Marek Gągolewski
Instytut Badań Systemowych PAN
Wydział Matematyki i Nauk Informacyjnych Politechniki Warszawskiej

Algorytmy
i podstawy programowania

3. Instrukcja warunkowa i pętle. Struktury. Funkcje – podstawy

Materiały dydaktyczne dla studentów matematyki
na Wydziale Matematyki i Nauk Informacyjnych Politechniki Warszawskiej
Ostatnia aktualizacja: 1 października 2016 r.

Spis treści

3.1. Instrukcje sterujące

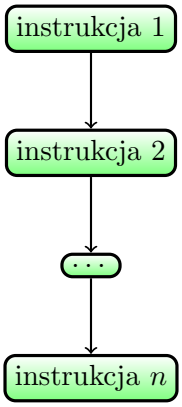
3.1.1. Bezpośrednie następstwo instrukcji

Znamy już następujące typy instrukcji:

– instrukcje deklarujące zmienne i stałe lokalne,
– instrukcję przypisania.

Wiemy też, w jaki sposób wypisywać wartości wyrażeń na ekran oraz jak pobierać wartości z klawiatury.

Zauważmy, iż do tej pory nasz kod w języku C++ nie miał żadnych rozgałęzień. Wszystkie instrukcje były wykonywane jedna po drugiej. Owo tzw. *bezpośrednie następstwo* instrukcji obrazuje schemat blokowy algorytmu (ang. *control flow diagram*, czyli schemat przepływu sterowania) na rys. 3.1.



Rys. 3.1. Schemat blokowy bezpośredniego następstwa instrukcji

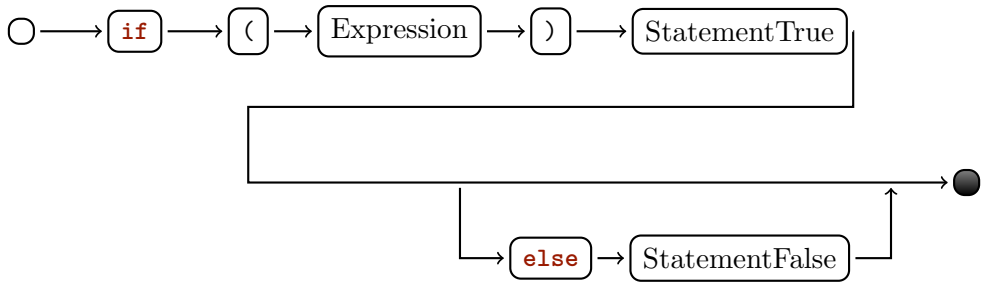
W kolejnych podrozdziałach poznamy konstrukcje, które pozwolą nam sterować przebiegiem programu. Dzięki nim będziemy mogli dostosowywać działanie programu do różnych szczególnych przypadków — w zależności od wartości przetwarzanych danych.

3.1.2. Instrukcja warunkowa if

Podczas zmagania się z różnymi problemami prawie zawsze spotykamy sytuację, gdy musimy dokonać jakiegoś wyboru. Na przykład, rozwiązując równanie kwadratowe inaczej postępujemy, gdy ma ono dwa pierwiastki rzeczywiste, a inaczej, gdy nie ma ich wcale. Policjant mierzący fotoradarem prędkość nadjeżdżającego samochodu inaczej postąpi, gdy stwierdzi, że dopuszczalna prędkość została przekroczona o 50 km/h (zwłaszcza w obszarze zabudowanym), niż gdyby się okazało, że kierowca jechał prawidłowo. Student przed sesją głowi się, czy przyłożyć się bardziej do programowania, algebry, do obu na raz (jedynie słuszna koncepcja) czy też dać sobie spokój i iść na imprezę itp.

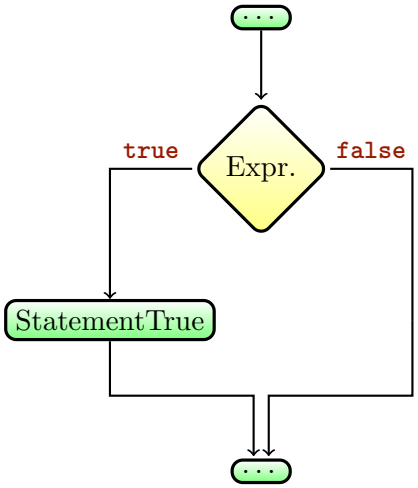
W języku C++ dokonywanie tego typu wyborów umożliwia *instrukcja warunkowa* `if` (od ang. *jeśli*). Wykonuje ona pewien fragment kodu *wtedy i tylko wtedy*, gdy pewien dany warunek logiczny jest spełniony. Może też, opcjonalnie, zadziałać w inny sposób w przeciwnym przypadku (`else`).

Jej składnię przedstawia poniższy diagram.



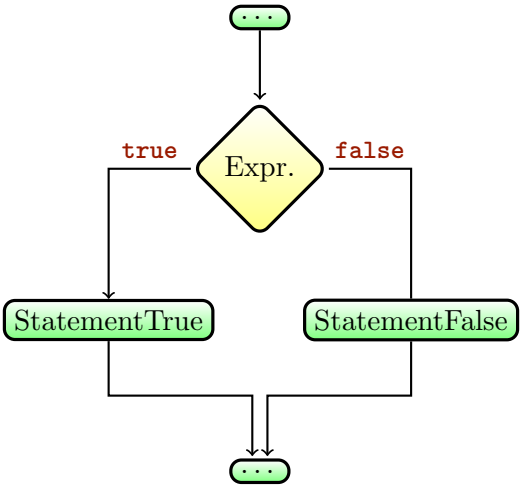
Expression jest wyrażeniem (koniecznie ujętym w nawiasy!) sprowadzalnym do typu `bool`, reprezentującym sprawdzany warunek logiczny. StatementTrue jest instrukcją wykonywaną wtedy i tylko wtedy, gdy wyrażenie Expression ma wartość `true`, Opcjonalne StatementFalse zaś — w przeciwnym przypadku.

Schemat blokowy instrukcji `if` bez części `else` przedstawia rys. 3.2.



