementacjach tak zwanej *semantyki przeniesienia* (ang. *move semantics*); pierwotny typ referencyjny (ten z deklaracją z pojedynczym znakiem `&`) jest teraz określany mianem *referencji l-wartościowej* albo *referencji do l-wartości*.

Użycie referencji do struktur

Referencje świetnie współpracują ze strukturami i klasami. Początkowo referencje wprowadzono właśnie na potrzeby tych dwóch typów, a nie na potrzeby wbudowanych typów podstawowych.

Referencji do struktur używa się w kontekście parametrów funkcji tak samo jak referencji do zmiennych typów podstawowych — wystarczy w deklaracji parametru odnoszącego się do struktury dodać operator referencji, `&`. Weźmy dla przykładu następującą definicję struktury:

```
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};
```

Oraz funkcję operującą na strukturze takiego typu, udostępnianej przez referencję; jej prototyp wyglądałby tak:

```
void set_pc(free_throws & ft);      // referencja do struktury
```

Jeśli funkcja nie ma modyfikować wartości struktury, warto uzupełnić deklarację parametru słowem `const`:

```
void display(const free_throws & ft);    // nie zmienia wartości struktury
```

Właśnie tak działa program z listingu 8.6. Ciekawym rozwiązaniem jest też zwracanie przez funkcję referencji do struktury. Działa to nieco inaczej niż zwracanie zwykłej struktury i trzeba pamiętać o zasięgu zwracanej referencji (wrócimy do tego niebawem).

Listing 8.6. *strc_ref.cpp*

```
// strc_ref.cpp -- użycie referencji do struktur
#include <iostream>
#include <string>
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};

void display(const free_throws & ft);
void set_pc(free_throws & ft);
free_throws & accumulate(free_throws & target, const free_throws & source);

int main()
{
    // inicjalizacje częściowe -- reszta składowych jest zerowana
```

```
    free_throws one = {"Ifelsa Branch", 13, 14};
    free_throws two = {"Andor Knott", 10, 16};
    free_throws three = {"Minnie Max", 7, 9};
    free_throws four = {"Whily Looper", 5, 9};
    free_throws five = {"Long Long", 6, 14};
    free_throws team = {"Throwgoods", 0, 0};
    // bez inicjalizacji
    free_throws dup;

    set_pc(one);
    display(one);
    accumulate(team, one);
    display(team);

    // wartości zwracanych użyjemy w roli argumentów wywołania
    display(accumulate(team, two));
    accumulate(accumulate(team, three), four);
    display(team);

    // wartości zwracane użyte w przypisaniach:
    dup = accumulate(team,five);
    std::cout << "Statystyka dla team:\n";
    display(team);
    std::cout << "Statystyka dla dup po przypisaniu:\n";
    display(dup);
    set_pc(four);

    // źle skonstruowane przypisanie
    accumulate(dup,five) = four;
    std::cout << "Statystyka dla dup po omyłkowym przypisaniu:\n";
    display(dup);
    return 0;
}
void display(const free_throws & ft)
{
    using std::cout;
    cout << "Imię: " << ft.name << '\n';
    cout << "Trafionych: " << ft.made << '\t';
    cout << "Rzutów: " << ft.attempts << '\t';
    cout << "Skuteczność: " << ft.percent << '\n';
}
void set_pc(free_throws & ft)
{
    if (ft.attempts != 0)
        ft.percent = 100.0f *float(ft.made)/float(ft.attempts);
    else
        ft.percent = 0;
}
free_throws & accumulate(free_throws & target, const free_throws & source)
{
    target.attempts += source.attempts;
    target.made += source.made;
    set_pc(target);
    return target;
}
```

Oto wynik działania programu z listingu 8.6:

```
Imię: Ifelsa Branch
Trafionych: 13 Rzutów: 14 Skuteczność: 92.8571
Imię: Throwgoods
Trafionych: 13 Rzutów: 14 Skuteczność: 92.8571
```

```
Imię: Throwgoods
Trafionych: 23 Rzutów: 30 Skuteczność: 76.6667
Imię: Throwgoods
Trafionych: 35 Rzutów: 48 Skuteczność: 72.9167
Statystyka dla team:
Imię: Throwgoods
Trafionych: 41 Rzutów: 62 Skuteczność: 66.129
Statystyka dla dup po przypisaniu:
Imię: Throwgoods
Trafionych: 41 Rzutów: 62 Skuteczność: 66.129
Statystyka dla dup po omyłkowym przypisaniu:
Imię: Whily Looper
Trafionych: 5 Rzutów: 9 Skuteczność: 55.5556
```

Uwagi o programie

Program rozpoczyna się od inicjalizacji kilku struktur. Pamiętamy, że jeśli inicjalizator zawiera mniej elementów niż liczba składowych inicjalizowanej struktury, pozostałe składowe będą inicjalizowane zerami (tu wyzerowane zostaną składowe `percent`). Pierwsze wywołanie funkcji w programie to:

```
set_pc(one);
```

Ponieważ parametrem `ft` funkcji `set_pc()` jest referencja, w ciele funkcji nazwa `ft` odnosi się do struktury `one`, a kod funkcji `set_pc()` ustawia wartość składowej `one.percent`. Przekazywanie przez wartość nie zadziała w tym przypadku, ponieważ funkcja zmieniłaby wtedy jedynie wartość składowej lokalnej (tymczasowej) kopii `one`. Alternatywą dla referencji jest (wiemy to z poprzedniego rozdziału) parametr typu wskaźnikowego, ale jest on odrobinę bardziej skomplikowany w obsłudze:

```
set_pcp(&one);    // wersja z parametrem typu wskaźnikowego -- &one zamiast one
...
void set_pcp(free_throws * pt)
{
    if (pt->attempts != 0)
        pt->percent = 100.0f *float(pt->made)/float(pt->attempts);
    else
    pt->percent = 0;
}
```

Następne wywołanie funkcji w programie to:

```
display(one);
```

Ponieważ funkcja `display()` wypisuje zawartość struktury bez modyfikowania wartości jej składowych, w deklaracji parametru funkcji umieszczono słowo `const`, więc parametr jest niemodyfikowalną referencją. W tym przypadku akurat można by przekazać strukturę przez wartość, ale przekazywanie przez referencję jest bardziej efektywne pod względem czasu wykonania i zajętości pamięci (przekazanie przez wartość oznaczałoby wykonanie tymczasowej kopii argumentu dla potrzeb funkcji).

Kolejne wywołanie funkcji w programie:

```
accumulate(team, one);
```

Funkcja `accumulate()` przyjmuje dwa argumenty w postaci struktur. Jej zadaniem jest dodanie wartości składowych `attempts` i `made` drugiego argumentu do odpowiednich składowych pierwszego argumentu. Modyfikowana jest więc tylko pierwsza z przekazanych struktur i reprezentujący ją parametr jest deklarowany jako zwykła referencja, a drugi parametr to referencja niemodyfikowalna (`const`):

```
free_throws & accumulate(free_throws & target, const free_throws & source);
```

A co z wartością zwracaną? W omawianym wywołaniu funkcji nie jest ona używana, więc równie dobrze funkcja mogłaby deklarować typ zwracany jako `void`. Ale już następne wywołanie w programie to:

```
display(accumulate(team, two));
```

O co tu chodzi? Prześledźmy operacje na zmiennej strukturowej `team`. Jest ona pierwszym argumentem wywołania funkcji `accumulate()`, co oznacza, że w ciele funkcji jest widoczna jako obiekt `target`. Funkcja `accumulate()` modyfikuje zmienną `team`, a następnie zwraca referencję do tejże zmiennej. Zauważmy, że instrukcja zwrócenia wartości wygląda tak:

```
return target;
```

Nic tu jawnie nie sygnalizuje, że dochodzi do zwrócenia referencji. Wniosek ten pochodzi wyłącznie z analizy nagłówka funkcji (a także jej prototypu):

```
free_throws & accumulate(free_throws & target, const free_throws & source)
```

Gdyby typ wartości zwracanej był deklarowany jako `free_throws` zamiast `free_throws &`, ta sama instrukcja `return` zwróciłaby kopię obiektu `target` (czyli kopię `team`). Ale skoro wartość zwracana jest referencją, to znaczy, że wartość zwrócona z powyższego wywołania to obiekt `team` tożsamy z tym przekazanym do wywołania `accumulate()`.

A co dalej? Funkcja `accumulate()` zwraca wartość, która jest użyta jako pierwszy argument wywołania funkcji `display()`, co oznacza, że funkcja `display()` otrzymuje w wywołaniu obiekt `team`. Ponieważ parametr funkcji `display()` jest referencją, w ciele funkcji nazwa `ft` odnosi się w istocie do obiektu `team`. Słowem, funkcja w tym wywołaniu wypisze zawartość `team`. W sumie instrukcja:

```
display(accumulate(team, two));
```

jest równoważna instrukcjom:

```
accumulate(team, two);
display(team);
```

Analogiczna analiza dotyczy instrukcji:

```
accumulate(accumulate(team, three), four);
```

która mogłaby równie dobrze zostać zapisana jako sekwencja instrukcji:

```
accumulate(team, three);
accumulate(team, four);
```

Następny wiersz programu używa operatora przypisania:

```
dup = accumulate(team, five);
```

Zgodnie z oczekiwaniem w `dup` ląduje kopia zawartości `team`.

Na koniec program korzysta z funkcji `accumulate()` w sposób niezgodny z jej przeznaczeniem:

```
accumulate(dup, five) = four;
```

Taka instrukcja, to jest przypisanie wartości do wywołania funkcji, zostanie z powodzeniem skompilowana, ponieważ wartością zwracaną przez funkcję jest referencja. Gdyby funkcja `accumulate()` zwracała strukturę przez wartość, kod nie skompilowałby się. W przypadku referencji wartością zwracaną jest referencja do `dup`, co oznacza, że powyższa instrukcja jest tożsama z:

```
accumulate(dup, five);    // dodanie danych five do dup
dup = four;    // zamazanie zawartości dup zawartością four
```

Druga instrukcja zasadniczo unieważnia wynik działania poprzedniej i choćby ze względu na to można sądzić, że pierwotne użycie `accumulate()` z przypisaniem do wartości zwracanej było niepoprawne.

Po co zwracać referencje?

Wróćmy jeszcze do zagadnienia zwracania referencji z wywołania funkcji. W przypadku zwracania zwyczajnej wartości mamy do czynienia z odpowiednikiem przekazywania parametru przez wartość: dochodzi do obliczenia wartości wyrażenia instrukcji `return` i przekazania obliczonej wartości do miejsca wywołania funkcji. Koncepcyjnie dochodzi wtedy do skopiowania wartości zwracanej do tymczasowej lokalizacji, w której jest dostępna dla programu wywołującego. Weźmy taki kod:

```
double m = sqrt(16.0);
cout << sqrt(25.0);
```

W pierwszej instrukcji dochodzi do skopiowania wartości `4.0` do tymczasowego obiektu, który następnie jest kopiowany do zmiennej `m`. W drugiej instrukcji wartość `5.0` jest również kopiowana do lokalizacji tymczasowej, z której jest następnie przekazywana do `cout` (to opis koncepcyjny; w praktyce kompilatory optymalizujące wywołania mogą skonsolidować poszczególne kroki zwracania wartości i eliminować operacje kopiowania). Teraz weźmy instrukcję:

```
dup = accumulate(team,five);
```

Gdyby `accumulate()` zwracała strukturę zamiast referencji do struktury, oznaczałoby to skopiowanie całej struktury do tymczasowej lokalizacji, a stamtąd do zmiennej `dup`. Ale przy zwracaniu referencji mamy bezpośrednie kopiowanie `team` do `dup`, co jest operacją dużo efektywniejszą.

> **Uwaga**
> Funkcja zwracająca referencję jest tak naprawdę aliasem zmiennej, której referencja dotyczy.

Ostrożnie ze zwracaniem referencji

Najważniejsze, co wiąże się ze zwracaniem referencji, to unikanie zwracania referencji do obszaru pamięci, który zostanie zwolniony w chwili zakończenia wykonywania funkcji. Chcemy uniknąć kodu podobnego do poniższego:

```
const free_throws & clone2(free_throws & ft)
{
    free_throws newguy;          // pierwszy krok ku poważnemu błędowi
    newguy = ft;    // kopiowanie danych
    return newguy;          // zwracanie referencji do kopii
}
```

Pojawi się tu niefortunny efekt zwracania referencji do zmiennej tymczasowej `newguy`, która przestanie istnieć w chwili zakończenia wykonywania funkcji (w rozdziale 9. omówimy trwałość różnego rodzaju zmiennych). Analogicznie trzeba unikać zwracania wskaźników na takie zmienne.

Najprostszym sposobem uniknięcia opisanego problemu jest zwracanie referencji przekazanej wcześniej funkcji jako parametru. Parametr referencyjny odnosi się do danych używanych przez funkcję wywołującą, wobec czego zwracana referencja odniesie się do tych samych danych. Tak działa funkcja `accumulate()` z listingu 8.6.

Druga metoda to alokacja pamięci operatorem `new`. Widzieliśmy już przykłady, gdzie operator `new` rezerwował miejsce na łańcuch, a funkcja zwracała wskaźnik na to miejsce. Oto jak można zrobić coś takiego z referencjami:

```
const free_throws & clone(free_throws & ft)
{
    free_throws * pt = new free_throws();
    *pt = ft; // kopiowanie informacji
    return *pt;         // zwrócenie referencji do kopii
}
```

Pierwsza instrukcja tworzy strukturę `free_throws` bez nazwy. Wskaźnik `pt` wskazuje strukturę, więc `*pt` jest strukturą. Wydaje się, że zwracana jest struktura, ale deklaracja funkcji mówi, że tak naprawdę zwracana jest referencja do struktury. Można byłoby zatem funkcji tej użyć następująco:

```
free_throws & jolly = clone(three);
```

Powoduje to, że `jolly` jest referencją do nowej struktury. Istnieje tu jednak pewien problem — należałoby użyć `delete` do zwolnienia pamięci zaalokowanej przez `new`, kiedy nie będzie już potrzebna. Ale wywołanie `clone()` ukrywa użycie operatora `new`, a zwracany obiekt jest referencją, a nie wskaźnikiem, przez co łatwo zapomnieć o konieczności późniejszego użycia `delete`. W tego typu przypadkach nieocenionym ułatwieniem są szablony automatyzujące zwalnianie pamięci, takie jak omawiane w rozdziale 16. szablony `auto_ptr` albo jeszcze lepiej `unique_ptr` z C++11.

Po co używać const przy zwracaniu referencji?

Na listingu 8.6 występowała taka instrukcja:

```
accumulate(dup,five) = four;
```

W jej wyniku najpierw zakumulowaliśmy wartość `five` w strukturze `dup` tylko po to, żeby po wyjściu z funkcji stracić obliczone wartości poprzez nadpisanie wartości `dup` wartością zmiennej `four`. Dlaczego w ogóle udało się skompilować taką instrukcję? Otóż przypisanie wymaga modyfikowalnej l-wartości po lewej stronie: wyrażenie po lewej stronie operatora przypisania powinno identyfikować blok pamięci, który można modyfikować. W tym przypadku funkcja zwracała referencję do `dup`, a taka referencja identyfikuje modyfikowalny blok pamięci, więc cała instrukcja jest dla kompilatora poprawna.

