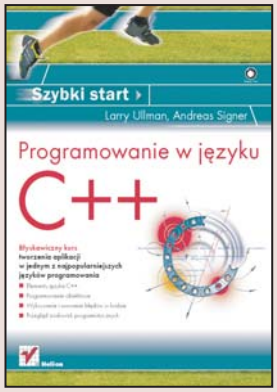
Programowanie w języku C++. Szybki start



Autorzy: Larry Ullman, Andreas Signer
Tłumaczenie: Przemysław Szeremiota, Andrzej Zawadzki
ISBN: 83-246-0447-2
Tytuł oryginału: C++ Programming: Visual QuickStart Guide
Format: B5, stron: 528

Błyskawiczny kurs tworzenia aplikacji w jednym z najpopularniejszych języków programowania

C++ to jeden z najpopularniejszych języków programowania. Mimo konkurencji ze strony innych, często nowocześniejszych języków, nadal jest powszechnie wykorzystywany, szczególnie przez twórców gier komputerowych, rozbudowanych aplikacji korporacyjnych i programów, od których wymaga się szczególnej szybkości i wydajności. Ten w pełni obiektowy język programowania, opracowany w połowie lat 80. w laboratoriach firmy Bell, jest stosunkowo łatwy do opanowania dzięki niewielkiemu zestawowi słów kluczowych, a oferuje ogromne możliwości.

„Programowanie w języku C++. Szybki start" to książka dla wszystkich osób, które chcą poznać ten język programowania, a nie mają czasu lub ochoty na wertowanie dziesiątek stron opisów teoretycznych. Przedstawia zasady pisania programów w C++ w sposób czytelny i obrazowy. Czytając ją, poznasz elementy języka C++, strukturę programów, zasady programowania obiektowego i sposoby realizacji różnych zadań programistycznych – od prostych operacji wejścia i wyjścia poprzez manipulowanie ciągami tekstowymi i liczbami aż do korzystania z szablonów i obsługi błędów. Każde z omawianych zagadnień zaprezentowane jest w postaci bogato ilustrowanej sekwencji czynności, co sprawi, że łatwo będzie Ci opanować opisywane w książce problemy.

- Kompilowanie i uruchamianie programów
- Typy danych i zmienne
- Instrukcje sterujące
- Operacje na plikach
- Definiowanie i stosowanie funkcji
- Programowanie obiektowe
- Zarządzanie pamięcią
- Modularyzacja kodu
- Szablony

Zostań programistą C++ w ekspresowym tempie

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Wprowadzenie 9

Rozdział 1. **Tworzymy prosty program** 17

Podstawy składni C++ ... 18
Kompilowanie programu w C++ ... 22
Wyświetlanie tekstu ... 26
Uruchamianie skompilowanego programu ... 30
Wstrzymywanie wykonania ... 32
Jak działają odstępy? ... 34
Dodawanie komentarzy do kodu źródłowego ... 36
Używanie środowiska programistycznego ... 39

Rozdział 2. **Proste zmienne i typy danych** 45

Deklarowanie zmiennych ... 46
Przypisywanie wartości zmiennym ... 52
Wypisywanie zmiennych ... 54
Formatowanie liczb ... 57
Jak działa konwersja typu? ... 60
Poznajemy znaki ... 64
Poznajemy łańcuchy znaków ... 67
Poznajemy stałe ... 70

Rozdział 3. **Operatory i instrukcje sterujące** 73

Operatory arytmetyczne ... 74
Instrukcja warunkowa if ... 80
Zastosowanie else i else if ... 84
Operator trójwartościowy ... 88
Operatory logiczne i porównania ... 92
Instrukcja switch ... 98
Operatory inkrementacji i dekrementacji ... 104
Pętla while ... 108
Pętla for ... 112

Rozdział 4. **Wejście, wyjście i pliki** **115**
Wczytywanie znaku ... **116**
Odrzucanie wejścia ... **121**
Wczytywanie liczb ... **124**
Wczytywanie łańcuchów znaków ... **127**
Wczytywanie wielu danych ... **130**
Wczytywanie całego wiersza ... **134**
Weryfikacja poprawności wejścia ... **137**
Zapisywanie do pliku ... **143**
Używanie pliku jako wejścia ... **148**

Rozdział 5. **Definiowanie własnych funkcji** **153**
Tworzenie prostych funkcji ... **154**
Tworzenie funkcji pobierających argumenty ... **159**
Ustalanie domyślnych wartości argumentów ... **165**
Tworzenie funkcji zwracających wartości ... **170**
Przeciążanie funkcji ... **176**
Zasięg zmiennych ... **180**

Rozdział 6. **Złożone typy danych** **185**
Praca z tablicami ... **186**
Praca ze wskaźnikami ... **192**
Struktury ... **210**
Powtórka z definiowania własnych funkcji ... **215**

Rozdział 7. **Przedstawiamy obiekty** **223**
Tworzymy prostą klasę ... **224**
Dodawanie metod do klas ... **228**
Tworzenie i używanie obiektów ... **233**
Definiowanie konstruktorów ... **237**
Definiowanie destruktorów ... **242**
Wskaźnik this ... **248**

Rozdział 8. **Dziedziczenie klas** **253**
Podstawy dziedziczenia ... **254**
Dziedziczenie konstruktorów i destruktorów ... **260**
Kontrola dostępu ... **265**
Przesłanianie metod ... **270**
Przeciążanie metod ... **274**
Nawiązywanie przyjaźni ... **277**

Rozdział 9. **Zaawansowane programowanie obiektowe** **283**
Metody i atrybuty statyczne ... 284
Metody wirtualne ... 291
Metody abstrakcyjne ... 298
Przeciążanie operatorów ... 304
Operator << ... 312
Dziedziczenie wielobazowe ... 317
Dziedziczenie wirtualne ... 324

Rozdział 10. **Diagnostyka i obsługa błędów** **329**
Techniki diagnostyczne ... 330
Kody błędów ... 336
Narzędzie assert() ... 342
Przechwytywanie wyjątków ... 348

Rozdział 11. **Dynamiczne zarządzanie pamięcią** **355**
Pamięć statyczna a pamięć dynamiczna ... 356
Przydzielanie obiektów ... 360
Dynamiczny przydział tablic ... 365
Zwracanie pamięci z funkcji i metod ... 370
Konstruktor kopiujący i operator przypisania ... 375
Statyczne rzutowanie typu obiektu ... 384
Dynamiczne rzutowanie typu obiektu ... 388
Unikanie wycieków pamięci ... 392

Rozdział 12. **Przestrzenie nazw i modularyzacja kodu** **395**
Praca z włączanymi plikami ... 396
Preprocesor C ... 410
Przestrzenie nazw ... 414
Zasięg a łączenie ... 422

Rozdział 13. **Praca z szablonami** **431**
Podstawy składni szablonów ... 432
Szablony z rozwinięciami w miejscu wywołania ... 444
Kontenery i algorytmy ... 448

Rozdział 14. **Zagadnienia dodatkowe** **459**
Znowu o ciągach ... 460
Operacje na plikach binarnych ... 474
Pobieranie argumentów z wiersza poleceń ... 489

Dodatek A **Narzędzia C++** **495**
Dev-C++ dla Windows ... **496**
Xcode dla systemu Mac OS X ... **501**
Narzędzia uniksowe ... **502**
Debugowanie z GDB ... **503**

Dodatek B **Zasoby dodatkowe** **505**
Strony WWW ... **506**
Tabele ... **508**

Skorowidz **513**

Definiowanie własnych funkcji

5

We wszystkich językach programowania funkcje służą do tego samego celu — grupowania instrukcji pod pewną etykietą. Późniejsze powtarzanie tych instrukcji jest bardzo łatwe, ponieważ do tego celu wystarczy użycie nazwy nadanej funkcji (czyli wywołanie jej). Spotkaliśmy się już z tym w praktyce, używając dostarczanej przez język C++ funkcji `cin.get()` i stosowanej przy pracach z plikami funkcji `close()`. W tym rozdziale dowiemy się, jak definiować i wywoływać własne funkcje.

Chociaż składnia potrzebna do definiowania funkcji jest prosta i logiczna, daje wiele możliwości. Na początek zdefiniujemy prostą funkcję, która nie pobiera argumentów i nie zwraca żadnych wartości. Następnie zajmiemy się takimi funkcjami, które zawsze wymagają przekazania im argumentów, oraz takimi, które pobierają je opcjonalnie. Potem utworzymy funkcje, które zwracają wartości pewnych obliczeń. Wychodząc od tych podstawowych tematów, przejdziemy do bardziej zaawansowanego zagadnienia — przeciążania funkcji. Rozdział kończy się omówieniem kwestii zasięgu zmiennych, czyli zasad, które opisują, w jakich miejscach w programie dostępna jest określona zmienna. Pojawią się także inne zagadnienia: funkcje rozwijane w miejscu wywołania i funkcje rekurencyjne oraz dodatkowe kwestie związane z deklarowaniem zmiennych.

Tworzenie prostych funkcji

Definiowanie funkcji w C++ składa się z dwóch kroków. Najpierw określa się składnię wywołania funkcji przez zadeklarowanie jej *prototypu*. Prototyp określa nazwę funkcji, liczbę i typ ewentualnie pobieranych przez nią argumentów oraz typ zwracanej wartości. Nie zawiera on jednak samej treści funkcji. Prototyp jest konieczny, by kompilator mógł sprawdzić, czy funkcja jest poprawnie używana w miejscach wywołania (kompilator porównuje prototyp z wywołaniami funkcji).

Składnia prototypu jest następująca:

```
typZwracany nazwaFunkcji (argumenty);
```

Jeśli funkcja nie zwraca żadnej wartości, to w miejscu wyrazu *typZwracany* umieścimy `void`. Wszystkie funkcje zdefiniowane w tym rozdziale będą zwracać co najwyżej proste wartości liczbowe, np. liczby typu `int` lub `float`. W kolejnych dwóch rozdziałach dowiemy się więcej o złożonych typach danych i o tym, jak sprawić, by funkcje zwracały wartości takich typów. Warto zauważyć, że nie ma znaczenia, czy pomiędzy nazwą funkcji a jej otwierającym nawiasem znajduje się spacja. W książce będziemy je czasem dodawać, aby uczynić kod bardziej czytelnym.

Reguły nazywania funkcji są takie same jak reguły nazywania zmiennych: można używać znaków alfanumerycznych oraz znaku podkreślenia. Należy pamiętać, że w języku C++ w nazwach funkcji rozróżniana jest wielkość liter. Wskazane jest więc obranie i konsekwentne trzymanie się jednej konwencji używania w nich wielkich liter. Zarówno w przypadku nazw zmiennych, jak i nazw funkcji niemożliwe jest używanie słów kluczowych języka (`int`, `break` itd.). Dozwolone jest natomiast dublowanie nazw istniejących już funkcji, ale dopóki nie zostanie omówione, kiedy i w jaki sposób można to wykorzystać, lepiej jest pozostać przy unikalnych i oczywistych nazwach.