Rys. 3.2. Schemat blokowy instrukcji warunkowej `if`

Jeśli jednak podinstrukcja `else` jest obecna, stosowany jest schemat zobrazowany na rys. 3.3.



Rys. 3.3. Schemat blokowy instrukcji warunkowej `if`...`else`

Rozważmy przykład, w którym wyznaczane jest minimum z dwóch liczb całkowitych.

```
int x, y;
cin >> x >> y;   // wprowadź x i y z klawiatury

if (x < y)       // tutaj nie ma średnika!
   cout << x;
else             // tutaj nie ma średnika!
   cout << y;
```

Zapamiętaj

Sam średnik (;) oznacza instrukcję pustą (która nie robi nic). Dlatego następujący fragment kodu:

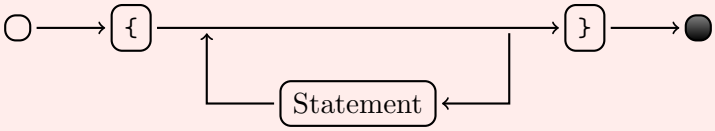
```
int x;
cin >> x;
if (x < 0);      // jeśli x < 0, nie rób nic specjalnego teraz
   cout << x;    // zawsze wypisuj x
```

wypisze wartość `x` na ekran *zawsze*.

Zapamiętaj

Jeżeli chcemy wykonać warunkowo więcej niż jedną instrukcję, tworzymy w tym celu tzw. blok instrukcji, czyli *instrukcję złożoną* (ang. *compound statement*), ograniczoną z obu stron nawiasami klamrowymi {...}.

Składnia instrukcji złożonej przedstawiona jest na poniższym diagramie.



Zauważmy, że po nawiasie zamykającym blok nie ma potrzeby stawiania średnika.

Informacja

Aby zwiększyć czytelność kodu, instrukcje wewnątrz bloku powinniśmy *wyróżnić wcięciem*. Jest to jedna z zasad dobrego pisania programów.

Instrukcje warunkowe można, rzecz jasna, zagnieżdżać. Oto przykład służący do znajdywania minimum z trzech liczb.

```
int x, y, z;
cin >> x >> y >> z;

if (x < y) { // klamerka może znaleźć się też w osobnym wierszu
   if (z < x)
      cout << z;
   else
      cout << x;
}
else {
   if (z < y)
      cout << z;
   else
      cout << y;
}
```

Informacja

Z zagnieżdżaniem instrukcji warunkowych trzeba jednak uważać. Słowo kluczowe `else` dotyczy najbliższej instrukcji `if`. Zatem poniższy kod zostanie wykonany przez komputer inaczej niż sugerują to wcięcia.

```
int x, y, z;
cin >> x >> y >> z;

if (x != 0)
   if (y > 0 && z > 0)
      cout << y*z/x;
else  // else dotyczy if (y > 0 && z > 0)
   cout << ":-(";
```

Aby dostosować ten kod do wyraźnej intencji programisty, należałoby objąć fragment

```
if (y > 0 && z > 0)cout << y*z/x;
```

nawiasami klamrowymi.

Warto pamiętać, że jako StatementFalse może się pojawić kolejna instrukcja `if... else`. Dzięki temu można obsłużyć w danym fragmencie programu więcej niż 2 rozłączne przypadki rozpatrywanego problemu na raz.

```
int x;
cout << "Określ swój nastrój w skali 1-3:";
cin >> x;

if (x == 1)
   cout << ":-(";
else if (x == 2)
   cout << ":-|";
else
   cout << ":-)";
```

Na marginesie, czasem przydać nam się może instrukcja `switch`. I tak:

```
int x = ...;
switch(x) { // tylko typ calkowitoliczbowy
    case 1:
        // ... gdy x == 1 ...
        break;
    case 2:
    case 3:
        // ... gdy x == 2 || x == 3 ...
        break;
    default:
        // ... w przeciwnych przypadkach ...
        break;
}
```

odpowiada:

```
int x = ...;
if (x == 1) {
        // ... gdy x == 1 ...
}
else if (x == 2 || x == 3) {
    // ... gdy x == 2 || x == 3 ...
}
else {
    // ... w przeciwnych przypadkach ...
}
```

Jak widzimy, `switch` właściwie nie wprowadza niczego nowego do arsenału naszych możliwości. W niektórych przypadkach jednak może prowadzić do bardziej czytelnego kodu.

3.1.3. Pętle

Oprócz instrukcji warunkowej `if`, rozgałęziającej przebieg sterowania „w dół", możemy skorzystać z tzw. *pętli* (ang. *loops* bądź *iterative statements*). Umożliwiają one wykonywanie tej samej instrukcji (albo, rzecz jasna, bloku instrukcji) wielokrotnie, być może na innych danych, *dopóki* pewien warunek logiczny jest spełniony.

Pomysł ten jest oczywiście bliski naszemu życiu. Pętle znajdują zastosowanie, jeśli zachodzi konieczność np. wykonania pewnych podobnych operacji na każdym z elementów danego zbioru. Np. życząc sobie zsumować wydatki poczynione na przyjemności podczas każdego dnia wakacji, rozpatrujemy kolejno sumę "odpływów" w dniu pierwszym, potem drugim, a potem trzecim, a potem itd., czyli tak naprawdę w dniu i-tym, gdzie $i = 1, 2, \ldots, n$.

Dalej, ileż to razy słyszeliśmy słowa mądrej mamy „dopóki (!nauczysz_się) zostajesz_w_domu;". To jest również przykład (niekoniecznie świadomie użytej) pętli.

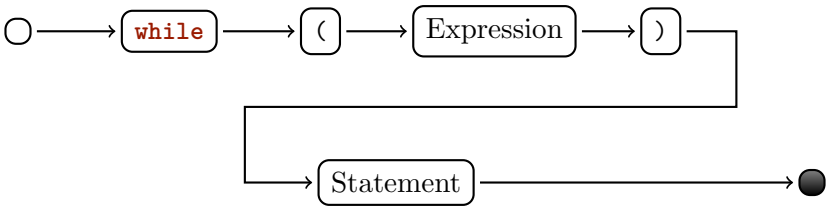
W języku C++ zdefiniowane zostały trzy instrukcje realizujące tego typu ideę.