Zwyczajne (niereferencyjne) typy wartości zwracanych to z kolei *r-wartości*, które nie mają adresu (nie identyfikują bloku pamięci). Takie wyrażenia mogą występować po prawej stronie operatora przypisania, ale nie można ich umieszczać po stronie lewej. Do r-wartości zaliczamy też literały takie jak `10.0` oraz wyrażenia w rodzaju `x + y`. Jest jasne, że próby pobrania adresu literału w rodzaju `10.0` nie mają sensu, ale dlaczego zwyczajna wartość zwracana z funkcji również jest r-wartością? Otóż wartość zwracana, dla przypomnienia, jest umieszczana w tymczasowej lokalizacji pamięci, która niekoniecznie istnieje choćby do końca wykonywania bieżącej instrukcji.

Załóżmy, że zamierzamy użyć referencji jako wartości zwracanej, ale chcielibyśmy zapobiec niepożądanym przypadkom użycia, takim jak przypisanie do wartości zwracanej z funkcji `accumulate()`. Aby to zrobić, wystarczy zadeklarować typ wartości zwracanej jako referencję niemodyfikowalną (ze słowem `const`):

```
const free_throws &
accumulate(free_throws & target, const free_throws & source);
```

Wartość zwracana jest teraz l-wartością, ale niemodyfikowalną, i instrukcja przypisania do wartości zwracanej staje się niepoprawna składniowo:

```
accumulate(dup,five) = four; // niedozwolone dla niemodyfikowalnej referencji zwracanej z funkcji
```

A co z pozostałymi wywołaniami funkcji w programie? Nawet po oznaczeniu typu wartości zwracanej słowem `const` poniższa instrukcja wciąż będzie poprawna:

```
display(accumulate(team, two));
```

Poprawność wynika z tego, że parametr funkcji `display()` jest również deklarowany jako referencja niemodyfikowalna (`const free_thows &`). Ale już poniższe wywołanie będzie niepoprawne, ponieważ pierwszy parametr `accumulate()` to referencja modyfikowalna:

```
accumulate(accumulate(team, three), four);
```

Czy to duże utrudnienie? Nie w tym przypadku, skoro powyższe wywołanie można łatwo przepisać na sekwencję:

```
accumulate(team, three);
accumulate(team, four);
```

I rzecz jasna wciąż mamy możliwość używania wartości zwracanej funkcji `accumulate()` po prawej stronie instrukcji przypisania.

Pominięcie `const` pozwala pisać krótszy, ale trudniejszy w zrozumieniu kod.

Zwykle lepiej unikać włączania do programu trudnych do zrozumienia cech, które później stają się przyczynkiem do powstawania trudnych do zrozumienia błędów. Uczynienie typu zwracanego referencją stałą chroni nas przed pokusą nieeleganckiego pójścia na skróty. Czasami warto też pominąć słowo `const`; przykładem tego jest omawiany w rozdziale 11. przeciążony operator <<.

Użycie referencji z obiektami

Klasyczną praktyką C++ jest przekazywanie do funkcji obiektów przez referencje. Referencji używa się na przykład w przypadku funkcji, którym przekazuje się obiekty klas `string`, `ostream`, `istream`, `ofstream` oraz `ifstream`.

Przyjrzyjmy się przykładowi, w którym użyto klasy `string` oraz pokazano pewne techniki programowania — niekoniecznie godne naśladowania. Chodzi o stworzenie funkcji dodającej podany łańcuch na koniec innego łańcucha. Listing 8.7 zawiera trzy funkcje, które realizują to zadanie, jedna z tych funkcji jest jednak źle zdefiniowana i może spowodować awarię programu, a nawet program może się nie skompilować.

Listing 8.7. *strquote.cpp*

```
// strquote.cpp -- różne techniki programowania
#include <iostream>
#include <string>
using namespace std;
string version1(const string & s1, const string & s2);
const string & version2(string & s1, const string & s2);  // efekty uboczne
const string & version3(string & s1, const string & s2);  // zła technika

int main()
{
    string input;
    string copy;
    string result;

    cout << "Podaj łańcuch: ";
    getline(cin, input);
    copy = input;
    cout << "Wprowadzony łańcuch: " << input << endl;
```

```
    result = version1(input, "***");
    cout << "Łańcuch po wzbogaceniu: " << result << endl;
    cout << "Oryginalny łańcuch: " << input << endl;

    result = version2(input, "###");
    cout << "Łańcuch po wzbogaceniu: " << result << endl;
    cout << "Oryginalny łańcuch: " << input << endl;

    cout << "Przywrócenie oryginalnego łańcucha.\n";
    input = copy;
    result = version3(input, "@@@");
    cout << "Łańcuch po wzbogaceniu: " << result << endl;
    cout << "Oryginalny łańcuch: " << input << endl;

    return 0;
}

string version1(const string & s1, const string & s2)
{
    string temp;

    temp = s2 + s1 + s2;
    return temp;
}

const string & version2(string & s1, const string & s2)    // efekty uboczne
{
    s1 = s2 + s1 + s2;
// można bezpiecznie zwrócić referencję przekazaną do funkcji
    return s1;
}

const string & version3(string & s1, const string & s2)    // zła technika
{
    string temp;

    temp = s2 + s1 + s2;
// zwracanie referencji do obiektu lokalnego jest niebezpieczne
    return temp;

}
```

Oto wynik uruchomienia programu z listingu 8.7:

```
Podaj łańcuch: To nie moja wina.
Wprowadzony łańcuch: To nie moja wina.
Łańcuch po wzbogaceniu: ***To nie moja wina.***
Oryginalny łańcuch: To nie moja wina.
Łańcuch po wzbogaceniu: ###To nie moja wina.###
Oryginalny łańcuch: ###To nie moja wina.###
Przywrócenie oryginalnego łańcucha.
```

W tym momencie działanie programu jest przerywane.

Uwagi o programie

Wersja 1. funkcji z listingu 8.7 jest najprostsza:

```
string version1(const string & s1, const string & s2)
{
    string temp;
```

```
    temp = s2 + s1 + s2;
    return temp;
}
```

Funkcja ta ma dwa parametry typu `string`. Do tworzenia nowego łańcucha mającego pożądane właściwości też używamy klasy `string`. Zauważmy, że oba parametry funkcji są referencjami stałymi. Funkcja ostatecznie da taki sam wynik, jaki dałaby, gdybyśmy przekazali jej obiekty `string`:

```
string version4(string s1, string s2)    // zadziałałaby tak samo
```

W tym przypadku `s1` i `s2` byłyby całkiem nowymi obiektami `string`. Zatem użycie referencji jest wydajniejsze, gdyż nie trzeba tworzyć nowych obiektów i kopiować do nich danych z obiektów oryginalnych. Użycie kwalifikatora `const` wskazuje, że funkcja użyje oryginalnych łańcuchów, ale nie będzie ich modyfikowała.

Obiekt `temp` to nowy obiekt lokalny dla funkcji `version1()`, który znika w chwili zakończenia działania funkcji. Zwrócenie `temp` jako referencji nie zadziała, więc funkcja jest typu `string`. Wobec tego zawartość zmiennej `temp` zostanie skopiowana w tymczasowe miejsce przeznaczone na wartości zwracane. Dalej, w funkcji `main()`, treść zwracanej wartości jest kopiowana do łańcucha o nazwie `result`:

```
result = version1(input, "***");
```

Przekazywanie łańcucha w formacie C jako referencji obiektu string

Być może część czytelników zauważyła pewną ciekawą cechę funkcji `version1()`: oba parametry, `s1` i `s2`, są typu `const string &`, ale przekazane parametry aktualne, `input` i `"***"`, są odpowiednio typów `string` i `const char *`. Parametr `input` jest typu `string`, więc bezproblemowo może się do niego odwoływać `s1`. Jak jednak program ma potraktować przekazanie wskaźnika `char` jako referencji do obiektu `string`?

Dzieją się tutaj dwie rzeczy. Po pierwsze, klasa `string` zawiera definicję konwersji typu `char*` na typ `string`, dzięki czemu możliwa jest inicjalizacja obiektu `string` łańcuchem w formacie języka C. Po drugie, trzeba pamiętać o omawianej wcześniej w tym rozdziale właściwości referencji `const` używanych jako parametry. Załóżmy, że typ argumentu nie pasuje do typu referencji, ale możliwa jest odpowiednia konwersja. W takiej sytuacji program utworzy tymczasową zmienną odpowiedniego typu, zainicjalizuje ją przekonwertowaną wartością i przekaże uzyskaną tak referencję do zmiennej tymczasowej. Wcześniej w tym rozdziale widzieliśmy na przykład, że parametr `const double &` może tak obsłużyć parametr typu `int`. Analogicznie parametr `const string &` może obsłużyć w ten sposób parametr `char *` lub `const char *`.

Wynika stąd, że jeśli parametr jest typu `const string &`, argument użyty w wywołaniu funkcji może być obiektem `string` lub łańcuchem w formacie C — literałem ujętym w cudzysłów, tablicą znakową lub wskaźnikiem `char`. Dlatego prawidłowo zadziała wywołanie:

```
result = version1(input, "***");
```

Funkcja `version2()` nie tworzy tymczasowego obiektu `string`, ale bezpośrednio modyfikuje łańcuch oryginalny:

```
const string & version2(string & s1, const string & s2)    // efekty uboczne
{
    s1 = s2 + s1 + s2;
// można bezpiecznie zwrócić referencję przekazaną do funkcji
    return s1;
}
```

Funkcja może zmieniać `s1`, gdyż parametr ten, w przeciwieństwie do `s2`, został zadeklarowany bez słowa kluczowego `const`.

Parametr s1 jest referencją obiektu z funkcji main() (obiektu input) i można bezpiecznie zwrócić s1 jako referencję. Skoro s1 jest referencją zmiennej input, wiersz:

```
result = version2(input, "###");
```

jest równoważny następującemu zapisowi:

```
version2(input, "###");  // input zmienione bezpośrednio w version2()
result = input;          // referencja do s1 to referencja do input
```

Jednak, jako że s1 jest referencją do input, wywołanie tej funkcji ma efekt uboczny w postaci zmiany także obiektu input:

```
Oryginalny łańcuch: To nie moja wina.
Łańcuch po wzbogaceniu: ###To nie moja wina.###
Oryginalny łańcuch: ###To nie moja wina.###
```

Jeśli zatem nie chcemy, aby oryginalny łańcuch był modyfikowany, taka technika projektowania jest nieprawidłowa.

Trzecia wersja funkcji z listingu 8.7 stanowi ostrzeżenie, czego nie należy robić:

```
const string & version3(string & s1, const string & s2)    // zła technika
{
    string temp;

    temp = s2 + s1 + s2;
// zwracanie referencji do obiektu lokalnego jest niebezpieczne
    return temp;
}
```

Pojawia się tu poważny błąd polegający na zwracaniu referencji do zmiennej zadeklarowanej lokalnie w funkcji version3(). Funkcja daje się skompilować (nawet bez ostrzeżeń)[2], ale przy próbie wykonania tej funkcji działanie programu jest przerywane. Powodem błędu jest przypisanie:

```
result = vesion3(input, "@@@");
```

w którym program próbuje odwołać się do nieużywanego już obszaru pamięci.

Obiekty po raz wtóry — obiekty, dziedziczenie i referencje

Klasy ostream i ofstream mają ciekawą cechę bycia referencjami „wprzód". Jak mówiliśmy w rozdziale 6., obiekty typu ofstream mogą używać metod ostream, co pozwala na wprowadzanie i zapisywanie danych z i do pliku w takiej samej postaci, jak to się robi w przypadku konsoli. Cechą języka, która umożliwia przekazywanie właściwości jednej klasy innej klasie, jest *dziedziczenie*, dokładnie omówione w rozdziale 13. Na razie tylko kilka słów wstępu: klasa ostream to *klasa bazowa* (bo klasa ofstream oparta jest na niej), a klasa ofstream to *klasa pochodna*. Klasa pochodna dziedziczy metody klasy bazowej, co znaczy, że obiekt ofstream może korzystać z takich cech klasy bazowej jak metody formatujące precision() czy setf().

Innym aspektem dziedziczenia jest to, że referencja klasy bazowej może odnosić się do obiektu klasy pochodnej bez konieczności rzutowania. Praktyczny efekt jest taki, że można zdefiniować funkcję mającą jako parametr referencję klasy bazowej i funkcja ta będzie mogła używać w tym parametrze

[2] Tak naprawdę zależy to od kompilatora — na przykład w wersji 6. kompilatora Borland na etapie kompilacji zgłaszany jest błąd *Attempting to return a reference to local variable 'temp'*, czyli *Próba zwrócenia referencji do zmiennej lokalnej 'temp'* — *przyp. tłum.*

obiektów tak klasy bazowej, jak i pochodnej. Na przykład funkcja z parametrem typu `ostream &` może odebrać obiekt `ostream` (choćby `cout`) lub obiekt `ofstream` (zadeklarowany przez nas).

Na listingu 8.8 pokazano to na przykładzie jednej funkcji, która zapisuje dane w pliku i na ekranie; zmienia się jedynie parametr wywołania tej funkcji. Program prosi o podanie ogniskowej obiektywu (głównego lustra lub soczewki) oraz pewnych cech okularu teleskopu. Powiększenie to długość ogniskowej podzielona przez ogniskową okularu, więc obliczenia są proste. Program korzysta też z metod formatowania, które — jak to już zostało zapowiedziane — działają równie dobrze z obiektem `cout`, jak i z obiektami klasy `ofstream` (w tym przypadku `fout`).

Listing 8.8. *filefunc.cpp*

```
//filefunc.cpp -- funkcja korzystająca z parametru ostream &
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;

void file_it(ostream & os, double fo, const double fe[],int n);
const int LIMIT = 5;
int main()
{
    ofstream fout;
    const char * fn = "ep-data.txt";
    fout.open(fn);
    if (!fout.is_open())
    {
        cout << "Nie mogę otworzyć " << fn << ". Do widzenia.\n";
        exit(EXIT_FAILURE);
    }
    double objective;
    cout << "Podaj ogniskową teleskopu w milimetrach:";
    cin >> objective;
    double eps[LIMIT];
    cout << "Podaj ogniskowe (w mm) " << LIMIT << " okularów:\n";
    for (int i = 0; i < LIMIT; i++)
    {
        cout << "Okular nr " << i + 1 << ": ";
        cin >> eps[i];
    }
    file_it(fout, objective, eps, LIMIT);
    file_it(cout, objective, eps, LIMIT);
    cout << "Gotowe\n";
    return 0;
}

void file_it(ostream & os, double fo, const double fe[],int n)
{
    ios_base::fmtflags initial;
    initial = os.setf(ios_base::fixed); // zachowaj formatowanie początkowe
    os.precision(0);
    os << "Ogniskowa obiektywu: " << fo << " mm\n";
    os.setf(ios::showpoint);
    os.precision(1);
    os.width(12);
    os << "f okularu";
    os.width(15);
    os << "powiększenie" << endl;
    for (int i = 0; i < n; i++)
    {
        os.width(12);
```

```
        os << fe[i];
        os.width(15);
        os << int (fo/fe[i] + 0.5) << endl;
    }
    os.setf(initial);    // przywrócenie początkowych ustawień formatowania
}
```

Oto wynik uruchomienia programu z listingu 8.8:

```
Podaj ogniskową teleskopu w milimetrach: 1800
Podaj ogniskowe (w mm) 5 okularów:
Okular nr 1: 30
Okular nr 2: 19
Okular nr 3: 14
Okular nr 4: 8.8
Okular nr 5: 7.5
Ogniskowa obiektywu: 1800 mm
   f okularu   powiększenie
        30.0             60
        19.0             95
        14.0            129
         8.8            205
         7.5            240
Gotowe
```

Wiersz:

```
file_it(fout, objective, eps, LIMIT);
```

powoduje wypisanie danych okularu do pliku *ep-data.txt*, a wiersz:

```
file_it(cout, objective, eps, LIMIT);
```

wypisanie tych samych danych na ekranie.