Pobieranie przez funkcję argumentów jest samo w sobie dość skomplikowanym zagadnieniem i zajmiemy się nim na następnych kilku stronach. Tymczasem musimy pamiętać, że jeśli funkcja nie pobiera argumentów, to nawiasy pozostawiamy puste.

Mając na uwadze powyższe informacje, utworzymy prototyp funkcji `sayHello` (ang. *przywitaj się*), która nie pobiera argumentów i nie zwraca żadnej wartości:

```
void sayHello();
```

Drugim krokiem przy tworzeniu własnej funkcji po zadeklarowaniu prototypu jest zdefiniowanie jej. Nazywa się to *implementowaniem* funkcji. Składnia przypomina deklarację prototypu, ale kończy się zapisaniem ciała funkcji, w którym znajduje się właściwa definicja wykonywanych instrukcji.

```
void sayHello() {
    std::cout << "Halo.";
}
```

Gdy mamy już za sobą zapisanie prototypu i zdefiniowanie funkcji, możemy ją wywoływać tak samo jak każdą inną:

```
sayHello();
```

Zanim przejdziemy do przykładowego programu, warto powiedzieć, gdzie w pliku źródłowym należałoby umieścić powyższy kod. Prototyp funkcji zazwyczaj powinien znajdować się przed definicją funkcji `main()`. Dzięki temu kompilator zna składnię funkcji w momencie jej wywołania w głównej funkcji programu. Definicje zdefiniowanych przez użytkownika funkcji umieszcza się za definicją funkcji `main()`.

Szkielet kodu naszego programu mógłby więc wyglądać następująco:

```
#include biblioteki
prototypy funkcji
int main() {
    // Zrób coś.
    // Wywołanie funkcji użytkownika.
}
definicje funkcji
```

Właściwym w tym miejscu pytaniem jest: „Kiedy powinno się deklarować własne funkcje?”. Zazwyczaj sekwencję instrukcji zamienia się w funkcję wtedy, gdy chce się jej wielokrotnie używać lub gdy stanowi ona odrębną logicznie całość. Jednocześnie nie należy definiować własnych funkcji, jeśli ich odpowiedniki istnieją już w C++.

W tym rozdziale stworzymy kilka funkcji. Pierwszy przykład będzie wykonywał to, co robią prawie wszystkie programy w tej książce: wypisze komunikat i wstrzyma wykonanie programu do momentu naciśnięcia klawisza *Enter* lub *Return*. Powtarzany już wielokrotnie kod zamienimy tym razem w funkcję.

Aby zdefiniować własne funkcje i używać ich:

1. Utwórz nowy, pusty dokument tekstowy w edytorze tekstu lub środowisku programistycznym (listing 5.1).

```
// prompt.cpp - Listing 5.1
#include <iostream>
```

2. Dodaj prototyp funkcji.

```
void promptAndWait();
```

   Definiowana funkcja nosi nazwę `promptAndWait` (ang. *zgłoś i czekaj*). Nie pobiera argumentów (nawiasy są puste) i nie zwraca żadnej wartości (co zaznacza się użyciem słowa `void`).

3. Rozpocznij definicję funkcji `main()`.

```
int main() {
```

   Pomimo że w aplikacji używamy utworzonych przez siebie funkcji, to i tak musimy zdefiniować funkcję `main()`. Powinno być już utrwalone, że to właśnie ona jest wywoływana automatycznie, gdy uruchamiany jest program. W `main()` należy więc umieścić główne instrukcje każdego programu.

4. Spraw, aby w funkcji `main()` wykonywane były jakieś instrukcje.

```
std::cout << "Funkcja main() wykonuje
jakieś instrukcje...\n\n";
```

Listing 5.1. *Zdefiniowana przez użytkownika funkcja promptAndWait() obejmuje często wykorzystywany fragment kodu. W razie potrzeby można ją wywołać w funkcji main(), używając instrukcji promptAndWait()*

Listing

```
1   // prompt.cpp - Listing 5.1
2
3   // Potrzebujemy pliku iostream, by móc
4   // korzystać z cout i cin.
5   #include <iostream>
6
7   /* Prototyp funkcji.
8   * Funkcja nie pobiera argumentów.
9   * Funkcja nie zwraca żadnej wartości.
10  */
11  void promptAndWait();
12
13  // Początek funkcji main().
14  int main() {
15
16      // Zrób coś.
17      std::cout << "Funkcja main()
        ↳wykonuje jakieś
        ↳instrukcje...\n\n";
18
19      // Wywołanie funkcji promptAndWait().
20      promptAndWait();
21
22      // Zwrócenie wartości 0, by zaznaczyć
          ↳brak problemów.
23      return 0;
24
25  } // Koniec funkcji main().
26
27
28  // Definicja funkcji promptAndWait().
29  void promptAndWait() {
30
31      // Oczekiwanie, aż użytkownik naciśnie
          ↳Enter lub Return.
32      std::cout << "Naciśnij Enter lub
        ↳Return, aby kontynuować.\n";
33      std::cin.get();
34
35  } // Koniec funkcji promptAndWait().
```

Ponieważ celem tej aplikacji jest zademonstrowanie, w jaki sposób definiuje się i wywołuje własne funkcje, to nie skupiamy się w niej na niczym innym.

Jeśli zaszłaby taka potrzeba, można sprawić, aby funkcja `main()` wykonywała coś pożytecznego, np. zamieniała jednostki pewnych wartości, wczytywała dane itd.

5. Wywołaj zdefiniowaną przez nas funkcję.

```
promptAndWait();
```

Do wywołania funkcji wystarczy zapisanie jej nazwy, po której umieszcza się puste nawiasy. Tak jak w przypadku wszystkich instrukcji w języku C++, wiersz kończy się średnikiem.

6. Zakończ funkcję `main()`.

```
    return 0;
} // Koniec funkcji main().
```

Jeśli programy zawierają więcej niż jedną funkcję, użyteczną konwencją jest zaznaczanie komentarzem, gdzie się kończy każda z nich.

7. Za funkcją `main()` rozpocznij definicję funkcji `promptAndWait()`.

```
void promptAndWait() {
```

Definicja rozpoczyna się dokładnie tak samo jak prototyp, ale potem zawiera nawias klamrowy, który zaznacza początek ciała funkcji.

8. Dodaj ciało funkcji.

```
    std::cout<< "Naciśnij Enter lub Return,
    ↳aby kontynuować.\n";
    std::cin.get();
```

Funkcja wypisuje komunikat, w którym prosi o naciśnięcie klawisza i czeka aż to nastąpi. Gdyby funkcja `main()` wczytywała jakieś dane (nie robi tego w przykładowym programie), to celowe byłoby dodanie w pierwszym wierszu funkcji instrukcji:

```
std::cin.ignore(100,'\n');
```

lub

```
std::cin.ignore(std::gcount()+1);
```

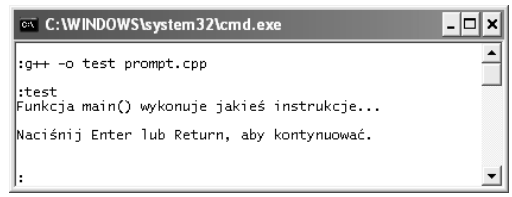
Powyższe wiersze stanowią zabezpieczenie na wypadek, gdyby np. nadmiarowy znak nowego wiersza pozostawał w buforze przed wywołaniem naszej funkcji (poprzedni rozdział zawiera omówienie składni i przyczyn użycia wspomnianych instrukcji).

9. Zakończ funkcję `promptAndWait()`.

```
} // Koniec funkcji promptAndWait().
```

Zamykający nawias klamrowy zaznacza koniec definicji funkcji. Dopisany za nim komentarz dodatkowo pomaga zorientować się, gdzie zaczynają się, a gdzie kończą definicje poszczególnych funkcji. Ponieważ funkcja nie zwraca żadnej wartości, nie pojawia się w niej instrukcja `return`.

10. Zapisz plik jako *prompt.cpp*, po czym skompiluj i uruchom program (rysunek 5.1).



Rysunek 5.1. *Mimo że program po uruchomieniu nie wydaje się robić nić nadzwyczajnego, to odbywające się za kulisami operacje — czyli wywoływanie zdefiniowanych przez nas funkcji — są naprawdę istotne*

Wskazówki

- Jeśli program kilkakrotnie używa tej samej funkcji lub jeśli mamy zbiór funkcji, których używamy w wielu aplikacjach, to celowe jest umieszczenie ich w osobnym pliku bibliotecznym. Więcej informacji na ten temat pojawi się w rozdziale 12. („Przestrzenie nazw i modularyzacja kodu”).
- Ponieważ programiści zajmują się głównie teoretycznymi rozważaniami na rozmaite tematy, to powstało wśród nich wiele teorii dotyczących tego, w jaki sposób należy nadawać nazwy definiowanym funkcjom. Jedyną powszechnie akceptowaną zasadą jest konsekwentne trzymanie się obranej konwencji nazewniczej (zarówno `promptAndWait`, `prompt_and_wait`, jak i `PromptAndWait` są zupełnie poprawne; nie należy jednak jednej funkcji nazwać `PromptAndWait`, a drugiej — `zrób_to_i_tamto`). Z oczywistych względów nazwy funkcji powinny też wskazywać na to, co funkcja robi.
- Możliwość definiowania własnych funkcji jest ważnym aspektem programowania obiektowego. Jak pokażemy w rozdziale 7. („Przedstawiamy obiekty”), obiektowe typy danych mogą mieć swoje własne funkcje, które nazywa się *metodami*.

Tworzenie funkcji pobierających argumenty

Chociaż prosta funkcja `promptAndWait()` dobrze realizuje swoje zadanie, to ilustruje jedynie drobną część wszystkich możliwości, jakie dają funkcje definiowane przez użytkownika. Kolejnym zagadnieniem, jakim się zajmiemy, jest pobieranie przez funkcje argumentów. Dla ścisłości — argumenty są wartościami, które mogą być przekazywane do funkcji i przez nie używane. Z zagadnieniem tym zetknęliśmy się już przy okazji używania `getline()`, którą stosowaliśmy, przekazując jej dwa argumenty: źródło danych (np. `cin`) oraz zmienną typu `string`, do której zapisywane były wprowadzone znaki.

Jeśli chcemy, aby funkcja pobierała argumenty, należy wymienić ich typy oraz nazwy wewnątrz nawiasów w prototypie i definicji:

```
// Prototyp:
void factorial (unsigned short num);
// Definicja:
void factorial (unsigned short num) {
    // Ciało funkcji.
    // Użycie zmiennej num.
}
```

Jeśli funkcja ma pobierać więcej argumentów, należy je kolejno oddzielić przecinkami, wypisując typ i nazwę:

```
void exponent (int num, int power);
```

Powyższa funkcja jest już zdefiniowana w C++ jako `pow()` (zobacz rozdział 3., „Operatory i instrukcje sterujące") i stanowi jedynie przykład składni.