- `while`,
- `for` oraz
- `do ... while`.

Przyjrzyjmy się im dokładniej.

3.1.3.1. Pętla while

Koncepcyjnie najprostszą konstrukcją realizującą pętlę jest instrukcja `while`. Jej składnię przedstawia poniższy diagram:

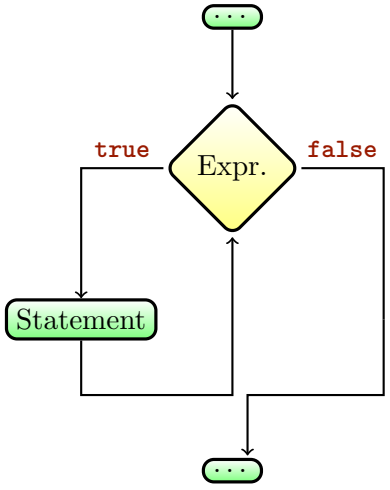


Expression jest wyrażeniem sprowadzalnym do typu `bool`. Najlepiej będzie, gdy istnieje możliwość jego zmiany na skutek wykonywania podanej w części Statement instrukcji. W przeciwnym wypadku stworzymy program, który nigdy się nie zatrzyma.

Schemat blokowy przedstawionej pętli zobrazowany jest na rys. 3.4.

Oto w jaki sposób możemy wypisać na ekranie kolejno liczby 1,2,...,100 i zsumować ich wartości (przypomnijmy sobie w tym miejscu problem młodego Gaussa z pierwszego wykładu).

```
int i = 1;
int suma = 0;
while (i <= 100) { // ten warunek nie będzie kiedyś spełniony...
    cout << i << endl;
    suma += i;
    i++;          /* ...gdyż jest zależny od wartości zmiennej i,
                     która się w tym miejscu zmienia */
}
cout << "Suma=" << suma << endl;
```



Rys. 3.4. Schemat blokowy pętli `while`

Zapamiętaj

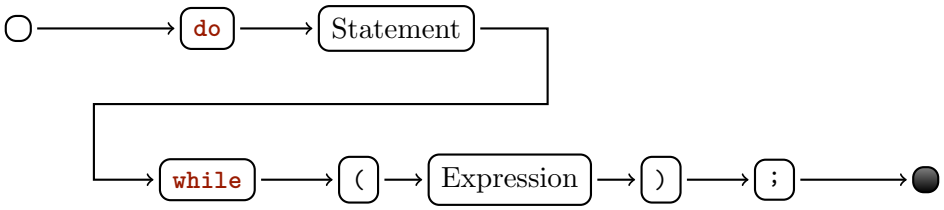
Powtórzmy, sam średnik (;) oznacza instrukcję pustą, dlatego pętla

```
while (true) ;
```

będzie w nieskończoność nie robić nic. Nie bierzmy z niej przykładu.

3.1.3.2. Pętla do...while

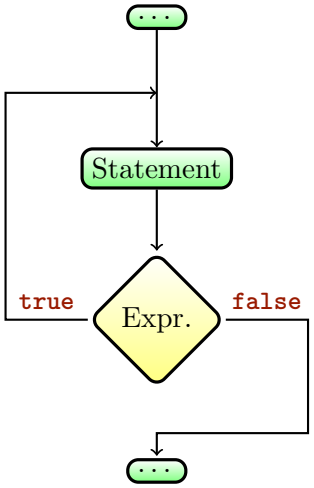
Inną odmianą pętli jest instrukcja `do ... while`, określona według składni:



Jak widać na schemacie blokowym na rys. 3.5, używamy jej, gdy chcemy zapewnić, by instrukcja Statement została wykonana *co najmniej* jeden raz.
Zapiszmy dla ćwiczenia powyższy przykład za pomocą pętli `do ... while`, urozmaicając go jednak nieco.

```
int i, suma = 0;
cin >> i;
do {
    suma += i;
    cout << i << endl;
    ++i;
}
while (i<=100);
```

Zauważmy, że tym razem wartość początkowa zmiennej `i` zostaje wprowadzona z klawiatury. Za pomocą wprowadzonej pętli wymuszamy, że wypisanie wartości na ekran i ustalenie odpowiedniej wartości zmiennej `suma` dokona się zawsze, także w przypadku, gdy `i > 100`.



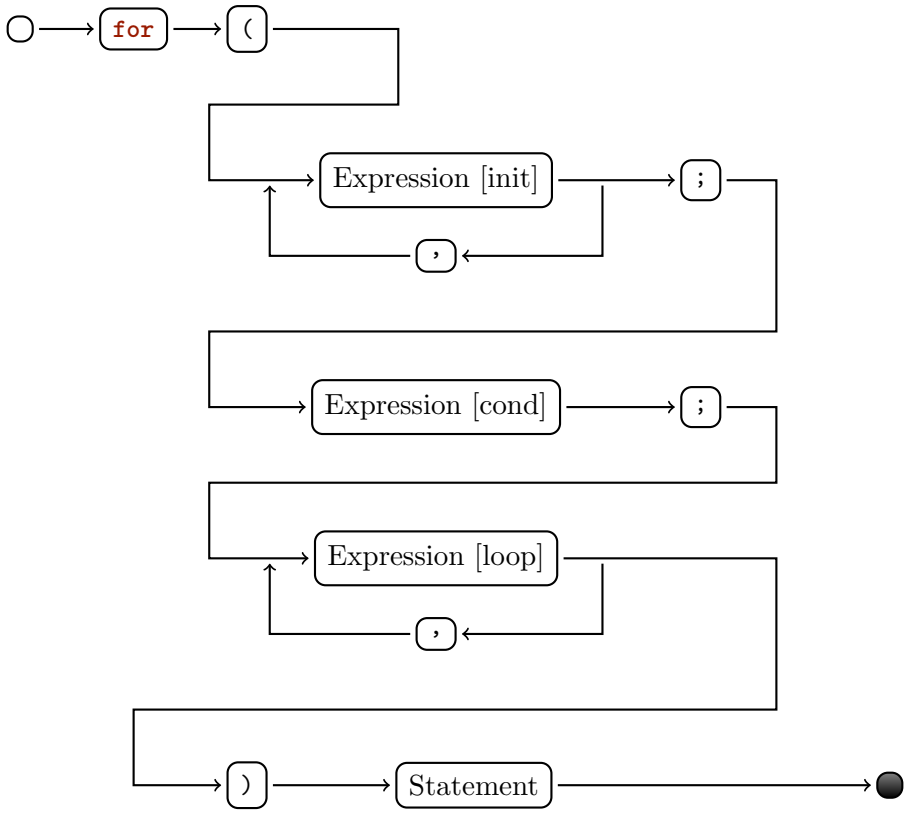
Rys. 3.5. Schemat blokowy pętli `do... while`