Uwagi o programie

Najważniejsze w kodzie z listingu 8.8 jest to, że parametr `os` typu `ostream &` może odnosić się do obiektu klasy `ostream`, jak `cout`, oraz do obiektu klasy `ofstream`, jak `fout`. Jednak program ten pokazuje także to, jak w obu wzmiankowanych typach obiektów można używać metod formatowania klasy `ostream`. Przejrzymy te metody, a niektóre z nich krótko omówimy; więcej szczegółów na ten temat pojawi się w rozdziale 17.

Metoda `setf()` pozwala ustawiać różne stany związane z formatowaniem. Na przykład wywołanie `setf(ios_base::fixed)` ustawia obiekt w tryb dziesiętnego zapisu stałoprzecinkowego. Wywołanie `setf(ios_base:showpoint)` ustawia obiekt w tryb pokazywania końcowej kropki dziesiętnej, nawet jeśli za nią są same zera. Metoda `precision()` określa liczbę cyfr pokazywanych na prawo od kropki dziesiętnej (o ile obiekt jest w trybie `fixed`). Wszystkie te ustawienia obowiązują do momentu, aż zostaną w kolejnym wywołaniu zmienione. Metoda `width()` ustala szerokość pola użytą przy następnym pokazywaniu danych. Ustawienie to dotyczy pojedynczej wartości, po czym jest przywracane ustawienie domyślne (a domyślnie pole ma szerokość zero, czyli jest rozszerzane tak, aby akurat pomieściło podaną wartość).

Funkcja `file_it()` korzysta z ciekawej pary wywołań metod:

```
ios_base::fmtflags initial;
initial = os.setf(ios_base::fixed);  // zapamiętanie aktualnego stanu formatowania
...
os.setf(initial);    // odtworzenie początkowego stanu formatowania
```

Metoda `setf` zwraca kopię wszystkich ustawień formatowania, które obowiązywały przed tym wywołaniem. `ios_base::fmtflags` to wymyślna nazwa typu potrzebnego do zapisania tych informacji. Wobec tego przypisanie zmiennej `initial` powoduje zapisanie ustawień, które obowiązywały przed wywołaniem funkcji `file_it()`. Zmienna `initial` może być później używana jako parametr `setf()` w celu wyzerowania wszystkich ustawień formatowania na wartości oryginalne. Wobec tego funkcja przywraca obiekt do stanu, w jakim znajdował się on przed przekazaniem go funkcji `file_it()`.

Szersza znajomość klas pomoże zrozumieć działanie tych metod oraz powód, dla którego na przykład stale pojawia się `ios_base`. Jednak aby użyć tych metod, nie musimy czekać do rozdziału 17.

I na koniec jeszcze jedna uwaga: każdy obiekt ma własne ustawienia formatowania. Kiedy program przekazuje `cout` do funkcji `file_it()`, ustawienia `cout` są zmieniane, a potem odtwarzane. Gdy program przekazuje funkcji `file_it()` obiekt `fout`, zmiana ustawień i ich odtwarzanie dotyczą obiektu `fout`.

Kiedy korzystać z referencji jako parametrów?

Istnieją dwa zasadnicze powody używania parametrów referencyjnych:

- umożliwienie modyfikacji danych w wywoływanej funkcji,
- przyspieszenie wykonywania programu dzięki przekazaniu samej referencji zamiast całego obiektu danych.

Drugi powód jest szczególnie istotny w przypadku większych obiektów danych, jak struktury i egzemplarze klas. Obydwa podane powody dotyczą także przekazywania przez wskaźniki. Nie dzieje się tak bez powodu — referencje są pewnego rodzaju interfejsem do wskaźników. Kiedy zatem używać referencji, kiedy wskaźników, a kiedy przekazywać parametry przez wartość? Oto garść wskazówek.

Funkcja korzysta z przekazanych danych, ale ich nie modyfikuje:

- Jeśli obiekt danych jest mały, na przykład jest to wbudowany typ danych lub mała struktura, należy go przekazać przez wartość.
- Jeśli obiekt danych jest tablicą, nie mamy wyboru — musimy użyć wskaźnika. Wskaźnik powinien być typu `const`.
- Jeśli obiekt danych jest duży, należy użyć wskaźnika `const` lub referencji `const`, aby w ten sposób zwiększyć szybkość działania programu. Zaoszczędzimy na czasie i pamięci, gdyż nie trzeba będzie kopiować struktury czy klasy.
- Jeśli obiekt danych jest egzemplarzem klasy, należy użyć referencji `const`. Semantyka klas często wymaga użycia referencji — to właśnie było głównym powodem dodania referencji do C++. Wobec tego standardowym sposobem przekazywania egzemplarzy klas jest użycie referencji.

Funkcja modyfikuje przekazane dane:

- Jeśli obiekt danych jest typem wbudowanym, należy użyć wskaźnika. Jeśli natkniemy się gdzieś w kodzie na zapis `poprawto(&x)`, gdzie `x` jest typu `int`, będzie dość oczywiste, że zadaniem funkcji jest modyfikacja wartości `x`.
- Jeśli obiekt danych jest tablicą, nie mamy wyboru i używamy wskaźnika.
- Jeśli obiekt danych jest strukturą, używamy referencji lub wskaźnika.
- Jeśli obiekt danych jest egzemplarzem klasy, używamy referencji.

Są to oczywiście tylko wytyczne i bywa, że konieczne jest dokonanie wyboru innego niż tu sugerowany. Na przykład obiekt `cin` korzysta z referencji do typów podstawowych, więc zamiast `cin >> &n` można użyć zapisu `cin >> n`.

Parametry domyślne

Teraz zajmijmy się kolejną sztuczką możliwą do zastosowania w C++ — parametrami domyślnymi. *Parametr domyślny* to wartość używana w przypadku, kiedy w wywołaniu funkcji zabraknie wskazanego parametru. Jeśli na przykład ustawimy funkcję `void wow(int n)` tak, aby parametr `n` miał wartość domyślną 1, to wywołanie `wow()` będzie równoważne wywołaniu `wow(1)`. Zwiększa to elastyczność użycia funkcji. Załóżmy, że mamy funkcję `left()` zwracającą pierwszych `n` znaków łańcucha, przy czym parametrami są łańcuch i `n`. Ściślej rzecz biorąc, funkcja zwraca wskaźnik na nowy łańcuch zawierający wybrany fragment łańcucha wejściowego. Przykładowo wywołanie `left("teoria", 3)` powoduje utworzenie łańcucha `"teo"`; zwracany jest wskaźnik na ten nowy łańcuch. Załóżmy teraz, że ustaliliśmy, iż wartością domyślną drugiego parametru jest 1. Wywołanie `left("teoria", 3)` zadziała jak dotąd, gdyż przekazana wartość 3 nadpisze wartość domyślną. Jednak już wywołanie `left("teoria")` nie spowoduje błędu, lecz zwróci wskaźnik na łańcuch `"t"`. Tego typu wartości domyślne są przydatne, jeśli program często pobiera łańcuchy jednoznakowe, ale czasami także dłuższe.

Jak ustalić wartość domyślną? Trzeba to zrobić w prototypie funkcji. Kompilator odszukuje prototyp funkcji w celu sprawdzenia, ile dana funkcja ma parametrów, więc także w prototypie jest miejsce na parametry domyślne. Wystarczy w prototypie przypisać wartość wybranemu parametrowi i voilá! Oto przykładowy prototyp funkcji `left()`:

```
char * left(const char * str, int n = 1);
```

Chcemy, aby nasza funkcja zwracała nowy łańcuch — stąd jej typem jest `char*`, czyli wskaźnik `char`. Pierwotny łańcuch ma być niezmieniany, dlatego używamy słowa kluczowego `const`. Parametr `n` ma mieć wartość domyślną 1, dlatego parametrowi temu przypisujemy jedynkę. Parametr domyślny zachowuje się jak wartość inicjalizująca — czyli w powyższym przykładzie zmienna `n` jest inicjalizowana wartością 1. Jeśli nie zmienialibyśmy `n`, wartością tej zmiennej pozostałoby 1, ale jeżeli przekażemy funkcji drugi parametr, zostanie on przypisany do `n` i nadpisze 1.

Kiedy używamy funkcji z listą parametrów, musimy wartości domyślne dodawać od strony prawej do lewej. Nie można zatem użyć niektórych parametrów domyślnych, jeśli za nimi są parametry bez wartości domyślnych:

```
int harpo(int n, int m = 4, int j = 5);         // DOBRZE
int chico(int n, int m = 6, int j);             // ŹLE
int groucho(int k = 1, int m = 2, int n = 3);   // DOBRZE
```

Przy prototypie funkcji `harpo()` jak powyżej można później wywoływać tę funkcję z jednym, dwoma lub trzema parametrami:

```
beeps = harpo(2);       // równoważne harpo(2,4,5)
beeps = harpo(1,8);     // równoważne harpo(1,8,5)
beeps = harpo(8,7,6);   // nie będą użyte żadne parametry domyślne
```

Argumenty są przypisywane odpowiednim parametrom od lewej strony do prawej; nie można żadnych z nich pomijać. Wobec tego poniższy zapis jest niedozwolony:

```
beeps = harpo(3, ,8);   // ŹLE, m nie zostanie ustawione na 4
```

Parametry domyślne nie są jakimś wielkim postępem w programowaniu; są po prostu wygodne. Kiedy zaczniemy pracę z klasami, okaże się, że parametry takie pozwalają ograniczyć liczbę konstruktorów, metod oraz metod przeciążonych.

Na listingu 8.9 pokazano użycie parametrów domyślnych. Jak widać, parametry domyślne występują tylko w prototypie. Definicja funkcji jest taka sama jak bez parametrów domyślnych.

Listing 8.9. *left.cpp*

```
// left.cpp -- funkcja obsługi łańcuchów z parametrem domyślnym
#include <iostream>
const int ArSize = 80;
char * left(const char * str, int n = 1);
int main()
{
    using namespace std;
    char sample[ArSize];
    cout << "Podaj łańcuch znakowy:\n";
    cin.get(sample,ArSize);
    char *ps = left(sample, 4);
    cout << ps << endl;
    delete [] ps;       // zwolnienie starego łańcucha
    ps = left(sample);
    cout << ps << endl;
    delete [] ps;       // zwolnienie nowego łańcucha
    return 0;
}

// funkcja zwraca wskaźnik na nowy łańcuch składający się
// z pierwszych n znaków łańcucha str
char * left(const char * str, int n)
{
    if(n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // kopiowanie znaków
    while (i <= n)
        p[i++] = '\0';  // wyzerowanie reszty łańcucha
    return p;
}
```

Oto wynik uruchomienia programu z listingu 8.9:

```
Podaj łańcuch znakowy:
bielactwo
biel
b
```

Uwagi o programie

Program z listingu 8.9 korzysta z operatora `new`, aby stworzyć nowy łańcuch na wybrane znaki. Może się zdarzyć, że złośliwy użytkownik poda ujemną liczbę znaków, a wtedy funkcja ustawi tę liczbę na 0 i zwróci pusty łańcuch. Inna możliwość to podanie przez nieodpowiedzialnego użytkownika większej liczby znaków, niż ma łańcuch. Przed tym funkcja chroni się następująco:

```
i < n && str[i]
```

Warunek `i < n` powoduje przerwanie pętli po skopiowaniu `n` znaków. Drugi warunek, `str[i]`, dotyczy kodu kopiowanego znaku. Jeśli pętla dojdzie do znaku `NUL` o kodzie 0, działanie pętli zostanie przerwane. Ostatnia pętla `while` kończy łańcuch znakiem `NUL` i wypełnia znakiem `NUL` pozostałą zaalokowaną pamięć.

Inną metodą ustalania wielkości nowego łańcucha jest ustawienie `n` na mniejszą z dwóch wartości — wartości przekazanej oraz długości łańcucha:

```
int len = strlen(str);
n = (n < len) ? n : len;    // mniejsza z n i len
char * p = new char[n+1];
```

W ten sposób zapewniamy, że new nie zaalokuje więcej miejsca, niż jest potrzebne na łańcuch. Może to być przydatne w przypadku wywołań typu `left("Cześć!", 32767)`. W pierwszym rozwiązaniu skopiujemy napis `"Cześć!"` do tablicy mającej 32 767 znaków, przy czym wszystkie te znaki poza pierwszymi sześcioma ustawimy na zero. Drugie rozwiązanie powoduje skopiowanie tego samego napisu do tablicy 7-znakowej. Jednak dodanie kolejnego wywołania funkcji, `strlen()`, powoduje zwiększenie programu, spowolnienie jego działania i wymaga włączenia pliku nagłówkowego *cstring* (lub *string.h*). Programiści C zawsze dążyli do zapewnienia jak najszybszego działania swoich programów, do bardziej zwartego kodu i do składania na barki programisty odpowiedzialności za prawidłowe wywoływanie funkcji. Jednak w C++ większy nacisk kładzie się na niezawodność. Ostatecznie powolny, ale dobrze działający program jest lepszy niż program działający szybciej, lecz z błędami. Jeśli czas potrzebny na wywołanie funkcji `strlen()` okaże się problemem, możemy funkcji `left()` umożliwić bezpośrednie wykrycie, czy mniejsze jest `n`, czy długość łańcucha:

```
int m = 0;
while ( m <= n && str[m] != '\n')
    m++;
char * p = new char[m+1];
// w reszcie kodu należy użyć m zamiast n
```

Pamiętajmy, że wyrażenie `str[m] != '\0'` otrzymuje wartość `true` dla każdego `str[m]` różnego od znaku pustego i `false`, kiedy `str[m]` to znak pusty. W wyrażeniach logicznych `&&` wartości niezerowe są konwertowane na wartość logiczną `true`, a warunek pętli `while` można zapisać również tak:

```
while (m<=n && str[m])
```

Przeciążanie funkcji

Polimorfizm funkcji to bardzo eleganckie rozszerzenie C++ względem C. O ile parametry domyślne pozwalają wywoływać tę samą funkcję z różnymi zestawami parametrów, o tyle *polimorfizm funkcji* (nazywany też *przeciążaniem funkcji*) pozwala używać wielu funkcji o takiej samej nazwie. Słowo *polimorfizm* oznacza posiadanie wielu postaci, zatem *polimorfizm funkcji* umożliwia funkcji przybieranie wielu form. Z kolei termin *przeciążanie funkcji* oznacza możliwość wiązania z jedną nazwą wielu funkcji — czyli nazwa jest przeciążana. Oba podane tu pojęcia oznaczają to samo, ale częściej mówi się o przeciążaniu funkcji — chyba dlatego, że bardziej kojarzy się z ciężką pracą. Korzystając z przeciążania, można zdefiniować całą rodzinę funkcji realizujących w zasadzie te same zadania, ale na różnych zestawach parametrów.

Przeciążone funkcje przypominają czasowniki mające więcej niż jedno znaczenie. Na przykład można kopać piłkę do bramki i można kopać w ziemi w poszukiwaniu skarbu. Kontekst pozwala (przynajmniej powinien pozwolić) określić, o jakie kopanie chodzi. Analogicznie w C++ na podstawie kontekstu określa się, której wersji przeciążonej funkcji należy użyć.