Przy wywołaniu funkcji pobierającej argumenty, należy przesłać jej odpowiednie wartości. Mieliśmy już okazję zobaczyć, że można to robić na wiele sposobów: od przekazywania literałów aż po wyniki obliczeń. Wszystkie poniższe wywołania są poprawne:

```
int n = 2;
int x = 3;
exponent (n, x);
exponent (n, 5);
exponent (3, 4);
```

Należy koniecznie zapamiętać, że przekazywane do funkcji wartości muszą być zawsze poprawnego typu i zapisane w dobrej kolejności. Wynika to stąd, iż pierwsza z wartości będzie przypisana pierwszemu z jej argumentów, druga — drugiemu itd. Nie ma możliwości przekazania jedynie drugiego argumentu do funkcji, bez uprzedniego przekazania pierwszego.

W kolejnym przykładowym programie zdefiniowana przez nas funkcja będzie pobierać dwa argumenty: temperaturę i pojedynczy znak, który wskaże, czy jest ona wyrażona w stopniach Celsjusza, czy Fahrenheita. Funkcja wykona następnie odpowiednie konwersje i wypisze wyniki w znany nam już sposób.

Aby zdefiniować funkcje, które pobierają argumenty, i używać ich:

1. Utwórz nowy, pusty dokument tekstowy w edytorze tekstu lub środowisku programistycznym (listing 5.2).

```
// temperature.cpp - Listing 5.2
#include <iostream>
```

2. Dodaj prototypy funkcji.

```
void promptAndWait();
void convertTemperature(float tempIn, char
↳typeIn);
```

Nowozdefiniowana funkcja nosi nazwę `convertTemperature` (ang. *zamień temperaturę*). Pobiera dwa argumenty, z których pierwszy jest liczbą typu `float`, a drugi znakiem. W tym programie użyjemy również funkcji `promptAndWait()`.

Listing 5.2. *Zdefiniowana przez użytkownika funkcja convertTemperature() pobiera dwa argumenty i wypisuje wynik konwersji*

Listing

```
1   // temperature.cpp - Listing 5.2
2
3   // Potrzebujemy pliku iostream, by móc
4   // korzystać z cout i cin.
5   #include <iostream>
6
7   /* Prototyp pierwszej funkcji.
8     * Funkcja nie pobiera argumentów.
9     * Funkcja nie zwraca żadnej wartości.
10    */
11  void promptAndWait();
12
13  /* Prototyp drugiej funkcji.
14  * Funkcja pobiera dwa argumenty:
15  * jeden typu float i jeden typu char.
16  * Funkcja nie zwraca żadnej wartości.
17  */
18  void convertTemperature(float tempIn,
    ↳char typeIn);
19
20  // Początek funkcji main().
21  int main() {
22
23      // Deklaracja zmiennych.
24      float temperatureIn;
25      char tempTypeIn;
26
27      // Prośba o podanie temperatury.
28      std::cout << "Podaj temperaturę
        ↳i określ,
        czy jest ona wyrażona w stopniach
        Fahrenheita, czy Celsjusza:
        ↳[##.# C/F] ";
29      std::cin>> temperatureIn >>
        ↳tempTypeIn;
30
31      // Sprawdzenie poprawności wartości
          ↳zmiennej tempTypeIn.
32      if  (
33          (tempTypeIn == 'C') ||
```

Listing 5.2. — *ciąg dalszy*

```
Listing
34              (tempTypeIn == 'c') ||
35              (tempTypeIn == 'F') ||
36              (tempTypeIn == 'f')
37              ) {
38
39              // Wywołanie funkcji konwertującej.
40              convertTemperature
                ↳(temperatureIn,
                ↳tempTypeIn);
41
42          } else { // Niepoprawny typ danych,
                     ↳wypisanie komunikatu
                     ↳ o błędzie.
43              std::cout << "Obliczenia nie
                ↳mogły zostać
                przeprowadzone ze względu na
                ↳niepoprawne dane
                ↳wejściowe.\n\n";
44          }
45
46          // Wywołanie funkcji promptAndWait().
47          promptAndWait();
48
49          // Zwrócenie wartości 0, by zaznaczyć
              ↳brak problemów.
50          return 0;
51
52      } // Koniec funkcji main().
53
54
55      // Definicja funkcji promptAndWait().
56      void promptAndWait() {
57          std::cin.ignore(100, '\n');
58          std::cout << "Naciśnij Enter lub
            ↳Return, aby kontynuować.\n";
59          std::cin.get();
60      } // Koniec funkcji promptAndWait().
61
62
63      // Definicja funkcji convertTemperature().
64      void convertTemperature(float tempIn,
        ↳char typeIn) {
65
```

3. Rozpocznij definicję funkcji `main()`.

```
int main() {
    float temperatureIn;
    char tempTypeIn;
```

W funkcji `main()` będziemy potrzebować dwóch zmiennych do przechowywania wprowadzonych przez użytkownika wartości.

4. Poproś użytkownika o wprowadzenie temperatury.

```
std::cout << "Podaj temperaturę i określ,
↳czy jest ona wyrażona w stopniach
Fahrenheita, czy Celsjusza: [##.# C/F] ";
std::cout >> temperatureIn >> tempTypeIn;
```

Powyższe instrukcje są bliźniaczo podobne do tych, które pojawiły się w rozdziale 4. („Wejście, wyjście i pliki"). W pierwszym wierszu z `cin` nastąpi próba wczytania wartości zmiennoprzecinkowej i przypisania jej do zmiennej `temperatureIn`. Następnie do zmiennej `tempTypeIn` wczytany zostanie pojedynczy znak.

5. Jeśli wprowadzone dane są poprawnego typu, wywołaj nową funkcję.

```
if  (
    ( tempTypeIn == 'C') ||
    ( tempTypeIn == 'c') ||
    ( tempTypeIn == 'F') ||
    ( tempTypeIn == 'f')
    ) {
    convertTemperature (temperatureIn,
    ↳tempTypeIn);
```

W tej wersji programu wykonujemy podstawowe sprawdzenie poprawności danych, aby upewnić się, że zmienna `tempTypeIn` ma dopuszczalną wartość. Oczywiście, można by dokonać dużo więcej tego typu sprawdzeń (ich przykłady zawiera rozdział 4). Rzeczą wartą zapamiętania jest jednak to, że przed użyciem swojej funkcji należałoby w jakiś sposób upewnić się co do poprawności pobieranych przez nią argumentów (pomijamy przypadek funkcji sprawdzającej poprawność danych).

6. Dokończ instrukcję `if`.

```
} else {
    std::cout << "Obliczenia nie mogły
    ↳zostać przeprowadzone ze względu na
    ↳niepoprawne dane wejściowe.\n\n";
}
```

Jeśli zmienna `tempTypeIn` nie będzie miała dozwolonej wartości, to funkcja `convertTemperature()` nie zostanie wywołana i ukaże się komunikat o błędzie (rysunek 5.2). Posiłkując się przykładem z rozdziału 4., można by spowodować, że program ponownie zapyta o dane wejściowe i je wczyta.

7. Dokończ funkcję `main()`.

```
    promptAndWait();
    return 0;
}
```

8. Za ciałem funkcji `main()` zdefiniuj funkcję `promptAndWait()`.

```
void promptAndWait() {
    std::cin.ignore(100, '\n');
    std::cout<< "Naciśnij Enter lub Return,
    ↳aby kontynuować.\n";
    std::cin.get();
}
```

Choć powyższy kod różni się nieznacznie od pojawiającego się w programie z listingu 5.1, to używa się go w ten sam sposób. Ponieważ program wczytuje dane od użytkownika, wywoływana jest funkcja `ignore()`, która pozwala pozbyć się ewentualnego nadmiarowego wejścia.

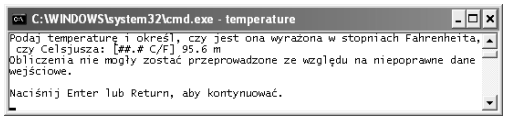
9. Rozpocznij definicję funkcji `convertTemperature()`.

```
void convertTemperature(float tempIn, char
↳typeIn) {
```

Listing 5.2. — *ciąg dalszy*

Listing

```
66          // Definicja stałych
67          // używanych w obliczeniach.
68          const unsigned short ADD_SUBTRACT
            ↳= 32;
69          const float RATIO = 5.0/9.0;
70
71          // Deklaracja zmiennych.
72          float tempOut;
73          char typeOut;
74
75          // Wykonanie odpowiedniej konwersji
              ↳w zależności od wartości tempTypeIn.
76          switch (typeIn) {
77
78              case 'C':
79              case 'c':
80                  // Obliczenia:
81                  tempOut = (tempIn /
                    ↳RATIO) + ADD_SUBTRACT;
82                  // Dla celów informacyjnych:
83                  typeOut = 'F';
84                  break;
85
86              case 'F':
87              case 'f':
88                  // Obliczenia:
89                  tempOut = (tempIn  -
                    ↳ADD_SUBTRACT) * RATIO;
90                  // Dla celów informacyjnych:
91                  typeOut = 'C';
92                  break;
93
94          } // Koniec instrukcji switch.
95
96          // Wypisanie wyników.
97          std::cout << tempIn << " stopni "
98          << typeIn << " jest równe "
99          << tempOut << " stopniom "
100         << typeOut << ".\n\n";
101
102     } // Koniec funkcji convertTemperature().
```



Rysunek 5.2. *Sprawdzanie poprawności wejścia, którym zajmowaliśmy się w rozdziale 4., jest bardzo ważnym aspektem każdego programu wczytującego dane*

Zgodnie z informacjami umieszczonymi w swoim prototypie funkcja pobiera dwa argumenty. Pierwszy z nich jest liczbą typu `float` o nazwie `tempIn`. Zostanie do niego przypisana wartość zmiennej `temperatureIn`, która jest pierwszym argumentem wywołania funkcji w `main()`. Analogicznie zmiennej `typeIn` zostanie przypisana wartość zmiennej `tempTypeIn` użytej w wywołaniu funkcji.

10. Zadeklaruj zmienne i stałe.

```
const unsigned short ADD_SUBTRACT = 32;
const float RATIO = 5.0/9.0;
float tempOut;
char typeOut;
```

Dwie zdefiniowane stałe są używane do obliczeń związanych z zamianą jednostek; zmienne używane są przy wypisywaniu wyników działania programu. Kod jest bardzo podobny do tego, który pojawił się w poprzednim rozdziale.

11. Wykonaj konwersję.

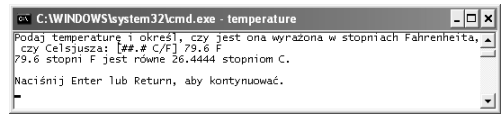
```
switch (typeIn) {
    case 'C':
    case 'c':
        tempOut = (tempIn / RATIO) +
        ↳ADD_SUBTRACT;
        typeOut = 'F';
        break;
    case 'F':
    case 'f':
        tempOut = (tempIn - ADD_SUBTRACT) *
        ↳RATIO;
        typeOut = 'C';
        break;
}
```

Nie pojawiają się tu żadne nowe elementy. W zależności od tego, w jakich jednostkach jest zadana temperatura wejściowa (Celsjusza lub Fahrenheita), wykonane zostaną odpowiednie obliczenia.