3.1.3.3. Pętla for

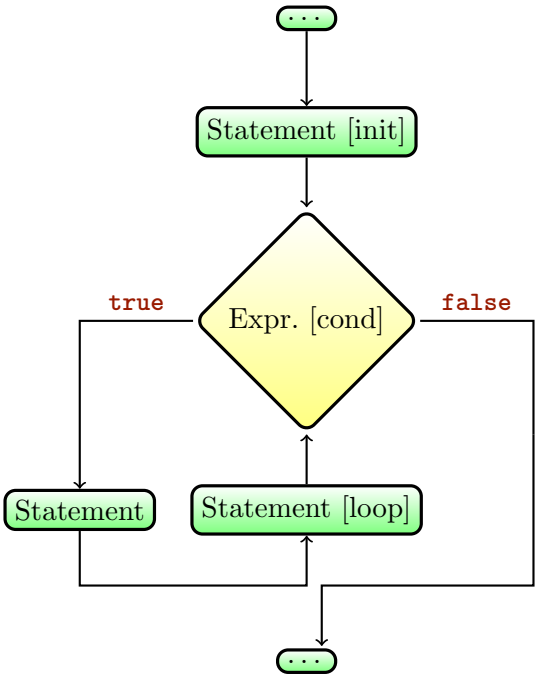
Dość często zachodzi potrzeba ustalenia przebiegu programu tak jak na schemacie blokowym przedstawionym na rys. 3.6.

Tutaj Statement [init] oznacza instrukcję inicjującą, wykonywaną tylko raz, na początku obliczeń (przygotowanie do wykonania pracy). Z kolei Statement [loop] jest wyrażeniem, które zostaje wykonane zawsze na końcu każdej iteracji i służy np. do aktualizacji licznika pętli (przygotowanie do kolejnej iteracji).

Tego typu schemat warto zakodować za pomocą pętli `for`, której (dość skomplikowaną na pierwszy rzut oka) składnię przedstawia poniższy diagram.



Expression [init] jest odpowiednikiem Statement [init] z rys. 3.6 bez końcowego średnika. Podobnie jest w przypadku Expression [loop]



Rys. 3.6. Schemat blokowy pętli `for`

Wróćmy do naszego wyjściowego przykładu. Można go także rozwiązać, korzystając z wprowadzonej właśnie pętli.

```
int suma = 0;
for (int i=1; i<=100; ++i) {
    cout << i << endl;
    suma += i;
}
cout << "Suma=" << suma << endl;
```

Informacja

Jeśli chcemy wykonać więcej niż jedną instrukcję inicjującą bądź aktualizującą, należy każdą z nich oddzielić przecinkiem (nie średnikiem, ani nie tworzyć dlań bloku).

Zatem powyższy przykład można zapisać i tak:

```
int suma = 0;
for (int i=1; i<=100; suma += i, ++i) // albo suma += (i++)
    cout << i << endl;
cout << "Suma=" << suma << endl;
```

3.1.3.4. break i continue

Niekiedy zachodzi konieczność zmiany domyślnego przebiegu wykonywania pętli.

Informacja

Instrukcja `break` służy do natychmiastowego wyjścia z pętli (niezależnie od wartości warunku testowego).

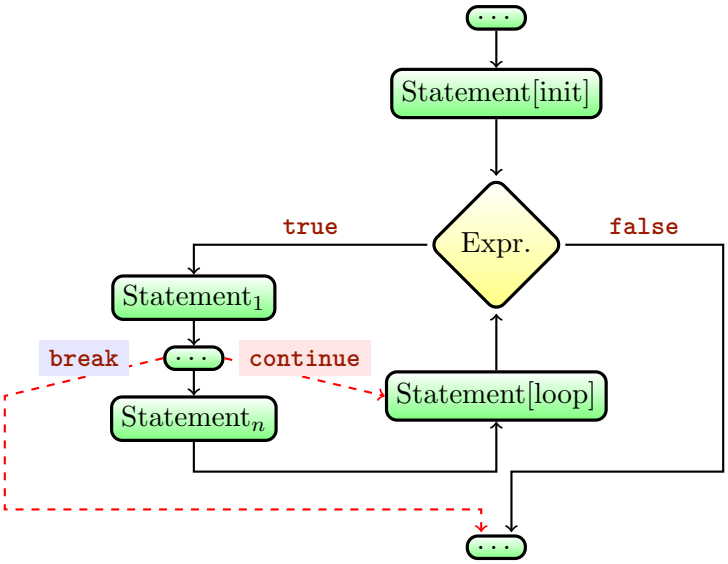
Z kolei instrukcja `continue` służy do przejścia do kolejnej iteracji pętli (ignorowane są instrukcje następujące po `continue`). W przypadku pętli `for` instrukcja aktualizująca jest jednak wykonywana.

Schemat blokowy z rys. 3.7 przedstawia zmianę przebiegu programu za pomocą omawianych instrukcji na przykładzie pętli `for`.

W przypadku zastosowania kilku pętli zagnieżdżonych, instrukcje te dotyczą tylko jednej (wewnętrznej) pętli.

Przykład (niezbyt pomysłowy): wypisywanie kolejnych liczb parzystych od 2 do 100:

```
for (int i=2; i<=100; ++i)
{
    if (i % 2 == 1)
        continue;
    cout << i;
}
```



Rys. 3.7. Instrukcje `break` i `continue` a pętla `for`

3.2. Struktury, czyli typy złożone

W poprzednim rozdziale poznaliśmy tak zwane skalarne typy podstawowe, w tym m.in. typ całkowity `int`, zmiennoprzecinkowy `double` oraz logiczny `bool`.