Kluczem do przeciążania funkcji jest lista parametrów, nazywana też *sygnaturą funkcji*. Jeśli dwie funkcje mają tyle samo takich samych (co do typu) parametrów, mają takie same sygnatury, to nazwy tych parametrów są bez znaczenia. Język C++ umożliwia definiowanie funkcji o takich samych nazwach, ale o różnych sygnaturach. Sygnatury mogą różnić się liczbą parametrów, ich typami lub jednym i drugim. Można na przykład zdefiniować ciąg funkcji `print()` mających następujące prototypy:

```
void print(const char * str, int width);  // nr 1
void print(double d, int width);          // nr 2
void print(long l, int width);            // nr 3
void print(int i, int width);             // nr 4
void print(const char *str);              // nr 5
```

Kiedy używamy funkcji `print()`, kompilator odbiera do wywołania prototyp mający taką samą sygnaturę:

```
print("Ciastka", 15);     // użyj nr 1
print("Syrop");           // użyj nr 5
print(1999.0, 10);        // użyj nr 2
print(1999, 12);          // użyj nr 4
print(1999L, 15);         // użyj nr 3
```

Na przykład wywołanie `print("Ciastka", 15);` korzysta z parametrów będących łańcuchem i liczbą całkowitą, więc pasuje do prototypu numer 1.

Gdy korzysta się z funkcji przeciążonych, trzeba zapewnić, że w wywołaniu użyte zostaną prawidłowe typy parametrów. Weźmy na przykład pod uwagę następujące instrukcje:

```
unsigned int year = 3210;
print(year, 6);             // niejednoznaczne wywołanie
```

Który prototyp `print()` pasuje do takiego wywołania? Żaden! Brak pasującego prototypu nie przekreśla jeszcze możliwości wywołania takiej funkcji, bo C++ będzie próbował wymusić dopasowanie przez standardowe konwersje typów. Gdyby na przykład prototyp numer 2 był *jedynym* prototypem funkcji `print`, wywołanie `print(year, 6)` spowodowałoby konwersję wartości `year` na typ `double`. Jednak powyżej trzy prototypy mają za pierwszy parametr liczbę, więc `year` można skonwertować na trzy inne typy. W przypadku takiej niejednoznaczności C++ odrzuci wywołanie funkcji jako błędne.

Niektóre sygnatury pozornie się różnią, ale mimo to nie mogą istnieć jednocześnie. Na przykład weźmy pod uwagę następujące dwa prototypy:

```
double cube(double x);
double cube(double & x);
```

Można by pomyśleć, że mamy do czynienia z przeciążaniem funkcji, gdyż sygnatury są różne, jednak dla kompilatora jest inaczej. Jeśli mamy wywołanie:

```
cout << cube(x);
```

parametr `x` pasuje do prototypu `double x` i do prototypu `double &x`. Kompilator nie jest w stanie „stwierdzić”, której funkcji ma użyć. Aby uniknąć tego typu problemów, kompilator, sprawdzając sygnatury funkcji, traktuje referencję do danego typu i sam ten typ jako równoważne.

Proces wybierania funkcji rozróżnia zmienne z modyfikatorem `const` i bez niego. Spójrzmy na następujące prototypy:

```
void dribble(char * bits);          // przeciążenie
void dribble(const char *cbits);    // przeciążenie
void dabble(char *bits);            // nie ma przeciążenia
void drivel(const char * bits);     // nie ma przeciążenia
```

Oto do których prototypów będą pasowały różne wywołania funkcji:

```
const char p1[20] = "Jaka jest pogoda?";
char p2[20] = "Jest ruch w interesie?";
dribble(p1);     // dribble(const char *);
dribble(p2);     // dribble(char *);
```

```
dabble(p1);      // brak dopasowania
dabble(p2);      // dabble(char *);
drivel(p1);      // drivel(const char *);
drivel(p2);      // drivel(const char *);
```

Funkcja `dribble()` ma dwa prototypy: jeden dla wskaźników stałych, drugi dla zwykłych; kompilator wybierze jeden z nich w zależności od tego, czy przekazany parametr jest stałą, czy nie. Funkcja `dabble()` pasuje jedynie do wywołań z parametrem niebędącym stałą, natomiast funkcja `drivel()` pasuje do wywołań z parametrem stałym i niestałym. Powodem tej różnicy w zachowaniu `dabble()` i `drivel()` jest to, że można przypisać wartość bez `const` do zmiennej `const`, ale nie odwrotnie.

Trzeba pamiętać, że o możliwości przeciążania decyduje sygnatura, a nie typ funkcji. Na przykład poniższe dwie deklaracje są ze sobą niezgodne:

```
long gronk(int n, float m);      // takie same sygnatury,
double gronk(int n, float m);    // więc sytuacja niedopuszczalna
```

Zatem funkcji `gronk()` nie można przeciążyć pokazaną metodą. Można użyć różnych typów wartości zwracanej, ale poza tym inne muszą być sygnatury:

```
long gronk(int n, float m);      // sygnatury są różne, więc
double gronk(float n, float m);  // teraz już jest w porządku
```

W dalszej części tego rozdziału, kiedy będziemy omawiać szablony funkcji, dokładniej przeanalizujemy też dobieranie funkcji.

Przeciążanie po parametrach referencyjnych

W klasach i szablonach STL często stosuje się parametry referencyjne i choćby dlatego warto wiedzieć, jak przeciążanie działa z różnymi typami referencyjnymi. Weźmy następujące trzy prototypy funkcji:

```
void sink(double & r1);  // dla modyfikowalnej l-wartości
void sank(const double & r2); // dla l-wartości (modyfikowalnej i niemodyfikowalnej) oraz r-wartości
void sunk(double && r3); // dla r-wartości
```

Parametr deklarowany jako l-wartość referencyjna `r1` pasuje do każdego argumentu będącego modyfikowalną l-wartością, na przykład do zwyczajnej zmiennej typu `double`. Parametr `r2` będzie dopasowany do argumentu będącego modyfikowalną l-wartością, niemodyfikowalną l-wartością bądź r-wartością (może to być na przykład wyrażenie sumy dwóch wartości typu `double`). Wreszcie referencja do r-wartości będzie pasować wyłącznie do r-wartości. Zauważmy, że `r2` można dopasować do tych samych argumentów co `r1` i `r3`, co prowadzi do pytania o zachowanie programu w przypadku niejednoznacznego przeciążenia funkcji. W takiej sytuacji realizowane jest dokładniejsze dopasowanie:

```
void staff(double & rs); // dla modyfikowalnej l-wartości
voit staff(const double & rcs); // dla r-wartości i niemodyfikowalnej l-wartości
void stove(double & r1); // dla modyfikowalnej r-wartości
void stove(const double & r2); // dla niemodyfikowalnej l-wartości
void stove(double && r3); // dla r-wartości
```

W ten sposób możemy dopasowywać zachowanie funkcji do typu argumentu wywołania:

```
double x = 55.5;
const double y = 32.0;
stove(x);    // wywołanie wersji stove(double &)
stove(y);    // wywołanie wersji stove(const double &)
stove(x+y); // wywołanie wersji stove(double &&)
```

Jeśli (załóżmy) pominęlibyśmy funkcję `stove(double &&)`, to wywołanie `stove(x+y)` zostanie rozprowadzone do funkcji `stove(const double &)`.

Przykład przeciążania funkcji

W tym rozdziale tworzyliśmy już funkcję `left()` zwracającą wskaźnik na pierwszych `n` znaków łańcucha. Dodajmy teraz jeszcze jedną funkcję `left()`, która zwraca pierwszych `n` cyfr liczby całkowitej. Możemy jej użyć na przykład do sprawdzania początkowych cyfr konta bankowego zapisanego jako długa liczba całkowita, w celu uzyskania informacji, w jakim banku konto to jest prowadzone.

Funkcja działająca na liczbach całkowitych jest nieco bardziej skomplikowana niż działająca na łańcuchach znakowych, gdyż poszczególne cyfry nie są przechowywane w odrębnych elementach tablicy. Jedno rozwiązanie polega na wyliczeniu najpierw, ile liczba ma cyfr. Dzielenie przez 10 powoduje odrzucenie jednej cyfry, więc stosując kolejne dzielenia, można policzyć cyfry. Można użyć na przykład następującej pętli:

```
unsigned digits = 1;
while (n /= 10)
    digits++;
```

Pętla zlicza, ile razy można usunąć z `n` cyfrę, tak aby jeszcze coś zostało. Przypomnijmy, że `n /= 10` to skrócony zapis `n = n / 10`. Jeśli `n` jest na przykład równe 8, warunek pętli spowoduje przypisanie `n` wartości 8/10, czyli 0, gdyż jest to dzielenie całkowitoliczbowe. Pętla od razu zostanie przerwana, a zmienna `digits` zostanie z wartością 1. Jednak już dla `n = 238` w pierwszym cyklu pętli mamy 238/10, czyli 23. Nie jest to zero, więc po zwiększeniu `digits` o jeden pętla jest kontynuowana. Następnie `n` przybiera wartość 23/10, czyli 2. Znowu wynik nie jest zerem, zatem `digits` ma już wartość 3. W następnym cyklu `n` jest równe 2/10, tym razem 0 — pętla jest przerywana, zmiennej `digits` pozostaje wartość 3.

Załóżmy teraz, że wiemy, iż liczba ma pięć cyfr, a chcemy zwrócić jej pierwsze 3 cyfry. Wystarczy najpierw podzielić liczbę przez 10, a potem ten wynik jeszcze raz przez 10. Każde dzielenie przez 10 powoduje odrzucenie jednej cyfry z prawej strony. Aby określić liczbę cyfr, jaką należy odrzucić, odejmujemy liczbę pokazywanych cyfr od całkowitej liczby cyfr. Żeby na przykład pokazać pierwsze 4 cyfry liczby 9-cyfrowej, odrzucamy ostatnich 5 cyfr. Rozwiązanie to możemy zapisać w formie programu następująco:

```
ct = digits - ct;
while (ct--)
    num /= 10;
return num;
```

Na listingu 8.10 kod ten włączono do nowej funkcji `left()`. Funkcja zawiera też kod dodatkowy, który obsługuje przypadki szczególne, jak żądanie zera cyfr lub żądanie większej liczby cyfr, niż ma dana liczba. Sygnatura nowej funkcji `left()` jest inna niż sygnatura starej, więc możemy obu funkcji używać w tym samym programie.

Listing 8.10. *leftover.cpp*

```
// leftover.cpp -- przeciążanie funkcji left()
#include <iostream>
unsigned long left(unsigned long num, unsigned ct);
char * left(const char * str, int n = 1);

int main()
{
     using namespace std;
    char * trip = "Hawaii!!";    // wartość testowa
    unsigned long n = 12345678; // wartość testowa
    int i;
    char * temp;
```

```
    for (i = 1; i < 10; i++)
    {
        cout << left(n, i) << endl;
        temp = left(trip,i);
        cout << temp << endl;
        delete [] temp; // wskazuje tymczasowy obszar pamięci
    }
    return 0;

}

// funkcja ta zwraca pierwszych ct cyfr liczby num
unsigned long left(unsigned long num, unsigned ct)
{
    unsigned digits = 1;
    unsigned long n = num;

    if (ct == 0 || num == 0)
        return 0;       // jeśli brak cyfr, zwraca 0
    while (n /= 10)
        digits++;
    if (digits > ct)
    {
    ct = digits - ct;
    while (ct--)
        num /= 10;
    return num;         // zwraca ct skrajnych lewych cyfr
    }
    else                // jeśli ct >= liczby cyfr,
        return num;     // zwraca całą liczbę
}

// funkcja zwraca wskaźnik nowego łańcucha zawierającego
// pierwszych n znaków łańcucha str
char * left(const char * str, int n)
{
    if(n < 0)
        n = 0;
    char * p = new char[n+1];
    int i;
    for (i = 0; i < n && str[i]; i++)
        p[i] = str[i];  // kopiowanie znaków
    while (i <= n)
        p[i++] = '\0';  // ustawienie reszty znaków na zera
    return p;
}
```

Oto wynik działania programu z listingu 8.10:

```
1
H
12
Ha
123
Haw
1234
Hawa
12345
Hawai
123456
Hawaii
1234567
```

```
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!
```

Kiedy korzystać z przeciążania funkcji?

Przeciążanie funkcji może wydawać się doprawdy fascynujące, ale nie należy go nadużywać. Przeciążanie funkcji należy stosować w przypadku funkcji realizujących te same zadania, ale na różnego rodzaju danych. Poza tym warto się zastanowić, czy podobnych efektów nie uda się uzyskać za pomocą parametrów domyślnych. Można na przykład zastąpić pojedynczą funkcję `left()` przetwarzającą łańcuchy dwoma funkcjami przeciążonymi:

```
char * left(const char * str, unsigned n);  // dwa parametry
char * left(const char * str);              // jeden parametr
```

Jednak użycie pojedynczej funkcji z parametrem domyślnym jest prostsze. Piszemy tylko jedną funkcję, program potrzebuje pamięci tylko na jedną funkcję… Jeśli zechcemy taką funkcję zmodyfikować, wystarczy poprawić tylko jeden fragment kodu. Jeśli jednak potrzebne są różne typy parametrów, a parametry domyślne nie rozwiązują problemu, to pora skorzystać z przeciążania funkcji.

Co to jest ozdabianie nazw?

W jaki sposób C++ rozróżnia poszczególne wersje funkcji przeciążonych? Przypisuje każdej z nich ukryte cechy identyfikacyjne. Kiedy korzystamy z narzędzia do edycji programów w środowisku programistycznym, kompilator robi pewne sztuczki określane mianem *ozdabiania nazw*. Polegają one na szyfrowaniu nazw funkcji zgodnie z parametrami tych funkcji podanymi w prototypie. Weźmy pod uwagę następujący prototyp:

```
long MyFunctionFoo(int, float);
```

Taka postać jest czytelna dla programisty; wiadomo, że funkcja ma dwa parametry typu `int` oraz `float` i zwraca wartość typu `long`. Kompilator dokumentuje ten interfejs, zamieniając wewnętrznie nazwę na dość niecodzienny napis, na przykład:

```
?MyFunctionFoo@@YAXH
```

Tak udekorowana nazwa oryginalna ma zakodowaną liczbę parametrów oraz ich typy. Inne sygnatury funkcji powodują dołączanie innych symboli, poza tym różne kompilatory stosują różne konwencje dekorowania.

Szablony funkcji

Współczesne kompilatory C++ wykorzystują jedną z nowszych cech C++ — szablony funkcji. *Szablon funkcji* to ogólny opis funkcji, czyli taki, w którym funkcja jest opisana przez typy ogólne — pod nie można później podstawiać typy konkretne, jak `int` czy `double`. Przekazując szablonowi typ jako parametr, można wymusić na kompilatorze wygenerowanie funkcji określonego typu. Szablony umożliwiają programowanie ogólne (ang. *generic programming*). Typy są przekazywane jak parametry, więc o szablonach mówi się czasem jako o *typach sparametryzowanych*. Sprawdźmy teraz, do czego szablony funkcji mogą się przydać i jak się ich używa.