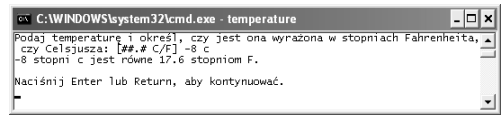
12. Wypisz wyniki i zakończ definicję funkcji.

```
        std::cout << tempIn << " stopni "
        << typeIn << " jest równe "
        << tempOut << " stopniom "
        << typeOut << ".\n\n";
    }
```

13. Zapisz plik jako *temperature.cpp*, po czym skompiluj i uruchom program (rysunek 5.3 i 5.4).



Rysunek 5.3. *Zamiana stopni Fahrenheita na stopnie Celsjusza*



Rysunek 5.4. *Zamiana stopni Celsjusza na stopnie Fahrenheita*

Wskazówki

- Technicznie rzecz biorąc, nadawanie nazw argumentom w prototypach funkcji nie jest konieczne — jest to jednak bardzo dobra praktyka.
- Bardzo często można się spotkać z terminami *parametry* i *argumenty* używanymi jako synonimy. Oba stosuje się do oznaczenia wartości przekazywanych do funkcji i pobieranych przez nie w momencie wywołania.
- Osoby mające doświadczenia z językiem C na pewno zauważą istotną różnicę między C a C++ dotyczącą deklaracji argumentów funkcji. Rozważmy przykładową deklarację:

  ```
  void nazwaFunkcji();
  ```

  W języku C oznacza ona funkcję o nieustalonych liczbie i typie argumentów. W C++ mówi ona, że funkcja nie pobiera żadnych argumentów. Właściwym odpowiednikiem w języku C byłoby więc:

  ```
  void nazwaFunkcji (void);
  ```

  Możesz pisać w ten sposób także w języku C++, jeśli bardziej odpowiedni wydaje Ci się jednoznaczny styl.
- Nawet jeśli funkcji pobiera wiele argumentów tego samego typu, to *nie można* używać skrótowego zapisu z oddzielającym zmienne przecinkiem:

  ```
  int n1, n2; // Poprawne
  ```

  Wiersz niepoprawny:

  ```
  void nazwaFunkcji (int n1, n2);
  ```

  Wiersz poprawny:

  ```
  void nazwaFunkcji (int n1, int n2);
  ```

Ustalanie domyślnych wartości argumentów

Innym sposobem, w jaki można modyfikować definicje swoich funkcji, jest wyznaczenie domyślnej wartości dla argumentu, dzięki czemu staje się on opcjonalny. Wykonuje się to przez dodanie przypisania wybranej wartości do argumentu w prototypie funkcji (nie w definicji):

```
void fN (int n1, int n2 = 6);
```

Dzięki takiej deklaracji oba poniższe wywołania są prawidłowe:

```
fN (1);
fN (5, 20);
```

Jeśli w wywołaniu funkcji dla argumentu o określonej wartości domyślnej przekażemy jakąś wartość, to zostanie ona przypisana do argumentu. Jeśli zaś nie przekażemy nic, to argumentowi zostanie przypisana wartość domyślna. W pierwszym z powyższych przykładów argument `n2` przyjmie więc wartość `6`, a w drugim — `20`.

Funkcje mogą zawierać dowolną liczbę argumentów z domyślnymi wartościami, jednak należy pamiętać o jednej zasadzie: wszystkie argumenty wymagane muszą wystąpić w deklaracji przed argumentami opcjonalnymi. Przykładowe dwa prototypy są poprawne:

```
void fN (int n1, int n2 = 6, int n3 = 4);
void fN (int n1 = 1, int n2 = 6);
```

Poniższe zaś są nieprawidłowe:

```
void fN (int n1 = 1, int n2); // ŹLE!
void fN (int n1, int n2 = 6, int n3); // ŹLE!
```

W kolejnym programie zdefiniujemy funkcję, która będzie dokonywać przewalutowania sumy w dolarach na sumę w euro, a następnie wypisze wyniki konwersji. Funkcja będzie pobierać dwa argumenty: współczynnik wymiany oraz liczbę zamienianych dolarów o domyślnej wartości 1.

Aby użyć domyślnych wartości:

1. Utwórz nowy, pusty dokument tekstowy w edytorze tekstu lub środowisku programistycznym (listing 5.3).

```
// currency.cpp - Listing 5.3
#include <iostream>
```

2. Dodaj dwa prototypy funkcji.

```
void promptAndWait();
void dollarsToEuros(float rate, unsigned
↳dollars = 1);
```

   Jedyną istotną różnicą pomiędzy funkcjami `convertTemperature()` z poprzedniego programu oraz `dollarsToEuros()` jest użycie domyślnej wartości argumentu. Zmienna `dollars` będzie domyślnie równa `1`, chyba że do funkcji w miejscu wywołania przekazana zostanie nowa wartość.

3. Rozpocznij definicję funkcji `main()`.

```
int main() {
    float conversionRate = 0.832339;
    unsigned dollarsIn;
```

   Kurs wymiany (`conversionRate`) jest zdefiniowany w programie jako niezmienna wartość, aby uniknąć wczytywania go za każdym razem przez użytkownika. Do przechowywania wczytanej kwoty w dolarach posłuży nam zmienna `dollarsIn` typu `unsigned int`.

4. Wypisz współczynnik wymiany, wywołując funkcję.

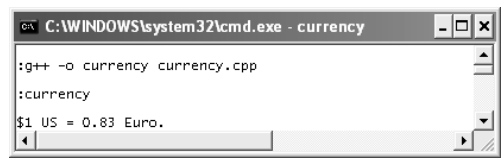
```
dollarsToEuros(conversionRate);
```

   Wywołanie funkcji z jednym tylko argumentem spowoduje, że użyta zostanie domyślna wartość zmiennej `dollars`, czyli `1`. Wynikiem tej operacji będzie wypisanie nagłówka zawierającego współczynnik zamiany (rysunek 5.5).

Listing 5.3. *Drugi argument funkcji dollarsToEuros() jest opcjonalny, ponieważ ma domyślną wartość 1*

Listing

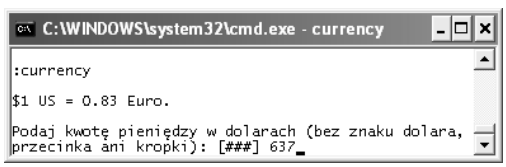
```
1    // currency.cpp - Listing 5.3
2
3    // Potrzebujemy pliku iostream, by móc
4    // korzystać z cout i cin.
5    #include <iostream>
6
7    /* Prototyp pierwszej funkcji.
8    * Funkcja nie pobiera argumentów.
9    * Funkcja nie zwraca żadnej wartości.
10   */
11   void promptAndWait();
12
13   /* Prototyp drugiej funkcji.
14   * Funkcja pobiera dwa argumenty:
15   * jeden typu float i jeden typu unsigned int.
16   * Argument typu unsigned int ma domyślną
       ↳wartość 1.
17   * Funkcja nie zwraca żadnej wartości.
18   */
19   void dollarsToEuros(float rate,
     ↳unsigned dollars = 1);
20
21   // Początek funkcji main().
22   int main() {
23
24       // Definicja zmiennych.
25       float conversionRate = 0.832339;
         ↳// $1 = 0.832339 Euro
```



Rysunek 5.5. *Przy wywołaniu funkcji dollarsToEuros() bez podania drugiego argumentu wypisywany jest pierwszy wiersz z domyślną kwotą jednego dolara*

Listing 5.3. *— ciąg dalszy*

```
Listing
26        unsigned dollarsIn;
27
28        // Wypisanie kursu walut przez wywołanie
            ↳funkcji.
29        dollarsToEuros(conversionRate);
30
31        // Prośba o podanie kwoty pieniędzy i
            ↳wczytanie jej.
32        std::cout << "Podaj kwotę
          ↳pieniędzy w dolarach (bez znaku
          ↳dolara, przecinka ani kropki):
          ↳[###] ";
33        std::cin >> dollarsIn;
34
35        // Wypisanie wysokości sumy po zamianie
            ↳walut.
36        dollarsToEuros(conversionRate,
          ↳dollarsIn);
37
38        // Wywołanie funkcji promptAndWait().
39        promptAndWait();
40
41        // Zwrócenie wartości 0, by zaznaczyć
            ↳brak problemów.
42        return 0;
43
44    } // Koniec funkcji main().
45
46
47    // Definicja funkcji promptAndWait().
48    void promptAndWait() {
49        std::cin.ignore(100, '\n');
50        std::cout << "Naciśnij Enter lub
          ↳Return, aby kontynuować.\n";
```



Rysunek 5.6. *Pytanie o liczbę zamienianych dolarów*

5. Poproś użytkownika o wprowadzenie wymienianej kwoty.

```
std::cout << "Podaj kwotę pieniędzy
w dolarach (bez znaku dolara, przecinka
↳ani kropki): [###] ";
std::cin >> dollarsIn;
```

Celem działania programu jest wczytanie kwoty w dolarach od użytkownika i zamiana jej na euro. Powyższy fragment kodu pyta użytkownika o dane wejściowe i wczytuje je (rysunek 5.6).

6. Wywołaj funkcję wykonującą zamianę.

```
dollarsToEuros(conversionRate, dollarsIn);
```

Funkcja `dollarsToEuros()` jest wywoływana ponownie, jednak tym razem z argumentem równym wprowadzonej przez użytkownika liczbie dolarów. Funkcja konwertująca użyje więc tyle dolarów, ile przypisano do zmiennej `dollarsIn` (zamiast domyślnej wartości `1`).

7. Dokończ funkcję `main()`.

```
    promptAndWait();
    return 0;
}
```

8. Zdefiniuj funkcję `promptAndWait()`.

```
void promptAndWait() {
    std::cin.ignore(100, '\n');
    std::cout<< "Naciśnij Enter lub Return,
    ↳aby kontynuować.\n";
    std::cin.get();
}
```

9. Rozpocznij definicję funkcji `dollarsToEuros()`.

```
void dollarsToEuros(float rate, unsigned
dolars) {
```

Należy pamiętać, żeby nie umieszczać w definicji funkcji domyślnych wartości argumentów, ponieważ skutkuje to błędami składniowymi. Wartości domyślne określa się jedynie w prototypach funkcji.

10. Ustaw formatowanie wyjścia.

```
std::cout.setf(std::ios_base::fixed);
std::cout.setf(std::ios_base::showpoint);
std::cout.precision(2);
```

Ponieważ wynikiem operacji będzie liczba zmiennoprzecinkowa (otrzymana przez pomnożenie współczynnika wymiany będącego liczbą rzeczywistą przez całkowitą liczbę dolarów), to celowe jest dodatkowe sformatowanie wyjścia. Powyższy kod pojawił się już w rozdziale 2. („Proste zmienne i typy danych"). Oznacza on, że wyświetlana ma być stała liczba cyfr po przecinku, przecinek dziesiętny ma być zawsze widoczny, a część całkowita ma być wyświetlana z dokładnością do dwóch miejsc po przecinku.