W języku C++ możemy tworzyć własne *typy złożone* będące reprezentacją *iloczynu kartezjańskiego* różnych innych zbiorów (typów). Są to tzw. *struktury*.

Struktury definiujemy w następujący sposób.

```
struct nazwaStruktury
{
    typ1 nazwaPola1;
    typ2 nazwaPola2;
    //....
    typN nazwaPolaN;
}; // średnik!
```

Zapamiętaj

Zauważmy, że po nawiasie klamrowym w definicji struktury występuje średnik. Jego pominięcie jest częstą przyczyną występowania różnych (dziwnych na pierwszy rzut oka) błędów kompilacji.

Zdefiniowanie struktury powoduje utworzenie nowego typu. *Zmienne* typu złożonego deklarujemy w standardowy sposób, czyli np.

```
nazwaStruktury identyfikatorZmiennej;
```

Do poszczególnych pól (składników) struktury możemy odwołać się za pomocą *kropki*, np. `identyfikatorZmiennej.nazwaPolaX`. Pola zmiennej typu złożonego traktujemy jak zwykłe zmienne odpowiednich typów. Intuicyjnie, zmienną typu złożonego można wyobrażać sobie jako swego rodzaju „paczkę”, która wewnątrz skrywa różne ciekawe „drobiazgi”.

Dla przykładu rozważmy definicję struktury reprezentującej punkt w $\mathbb{R}^2 = \mathbb{R} \times \mathbb{R}$. Łatwo się domyślić, że taka „paczka” powinna składać się z dwóch pól typu `double`.

```
struct Punkt
{
    double x;
    double y;
};
```

Oto przykładowe użycie tak zdefiniowanej struktury:

```
1  int main() {
2      Punkt p;          // tzn. niech p∈ℝ×ℝ
3      p.x = 0.0;
4      p.y = 0.5;
5      // wypisz na ekran:
6      cout << "(" << p.x << ", " << p.y << ")" << endl;
7
8      Punkt r;
9      cin >> r.x;
10     cin >> r.y;
11     cout << "(" << p.x+r.x << ", " << p.y+r.y << ")" << endl;
12
13     return 0;
14 }
```

Zapamiętaj

Zwróćmy uwagę, że instrukcja

```
cout << p;          // :-(
```

spowoduje wystąpienie błędu kompilacji. Komputer bowiem nie wie, jak miałby wykonać polecenie „wypisz coś złożonego” — musiałby zgadnąć, czy chodzi nam np. o wypisanie najpierw pola `x`, czy może pola `y`, czy chcemy je oddzielić spacją, a może znakiem nowej linii, a może... itp.

3.3. Funkcje — informacje podstawowe

3.3.1. Funkcje w matematyce

Niech X i Y będą dowolnymi niepustymi zbiorami. Powiemy, że f jest *funkcją*[1] odwzorowującą (przekształcającą) X w Y, ozn. $\mathsf{f} : X \to Y$, jeśli dla każdego $x \in X$ istnieje dokładnie jeden element $y \in Y$ taki, że $\mathsf{f}(x) = y$.

Zbiór X, złożony z elementów, dla których funkcja f została zdefiniowana, nazywamy *dziedziną* (bądź zbiorem argumentów). Zbiór Y z kolei, do którego należą wartości funkcji, nazywamy jej *przeciwdziedziną*.

Przyjrzyjmy się dwom przykładom.

[1] Por. H. Rasiowa, *Wstęp do matematyki współczesnej*, Warszawa, PWN 2003.

Przykład 1. Niech

$$\overbrace{\mathsf{kwadrat}:}^{\text{nazwa funkcji}} \quad \overbrace{\mathbb{R}}^{\text{dziedzina}} \quad \to \quad \overbrace{\mathbb{R}}^{\text{przeciwdziedzina}}$$

będzie funkcją taką, że

$$\mathsf{kwadrat}(x) := x \cdot x.$$

Powyższa funkcja oblicza po prostu kwadrat danej liczby rzeczywistej. Zauważmy, że definicja każdej funkcji składa się z dwóch części. Pierwsza, „niech...", to tzw. *deklarator* określający, że od tej pory identyfikator kwadrat oznacza funkcję o danej dziedzinie i danej przeciwdziedzinie. Druga, „taka, że", to tzw. *definicja właściwa*, która prezentuje abstrakcyjny *algorytm*, przepis, za pomocą którego dla konkretnego elementu dziedziny (powyżej ten element oznaczyliśmy symbolem x, choć równie dobrze mogłoby to być t, ζ, bądź $\widehat{\alpha}^*$) możemy uzyskać wartość wynikową, tj. $\mathsf{kwadrat}(x)$. ■

Przykład 2. Niech

$$\mathsf{sindod}: \quad \mathbb{R} \quad \to \quad \mathbb{R}$$

będzie funkcją taką, że

$$\mathsf{sindod}(u) := \begin{cases} \mathsf{sin}(u) & \text{dla } \mathsf{sin}(u) \geqslant 0, \\ 0 & \text{w p.p..} \end{cases}$$

Tym razem widzimy, że w definicji wprowadzonej funkcji pojawia się odwołanie do innego odwzorowania, tj. do dobrze znanej funkcji sin. Oczywiście w tym przypadku niejawnie odwołujemy się do wiedzy Czytelnika, że sin jest przekształceniem $\mathbb{R}$ w np. $\mathbb{R}$, danym pewnym wzorem. ■

W niniejszym rozdziale przyjrzymy się sposobom definicji funkcji w języku C++ oraz różnym przykładom ich użycia. Dowiemy się, czym różnią się funkcje w C++ od ich matematycznych odpowiedników. Poznamy także bogatą bibliotekę gotowych do użycia funkcji wbudowanych, które możemy w każdej chwili wykorzystać do własnych potrzeb.

3.3.2. Definiowanie funkcji w języku C++

Na etapie projektowania programu często można wyróżnić wiele *modułów* (części, podprogramów), z których każdy jest odpowiedzialny za pewną logicznie wyodrębnioną, niezależną czynność. Fragmenty mogą cechować się dowolną złożonością. Mogą też same korzystać z innych podmodułów.