Definiowaliśmy wcześniej (na listingu 8.4) funkcję, która zamieniała wartości dwóch zmiennych typu `int`. Załóżmy, że chcielibyśmy zamienić wartości dwóch zmiennych typu `double`. Jedno rozwiązanie to powielenie poprzedniego kodu, ale z zastąpieniem typu `int` typem `double`. Gdybyśmy chcieli zamieniać

wartości typu `char`, ponownie skorzystalibyśmy z tej samej techniki. Przeprowadzanie takich banalnych modyfikacji byłoby marnowaniem naszego cennego czasu, poza tym zawsze za którymś razem można popełnić błąd. Jeśli wprowadzamy zmiany ręcznie, zawsze możemy przeoczyć któreś wystąpienie `int`. Jeżeli realizujemy podstawienie globalne, na przykład zamieniając `int` na `double`, możemy zrobić coś takiego:

```
int x;
short interval;
```

zostanie zamienione na:

```
double x;           // zamierzona zamiana typu
short doubleerval;  // niezamierzona zmiana nazwy zmiennej
```

Szablony funkcji C++ automatyzują proces generowania różnych odmian funkcji, przez co pozwalają oszczędzić czas i zwiększyć niezawodność kodu.

Szablony funkcji umożliwiają zdefiniowanie funkcji z dowolnym typem. Na przykład można przygotować szablon do zamiany wartości następująco:

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
    temp = a;
    a = b;
    b = temp;
}
```

W pierwszym wierszu deklarujemy tworzenie szablonu, ustalamy też, że typ parametryzujący będzie się nazywał `AnyType`. Słowa kluczowe `template` i `typename` są obowiązkowe, tyle że zamiast słowa `typename` można użyć `class`. Obowiązkowe są też nawiasy kątowe. Nazwa typu (u nas `AnyType`) jest dowolna, byle tylko spełniała standardowe zasady nazewnictwa obowiązujące w C++; wielu programistów preferuje w tej sytuacji proste nazwy, jak `T`, które (trzeba to przyznać) istotnie skracają kod. Reszta kodu opisuje algorytm zamieniający dwie wartości typu `AnyType`. Szablon nie powoduje utworzenia żadnej nowej funkcji, ale daje kompilatorowi wskazówki, jak takie funkcje należałoby definiować. Jeśli zechcemy, aby funkcja zamieniała miejscami wartości `int`, kompilator utworzy potrzebną definicję funkcji, zamieniając wszystkie wystąpienia typu `AnyType` na typ `int`. Jeśli potrzebna nam będzie funkcja zamieniająca wartości typu `double`, kompilator zgodnie z tym samym szablonem utworzy i taką funkcję.

Przed ustaleniem standardu C++98 ze słowem kluczowym `typename` w C++ w tym konkretnym kontekście (to jest w kontekście parametrów szablonów) stosowano słowo `class`. Powyższy szablon można by więc przepisać tak:

```
template <class AnyType>
void Swap(AnyType &a, AnyType &b)
{
    AnyType temp;
   temp = a;
    a = b;
    b = temp;
}
```

Słowo kluczowe `typename` nieco lepiej objaśnia, że `AnyType` jest typem; jednak w dużych bibliotekach gotowego już kodu używane jest starsze słowo kluczowe `class`. Standard C++ traktuje w omawianym kontekście oba słowa identycznie. Tutaj będziemy również stosować obie formy, choćby po to, żeby oswoić czytelnika z obiema wciąż powszechnie stosowanymi składniami deklaracji typowego parametru szablonu.

Wskazówka

Szablonów należy używać, kiedy potrzebne nam są funkcje stosujące ten sam algorytm do różnych typów danych. Jeśli nie zależy nam na zgodności wstecz i stać nas na wysiłek wpisania dłuższego słowa, zamiast `class` należy stosować raczej `typename`.

Aby poinformować kompilator, że potrzebna nam będzie określona funkcja `Swap()`, wystarczy użyć jej w programie. Kompilator sprawdzi, jakiego typu są jej parametry, i potrzebną funkcję wygeneruje. Na listingu 8.11 pokazano, jak to działa. Program jest zbudowany jak w przypadku zwykłych funkcji, prototyp szablonu funkcji jest na górze pliku, a sam szablon zdefiniowano za funkcją `main()`.W przykładzie posługujemy się też tradycyjną już nazwą parametru typowego `T` zamiast `AnyType`.

Listing 8.11. *funtemp.cpp*

```
// funtemp.cpp -- użycie szablonu funkcji
#include <iostream>
// prototyp szablonu funkcji
template <typename T> // albo class T
void Swap(T &a, T &b);

int main()
{
    using namespace std;
    int i = 10;
    int j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Użycie funkcji obsługującej typ int, "
            "wygenerowanej przez kompilator:\n";
    Swap(i,j);  // generuje void Swap(int &, int &)
    cout << "Teraz i, j = " << i << ", " << j << ".\n";

    double x = 24.5;
    double y = 81.7;
    cout << "x, y = " << x << ", " << y << ".\n";
    cout << "Użycie funkcji obsługującej typ double, "
            "wygenerowanej przez kompilator:\n";
    Swap(x,y);  // generuje void Swap(double &, double &)
    cout << "Teraz x, y = " << x << ", " << y << ".\n";
    // cin.get();
    return 0;
}

// definicja szablonu funkcji
template <typename T> // lub class T
void Swap(T &a, T &b)
{
    T temp;    // zmienna temp typu T
    temp = a;
    a = b;
    b = temp;
}
```

Pierwsza funkcja `Swap()` z listingu 8.11 ma dwa parametry typu `int`, więc kompilator wygeneruje wersję `int` tej funkcji: wszystkie wystąpienia `T` zostaną zamienione na `int`, zatem uzyskamy definicję:

```
void Swap(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Tego kodu nie widać, ale kompilator go wygeneruje i umieści w programie wynikowym. Druga funkcja `Swap()` ma parametry typu `double`, więc w nowej wersji kompilator zastąpi wszystkie wystąpienia `T` słowem `double`:

```
void Swap(double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

Oto wynik działania programu z listingu 8.11:

```
i, j = 10, 20.
Użycie funkcji obsługującej typ int, wygenerowanej przez kompilator:
Teraz i, j = 20, 10.
x, y = 24.5, 81.7.
Użycie funkcji obsługującej typ double, wygenerowanej przez kompilator:
Teraz x, y = 81.7, 24.5.
```

Zauważmy, że szablony funkcji nie powodują skrócenia programu wynikowego. Na listingu 8.11 nadal mamy dwie osobne definicje funkcji, zupełnie jakbyśmy każdą z nich zdefiniowali ręcznie. Ostateczny kod nie zawiera szablonów, ale jedynie ich konkretne wcielenia. Zaletą szablonów jest to, że upraszczają definiowanie wielu podobnych funkcji oraz ograniczają ryzyko popełnienia błędów.

Zazwyczaj szablony są w całości umieszczane w osobnych plikach nagłówkowych, włączanych następnie do plików korzystających z tych szablonów; o samych plikach nagłówkowych będzie jeszcze mowa w rozdziale 9.

Przeciążone szablony

Szablonów używa się, kiedy potrzebne są różne funkcje oparte na tym samym algorytmie, ale działające na różnych typach, jak na listingu 8.11. Może się jednak zdarzyć, że nie wszystkie typy będą wykorzystywały ten sam algorytm. Aby obsłużyć i taki przypadek, można przeciążać definicje szablonów tak samo, jak przeciąża się definicje zwykłych funkcji. Tak jak w przypadku zwykłego przeciążania przeciążone szablony muszą mieć odrębne sygnatury funkcji. Na przykład na listingu 8.12 dodano nowy szablon zamiany wartości, obsługujący elementy dwóch tablic. Nasz pierwszy szablon miał sygnaturę `(T &, T &)`, a nowy — `(T [],T [], int)`. Zwróćmy uwagę na to, że ostatni element tej sygnatury jest konkretnym typem `int`, a nie typem ogólnym. Nie wszystkie parametry szablonów muszą być typami ogólnymi.

Kiedy w pliku *twotemps.cpp* kompilator natknie się na pierwsze wywołanie `Swap()`, „stwierdzi”, że użyto dwóch parametrów typu `int`, więc wywołanie to pasuje do pierwotnego szablonu. Jednak drugie wywołanie ma za parametry dwie tablice typu `int` oraz liczbę `int`, więc pasuje do nowego szablonu.

Listing 8.12. *twotemps.cpp*

```
// twotemps.cpp -- użycie przeciążonych szablonów funkcji
#include <iostream>
template <typename T>      // szablon oryginalny
void Swap(T &a, T &b);

template <typename T>      // nowy szablon
void Swap(T *a, T *b, int n);

void Show(int a[]);
const int Lim = 8;
```

```
int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Użycie funkcji obsługującej typ int, "
            "wygenerowanej przez kompilator:\n";
    Swap(i,j);               // pasuje do szablonu oryginalnego
    cout << "Teraz i, j = " << i << ", " << j << ".\n";

    int d1[Lim] = {0,7,0,4,1,7,7,6};
    int d2[Lim] = {0,7,2,0,1,9,6,9};
    cout << "Tablice początkowo:\n";
    Show(d1);
    Show(d2);
    Swap(d1,d2,Lim);         // pasuje do nowego szablonu
    cout << "Tablice po zamianie:\n";
    Show(d1);
    Show(d2);
// cin.get();
    return 0;
}

template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}

void Show(int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];
    cout << endl;
}
```

Oto wynik działania programu z listingu 8.12:

```
i, j = 10, 20.
Użycie funkcji obsługującej typ int, wygenerowanej przez kompilator:
Teraz i, j = 20, 10.
Tablice początkowo:
07/04/1776
06/20/1969
```

```
Tablice po zamianie:
06/20/1969
07/04/1776
```

Ograniczenia szablonów

Załóżmy, że dany jest następujący szablon funkcji:

```
template <class T>    // albo template <typename T>
void f(T a, T b)
{...}
```

Kod szablonu często bazuje na silnych założeniach co do operacji możliwych na danym typie. Na przykład poniższa instrukcja w kodzie szablonu opiera się na założeniu, że dla obiektów zdefiniowano operację przypisania, co nie będzie możliwe dla `T` będącego typem tablicowym:

```
a = b;
```

Podobnie poniższa instrukcja bazuje na założeniu, że dla `a` i `b` zdefiniowana jest operacja >, co nie będzie spełnione dla `T` będącego zwyczajną strukturą:

```
if (a > b)
```

Operator > jest co prawda zdefiniowany dla nazw tablic, ale ponieważ nazwy tablic są adresami, porównanie dotyczy wartości adresów tablic, co niekoniecznie jest pożądaną semantyką porównania (zapewne bliższe oczekiwań byłoby porównanie rozmiarów tablic). A poniższa instrukcja zakłada, że dla `T` zdefiniowano operację mnożenia, co jest nieprawdą w przypadku tablic, wskaźników i struktur:

```
T c = a*b;
```

Krótko mówiąc, łatwo jest napisać szablon funkcji, która nie będzie działać dla niektórych (albo dla wielu) typów. Z drugiej strony niekiedy uogólnienie funkcji ma rzeczywiście sens, nawet jeśli zwyczajna składnia C++ na to nie pozwala. Na przykład przydałaby się możliwość dodawania struktur zawierających współrzędne pozycji, mimo że dla struktur nie istnieje operator +. W C++ można wtedy przeciążyć operator + tak, aby działał z odpowiednio zdefiniowaną strukturą lub klasą; takie podejście będziemy omawiać w rozdziale 11. Taka struktura nadawałaby się również do użycia w szablonie, który do poprawnego działania wymaga operacji dodawania. Alternatywą jest udostępnienie specjalizowanych definicji szablonów dla wybranych typów; to będzie przedmiotem omówienia w następnym punkcie.

Specjalizacje jawne

Załóżmy, że zdefiniowaliśmy następującą strukturę:

```
struct job
{
    char name[40];
    double salary;
    int floor;
};
```

Przyjmijmy jeszcze, że chcemy móc zamienić zawartość dwóch takich struktur. W pierwotnym szablonie do zamiany używamy następującego kodu:

```
temp = a;
a = b;
b = temp;
```

Wobec tego, że C++ pozwala przypisywać jedne struktury innym, kod ten zadziała poprawnie, nawet jeśli `T` jest strukturą `job`. Załóżmy jednak, że chcielibyśmy zamienić wartościami pola `salary` i `floor`, zostawiając `name` bez zmian. Potrzebny tu będzie inny kod, ale parametry `Swap()` będą takie same jak w pierwszym przypadku (referencje do dwóch struktur `job`), wobec czego nie można zastosować przeciążania szablonów.

Można jednak podać specjalizowaną definicję funkcji, nazywaną *specjalizacją jawną*, która będzie miała potrzebny kod. Jeśli kompilator znajdzie specjalizowaną definicję pasującą dokładnie do wywołania funkcji, użyje tej właśnie definicji, nie szukając nawet szablonów.

Mechanizm specjalizacji zmieniał się wraz z rozwojem języka C++. Przyjrzymy się aktualnej wersji, zgodnej ze standardem C++.

Specjalizacja trzeciej generacji (standard C++ ISO/ANSI)

Po wczesnych eksperymentach w tym zakresie standard C++98 ustanowił, co następuje:

- Dla danej nazwy funkcji można mieć funkcję nieszablonową, szablon funkcji oraz jawną specjalizację szablonu funkcji, a także przeciążone wersje ich wszystkich.
- Prototyp i definicja jawnej specjalizacji muszą być poprzedzone zapisem `template <>`, powinny wymieniać nazwę specjalizowanego typu.
- Specjalizacja przykrywa zwykły szablon, a funkcja zwykła przykrywa je wszystkie.

Oto wszystkie trzy postaci prototypów funkcji zamieniającej struktury `job`:

```
// prototyp zwykłej funkcji
void Swap(job &, job &);

// szablon prototypu
template <typename T>
void Swap(T &, T &);

// jawna specjalizacja typu job
template <> void Swap<job>(job&, job &);
```

Jak już wspomnieliśmy wcześniej, jeśli istnieje więcej niż jeden z tych prototypów, kompilator wybiera nieszablonową wersję przed jawną specjalizacją i przed szablonem, a jawna specjalizacja jest wybierana przed wersją wygenerowaną przed szablonem. Na przykład w poniższym kodzie pierwsze wywołanie `Swap()` korzysta z szablonu ogólnego, a drugie korzysta z jawnej specjalizacji bazującej na typie `job`:

```
...
template <class T>      // szablon
void Swap(T &, T &);

// jawna specjalizacja dla typu job
template <> void Swap<job>(job &, job &);
int main()
{
    double u, v;
    ...
    Swap(u,v);      // użyj szablonu
    job a, b;
    ...
    Swap(a,b);      // użycie void Swap<job>(job &, job &)
}
```

Zapis <job> w Swap<job> jest opcjonalny, gdyż z typów parametrów funkcji wynika, że jest to specjalizacja job. Wobec tego ten sam prototyp można zapisać jako:

```
template <> void Swap(job &, job &);    // prostsza forma
```

Gdyby przyszło nam pracować ze starszym kompilatorem, wrócilibyśmy do użycia wywołania starszego niż standard C++. Najpierw jednak sprawdźmy, jak powinny działać specjalizacje jawne.

Przykład jawnej specjalizacji

Na listingu 8.13 pokazano działanie specjalizacji. Kod ten jest zgodny ze standardem ANSI C++.

Listing 8.13. *twoswap.cpp*

```
// twoswap.cpp -- specjalizacja nadpisuje szablon
#include <iostream>
template <typename T>
void Swap(T &a, T &b);

struct job
{
    char name[40];
    double salary;
    int floor;
};