11. Wypisz wyniki i zakończ definicję funkcji.

```
    std::cout << "\n$" << dollars
    << " US = " << (rate * dollars)
    << " Euro.\n\n";
}
```

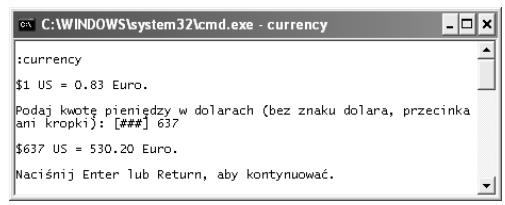
Ta część funkcji powinna być dość oczywista, ponieważ jej sednem jest jedynie mnożenie współczynnika wymiany przez liczbę dolarów. Aby wypisywany na ekranie tekst był nieco bardziej estetyczny, na początku i na końcu komunikatów znalazły się znaki nowego wiersza.

12. Zapisz plik jako *currency.cpp*, po czym skompiluj i uruchom program (rysunek 5.7).

Listing 5.3. — *ciąg dalszy*

Listing

```
51        std::cin.get();
52    } // Koniec funkcji promptAndWait().
53
54
55    // Definicja funkcji dollarsToEuros().
56    void dollarsToEuros(float rate,
      ↳unsigned dollars) {
57
58        // Ustawianie formatowania.
59        std::cout.setf
          ↳(std::ios_base::fixed);
60        std::cout.setf
          ↳(std::ios_base::showpoint);
61        std::cout.precision(2);
62
63        // Wypisanie wyników.
64        std::cout << "\n$" << dollars
65        << " US = " << (rate * dollars)
66        << " Euro.\n\n";
67
68    } // Koniec funkcji dollarsToEuros().
```



Rysunek 5.7. *Podana przez użytkownika kwota w dolarach jest zamieniana na euro i wypisywana na ekranie*

Wskazówki

- Możliwość ustalania domyślnych wartości argumentów jest kolejnym z elementów języka C++ niedostępnym w C.
- Niektórzy programiści preferują, aby domyślna wartość była zapisywana także w definicji funkcji. Robi się to, wykomentowując przypisanie wartości:

```
void fN (int n1, int n2 /* = 6 */) {
    // Ciało funkcji.
}
```


Funkcje rozwijane w miejscu wywołania

W C++ istnieje także inny typ funkcji możliwy do zdefiniowania przez użytkownika — funkcje rozwijane w miejscu wywołania, z atrybutem `inline`. Są one definiowane w inny sposób niż pozostałe funkcje pojawiające się w tym rozdziale i inaczej traktowane przez kompilator. Najprostszym sposobem utworzenia funkcji z atrybutem `inline` jest użycie słowa kluczowego `inline` i zdefiniowanie jej przed definicją funkcji `main()`.

```
#include <iostream>
inline void fN() {
    // Ciało funkcji.
}
int main (void) {...
```

Ponieważ w przypadku funkcji rozwijanych w miejscu wywołania całość definicji pojawia się przed funkcją `main()`, to nie ma potrzeby osobnego zapisywania prototypów.

Jasne scharakteryzowanie zysków, jakie wiążą się z używaniem takich funkcji, jest samo w sobie dość trudnym zadaniem. Czasami *mogą* być one szybsze. Jednocześnie często takimi nie są. W tej książce nie pojawia się chyba drugie, równie niejasne zdanie, więc pokrótce przybliżymy, co ono właściwie oznacza. Po pierwsze, kompilator, rozpoczynając pracę z naszym kodem, może potraktować zdefiniowane przez nas funkcje w dowolny sposób — tak jakby miały nadany atrybut `inline` lub nie. Nie musi tu mieć nawet znaczenia miejsce ani sposób, w jaki je zdefiniowaliśmy. Wynika stąd fakt, że nawet jeśli użyjemy atrybutu `inline`, to kompilator może go zignorować. Po drugie, zwiększenie wydajności, jakie daje rozwinięcie funkcji w miejscu wywołania, bardzo się różni w zależności od konkretnej funkcji i programu.

Jeśli nadamy funkcji atrybut `inline`, to efektem tego będzie zastąpienie instrukcji wywołania tej funkcji całością jej kodu, tak jakby był on napisany w miejscu wywołania.

Podsumowując: kiedy tworzy się własne funkcje na etapie nauki C++, dobrze jest pamiętać o istnieniu funkcji z atrybutem `inline`, ale nie należy ich używać. Dopiero po nauczeniu się języka oraz — co ważniejsze — zdobyciu wiedzy, jak skutecznie poprawia się efektywność kodu, można zacząć eksperymentować z poruszonym zagadnieniem i sprawdzać, czy jego stosowanie powoduje jakąkolwiek poprawę działania programu.

Tworzenie funkcji zwracających wartości

Została nam do omówienia jeszcze jedna ważna kwestia dotycząca funkcji: możliwość zwracania wartości. Spotkaliśmy się już z tym zagadnieniem wcześniej, ponieważ funkcja `size()` używana wraz ze zmiennymi typu `string` zwracała ilość znaków w łańcuchu, a zdefiniowana w pliku `cmath` funkcja `pow()` zwracała liczbę podniesioną do zadanej potęgi. Aby sprawić, że funkcja zwraca wartości, musimy użyć instrukcji `return` (jak już robiliśmy w przypadku funkcji `main()`). Może to być dowolna pojedyncza wartość:

```
return 1;
```

Zwracana wartość może być nawet wyrażona w postaci zmiennej:

```
int num = 6.25;
return num;
```

Poza poprawnym użyciem instrukcji `return` niezbędne jest także wskazanie w prototypie i definicji funkcji typu wartości zwracanej. Robi się to przez poprzedzenie nazwy funkcji identyfikatorem typu:

```
// Prototyp:
float multiply (float x, float y);
// Definicja:
float multiply (float x, float y) {
    return x * y;
}
```

Wartości zwróconych przez funkcję można używać w przypisaniach do zmiennych, w obliczeniach lub nawet jako argumentów wywołania innych funkcji:

```
float num = multiply (2.6, 890.245);
float sum = num + multiply (0.32, 6874.1);
std::cout << multiply (0.5, 32.2);
```

Listing 5.4. *Program używa osobnej funkcji do obliczenia i zwrócenia wartości silni liczby wprowadzonej przez użytkownika*

Listing

```
1   // factorial.cpp - Listing 5.4
2
3   // Potrzebujemy pliku iostream, by móc
4   // korzystać z cout i cin.
5   #include <iostream>
6
7   /* Prototyp pierwszej funkcji.
8   * Funkcja nie pobiera argumentów.
9   * Funkcja nie zwraca żadnej wartości.
10  */
11  void promptAndWait();
12
13  /* Prototyp drugiej funkcji.
14  * Funkcja pobiera jeden argument typu
      ↳unsigned short.
15  * Funkcja zwraca wartość typu unsigned long.
16  */
17  unsigned long returnFactorial(unsigned
    ↳short num);
18
19  // Początek funkcji main().
20  int main() {
21
22      // Deklaracja zmiennych.
23      unsigned short numberIn;
24
25      // Prośba o wprowadzenie liczby i
          ↳wczytanie jej.
26      std::cout << "Wprowadź nieujemną
        ↳liczbę całkowitą o małej
        ↳wartości: [##] ";
27      std::cin >> numberIn;
28
29      // Miejsce na sprawdzenie poprawności
          ↳danych -
30      // np. czy liczba mieści się w zakresie od 1
do 13.
31
```

Kolejny przykładowy program zaprezentuje w praktyce zwracanie wartości przez funkcje. Zdefiniujemy funkcję pobierającą pojedynczy argument typu całkowitego, zwracającą jego silnię. Program w rozdziale 3. demonstrował już, czym jest silnia: operacja ta jest oznaczana w matematyce jako n!, a jej wynik jest równy iloczynowi wszystkich liczb od 1 do n (tak więc 4! jest równe 1 * 2 * 3 * 4).

Aby zdefiniować funkcje zwracające wartości:

1. Utwórz nowy, pusty dokument tekstowy w edytorze tekstu lub środowisku programistycznym (listing 5.4).

```
// factorial.cpp - Listing 5.4
#include <iostream>
```

2. Dodaj dwa prototypy funkcji.

```
void promptAndWait();
unsigned long
returnFactorial(unsigned short num);
```

   Nowa funkcja `returnFactorial()` pobiera jeden argument o nazwie `num` typu `unsigned short` oraz zwraca wartość typu `unsigned long`. Typy te są dopasowane do wielkości liczb, ponieważ silnia nawet bardzo małej liczby bardzo szybko osiągnie górną granicę zakresu typu `long`.

3. Rozpocznij definicję funkcji `main()`.

```
int main() {
    unsigned short numberIn;
```

   Główna część programu wymaga użycia tylko jednej zmiennej, w której zapisywać będziemy wprowadzoną przez użytkownika liczbę. Jej typ jest celowo taki sam, jak typ argumentu funkcji `returnFactorial()`.

4. Poproś użytkownika o wprowadzenie liczby i przypisz ją do zmiennej `numberIn`.

```
std::cout << "Wprowadź nieujemną liczbę
↳całkowitą o małej wartości: [##] ";
std::cin >> numberIn;
```

Moglibyśmy w tym miejscu wykonać operacje sprawdzające poprawność wprowadzonych danych (np. zweryfikować, czy wprowadzona wartość jest liczbą całkowitą mieszczącą się w przedziale od 1 do 12). Na górną granicę wybrano liczbę 12, ponieważ 13! przekracza zakres zmiennych typu `unsigned int` na 32-bitowych komputerach (nie należy się martwić, jeśli ostatnie zdanie wydaje się nie mieć większego sensu — po lekturze rozdziału 10., „Diagnostyka i obsługa błędów", wszystko powinno stać się jasne).

5. Wypisz silnię wprowadzonej liczby.

```
std::cout << "Silnia liczby " << numberIn
<< " wynosi " << returnFactorial (numberIn)
<< ".\n\n";
```

Wprowadzona wartość jest ponownie wysyłana na wyjście wraz z obliczonym wynikiem działania (rysunek 5.8). Ponieważ funkcja `returnFactorial()` zwraca wartość, to jej wywołanie można umieścić bezpośrednio w instrukcji wysyłania danych do strumienia `cout`. Wynikiem działania instrukcji będzie wypisanie liczby na ekranie.