Jednym z zadań programisty jest wtedy powiązanie tych fragmentów w spójną całość, m.in. poprzez zorganizowanie odpowiedniego przepływu sterowania i wymiany informacji (proces taki może przypominać budowanie domku z klocków różnych kształtów — w odróżnieniu od rzeźbienia go z jednego, dużego kawałka drewna).

Rozważmy dwa przykładowe zalążki programów, obrazujące pewne sytuacje z tzw. życia.

Przykład 3. Najpierw przyjrzyjmy się czynnościom potrzebnym do wyruszenia samochodem na wycieczkę (albo do rozpoczęcia egzaminu praktycznego na prawo jazdy).

```
1 int main() {
2     ustawFotel();
3     ustawLusterka();
4     zapnijPasy();
5     przekręćKluczyk();
```

```
6     sprawdźKontrolki();
7     uruchomRozrusznik();
8     return 0;
9  }
```

Programiście, który wyróżnił ciąg czynności potrzebnych do wykonania pewnego zadania pozostaje tylko szczegółowe określenie (implementacja), na czym one polegają. Zauważmy, że dzięki takiemu sformułowaniu rozwiązania możliwe jest łatwiejsze zapanowanie nad złożonością kodu. ■

Przykład 4. Kolejny przykład dotyczy organizacji nauki w semestrze pewnego pilnego studenta.

```
1  int main() {
2     // ...
3     do {
4        if (dzieńTygodnia == niedziela)
5           continue;
6
7        pouczSięTrochę();
8
9        if (zmęczony) {
10          if (godzinNaukiDziś > 5)
11             możeszWreszcieIśćNaRandkę();
12          else
13             odpocznijChwilę();
14       }
15    } while (!nauczony);
16    // ...
17 }
```

■

Najprostszym sposobem podziału programu w języku C++ na (pod)moduły jest użycie funkcji[2].

Zapamiętaj

Funkcja (zwana też czasem procedurą, metodą, podprogramem) to odpowiednio wydzielony fragment kodu programu wykonujący pewne instrukcje. Do funkcji można odwoływać się z innych miejsc programu.

Informacja

Funkcje służą do dzielenia kodu programu na mniejsze, łatwiejsze w tworzeniu, zarządzaniu i testowaniu fragmenty o ściśle określonym działaniu. Fragmenty te są względnie niezależne od innych części.

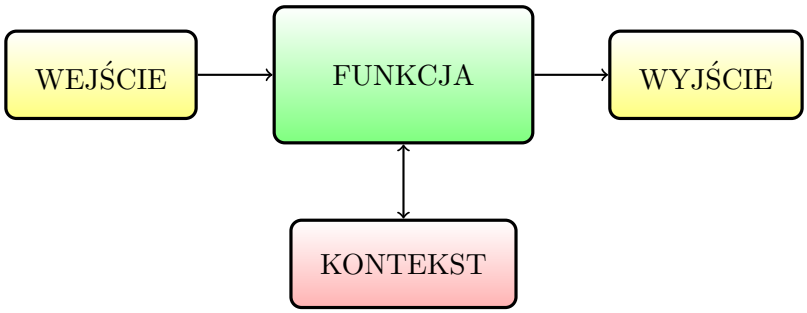
Dodatkowo, dzięki funkcjom uzyskujemy m.in. możliwość wykorzystania tego samego kodu wielokrotnie.

[2] W semestrze II poznamy jeszcze inny sposób, pozwalający na pisanie naprawdę dużych programów: tzw. programowanie obiektowe.

Projektując każdą funkcję należy wziąć pod uwagę, w jaki sposób ma ona wchodzić w interakcję z innymi funkcjami, to znaczy jakiego typu *dane wejściowe* powinna ona przyjmować i jakiego typu *dane wyjściowe* będą wynikiem jej działania.

Zapamiętaj

Bezpośrednim skutkiem działania funkcji jest uzyskanie jakiejś konkretnej wartości na wyjściu. *Skutkiem pośrednim* działania funkcji może być z kolei zmiana tzw. *kontekstu*, tj. stanu komputera (np. wypisanie czegoś na ekran, pobranie wartości z klawiatury itp.).



Rys. 3.8. Schemat przepływu danych w funkcjach

Powyższa idea została zobrazowana na rys. 3.8.

Przyjrzyjmy się najpierw najważniejszym różnicom między funkcjami w matematyce a funkcjami w języku C++.

1. *Funkcje w matematyce nie mają skutków pośrednich*. Jedynym efektem ich działania może być tylko przekształcenie konkretnego elementu dziedziny w ściśle określony element przeciwdziedziny. Funkcje w C++ z kolei mogą, niejako „przy okazji", wypisać coś na ekran, zapisać na dysk twardy plik pobrany z internetu, bądź też odegrać piosenkę w formacie MP3.

2. *Dziedzina i przeciwdziedzina funkcji matematycznej nie może być zbiorem pustym*. W języku C++ został określony specjalny typ `void` ($\emptyset$, od ang. *pusty*), który pozwala na określenie funkcji „robiącej coś z niczego" (korzystającej tylko z kontekstu, np. danych pobranych z klawiatury), „niby-nic z czegoś" (tylko np. zmieniającej kontekst), bądź nawet „niby-nic z niczego".

3. *Przeciwdziedziną funkcji w języku C++ może być tylko jeden typ*, choć może to być typ złożony (struktura)[3].

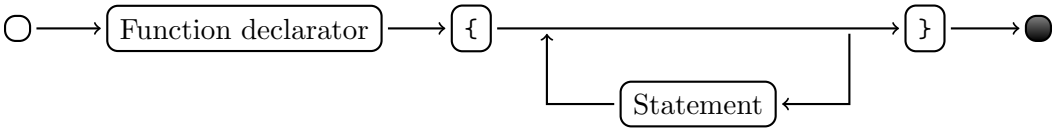
Ciekawostka

De facto więc, funkcje w języku programowania mogłyby być postrzegane jako funkcje matematyczne, jeśliby ich dziedziną było zawsze $X \times \mathbb{K}$, a przeciwdziedziną $Y \times \mathbb{K}$, gdzie $\mathbb{K}$ oznacza

[3] W pewnym sensie da się ominąć to ograniczenie korzystając z argumentów przekazanych przez referencję (o czym w innym rozdziale).