// jawna specjalizacja
template <> void Swap<job>(job &j1, job &j2);
void Show(job &j);

int main()
{
    using namespace std;
    cout.precision(2);
    cout.setf(ios::fixed, ios::floatfield);
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Użycie generowanej przez kompilator funkcji "
            "zamieniającej wartości int:\n";
    Swap(i,j);     // generuje void Swap(int &, int &)
    cout << "Teraz i, j = " << i << ", " << j << ".\n";

    job sue = {"Susan Yaffee", 73000.60, 7};
    job sidney = {"Sidney Taffee", 78060.72, 9};
    cout << "Przed zamianą struktur job:\n";
    Show(sue);
    Show(sidney);
    Swap(sue, sidney); // używa void Swap(job &, job &)
    cout << "Po zamianie struktur job:\n";
    Show(sue);
    Show(sidney);
// cin.get();
    return 0;
}

template <typename T>
void Swap(T &a, T &b)     // wersja ogólna
{
    T temp;
    temp = a;
    a = b;
    b = temp;
```

```
}

// zamienia tylko pola salary i floor struktury job

template <> void Swap<job>(job &j1, job &j2)  // specjalizacja
{
    double t1;
    int t2;
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;
    t2 = j1.floor;
    j1.floor = j2.floor;
    j2.floor = t2;
}

void Show(job &j)
{
    using namespace std;
    cout << j.name << ": " << j.salary
         << "zł na piętrze " << j.floor << endl;
}
```

Oto wyniki działania programu z listingu 8.13:

```
i, j = 10, 20.
Użycie generowanej przez kompilator funkcji zamieniającej wartości int:
Teraz i, j = 20, 10.
Przed zamianą struktur job:
Susan Yaffee: 73000.60zł na piętrze 7
Sidney Taffee: 78060.72zł na piętrze 9
Po zamianie struktur job:
Susan Yaffee: 78060.72zł na piętrze 9
Sidney Taffee: 73000.60zł na piętrze 7
```

Konkretyzacje i specjalizacje

Aby lepiej zrozumieć szablony, zajmijmy się *konkretyzacją* i *specjalizacją*. Pamiętajmy, że włączenie szablonu funkcji samo z siebie nie powoduje wygenerowania definicji funkcji; jest to tylko plan generowania takiej definicji. Kiedy kompilator generuje definicję funkcji dla danego typu na podstawie szablonu, wynikiem jest *konkretyzacja* (czyli utworzenie egzemplarza) szablonu. Na przykład na listingu 8.13 wywołanie funkcji `Swap(i,j)` powoduje, że kompilator wygeneruje egzemplarz `Swap()` dla typu `int`. Szablon *nie jest* definicją funkcji, natomiast konkretyzacja dla typu `int` już *jest* definicją funkcji. Tego typu konkretyzację nazywamy *niejawną*, gdyż kompilator „wnioskuje” o konieczności przygotowania definicji po ustaleniu, że program używa funkcji `Swap()` z parametrami typu `int`.

Początkowo niejawna konkretyzacja była jedynym sposobem generowania definicji funkcji na podstawie szablonów, ale obecnie C++ pozwala też na *konkretyzacje jawne*. Znaczy to, że można jawnie nakazać kompilatorowi utworzenie konkretnego egzemplarza, na przykład `Swap<int>()`. Składnia jest prosta — deklaruje się wszystkie potrzebne funkcje, podając typy w <> i poprzedzając deklarację słowem kluczowym `template`:

```
template void Swap<int>(int, int);     // jawna konkretyzacja
```

Kompilator pozwalający na jawne konkretyzowanie szablonu po natknięciu się na powyższą deklarację na podstawie szablonu `Swap()` utworzy egzemplarz z typem `int`. Deklaracja ta znaczy zatem: „użyj szablonu `Swap()` do wygenerowania definicji funkcji dla typu `int`”.

Porównajmy teraz jawną konkretyzację szablonu z jego jawną specjalizacją, w której używa się jednej z równoważnych deklaracji:

```
template <> void Swap<int>(int &, int &);     // jawna specjalizacja
template <> void Swap(int &, int &);          // jawna specjalizacja
```

Różnica jest taka, że dwie powyższe deklaracje nie oznaczają użycia szablonu `Swap()` do wygenerowania definicji funkcji; oznaczają „użyj odrębnej, wyspecjalizowanej definicji funkcji jawnie zdefiniowanej dla typu `int`". Prototypy te muszą być powiązane z odpowiadającymi im definicjami funkcji. Deklaracja jawnej specjalizacji po słowie kluczowym `template` ma <>, a w jawnej konkretyzacji nawiasów kątowych <> już nie ma.

Ostrzeżenie

Próba użycia w jednym pliku (a ogólnie w jednej jednostce translacji) jawnej konkretyzacji i jawnej specjalizacji dla tego samego typu danych jest błędem.

Jawne konkretyzacje mogą być tworzone również poprzez użycia funkcji w programie. Weźmy na przykład taki kod:

```
template <class T>
T Add(T a, T b) // przekazywanie przez wartość
{
    return a + b;
}
...
int m = 6;
double x = 10.2;
cout << Add<double>(x, m) << endl; // jawna konkretyzacja
```

Szablon nie dopasowałby wywołania `Add(x, m)`, ponieważ oczekuje, że oba przekazywane argumenty będą tego samego typu. Ale wywołanie w postaci `Add<double>(x, m)` wymusza konkretyzację szablonu dla typu `double`, a także rzutowanie typów argumentów wywołania funkcji na `double` w przypadku drugiego argumentu skonkretyzowanej funkcji `Add<double>(double, double)`.

A co w przypadku analogicznej operacji na szablonie `Swap()`?

```
int m = 5;
double x = 14.3;
Swap<double>(m, x); // prawie dobrze
```

Powyższy kod wygeneruje co prawda jawną konkretyzację szablonu dla typu `double`, ale niestety szablonu nie uda się skompilować, ponieważ pierwszy parametr funkcji Swap ma typ `double &`, a taki typ nie może się odnosić do zmiennej `m` typu `int`.

Niejawna konkretyzacja, jawna konkretyzacja oraz jawna specjalizacja określane są zbiorczo mianem *specjalizacji*. Ich wspólną cechą jest to, że reprezentują definicję funkcji używającej konkretnych typów, a nie ogólny opis funkcji.

Dodanie jawnej konkretyzacji szablonu doprowadziło do opracowania nowej składni: `template` i `template <>`; służy to odróżnieniu jawnej konkretyzacji od jawnej specjalizacji. Jak w wielu innych przypadkach zwiększenie możliwości odbywa się kosztem skomplikowania składni. W poniższym fragmencie kodu zestawiono omawiane tu pojęcia:

```
...
template <class T>
void Swap (T &, T &);     // prototyp szablonu
```

```
template <> void Swap<job>(job &, job &);      // jawna specjalizacja dla typu job
int main(void)
{
  template void Swap<char>(char &, char &);  // jawna konkretyzacja dla typu char
  short a, b;
  ...
  Swap(a,b);      // niejawna konkretyzacja szablonu dla typu short
  job n, m;
  ...
  Swap(n, m);      // użycie jawnej specjalizacji dla typu job
  char g, h;
  ...
  Swap(g, h);      // użycie jawnie skonkretyzowanego szablonu dla typu char
  ...
}
```

Kiedy kompilator dochodzi do jawnej konkretyzacji szablonu dla typu `char`, na podstawie definicji szablonu generuje wersję `char` funkcji `Swap()`. W innych przypadkach użycia `Swap()` kompilator dopasowuje szablon do argumentów wykorzystanych w wywołaniu funkcji. Kiedy na przykład kompilator dochodzi do wywołania `Swap(a,b)`, generuje wersję `short` funkcji `Swap()`, gdyż oba parametry są typu `short`. Kiedy kompilator dochodzi do wywołania `Swap(n,m)`, używa odrębnej definicji (jawnej specjalizacji) przygotowanej dla typu `job`. Gdy kompilator dochodzi do wywołania `Swap(g,h)`, używa specjalizacji szablonu wygenerowanej wcześniej podczas przetwarzania konkretyzacji jawnej.

Którą wersję funkcji wybierze kompilator?

Biorąc pod uwagę przeciążanie funkcji, szablony funkcji oraz przeciążanie szablonów funkcji, C++ potrzebuje jasno zdefiniowanej strategii pozwalającej decydować, której definicji funkcji użyć w poszczególnych wywołaniach, szczególnie jeśli przekazywanych jest wiele parametrów — i ma taką strategię. Proces ten jest nazywany *rozwiązywaniem przeciążeń*. Kompletny opis tej strategii wymagałby osobnego rozdziału, więc tylko przyjrzyjmy się z grubsza, jak się odbywa ten proces.

- **Etap 1.** — zbieranie listy funkcji potencjalnie przydatnych w danej sytuacji. Wybierane są funkcje i szablony funkcji mające taką samą nazwę jak funkcje wywołane.
- **Etap 2.** — spośród funkcji potencjalnie przydatnych wybierane są funkcje pasujące. Są to funkcje mające odpowiednią liczbę parametrów, dla których istnieją niejawne metody konwersji lub dokładne dopasowanie typu parametru. Na przykład jeśli funkcja jest wywoływana z parametrem typu `float`, przekazana wartość zostanie skonwertowana na typ `double`, aby pasowała do parametru `double`, ale w przypadku konkretyzacji będzie ona już dotyczyć typu `float`.
- **Etap 3.** — sprawdzenie, czy istnieje funkcja pasująca najlepiej. Jeśli tak, zostanie ona użyta. Jeżeli nie, wywołanie funkcji kończy się błędem.

Przyjrzyjmy się przypadkowi z pojedynczym parametrem funkcji, na przykład:

```
may('B');      // parametr aktualny jest typu char
```

Najpierw kompilator dobiera „podejrzanych”, czyli funkcje i szablony funkcji mające nazwę `may()`. Następnie wybierane są funkcje i szablony mające jeden parametr. Grono kandydatów obejmuje wszystkie poniższe funkcje o tej samej nazwie i z jednym argumentem wywołania:

```
void may(int);                          // #1
float may(float, float = 3);            // #2
void may(char);                         // #3
```

```
char * may(const char *);               // #4
char may(const char &);                 // #5
template<class T> void may (const T &); // #6
template<class T> void may (T *);       // #7
```

Pamiętajmy, że sprawdzane są dalej typy parametrów, a nie typy wartości zwracanych. Jednak dwóch kandydatów, #4 i #7, nie pasuje, gdyż typu liczbowego nie można niejawnie (czyli bez jawnego rzutowania) przekształcić na wskaźnik. Drugi szablon nadaje się, bo służy do wygenerowania specjalizacji, przy czym typ `T` to typ `char`. W ten sposób mamy pięć potencjalnie użytecznych funkcji, z których każda mogłaby zostać użyta, gdyby była jedyną deklaracją.

Następnie kompilator sprawdza, która z funkcji pasuje najlepiej. Istotna jest tu konwersja konieczna do wywołania funkcji. Ogólnie rzecz biorąc, dopasowania od najlepszych do najgorszych układają się następująco:

1. Dokładne dopasowania, w tym zwykłe funkcje przed szablonami.
2. Konwersje będące promocjami typów (na przykład automatyczna konwersja `char` i `short` na typ `int` oraz `float` na `double`).
3. Konwersje standardowe, na przykład konwersja typu `int` na `char` lub typu `long` na typ `double`.
4. Konwersje użytkownika, na przykład zdefiniowane w deklaracjach klas.

Na przykład funkcja #1 jest lepsza od funkcji #2, gdyż konwersja typu `char` na `int` to promocja (rozdział 3.), a konwersja `char` na `float` to konwersja standardowa (rozdział 3.). Funkcje #3, #5 i #6 są lepsze od #1 i #2, gdyż są to dokładne dopasowania. Funkcje #3 i #5 są lepsze od #6, ponieważ ta ostatnia jest szablonem. Pojawia się kilka pytań związanych z opisanymi zasadami wywoływania. Czym jest dopasowanie dokładne? I co jeśli mamy dwa takie dopasowania jak #3 i #5?

Zwykle, jak w powyższym przykładzie, dwa dokładne dopasowania świadczą o błędzie, chociaż od tej zasady bywają wyjątki. Jak widać, bez dalszej analizy tego tematu się nie obędzie!

Dopasowania dokładne i najlepsze

C++ dopuszcza pewne najprostsze konwersje w przypadku dopasowania dokładnego. Zestawiono je w tabeli 8.1; *Type* oznacza dowolny typ. Na przykład parametr wywołania `int` dokładnie pasuje do parametru `int &`. Zauważmy, że *Type* może mieć postać typu `char &`, więc te oczywiste konwersje to zamiana `char &` na `const char &`. Wiersz z zapisem *Type* (*lista-parametrów*) oznacza przekazanie wskaźnika funkcji o takiej samej sygnaturze i takiej samej wartości zwracanej. Przypomnijmy sobie wskaźniki funkcji z rozdziału 7. Pamiętajmy też, że można przekazać nazwę funkcji jako parametr funkcji oczekującej wskaźnika na funkcję. Słowo kluczowe `volatile` omówimy dalej w rozdziale 9.

Tabela 8.1. *Oczywiste konwersje dozwolone przy dokładnym dopasowaniu*

Z parametru wywołania	Na parametr
`Type`	`Type &`
`Type &`	`Type`
`Type []`	`* Type`
`Type (lista-parametrów)`	`Type (*) (lista-parametrów)`
`Type`	`const Type`
`Type`	`volatile Type`
`Type *`	`const Type *`
`Type *`	`volatile Type *`

Załóżmy, że mamy następujący kod funkcji:

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(ink);
```

Wtedy dokładne będą dopasowania wszystkich poniższych prototypów:

```
void recycle(blot);          // #1 blot-blot
void recycle(const blot);    // #2 blot-(const blot)
void recycle(blot &);        // #3 blot-(blot &)
void recycle(const blot &);  // #4 blot-(const blot &)
```

Zgodnie z oczekiwaniami, jeśli mamy kilka pasujących prototypów, kompilator nie potrafi zakończyć procesu rozwiązywania przeciążeń. Nie ma funkcji pasującej najlepiej, więc kompilator zgłasza błąd — prawdopodobnie w komunikacie pojawi się gdzieś słowo *ambiguous*, *niejednoznaczny*.

Jednak czasami rozwiązanie przeciążeń jest możliwe nawet wtedy, kiedy mamy dwie dokładnie dopasowane funkcje. Po pierwsze, wskaźniki i referencje nie-`const` są preferowane przy dopasowywaniu do wskaźników i referencji nie-`const`. Gdyby zatem w przykładzie z `recycle()` do dyspozycji były tylko funkcje #3 i #4, wybrana zostałaby #3, bo jej parametr nie zawiera słowa kluczowego `const`. Jednak rozróżnienie między `const` a nie-`const` dotyczy tylko danych wskaźnikowych i referencyjnych. Gdyby dostępne były tylko funkcje #1 i #2, błąd niejednoznaczności wystąpiłby i tak.

Inny przypadek, kiedy jedno dokładne dopasowanie jest lepsze od innego, to sytuacja, gdy jedna funkcja jest szablonem, a druga nie. Wtedy za lepiej dopasowaną uznawana jest funkcja niebędąca szablonem; zasada ta obejmuje także jawne specjalizacje.