6. Dokończ funkcję `main()`.

```
    promptAndWait();
    return 0;
}
```

7. Zdefiniuj funkcję `promptAndWait()`.

```
void promptAndWait() {
    std::cin.ignore(100, '\n');
    std::cout<< "Naciśnij Enter lub Return,
    ↳aby kontynuować.\n";
    std::cin.get();
}
```

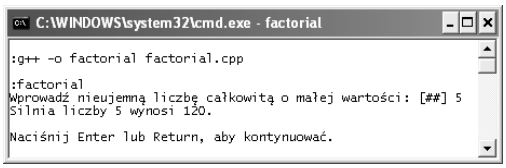
Listing 5.4. — *ciąg dalszy*

Listing

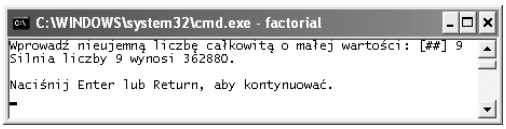
```
32        // Wypisanie wyników.
33        std::cout << "Silnia liczby " <<
          ↳numberIn
34        << " wynosi " << returnFactorial
          ↳(numberIn) << ".\n\n";
35
36        // Wywołanie funkcji promptAndWait().
37        promptAndWait();
38
39        // Zwrócenie wartości 0, by zaznaczyć
            ↳brak problemów.
40        return 0;
41
42    } // Koniec funkcji main().
43
44
45    // Definicja funkcji promptAndWait().
46    void promptAndWait() {
47        std::cin.ignore(100, '\n');
48        std::cout << "Naciśnij Enter lub
          ↳Return, aby kontynuować.\n";
49        std::cin.get();
50    } // Koniec funkcji promptAndWait().
51
52
53    // Definicja funkcji returnFactorial().
54    unsigned long returnFactorial(unsigned
      ↳short num) {
55
56        unsigned long sum = 1;
57
58        // Pętla przechodzi przez wszystkie liczby
59        // od 1 do num, wyznaczając iloczyn.
50        for (int i = 1; i <= num; ++i) {
61
62            // Mnożenie sumarycznego wyniku
                ↳przez współczynnik.
63            sum *= i;
```

Listing 5.4. *— ciąg dalszy*

```
64
65          } // Koniec pętli.
66
67          return sum;
68
69      } // Koniec funkcji returnFactorial().
```



Rysunek 5.8. *Wprowadzona liczba i jej silnia są wyświetlane na ekranie*



Rysunek 5.9. *Funkcja obliczy i zwróci wartości dowolnej nieujemnej liczby całkowitej o małej wartości*

8. Rozpocznij definicję funkcji `returnFactorial()`.

```
unsigned long returnFactorial (unsigned
↳short num) {
```

Definicja rozpoczyna się dokładnie tak samo jak prototyp.

9. Oblicz wartość silni.

```
unsigned long sum = 1;
for (int i = 1; i <= num; ++i) {
    sum *= i;
}
```

Powyższy kod został już przedstawiony w rozdziale 3. Wynik otrzymywany jest przez pomnożenie wszystkich liczb od 1 aż do wprowadzonej wartości. W tym celu zmienna `sum` inicjalizowana jest wartością `1`, której następnie przypisywany jest wynik mnożenia jej samej przez współczynnik `i`.

10. Zwróć wynik obliczeń i zakończ funkcję.

```
    return sum;
}
```

Zwracana zmienna `sum` zawiera obliczoną wartość silni. Zauważmy, że typ zmiennej (`unsigned long`) jest taki sam, jak zadeklarowany typ wartości zwracanej przez funkcję.

11. Zapisz plik jako *factorial.cpp*, po czym skompiluj i uruchom program (rysunek 5.9).

Wskazówki

- W funkcji możemy zapisać kilka instrukcji `return`, zawsze jednak wykona się tylko jedna z nich. Często można się spotkać z kodem podobnym do poniższego:

```
if (warunek) {
    return 1;
} else {
    return 0;
}
```

- Instrukcja `return` zatrzymuje działanie funkcji. Jakikolwiek kod występujący po instrukcji `return` nie będzie już wykonany:

```
int myF () {
    return 1;
    std::cout << "Funkcja zwróciła liczbę
1";
}
```

  Komunikat *Funkcja zwróciła liczbę 1* nie pojawi się na ekranie, ponieważ program nie wykona wiersza zawierającego odwołanie do `cout`.

- Prawdziwi formaliści mogą zapisywać:

```
return;
```

  nawet w funkcjach, które nie zwracają żadnej wartości.

Funkcje rekurencyjne

Innym rodzajem funkcji, które można zdefiniować w C++, a które nie są szczegółowo omawiane w książce, są funkcje *rekurencyjne* (ang. *recursive*). Rekurencja polega na wywoływaniu przez funkcję samej siebie i stworzeniu swoistej pętli. Funkcji takich raczej nie spotyka się zbyt często, ale czasami mogą w zdecydowany sposób uprościć rozwiązywanie złożonych problemów.

Załóżmy, że chcemy napisać program obliczający następną liczbę pierwszą większą od pewnej wczytanej wartości (podanie liczby 3 ma sprawić, że program zwróci liczbę 5 jako kolejną liczbę pierwszą, a wczytanie liczby 13 spowodować ma zwrócenie wartości 17). Szkielet algorytmu można przedstawić następująco:

1. Zwiększ wczytaną wartość o 1 (ponieważ zależy nam na liczbie pierwszej większej od wczytanej wartości).
2. Sprawdź, czy nowa wartość jest liczbą pierwszą (sprawdzając resztę z dzielenia przez wszystkie mniejsze od bieżącej wartości liczby pierwsze).
3. Jeśli wartość nie jest liczbą pierwszą, wróć do kroku 1.
4. Jeśli wartość jest liczbą pierwszą, zwróć ją.

Powyższe instrukcje można by zapisać w postaci funkcji rekurencyjnej:

```
unsigned int findPrime(unsigned int num) {
    bool isPrime = false; // Znacznik zakończenia obliczeń.
    ++num; // Inkrementacja.
    // Obliczenia sprawdzające,
    // czy liczba jest pierwsza.
    if (isPrime) { // Stop.
        return num;
    } else { // Sprawdzenie kolejnej liczby.
        findPrime(num);
    }
}
```

Powyższy kod utworzy zapętlone wywołanie, które będzie się wykonywać dopóki zmienna `isPrime` ma wartość `true`. Każde kolejne wywołanie funkcji `findPrime()` sprawdzać będzie, czy następna liczba nie jest szukaną.

Innym przykładem używania wywołań rekurencyjnych jest odczytywanie struktury katalogów na dysku. Można zdefiniować funkcję, która przegląda zawartość całego katalogu i w momencie napotkania innego katalogu wywoła samą siebie, używając znalezionego katalogu jako swojego nowego punktu startowego.

Najważniejszym zagadnieniem związanym z funkcjami rekurencyjnymi jest poprawne zdefiniowanie warunku zakończenia wywołań. W innym przypadku program wpadnie w pętlę nieskończoną lub po prostu zakończy pracę, gdyż zabraknie mu pamięci.

Przeciążanie funkcji

Przeciążanie funkcji w C++ jest trochę bardziej skomplikowanym zagadnieniem niż to, czym zajmowaliśmy się do tej pory, ale stanowi w dużej mierze o sile i elastyczności języka. Przeciążanie polega na zdefiniowaniu więcej niż jednej funkcji o takiej samej nazwie (i tym samym przeznaczeniu), ale o innej liście argumentów (zarówno co do typu, jak i liczby argumentów). Pobieżna analiza funkcji `convertTemperature()` (zobacz listing 5.2) pozwala dostrzec, dlaczego może być to przydatne.

Nasza funkcja pobiera zasadniczo jeden argument typu `float` oraz jeden typu `char`. Próba konwersji liczby całkowitej byłaby niemożliwa, ponieważ aktualna wersja funkcji oczekuje wartości typu `float` (kompilator dopuściłby takie wywołanie, jednak wartość `64` byłaby traktowana jako `64.0` — liczba rzeczywista, a nie całkowita). Moglibyśmy utworzyć nową funkcję, np. o nazwie `convertIntegerTemperature()`, ale wtedy w programie należałoby pamiętać o tym, aby dla jednego typu argumentów wywoływać funkcje o jednej nazwie, a dla innego o drugiej. Lepszym rozwiązaniem jest ponowne zdefiniowanie tej funkcji, ale w taki sposób, aby mogła pobierać także wartości całkowite.

Dwa prototypy deklarowanych funkcji mogłyby wyglądać następująco:

```
void convertTemperature (float tempIn, char
↳typeIn);
void convertTemperature (int tempIn, char
↳typeIn);
```

Wybór odpowiedniej funkcji przy wywołaniu odbywa się na podstawie sprawdzenia typów argumentów.

```
convertTemperature (54.9, 'F');
convertTemperature (16, 'C');
```

Pomimo że obie funkcje wykonują w tym przypadku te same operacje, kompilator na podstawie typów przekazywanych argumentów wygeneruje odpowiednie wywołania dwóch różnych funkcji.

Listing 5.5. *Dzięki dodaniu kolejnej definicji funkcji dollarsToEuros() funkcje o tej samej nazwie mogą być wykorzystywane do działania z liczbami rzeczywistymi oraz całkowitymi*

Listing

```
1   // overload.cpp - Listing 5.5
2
3   // Potrzebujemy pliku iostream, by móc
4   // korzystać z cout i cin.
5   #include <iostream>
6
7   /* Prototyp pierwszej funkcji.
8   * Funkcja nie pobiera argumentów.
9   * Funkcja nie zwraca żadnej wartości.
10  */
11  void promptAndWait();
12
13  /* Prototyp drugiej funkcji.
14  * Funkcja pobiera dwa argumenty:
15  * jeden typu foat i jeden typu char.
16  * Funkcja nie zwraca żadnej wartości.
17  */
18  void dollarsToEuros(float rate,
    ↳unsigned dollars);
19
20  /* Prototyp trzeciej funkcji (przeciążenie).
21  * Funkcja pobiera dwa argumenty, oba typu
     ↳float.
22  * Funkcja nie zwraca żadnej wartości.
23  */
24  void dollarsToEuros(float rate, float
    ↳dollars);
25
26  // Początek funkcji main().
27  int main() {
28
29      // Deklaracja zmiennych.
30      float conversionRate = 0.832339;
        ↳// $1 = 0.832339 Euro
31      unsigned dollarsIn;
32      float dollarsInFloat;
33
```

Możemy sobie zadać pytanie, kiedy ta składnia mogłaby się nam przydać? Najczęściej przeciążonych funkcji używa się do zapewnienia, aby ten sam kod mógł operować na różnych typach danych.