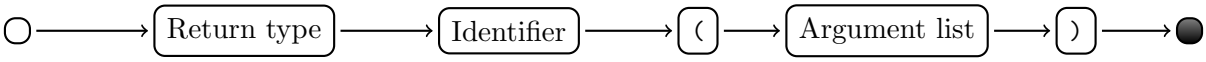
kontekst działania komputera (bardzo trudny w opisie, na niego bowiem składają się wszystkie urządzenia wejściowe i wyjściowe komputera, w tym urządzenia wideo, audio, sieciowe, pamięć masowa itp.).

Przejdźmy do opisu składni *definicji funkcji* w języku C++. Możemy ją przedstawić w następującej postaci.

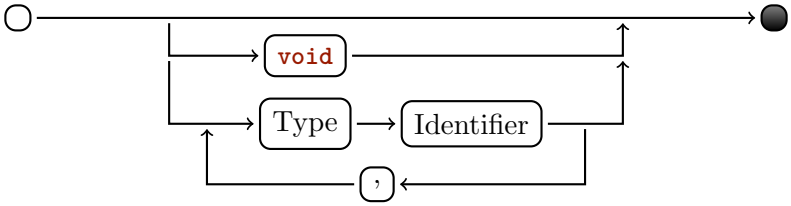


Blok instrukcji zawarty między nawiasami klamrowymi to tzw. *ciało funkcji* (definicja właściwa) — tutaj zawarta jest implementacja algorytmu realizowanego przez funkcję.

Z kolei Function declarator to *deklarator funkcji* postaci:



gdzie Identifier jest pewnym identyfikatorem oznaczającym nazwę definiowanej funkcji, Return type jest typem określającym przeciwdziedzinę (może to być `void`), a Argument list to lista argumentów określająca dziedzinę:



Zauważmy, że pusta lista argumentów jest tożsama z `void`. W ten sposób można określać funkcje pobierające dane tylko z tzw. kontekstu.

Informacja

Do zwracania wartości przez funkcję (należących do określonej przeciwdziedziny) służy instrukcja `return`.

Instrukcja `return` działa podobnie jak `break` w przypadku pętli, tj. natychmiast przerywa wykonywanie funkcji, w której aktualnie znajduje się sterowanie. Dzięki temu następuje od razu powrót do miejsca, w którym nastąpiło wywołanie danej funkcji.

W przypadku funkcji o przeciwdziedzinie `void`, instrukcja `return` (w takim przypadku nieprzyjmująca żadnych argumentów) może zostać pominięta. Po wykonaniu ostatniej instrukcji następuje automatyczny powrót do funkcji wywołującej.

Zapamiętaj

Funkcje są podstawowym „budulcem” programów pisanych w języku C++. Każda wykonywana instrukcja (np. pętla, instrukcja warunkowa, instrukcja przypisania) musi należeć do jakiejś funkcji.

Funkcji w języku C++ nie można zagnieżdżać, tzn. nie można tworzyć funkcji w funkcji.

Przykład 5. Najprostszy, pełny kod źródłowy działającego programu w języku C++ można zapisać w sposób następujący.

```
int main()
{
    return 0; // zakończenie programu (,,sukces'')
}
```

Widzimy zatem (nareszcie!), że `main()` jest funkcją, która nie przyjmuje żadnych parametrów wejściowych, a na wyjściu zwraca wartość całkowitą. Robi zatem „coś" z „niczego"! ■

Ciekawostka

Wartość zwracaną przez funkcję `main()` odczytuje system operacyjny. Dzięki temu wiadomo, czy program zakończył się prawidłowo (wartość równa 0), czy też jego działanie nie powiodło się (wartość różna od 0). Z reguły informacja ta jest ignorowana i na jej podstawie nie jest podejmowane żadne specjalne działanie.

Informacja

Każdy program w języku C++ musi zawierać definicję funkcji `main()`, inaczej proces kompilacji nie powiedzie się. `main()` (od ang. *główny*) stanowi punkt startowy każdego programu. Instrukcje zawarte w tej funkcji zostają wykonane jako pierwsze zaraz po załadowaniu programu do pamięci komputera.

3.3.3. Wywołanie funkcji

Rozpatrzmy jeszcze raz definicję funkcji kwadrat.

Przykład 1 (cd.). Niech

$$\overbrace{\text{kwadrat}:}^{\text{nazwa funkcji}}\ \overbrace{\mathbb{R}}^{\text{dziedzina}}\ \to\ \overbrace{\mathbb{R}}^{\text{przeciwdziedzina}}$$

będzie funkcją taką, że

$$\text{kwadrat}(x) := x \cdot x.$$

Przyjrzyjmy się implementacji rozpatrywanej funkcji i jej przykładowemu zastosowaniu.

```
#include <iostream>
using namespace std;

                          // ,,Niech...''
double kwadrat(double x)
{                         // ,,taka, że...''
    return x*x;
}


int main() // funkcja bezargumentowa
```

```
12 {
13     double y = 0.5;
14     cout << kwadrat(y)    << endl; // wywołanie funkcji
15     cout << kwadrat(2.0) << endl; // można i tak
16     return 0;
17 }
```

Zapamiętaj

Zauważmy, że definicja funkcji `kwadrat()` została zamieszczona *przed* definicją funkcji `main()`, która z `kwadrat()` korzysta.

W powyższej funkcji `main()` zapis `kwadrat(y)` oznacza *wywołanie* (nakaz ewaluacji) funkcji `kwadrat()` na argumencie równym wartości przechowywanej w zmiennej `y`.

W momencie wywołania działanie funkcji `main()` zostaje wstrzymane, a kontrolę nad działaniem programu przejmuje funkcja `kwadrat()`. Parametr `x` otrzymuje wartość `0.5` i od tego momentu można traktować go wewnątrz tej funkcji jak zwykłą zmienną. Po wywołaniu instrukcji `return` przebieg sterowania wraca do funkcji `main()`. ■

Zapamiętaj

Zapis `kwadrat(y)` jest traktowany jak wyrażenie (ang. *expression*) o typie takim, jak przeciwdziedzina funkcji (czyli tutaj: `double`).