Jeśli mamy dwa dokładnie dopasowane szablony, ale jeden z nich jest bardziej wyspecjalizowany, jest on uznawany za lepszy. Wobec tego jawna specjalizacja jest preferowana przed specjalizacją niejawną:

```
struct blot {int a; char b[10];};
template <class Type> void recycle (Type t);     // szablon
template <> void recycle<blot> (blot & t);       // specjalizacja blot
blot ink = {25, "spots"};
...
recycle(ink);     // użycie specjalizacji
```

Termin *najbardziej wyspecjalizowana* nie musi oznaczać specjalizacji jawnej; wskazuje raczej, że przy ustalaniu użytych typów kompilator musi przeprowadzić mniej konwersji. Weźmy na przykład pod uwagę dwa przykłady szablonów:

```
template <class Type> void recycle (Type t);      // #1
template <class Type> void recycle (Type * t);    // #2
```

Załóżmy, że program zawiera powyższe szablony oraz kod:

```
struct blot {int a; char b[10];};
blot ink = {25, "spots"};
...
recycle(&ink);     // adres struktury
```

Wywołanie `recycle(&ink)` pasuje do szablonu #1, gdzie `Type` jest interpretowany jako `blot *`. Wywołanie `recycle(&ink)` pasuje także do szablonu #2, tym razem `Type` to `blot`. Powoduje to dwie niejawne konkretyzacje szablonu, `recycle<blot *>(blot *)` oraz `recycle<blot>(blot *)`.

Z dwóch powyższych szablonów bardziej wyspecjalizowany jest szablon `recycle<blot *>(blot *)`, gdyż jego generowanie wymaga mniej konwersji. Wobec tego szablon #2 wskazuje już jawnie, że parametr funkcji był wskaźnikiem na typ `Type`, więc może być bezpośrednio utożsamiony z `blot`. Jednak w szablonie #1 `Type` jest parametrem funkcji, zatem `Type` musi być interpretowany jako wskaźnik `blot`. Wobec tego w szablonie #2 `Type` był specjalizowany jako wskaźnik i dlatego jest „bardziej wyspecjalizowany".

Zasady znajdowania najbardziej wyspecjalizowanego szablonu to *zasady uporządkowania częściowego* szablonów funkcji. Tak jak jawna konkretyzacja szablonów są one elementami języka C++ wprowadzonymi standardem C++98.

Przykład zasad uporządkowania częściowego

Teraz przyjrzyjmy się kompletnemu programowi używającemu uporządkowania częściowego do decydowania, którego szablonu należy użyć. Na listingu 8.14 pokazane są dwie definicje szablonów wyświetlających zawartość tablicy. Pierwsza definicja, szablon A, zakłada, że tablica zostanie przekazana jako parametr z danymi do pokazania. Szablon B zakłada, że elementami tablicy są wskaźniki na wyświetlane dane.

Listing 8.14. *tempover.cpp*

```
// tempover.cpp -- przeciążanie szablonów
#include <iostream>

template <typename T>             // szablon A
void ShowArray(T arr[], int n);

template <typename T>             // szablon B
void ShowArray(T * arr[], int n);

struct debts
{
    char name[50];
    double amount;
};

int main()
{
    using namespace std;
    int things[6] = {13, 31, 103, 301, 310, 130};
    struct debts mr_E[3] =
    {
        {"Ima Wolfe", 2400.0},
        {"Ura Foxe", 1300.0},
        {"Iby Stout", 1800.0}
    };
    double * pd[3];

// ustawienie wskaźników na pola amount struktur z tablicy Mr_E
    for (int i = 0; i < 3; i++)
        pd[i] = &mr_E[i].amount;

    cout << "Wyliczanie rzeczy pana E.:\n";
// things to tablica int
ShowArray(things, 6);  // używamy szablonu A
    cout << "Wyliczanie długów pana E.:\n";
// pd to tablica wskaźników na double
    ShowArray(pd, 3);       // używa szablonu B (bardziej wyspecjalizowanego)
    return 0;
}
```

```
template <typename T>
void ShowArray(T arr[], int n)
{
    using namespace std;
    cout << "szablon A\n";
    for (int i = 0; i < n; i++)
        cout << arr[i] << ' ';
    cout << endl;
}

template <typename T>
void ShowArray(T * arr[], int n)
{
    using namespace std;
    cout << "szablon B\n";
    for (int i = 0; i < n; i++)
        cout << *arr[i] << ' ';
    cout << endl;
}
```

Weźmy pod uwagę wywołanie:

```
ShowArray(things, 6);
```

Identyfikator `things` to nazwa tablicy `int`, więc pasuje on do szablonu:

```
template <typename T>            // szablon A
void ShowArray(T arr[], int n);
```

gdzie jako `T` jest brane `int`.

Teraz weźmy pod uwagę wywołanie:

```
ShowArray(pd, 3);
```

Tym razem `pd` to nazwa tablicy `double *`. Można je dopasować do szablonu A:

```
template <typename T>            // szablon A
void ShowArray(T arr[], int n);
```

Tym razem `T` zostanie zinterpretowane jako `double *`. Tak wygenerowana funkcja wyświetliłaby zawartość tablicy `pd` — trzy adresy. Wywołanie funkcji będzie pasowało także do szablonu B:

```
template <typename T>            // szablon B
void ShowArray(T * arr[], int n);
```

Tym razem `T` to typ `double`, a funkcja wyświetli dereferencje elementów `*arr[i]`, czyli wartości `double` wskazywane w tablicy. Z pokazanych dwóch szablonów bardziej specjalizowany jest B, gdyż przyjmuje założenie, że wszystkie elementy tablicy są wskaźnikami, więc ten szablon zostanie użyty.

Oto wyniki działania programu z listingu 8.14:

```
wyliczanie rzeczy pana E.:
szablon A
13 31 103 301 310 130
Wyliczanie długów pana E.:
szablon B
2400 1300 1800
```

Gdybyśmy z programu usunęli szablon B, kompilator użyłby do pokazania zawartości `pd` szablonu A, więc pokazane zostałaby adresy, a nie wartości. Łatwo to sprawdzić samemu.

Krótko mówiąc, proces rozwiązywania przeciążeń szuka najlepiej pasującej funkcji. Jeśli jest tylko jedna taka funkcja, jest ona wybierana. Jeżeli jest więcej niż jedna funkcja pasująca, ale tylko jedna z nich nie jest szablonem, ta jest wybierana. Gdy jest więcej kandydatów na dopasowanie i są to same szablony, a tylko jedna funkcja jest bardziej specjalizowana od reszty, to ta jest wybierana. Jeśli istnieją dwie lub więcej dobrych funkcji niebędących szablonami, wszystkie są tak samo wyspecjalizowane, to wywołanie funkcji jest niejednoznaczne i powoduje błąd. Kiedy nie ma żadnych pasujących wywołań, to oczywiście jest to błąd.

Prowokowanie dopasowania

W niektórych sytuacjach można sprowokować pożądane dopasowanie przeciążenia poprzez odpowiedni zapis wywołania funkcji. Weźmy program z listingu 8.15, który, nawiasem mówiąc, eliminuje prototyp szablonu i zawiera definicję szablonu funkcji na samym początku pliku. Tak jak w przypadku zwyczajnych funkcji, definicje funkcji szablonowych mogą pełnić równocześnie rolę własnego prototypu — jeśli funkcja (również szablon funkcji) jest definiowana przed miejscem pierwszego użycia, nie potrzebuje prototypu.

Listing 8.15. *choices.cpp*

```
// choices.cpp -- wybór szablonu
#include <iostream>
template<class T>  // albo template <typename T>
T lesser(T a, T b) // #1
{
    return a < b ? a : b;
}

int lesser (int a, int b) // #2
{
    a = a < 0 ? -a : a;
    b = b < 0 ? -b : b;
    return a < b ? a : b;
}

int main()
{
    using namespace std;
    int m = 20;
    int n = -30;
    double x = 15.5;
    double y = 25.9;

    cout << lesser(m, n) << endl;       // #2
    cout << lesser(x, y) << endl;       // #1 (z double jako T)
    cout << lesser<>(m, n) << endl;     // #1 (z int jako T)
    cout << lesser<int>(x, y) << endl; // #1 (z int jako T)
    return 0;
}
```

(Ostatnie wywołanie funkcji konwertuje argumenty typu `double` na typ `int`; niektóre kompilatory mogą w tym miejscu ostrzegać o ryzyku związanym z taką konwersją).

Oto wyniki działania programu z listingu 8.15:

```
20
15.5
-30
15
```

Program z listingu 8.15 zawiera szablon funkcji, która zwraca mniejszy z pary argumentów wywołania (w przypadku wersji zwyczajnej porównanie dotyczy wartości bezwzględnej argumentów). Jeśli definicja funkcji znajduje się przed miejscem jej pierwszego użycia, pełni rolę prototypu funkcji; można wtedy pominąć same prototypy. Weźmy pierwszą instrukcję wywołania funkcji:

```
cout << lesser(m, n) << endl; // #2
```

Typy obu argumentów wywołania funkcji pasują zarówno do szablonu, jak i do zwyczajnej wersji funkcji, więc kompilator wybiera funkcję zwyczajną; wynikiem wywołania będzie tu 20.

Następne wywołanie funkcji pasuje z kolei do szablonu, przy `T` równym `double`:

```
cout << lesser(x, y) << endl; // #1 (z double jako T)
```

Z kolei w instrukcji:

```
cout << lesser<>(m, n) << endl; // #1 (z int jako T)
```

Obecność nawiasów ostrych przy nazwie funkcji (`lesser<>(m, n)`) oznacza, że kompilator powinien wybrać wersję szablonową, a nie zwykłą, więc kompilator po analizie typów argumentów (`int`) konkretyzuje szablon funkcji dla `T` równego `int`.

Wreszcie w ostatniej instrukcji wywołania:

```
cout << lesser<int>(x, y) << endl; // #1 (z int jako T)
```

mamy jawne żądanie konkretyzacji szablonu z typem `int` jako `T` i ta właśnie funkcja jest tu wywoływana. Wartości `x` i `y` są w ramach wywołania funkcji rzutowane na typ `int`, a sama funkcja zwraca także wartość typu int, dlatego program wypisuje 15 zamiast 15.5.

Funkcje z różnymi typami argumentów

Wszystko się komplikuje jeszcze bardziej, jeśli mamy wywołanie funkcji z wieloma parametrami, które pasuje do wielu prototypów. Kompilator musi spróbować dopasować wszystkie parametry i jeśli znajdzie opcję lepszą od pozostałych, wybierze ją. Aby jedna funkcja była lepsza od innej, jej parametry muszą być równie dobrze dopasowane, a chociaż jeden z nich musi być dopasowany lepiej.

W książce tej nie chcemy analizować procesu dopasowywania na trudnych przykładach. Opisane tu zasady pozwalają oceniać dopasowanie wszystkich możliwych prototypów i szablonów.

Ewolucja szablonów funkcji

U zarania kariery języka C++ większość programistów nie przykładała uwagi do szablonów jako skutecznego mechanizmu uogólniania kodu. Ale sprytni i dociekliwi programiści zdołali przesunąć granicę użyteczności szablonów w sposób wręcz niewyobrażalny. Uwagi programistów, którzy parali się na poważnie programowaniem z użyciem szablonów, zostały uwzględnione w standardzie C++98 oraz w dodatkach do standardowej biblioteki szablonów. Postęp odbywał się dalej, a kolejni programiści eksplorują możliwości programowania uogólnionego i niekiedy odbijają się od ograniczeń mechanizmu szablonów. Z kolei ich opinie i pomysły stanowią podstawę dla zmian zatwierdzonych w standardzie C++11. Warto pokrótce przedstawić podnoszone problemy i zaproponowane w standardzie rozwiązania.

Co to za typ?

Przy pisaniu szablonu w C++98 nie zawsze można było łatwo ustalić typ do użycia w deklaracjach występujących w kodzie szablonu. Weźmy następujący (fragmentaryczny) przykład:

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    ?typ? xpy = x + y;
    ...
}
```

Jaki powinien być deklarowany typ dla xpy? Nie wiadomo przecież, jak użytkownicy będą stosowali szablon ft() i dla jakich typów będą go konkretyzować. Właściwy może się okazać typ T1, T2 albo wręcz zupełnie inny typ. Jeśli na przykład T1 to double, a T2 to int, właściwym typem dla sumy byłby typ double. A jeśli T1 to short, a T2 to int, właściwym typem dla sumy jest int. Dla T1 będącego short i T2 będącego char dodawanie oznacza wykonanie automatycznej promocji całkowitoliczbowej, a typem wynikowym dodawania jest int. Co więcej, jeśli T1 i T2 nie są typami wbudowanymi, tylko np. strukturami czy klasami, to mogą mieć własne przeciążone operatory +, co jeszcze bardziej komplikuje kwestię doboru typu wynikowego sumy. W C++98 dylemat wyboru typu dla xpy nie ma prostego rozwiązania.

Słowo decltype (C++11)

W C++11 zaproponowano rozwiązanie w postaci nowego słowa kluczowego: decltype. Można je stosować w taki sposób:

```
int x;
decltype(x) y; // y będzie tego samego typu co x
```

Argumentem wyrażenia decltype może być wyrażenie, więc w przykładzie z szablonem ft() można by rozwiązać problem typu xpy w następujący sposób:

```
decltype(x + y) xpy; // xpy będzie tego samego typu co wyrażenie x + y
xpy = x + y;
```

Instrukcje te można połączyć w deklarację z równoczesną inicjalizacją:

```
decltype(x + y) xpy = x + y;
```

Nasz szablon ft() możemy więc poprawić następująco:

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    decltype(x + y) xpy = x + y;
    ...
}
```

Słowo decltype jest narzędziem nieco bardziej skomplikowanym, niżby się zdawało na pierwszy rzut oka. Kompilator jest tu zmuszany do wykonania całej procedury wywnioskowania typu deklaracji. Załóżmy, że mamy następujący kod:

```
decltype(wyrażenie) zmienna;
```

Oto nieco uproszczona wersja koniecznej procedury wnioskowania.

Etap 1.: Jeśli *wyrażenie* jest prostym identyfikatorem nieujętym w nawias, to *zmienna* jest tego samego typu co identyfikator, z wszystkimi kwalifikatorami, takimi jak const, włącznie:

```
double x = 5.5;
double y = 7.9;
double &rx = x;
const double * pd;
decltype(x) w;       // w jest typu double
decltype(rx) u = y; // u jest typu double &
decltype(pd) v;      // v jest typu const double *
```

Etap 2.: Jeśli *wyrażenie* jest wywołaniem funkcji, *zmienna* otrzyma typ zgodny z typem wartości zwracanej tej funkcji:

```
long indeed(int);
decltype (indeed(3)) m; // m jest typu int
```

Uwaga

Nie dochodzi tu bynajmniej do obliczenia wartości *wyrażenia* (poprzez realizację wywołania funkcji). Typ wartości zwracanej jest ustalany na podstawie prototypu funkcji; jest to wystarczające do jednoznacznego określenia typu deklarowanej zmiennej.

Etap 3.: Jeśli *wyrażenie* jest l-wartością, to *zmienna* jest referencją do typu wyrażenia. Może się wydawać, że reguła ta oznacza, iż wcześniejszy przykład z `w` powinien być rozstrzygnięty jako typ referencyjny, skoro `w` jest l-wartością. Pamiętajmy jednak, że przypadek zmiennej `w` został załatwiony już na 1. etapie procedury i kompilator nie będzie jej kontynuował. Niniejszy etap dotyczy wyłącznie *wyrażeń* niebędących prostymi identyfikatorami. A więc czym? Cóż, w pierwszej kolejności przychodzi na myśl identyfikator ujęty w nawias:

```
double xx = 4.4;
decltype ((xx)) r2 = xx; // r2 jest typu double &
decltype(xx) w = xx;     // w jest typu double (dopasowanie z etapu 1.)
```

Co ciekawe, poza kontekstem `decltype` same nawiasy nie zmieniają ani wartości, ani l-wartościowości wyrażenia. Na przykład poniższe dwie instrukcje są traktowane jako tożsame:

```
xx = 98.6;
(xx) = 98.6; // () nie wpływa na zachowanie xx
```

Etap 4.: Jeśli nie udało się rozstrzygnąć typu na poprzednich etapach, *zmienna* będzie tego samego typu co *wyrażenie*:

```
int j = 3;
int &k = j
int &n = j;
decltype(j+6) i1;  // i1 jest typu int
decltype(100L) i2; // i2 jest typu long
decltype(k+n) i3;  // i3 jest typu int
```

Zwróćmy uwagę, że chociaż `k` i `n` są referencjami, to wyrażenie `k+n` nie ma typu referencyjnego: to zwyczajna suma dwóch wartości typu `int` i jako taka ma typ `int`.