Listing 5.5. — *ciąg dalszy*

```
Listing
34          // Prośba o podanie kwoty pieniędzy i
              ↳wczytanie jej.
35          std::cout << "Podaj kwotę
            ↳pieniędzy w dolarach (bez znaku
            ↳dolara, przecinka ani kropki):
            ↳[###] ";
36          std::cin >> dollarsIn;
37
38          // Wypisanie kwoty po zamianie walut.
39          dollarsToEuros(conversionRate,
            ↳dollarsIn);
40
41          // Powtórzenie dla zmiennej typu float.
42          std::cout << "Podaj kwotę
            ↳pieniędzy w dolarach (bez znaku
            ↳dolara, przecinka ani kropki):
            ↳[###] ";
43          std::cin >> dollarsInFloat;
44
45          // Wypisanie wysokości sumy po zamianie
              ↳walut.
46          dollarsToEuros(conversionRate,
            ↳dollarsInFloat);
47
48          // Wywołanie funkcji promptAndWait().
49          promptAndWait();
50
51          // Zwrócenie wartości 0, by zaznaczyć
              ↳brak problemów.
52          return 0;
53
54      } // Koniec funkcji main().
55
56
57      // Definicja funkcji promptAndWait().
58      void promptAndWait() {
59          std::cin.ignore(100, '\n');
60          std::cout << "Naciśnij Enter lub
            ↳Return, aby kontynuować.\n";
61          std::cin.get();
62      } // Koniec funkcji promptAndWait().
63
```

Moglibyśmy zdefiniować jedną wersję pewnej funkcji tak, aby pobierała liczby całkowite, inną — żeby oczekiwała liczb rzeczywistych itd. Podobne działanie demonstrował poprzedni przykład. W praktyce pokażemy je w kolejnym programie. W rozdziale 7. („Przedstawiamy obiekty") oraz rozdziale 8. („Dziedziczenie klas") jeszcze raz pojawi się temat przeciążania funkcji oraz powiązanego z tym przysłaniania (ang. *overriding*).

Aby przeciążyć funkcję:

1. Otwórz plik *currency.cpp* (listing 5.3) w swoim edytorze tekstu lub środowisku programistycznym.
2. Zmień prototyp funkcji `dollarsToEuros()` tak, aby zmienna `dollars` nie miała już przypisanej domyślnej wartości (listing 5.5).

   ```
   void dollarsToEuros(float rate, unsigned
   dollars);
   ```

   Poleganie na wartościach domyślnych przy przeciążaniu funkcji może spowodować problemy z niejednoznacznością wywołań. Usuniemy więc problematyczną część deklaracji.
3. Dodaj drugi prototyp funkcji `dollarsToEuros()`.

   ```
   void dollarsToEuros(float rate, float
   ↳dollars);
   ```

   Powyższy prototyp jest taki sam jak poprzedni, z wyjątkiem typu drugiego argumentu. Ponieważ w nazwach funkcji w C++ rozróżniana jest wielkość liter, nazwy funkcji muszą być dokładnie takie same.
4. Wewnątrz funkcji `main()` zadeklaruj kolejną zmienną.

   ```
   float dollarsInFloat;
   ```

   W celu sprawdzenia, jak działa przeciążanie, w programie wczytamy najpierw liczbę całkowitą, a następnie liczbę rzeczywistą. Druga z nich przechowywana będzie w powyższej zmiennej.

   Kolejne trzy kroki również będą wymagały wprowadzenia zmian do treści funkcji `main()`.

5. Usuń pierwsze wywołanie funkcji `dollarsToEuros()`.

W pierwotnej wersji programu funkcja była wywoływana z użyciem domyślnej wartości drugiego argumentu (czyli liczby dolarów). Gdybyśmy zachowali domyślne wartości w obu funkcjach i spróbowali wywołać jedną z nich, przekazując tylko jeden argument, to kompilator wygenerowałby błąd, gdyż nie byłby w stanie wybrać odpowiedniej funkcji (rysunek 5.10).

6. Poproś użytkownika o wprowadzenie liczby rzeczywistej odpowiadającej liczbie dolarów.

```
std::cout << "Podaj kwotę pieniędzy
↳w dolarach (bez znaku dolara, przecinka
↳ani kropki): [###] ";
std::cin >> dollarsInFloat;
```

7. Wywołaj funkcję `dollarsToEuros()`, używając wprowadzonej wartości jako drugiego argumentu.

```
dollarsToEuros(conversionRate,
↳dollarsInFloat);
```

Ponieważ zmienna `dollarsInfloat` jest typu `float`, wywołana zostanie druga ze zdefiniowanych funkcji `dollarsToEuros()`.

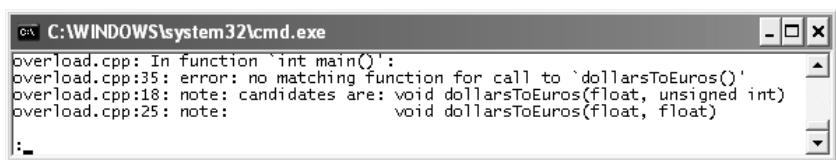
8. Po istniejących definicjach dodaj definicję drugiej funkcji `dollarsToEuros()`.

```
void dollarsToEuros(float rate, float
↳dollars) {
    std::cout.setf(std::ios_base::fixed);
    std::cout.setf
    ↳(std::ios_base::showpoint);
```

Listing 5.5. — *ciąg dalszy*

Listing

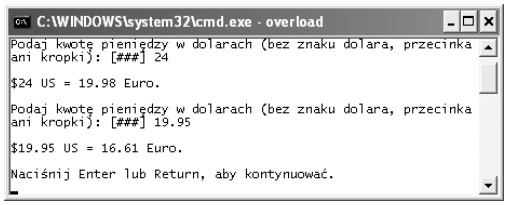
```
64
65    // Definicja funkcji dollarsToEuros().
66    void dollarsToEuros(float rate,
      ↳unsigned dollars) {
67
68        // Ustawianie formatowania.
69        std::cout.setf
          ↳(std::ios_base::fixed);
70        std::cout.setf
          ↳(std::ios_base::showpoint);
71        std::cout.precision(2);
72
73        // Wypisanie wyników.
74        std::cout << "\n$" << dollars
75        << " US = " << (rate * dollars)
76        << " Euro.\n\n";
77
78    } // Koniec funkcji dollarsToEuros().
79
80
81    // Definicja funkcji dollarsToEuros()
       ↳(przeciążenie).
82    void dollarsToEuros(float rate, float
      ↳dollars) {
83
84        // Ustawianie formatowania.
85        std::cout.setf
          ↳(std::ios_base::fixed);
86        std::cout.setf
          ↳(std::ios_base::showpoint);
87        std::cout.precision(2);
88
```



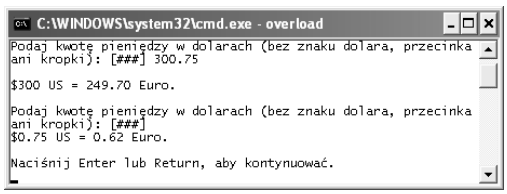
Rysunek 5.10. *Ponieważ w obu definicjach występuje domyślna wartość argumentu, kompilator nie zdecyduje, która z funkcji jest wywoływana*

Listing 5.5. *— ciąg dalszy*

```
Listing
89          // Wypisanie wyników.
90          std::cout << "\n$" << dollars
91          << " US = " << (rate * dollars)
92          << " Euro.\n\n";
93
94      } // Koniec funkcji dollarsToEuros()
            (przeciążenie).
```



Rysunek 5.11. *Możemy teraz wywoływać w ten sam sposób funkcję obsługującą zmienne typu int lub float. Po wprowadzeniu wartości całkowitej (np. 24) użyta zostanie pierwsza wersja funkcji. Przy przekazaniu liczby rzeczywistej — druga wersja*



Rysunek 5.12. *Program można by ulepszyć, dodając dokładniejsze sprawdzanie poprawności danych i zarządzanie buforem wejściowym. Pozwoliłoby to uniknąć niepożądanych wyników, jak na rysunku*

```
    std::cout << "\n$" << dollars
        << " US = " << (rate * dollars)
        << " Euro.\n\n";
}
```

Jedyną różnicą pomiędzy funkcjami jest typ drugiego argumentu, w tym przypadku: `float`.

9. Zapisz plik jako *overload.cpp*, po czym skompiluj i uruchom program (rysunek 5.11).

Wskazówki

- Możemy zmodyfikować obie funkcje `dollarsToEuros()` tak, aby zwracały obliczoną liczbę zamiast ją wypisywać. Inne możliwe rozszerzenie programu to odrzucenie ewentualnego nadmiarowego wejścia przed ponownym wczytaniem kwoty w dolarach. W obecnej wersji wpisanie jako kwoty liczby `300.75` spowoduje, że `cin` przypisze do pierwszej zmiennej liczbę `300`, pozostawiając `.75` w buforze. Liczba ta zostanie zinterpretowana jako oczekiwania wartość rzeczywista (rysunek 5.12).
- Nie można przeciążać funkcji poprzez zmianę jedynie typu wartości zwracanej. Przeciążone funkcje *mogą* się jednak między sobą nim różnić.
- Ponieważ obie funkcje `dollarsToEuros()` pobierają dwa argumenty i różnią się między sobą drugim z nich, celowe byłoby odwrócenie kolejności argumentów i umieszczenie najpierw kwoty w dolarach, aby jasno było widać różnice pomiędzy funkcjami.
- Podkreślmy jeszcze raz, że funkcje przeciąża się w celu wykonywania tych samych operacji na różnych typach danych. Jeśli potrzebujemy funkcji, która pobiera w różnych sytuacjach np. dwa lub trzy argumenty, to lepiej zdefiniować tylko jedną funkcję z trzecim argumentem opcjonalnym (określając jego wartość domyślną).

Zasięg zmiennych

Skoro już wiadomo, jak pisze się funkcje, nadszedł czas, by powrócić do tematu zmiennych. Chociaż nie mówiliśmy o tym wcześniej, ze zmiennymi związane jest pojęcie *zasięgu* (ang. *scope*) — granic świata, w którym każda z nich istnieje. Jak już pokazane zostało poprzednio, po zadeklarowaniu zmiennej można się do niej odnosić. Jednak obszar, w którym jest to dozwolone, zależy od miejsca deklaracji zmiennej. Oto kilka najważniejszych zasad:

- Zmienna zadeklarowana wewnątrz bloku kodu jest dostępna jedynie wewnątrz tego bloku. Blokiem nazywamy ciąg instrukcji zawarty pomiędzy nawiasami klamrowymi, np. w pętlach, instrukcjach warunkowych czy nawet:

```
{
// Oto blok.
}
```

- Zmienna zadeklarowana wewnątrz funkcji jest dostępna jedynie w tej funkcji. Zmienne takie nazywamy *lokalnymi* (ang. *local*).
- Zmienne zadeklarowanie poza ciałem jakiejkolwiek funkcji są dostępne w każdej zdefiniowanej po niej funkcji. Zmienne takie nazywamy *globalnymi* (ang. *global*).

Pierwszą zasadę demonstruje poniższy kod:

```
for (int i = 1; i <= 10; ++ i) {
    std::cout << i << "\n";
}
/* zmienna i nie istnieje poza pętlą! */
```

a także poniższy (choć trochę mniej użyteczny) fragment kodu:

```
if (1) {
    bool isTrue = true;
} // zmienna isTrue już nie istnieje!
```

Listing 5.6. *Poniższy prosty program pokazuje, jak zmienia się zasięg istnienia różnych zmiennych, w zależności od miejsca ich definicji*

Listing

```
1   // scope.cpp - Listing 5.6
2
3   // Potrzebujemy pliku iostream, by móc
4   // korzystać z cout i cin.
5   #include <iostream>
6
7   // Potrzebujemy pliku string, by móc
8   // operować łańcuchami znaków.
9   #include <string>
10
11  /* Prototyp pierwszej funkcji.
12  * Funkcja nie pobiera argumentów.
13  * Funkcja nie zwraca wartości.
14  */
15  void promptAndWait();
16
17  /* Prototyp drugiej funkcji.
18  * Funkcja nie pobiera argumentów.
19  * Funkcja nie zwraca żadnej wartości.
20  */
21  void nameChange();
22
23  // Deklaracja zmiennej globalnej.
24  std::string gName;
25
26  // Początek funkcji main().
27  int main() {
28
29      // Deklaracja zmiennej lokalnej.
30      std::string name = "Andi";
31
32      // Przypisanie nowej wartości zmiennej
        ↳globalnej.
33      gName = name;
34
35      // Wypisanie wartości zmiennych
```

Listing 5.6. — *ciąg dalszy*

```
Listing
36        std::cout << "Na początku wartość
          ↳zmiennej name to " << name << ",
          ↳a gName " << gName << ".\n";
37
38        // Wywołanie funkcji.
39        nameChange();
40
41        // Ponowne wypisanie wartości.
42        std::cout << "Po wywołaniu funkcji
          ↳wartość zmiennej name to "
          ↳<< name << ", a gName to " <<
          ↳gName << ".\n";
43
44        // Wywołanie funkcji promptAndWait().
45        promptAndWait();
46
47        // Zwrócenie wartości 0, by zaznaczyć
            ↳brak problemów.
48        return 0;
49
50    } // Koniec funkcji main().
51
52
53    // Definicja funkcji promptAndWait().
54    void promptAndWait() {
55        std::cout << "Naciśnij Enter lub
          ↳Return, aby kontynuować.\n";
56        std::cin.get();
57    } // Koniec funkcji promptAndWait().
58
59
60    // Definicja funkcji nameChange().
61    void nameChange() {
62
63        // Deklaracja zmiennej lokalnej.
64        std::string name = "Larry";
65
66        // Wypisanie wartości.
67        std::cout << "Wewnątrz funkcji
          ↳początkowa wartość zmiennej name
          ↳to " << name << ", a gName " <<
          ↳gName << ".\n";
```

Zgodnie z drugą zasadą zmienna globalna definiowana jest poza jakąkolwiek funkcją:

```
unsigned short age = 60;
int main() {...}
void sayHello() {...}
```

Ponieważ zmienna `age` istnieje w zasięgu globalnym, to można się do niej odwoływać zarówno w funkcji `main()`, jak i `sayHello()`. Kolejny program zademonstruje w praktyce kwestie związane z zasięgiem zmiennych, uwzględniając zwłaszcza trzecią z omówionych zasad, która bardzo często jest mylnie rozumiana przez początkujących programistów. W aplikacji zdefiniujemy zmienną globalną, aby można było zobaczyć, w jaki sposób różni się ona od zmiennych lokalnych.

Aby zapoznać się z zasięgiem zmiennych:

1. Utwórz nowy, pusty dokument tekstowy w edytorze tekstu lub środowisku programistycznym (listing 5.6).

   ```
   // scope.cpp - Listing 5.6
   #include <iostream>
   #include <string>
   ```

2. Dodaj dwa prototypy funkcji.

   ```
   void promptAndWait();
   void nameChange();
   ```

   Nowa funkcja `nameChange()` (ang. *zmień imię*) nie pobiera argumentów i nic nie zwraca. Użyjemy jej do podkreślenia różnic pomiędzy zmiennymi istniejącymi wewnątrz funkcji `main()`, zmiennymi lokalnymi innych funkcji oraz zmiennymi zdefiniowanymi poza funkcjami.

3. Zdefiniuj zmienną globalną.

   ```
   std::string gName;
   ```

   Powyższa zmienna, utworzona poza funkcjami, będzie dostępna we wszystkich zdefiniowanych następnie funkcjach.

4. Rozpocznij definicję funkcji `main()`.

```
int main() {
    std::string name = "Andi";
```

Zdefiniowana powyżej `name` (ang. *imię*) jest zmienną lokalną funkcji `main()`. Może być teraz używana gdziekolwiek wewnątrz funkcji, poczynając od tego miejsca (aż do nawiasu klamrowego zamykającego funkcję).

5. Przypisz wartość do `gName` i wypisz wartości obu zmiennych.

```
gName = name;
std::cout << "Na początku wartość zmiennej
↳name to " << name << ", a gName " <<
↳gName << ".\n";
```

W pierwszym wierszu do globalnej zmiennej `gName` przypisywana jest wartość zmiennej `name`. Obie zmienne mają więc te same wartości, co potwierdzamy wypisaniem ich na ekranie.

6. Wywołaj funkcję `nameChange()`i wypisz ponownie wartości zmiennych.

```
nameChange();
std::cout << "Po wywołaniu funkcji wartość
↳zmiennej name to " << name << ",
↳a gName " << gName << ".\n";
```

Jak się niebawem przekonamy, funkcja `nameChange()` przypisze wartości obu zmiennym. Dla zademonstrowania, jaki wpływ ma to na zmienne dostępne w funkcji `main()`, po wywołaniu funkcji wypiszemy na ekran ich wartości.

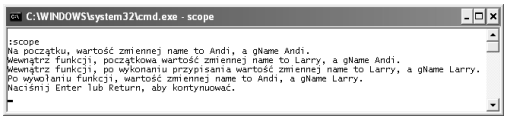
7. Dokończ funkcję `main()`.

```
    promptAndWait();
    return 0;
}
```

Listing 5.6. — *ciąg dalszy*

Listing

```
68
69          // Przypisanie nowej wartości zmiennej
              ↳globalnej.
70          gName = name;
71
72          // Ponowne wypisanie wartości.
73          std::cout << "Wewnątrz funkcji po
            ↳wykonaniu przypisania wartość
            ↳zmiennej name to " << name << ",
            ↳a gName " << gName << ".\n";
74
75      } // Koniec funkcji nameChange().
```



Rysunek 5.13. *Wartość zmiennej gName (zdefiniowanej w zasięgu globalnym) może być zmieniana wewnątrz obu funkcji, podczas gdy wartość lokalnej zmiennej name — nie*

8. Zdefiniuj funkcję `promptAndWait()`.

```
void promptAndWait() {
    std::cout<< "Naciśnij Enter lub Return,
    ↪aby kontynuować.\n";
    std::cin.get();
}
```

Ponieważ program nie wczytuje żadnych danych z wejścia, nie używamy funkcji `ignore()` (podobnie jak w pierwszym programie w tym rozdziale).

9. Rozpocznij definicję funkcji `nameChange()`.

```
void nameChange() {
    std::string name = "Larry";
```

Funkcja definiuje własną zmienną o nazwie `name`. Pomimo że zmienna ta ma taki sam identyfikator oraz typ jak zmienna w funkcji `main()`, to są one zupełnie *odrębne i niezależne*.

10. Wypisz aktualną wartość zmiennych.

```
std::cout << "Wewnątrz funkcji początkowa
↪wartość zmiennej name to " << name << ",
↪a gName " << gName << ".\n";
```

11. Zmień wartość `gName` i ponownie wypisz wartości obu zmiennych.

```
gName = name;
std::cout << "Wewnątrz funkcji po wykonaniu
↪przypisania wartość zmiennej name to " <<
↪name << ", a gName " << gName << ".\n";
```

Instrukcja w pierwszym wierszu przypisuje wartość zmiennej `name` (wewnątrz funkcji jest nią *Larry*) do zmiennej globalnej `gName`. Następnie wypisywana jest wartość obu zmiennych.

12. Zakończ funkcję.

```
}
```

13. Zapisz plik jako *scope.cpp*, po czym skompiluj i uruchom program (rysunek 5.13).

Wskazówki

- Załóżmy, że zmienne w wywołaniu funkcji używane są w poniższy sposób:

```
int myVar = 20;
fN(myVar);
```

  W powyższym przykładzie zmienna tak naprawdę nie jest przekazywana do funkcji, ale jedynie jej wartość. Zmienna `myVar` z perspektywy funkcji `fN` nie istnieje.

- Przyjmijmy kolejny przykładowy fragment kodu:

```
void fN(int myVar);
int main() {
    int myVar = 20;
    fN(myVar);
}
void fN(int myVar) {
    // Zrób coś.
}
```

  Zmienna `myVar` zdefiniowana wewnątrz funkcji `main()` i użyta w wywołaniu `fN()` jest również odrębną zmienną wobec `myVar`, która jest argumentem funkcji `fN()`.

- Może się wydawać, że skoro zmienne globalne są dostępne w każdym miejscu programu, to najlepiej byłoby używać wyłącznie ich. To pozornie wygodne rozwiązanie z punktu widzenia poprawnych praktyk programistycznych jest jednak bardzo złą praktyką.

- Należy zwrócić szczególną uwagę na to, by nie nadać zmiennym lokalnym i globalnym tych samych nazw. W takim przypadku zmienna lokalna będzie miała pierwszeństwo w danym bloku kodu i zmienna globalna będzie niewidoczna.

- Jednym ze sposobów na ominięcie rygorów związanych z zasięgiem zmiennych jest przekazywanie wartości do funkcji przez referencję. Zagadnieniem tym zajmiemy się w rozdziale 6. („Złożone typy danych”).

Kwestie związane z zasięgiem i definicjami zmiennych

Poza globalnym i lokalnym zasięgiem zmiennych istotnym zagadnieniem jest również *wewnętrzne* i *zewnętrzne* łączenie symboli (ang. *internal linkage* i *external linkage*). Chociaż nie pracowaliśmy jeszcze z programami składającymi się z wielu plików źródłowych (gdzie kod jest rozbity na odrębne, oddzielnie kompilowane części), to warto nadmienić, że łączenie jest związane z zasięgiem zmiennych rozpatrywanym pomiędzy poszczególnymi plikami. Temat ten jest omawiany w rozdziale 12. („Przestrzenie nazw i modularyzacja kodu”).

Zmienne wewnątrz funkcji (nazywa się je zmiennymi lokalnymi lub automatycznymi) można także deklarować z atrybutem `static` (ang. *statyczny*):

```
void count() {
    static int counter = 1;
    counter++;
}
```

Zmienne statyczne wewnątrz funkcji tym różnią się od zmiennych niestatycznych, że zachowują swoją wartość przez cały czas życia programu (nawet w czasie wielokrotnego wywoływania tej samej funkcji). Przy pierwszym wywołaniu funkcji `count()` zmienna `counter` (ang. *licznik*) ma wartość `1` i zostaje inkrementowana do wartości `2`. Przy drugim wywołaniu funkcji zmienna `counter` ma wartość `2` i zostaje inkrementowana do wartości `3`.

Gdyby `counter` była zadeklarowana w funkcji bez atrybutu `static`, to za każdym wywołaniem funkcji zmienna byłaby od nowa definiowana, inicjalizowana liczbą `1`, a później inkrementowana do wartości `2`.