Jeśli potrzebna jest większa liczba deklaracji tego samego typu, można ze słowem kluczowym `decltype` zastosować słowo `typedef`:

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    typedef decltype(x + y) xytype;
    xytype xpy = x + y;
```

```
    xytype arr[10];
    xytype & rxy = arr[2]; // rxy jest referencją
    ...
}
```

Alternatywna składnia funkcji (opóźniona deklaracja typu zwracanego)

Mechanizm `decltype` sam w sobie nie eliminuje wszystkich powiązanych problemów deklaracji typów w szablonach. Weźmy na przykład taki (niekompletny) szablon funkcji:

```
template<class T1, class T2>
?type? gt(T1 x, T2 y)
{
    ...
    return x + y;
}
```

Ponownie nie wiemy tu, jaki typ otrzymamy z dodawania `x` i `y`. Wydawałoby się, że deklaracja `decltype(x + y)` mogłaby rozwiązać kwestię typu wartości zwracanej, ale niestety w tym miejscu w kodzie szablonu parametry `x` i `y` nie zostały jeszcze zadeklarowane, więc nie mamy ich (i ich typów) w zasięgu deklaracji typu zwracanego funkcji. Słowo `decltype` z wyrażeniem angażującym parametry funkcji może występować jedynie *po* zadeklarowaniu tych parametrów. Dlatego C++11 dopuszcza nową składnię deklarowania i definiowania funkcji. Dla typów wbudowanych działa to tak jak poniżej. Prototyp:

```
double h(int x, float y);
```

Można zapisać za pomocą składni alternatywnej:

```
auto h(int x, float y) -> double;
```

W ten sposób można przesunąć deklarację typu wartości zwracanej za deklaracje typów parametrów funkcji. Kombinacja `-> double` z powyższego przykładu to tak zwana *opóźniona deklaracja typu zwracanego* (ang. *trailing return type*). Widać tu też kolejny element C++11 w postaci słowa `auto`, które pełni tu rolę symbolu zastępczego dla właściwego typu wartości zwracanej z funkcji, podawanego za listą parametrów. Podobnie można zrealizować definicję funkcji:

```
auto h(int x, float y) -> double
{/* ciało funkcji */};
```

Połączenie tej składni ze słowem `decltype` pozwala na ostateczne rozwiązanie problemu definicji naszego szablonu `gt()`:

```
template<class T1, class T2>
auto gt(T1 x, T2 y) -> decltype(x + y)
{
    ...
    return x + y;
}
```

Teraz `decltype` znajduje się za deklaracjami parametrów, więc `x` i `y` są nazwami w zasięgu kompilatora i mogą być użyte do określenia typu wartości zwracanej.

Podsumowanie

Obsługa funkcji w C++ jest znacznie rozbudowana względem języka C. Jeśli użyjemy słowa kluczowego `inline` w definicji funkcji i definicję tę umieścimy przed pierwszym wywołaniem funkcji, sugerujemy kompilatorowi uczynienie tej funkcji funkcją *inline*. Funkcje takie nie powodują skoku do odrębnej sekcji kodu, ale wywołanie funkcji jest zastępowane jej pełnym kodem. Cechę tę należy wykorzystywać jedynie w odniesieniu do krótkich funkcji.

Zmienna referencyjna to swoista odmiana wskaźnika, która umożliwia tworzenie aliasu (czyli innej nazwy) zmiennej. Zmienne referencyjne są używane przede wszystkim jako parametry funkcji przekazujących struktury i obiekty klas. Normalnie identyfikator deklarowany jako referencja danego typu może odnosić się tylko do danych tego typu. Jeśli jednak dana klasa jest pochodną innej klasy, jak klasa `ofstream` pochodząca od `ostream`, referencja typu bazowego może też odwoływać się do typu pochodnego.

W prototypach funkcji C++ można definiować wartości domyślne parametrów. Jeśli w wywołaniu funkcji takie parametry zostaną pominięte, program użyje ich wartości domyślnych. Jeżeli wywołanie funkcji zawiera wartość parametru, program użyje tej wartości, a nie wartości domyślnej. Parametry domyślne mogą występować tylko po prawej stronie listy parametrów. Jeśli zatem podamy wartość domyślną jakiegoś parametru, musimy podać też wartości wszystkich następnych parametrów.

Sygnatura funkcji to lista jej parametrów. Można zdefiniować dwie funkcje o takiej samej nazwie, o ile tylko mają różne sygnatury. Nazywamy to *polimorfizmem funkcji* lub *przeciążaniem funkcji*. Zwykle przeciąża się funkcje realizujące takie same zadania na różnych typach danych.

Szablony funkcji automatyzują proces przeciążania funkcji. Definiuje się funkcję, używając typu ogólnego oraz konkretnego algorytmu, a kompilator generuje później potrzebne definicje funkcji z konkretnymi typami danych używanymi w programie.

Pytania sprawdzające

1. Jakie funkcje są dobrymi kandydatami na funkcje *inline*?
2. Załóżmy, że funkcja `piosenka()` ma następujący prototyp:

   ```
   void piosenka(const char* tytul, int razy);
   ```

   a) Jak zmodyfikować prototyp, aby parametrowi `razy` nadać wartość domyślną 1?

   b) Jakie zmiany trzeba byłoby wtedy wprowadzić w definicji funkcji?

   c) Czy można parametrowi `tytul` nadać wartość domyślną `"Śpij, aniele mój"`?
3. Napisz przeciążoną wersję funkcji `iquote()` wyświetlającej parametr ujęty w podwójny cudzysłów. Napisz trzy jej wersje: dla liczb `int`, dla liczb `double` oraz dla napisów `string`.
4. Oto szablon struktury:

   ```
   struct pudelko
   {
       char producent[40];
       float wysokosc;
       float szerokosc;
       float dlugosc;
       float pojemnosc;
   };
   ```

   a) Napisz funkcję, która będzie miała jako parametr referencję struktury `pudelko` oraz która pokaże wartości wszystkich pól.

   b) Napisz funkcję, która będzie miała jako parametr referencję struktury `pudelko` oraz która wyliczy wartość pola `pojemnosc` jako iloczyn pozostałych wymiarów.

5. Jak należałoby zmienić listing 7.15, aby funkcje `fill()` i `show()` korzystały z parametrów typu referencyjnego?
6. Poniżej opisane są pewne pożądane zachowania. Wskaż, czy można je osiągnąć za pomocą parametrów domyślnych, przeciążania funkcji, jednego i drugiego, czy wcale. Podaj odpowiednie prototypy.
   a) `masa(gestosc, objetosc)` zwraca masę obiektu o zadanej gęstości i objętości. `masa(gestosc)` zwraca masę objętości 1 metra sześciennego przy ustalonej gęstości. Wszystkie wielkości są typu `double`.
   b) `repeat(10, "Czuję się świetnie")` pokazuje wskazany łańcuch 10-krotnie, a `repeat("Ale ty jesteś głupi.")` wyświetla podany łańcuch 5-krotnie.
   c) `srednia(3,6)` zwraca średnią dwóch parametrów `int`; `srednia(3.0, 6.0)` zwraca średnią wartości typu `double` jako `double`.
   d) Wywołanie `mangle("Miło mi cię widzieć")` zwraca znak `M` lub wskaźnik łańcucha `"Miło mi cię widzieć"` w zależności od tego, czy wynik jest przypisywany zmiennej `char`, czy zmiennej `char*`.
7. Napisz szablon funkcji zwracającej większy z jej parametrów.
8. Mając szablon z poprzedniego ćwiczenia oraz strukturę `pudelko` z ćwiczenia 4., podaj specjalizację szablonu, która będzie pobierała dwa parametry typu `pudelko` i zwracała jeden obiekt pudełko o większej pojemności.
9. Jakie typy zostaną przypisane do zmiennych `v1`, `v2`, `v3`, `v4` i `v5` w poniższym kodzie (załóżmy, że jest to fragment kompletnego programu)?

```
int g(int x);
...
float m = 5.5f;
float & rm = m;
decltype(m) v1 = m;
decltype(rm) v2 = m;
decltype((m)) v3 = m;
decltype (g(100)) v4;
decltype (2.0 * m) v5;
```

Ćwiczenia programistyczne

1. Napisz funkcję, która normalnie pobiera jeden parametr, adres łańcucha, po czym zaraz pokazuje ten łańcuch. Jeśli jednak podany zostanie niezerowy drugi parametr, napis ma się pojawić tyle razy, ile razy dotąd wywołano tę funkcję. Zauważmy, że drugi parametr nie mówi, ile razy należy pokazać napis. Owszem, funkcja jest nieco bzdurna, ale jej napisanie będzie dobrym ćwiczeniem utrwalającym wiedzę. Użyj opisanej funkcji w prostym programie, który pokaże jej działanie.
2. Struktura `Batonik` ma trzy pola: markę producenta, wagę (z częścią ułamkową) oraz liczbę kalorii (całkowitoliczbowo). Napisz program korzystający z funkcji, której parametry to referencja do typu `Batonik`, wskaźnik do typu `char`, wartość `double` oraz wartość `int`; funkcja ma używać ostatnich trzech wartości do ustawienia odpowiednich pól struktury. Ostatnie trzy parametry

mają mieć wartości domyślne "Millennium Munch", 2,85 i 350. Poza tym program powinien korzystać z funkcji pobierającej referencję do Batonika i wyświetlającej zawartość struktury. Należy użyć w miarę potrzeb const.

3. Napisz funkcję pobierającą referencję do obiektu string jako parametr i zamieniającą zawartość tego łańcucha na wielkie litery. Użyj funkcji toupper() opisanej w tabeli 6.4. Napisz program, który w pętli pozwoli przećwiczyć wprowadzanie danych do tej funkcji. Oto wynik przykładowego uruchomienia tego kodu:

```
Podaj łańcuch (q, aby skończyć): odejdź stąd
ODEJDŹ STĄD
Następny łańcuch (q, aby skończyć): niezły pasztet!
NIEZŁY PASZTET!
Następny łańcuch (q, aby skończyć): q
Do widzenia
```

4. Oto szkielet pewnego programu:

```
#include <iostream>
using namespace std;
#include <cstring>       // dla strlen(), strcpy()
struct stringy {
    char * str;          // wskazuje łańcuch
    int ct;              // długość łańcucha (bez \0')
};

// tutaj prototypy set(), show() i show()
{
    stringy beany;
    char testing[] = "Rzeczywistość to już nie to, co kiedyś.";

    set(beany, testing);     // pierwszy parametr jako referencja,
        // alokacja pamięci na wynik sprawdzania,
        // ustawienie pola str struktury beany tak, by wskazywała nowy blok;
        // kopiowanie testing do nowego bloku,
        // ustawienie pola ct zmiennej beany
    show(beany);       // pokazuje napis z pola raz
    show(beany, 2);    // pokazuje napis z pola dwukrotnie
    testing[0] = 'D';
    testing[1] = 'u';
    show(testing);     // pokazuje łańcuch testing raz
    show(testing, 3); // pokazuje łańcuch testing trzykrotnie
    show("Gotowe!");
    return 0;
}
```

Uzupełnij powyższy szkielet, definiując opisane funkcje i dodając im prototypy. Zauważmy, że potrzebne są dwie funkcje show(), obie z parametrami domyślnymi. W razie potrzeby użyj parametrów const. Zauważmy, że set() powinno korzystać z new do zaalokowania pamięci na łańcuch. Użyte tutaj techniki są podobne jak w przypadku projektowania i implementacji klas (w niektórych kompilatorach konieczna może być zmiana nazw plików nagłówkowych i usunięcie dyrektywy using).

5. Napisz szablon funkcji max5() pobierającej jako parametr tablicę pięciu wartości typu T, zwracającej największy element z tablicy. Wielkość tablicy jest ustalona, można ją na stałe zapisać w kodzie pętli, bez przekazywania jako parametr. Przetestuj program przy użyciu funkcji z tablicą pięciu wartości int i pięciu wartości double.

6. Napisz szablon funkcji maxn() pobierającej jako parametry tablicę typu T oraz liczbę elementów tej tablicy, zwracającej największy element tablicy. Przetestuj w programie 5-elementową tablicę wartości int i 4-elementową tablicę double. Program powinien korzystać ze specjalizacji przyjmującej

jako parametr tablicę typu `char`, zwracającej adres najdłuższego napisu. Jeśli jest kilka najdłuższych łańcuchów, funkcja powinna zwracać adres pierwszego z nich. Sprawdź specjalizację z tablicą pięciu łańcuchów.

7. Zmodyfikuj listing 8.14 tak, aby stosował dwa szablony funkcji o nazwie `SumArray()`, zwracającej sumę elementów tablicy, a nie wypisującej zawartości tej tablicy. Program powinien podawać też łączną liczbę przedmiotów i sumę zadłużenia.

Skorowidz

D

E

F

G

H

I

J

K

L

Ł

M

N

O

P

R

S

Ś

T

U

W

Z

Język C++. Szkoła programowania to starannie sprawdzony, sumiennie przygotowany i kompletny przewodnik po programowaniu w C++, przeznaczony dla programistów. Ten klasyczny już materiał pomocniczy dla wykładowców uczy zasad programowania, począwszy od kodu strukturalnego i projektowania metodą dekompozycji i analizy, przez klasy, dziedziczenie, szablony i wyjątki, po wyrażenia lambda, inteligentne wskaźniki i semantykę przeniesienia. Stephen Prata jako autor i wykładowca proponuje przemyślane, przejrzyste i wnikliwe wprowadzenie do języka C++. Wraz z mechanizmami samego języka omawia fundamentalne pojęcia i techniki programowania.

Szóste wydanie książki zostało uaktualnione i rozszerzone o omówienie elementów wprowadzonych do języka w nowym standardzie C++11. Książka przyda się nie tylko studentom kierunków informatycznych, ale również programistom biegłym w innych językach programowania, którzy chcieliby poznać język C++ lub pogłębić swoją wiedzę dotyczącą niuansów stosowania tego języka.

Dzięki tej książce:

- poznasz nowe elementy języka wprowadzone w standardzie C++ 11
- zaliczysz kurs programowania w języku C++
- błyskawicznie znajdziesz informacje potrzebne w codziennej pracy
- opanujesz język C++

Kompletny podręcznik do nauki C++!

Stephen Prata uczył astronomii, fizyki i informatyki w College of Marin w Kentfield (Kalifornia). Dyplom magisterski obronił w Kalifornijskim Instytucie Technologicznym, a doktorat na Uniwersytecie Kalifornijskim w Berkeley. Jest autorem i współautorem kilkunastu książek, w tym poprzednich wydań *Języka C++. Szkoły programowania* i *Języka C. Szkoły programowania*, nagrodzonych w 1990 roku wyróżnieniem Best How-to Computer Book Award stowarzyszenia Computer Press Association.

ISBN 978-83-246-4336-3
9 788324 643363
Cena: 99,00 zł