Obliczenie wartości takiego wyrażenia następuje poprzez wykonanie kodu funkcji na danej wartości przekazanej na wejściu.

Wywołanie funkcji można więc wyobrażać sobie jak „zlecenie" wykonania pewnej czynności przekazane jakiemuś „podwykonawcy". „Masz tu coś. Chcę uzyskać z tego to i to. Nie interesuje mnie, jak to zrobisz (to już twoja sprawa), dla mnie się liczy tylko wynik".

Z tych powodów w powyższej funkcji `main()` moglibyśmy także napisać:

```
1 double y = 0.5;
2 y = kwadrat(y);
3 // bądź
4 double z = kwadrat(4.0)-3.0*kwadrat(-1.5);
5 // itp.
```

Co więcej, należy pamiętać, że instrukcja

```
cout << kwadrat(kwadrat(0.5));
```

zostanie wykonana w sposób podobny do następującego:

```
double zmPomocnicza1 = kwadrat(0.5);
double zmPomocnicza2 = kwadrat(zmPomocnicza1);
cout << zmPomocnicza2;
```

Wartości pośrednie są obliczanie i odkładane „na boku". Tym samym, podanie „zewnętrznej" funkcji `kwadrat()` argumentu „`kwadrat(0.5)`" jest tożsame z przekazaniem jej wartości `0.25`.

Jest to o tyle istotne, iż w przeciwnym przypadku „`kwadrat(0.5)`" musiałby być wyliczony dwukrotnie (mamy przecież `return x*x;` w definicji). Nic takiego na szczęście jednak się nie dzieje.

3.3.4. Zasięg zmiennych

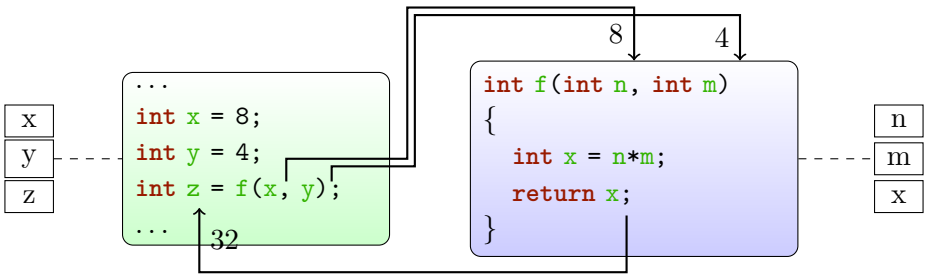
Przyjrzyjmy się *zasięgowi* definiowanych przez nas zmiennych. Zasięg określa, jak długo dany obiekt istnieje i w jaki sposób dane zmienne są widoczne z innych miejsc programu.

Zmienne, które definiujemy wewnątrz każdej funkcji to tzw. *zmienne lokalne*. *Tworzone* są one, gdy następuje wywołanie funkcji, a *usuwane*, gdy następuje jej opuszczenie.

Zmienne lokalne nie są współdzielone między funkcjami. I tak, zmienna `x` w funkcji `f()` to zmienna inna niż `x` w funkcji `g()`. Zatem jedynym sposobem wymiany danych między funkcjami jest zastosowanie parametrów wejściowych i wartości zwracanych.

Parametry funkcji spełniają *rolę zmiennych lokalnych*, których wartości są przypisywane automatycznie przy wywołaniu funkcji.

Przyjrzyjmy się rys. 3.9. Jak powiedzieliśmy, `x` po lewej i `x` po prawej to dwie różne zmienne. Zmienne lokalne `n`,`m`,`x` są tworzone na użytek wewnętrzny aktualnego wywołania funkcji `f()`. Mają one „pomóc” funkcji w spełnieniu swego zadania, jakim jest uzyskanie na wyjściu pewnej wartości, która jest konieczna do działania bloku kodu po lewej stronie. Zmienne te zostaną skasowane zaraz po tym, gdy funkcja zakończy swoje działanie (nie są one już do niczego potrzebne). W tym sensie można traktować taką funkcję jak *czarną skrzynkę*, gdyż to, jakie procesy wewnątrz niej zachodzą, nie ma bezpośredniego wpływu na obiekt, który posiłkuje się `f()` do osiągnięcia swoich celów.



Rys. 3.9. Zasięg zmiennych

Spójrzmy na blok kodu po lewej stronie. Zmienne `x` i `y` są przekazywane do `f()` *przez wartość*. Znaczy to, że wartości parametrów są obliczane przed wywołaniem funkcji (w tym przypadku pobierane są po prostu wartości przechowywane w tych zmiennych), a wyniki tych operacji są *kopiowane* do argumentów wejściowych. Widoczne są one w `f()` jako zmienne lokalne, odpowiednio, `n` i `m`.

Z wartością zwracaną za pomocą instrukcji `return` (prawa strona rysunku) kompilator postępuje w sposób analogiczny, tzn. nie przekazuje obiektu `x` jako takiego, lecz wartość, którą on przechowuje. Jeśli w tym miejscu stałoby np. złożone wyrażenie arytmetyczne albo stała, reguła ta byłaby oczywiście zachowana.

Zapamiętaj

Powtórzmy, zmienne przekazywane *przez wartość* są *kopiowane*.

Co więcej, zmienne lokalne nie są przypisane na stałe samym funkcjom, tylko ich wywołaniom. Obiekty utworzone dla jednego wywołania funkcji są *niezależne* od zmiennych dla innych wywołań. Jest to bardzo istotne w przypadku techniki zwanej rekurencją, która polega na tym, że funkcja wywołuje samą siebie (zob. kolejne rozdziały). Innymi słowy, wywołując tę samą funkcję `f()` raz po raz, za każdym razem tworzony jest nowy, niezależny zestaw zmiennych lokalnych.

Ciekawostka

W języku C++ dostępne są także zmienne globalne. Jednakże stosowanie ich nie jest zalecane. Nie będziemy ich zatem omawiać.

3.3.5. Przekazywanie parametrów przez referencję

Wiemy już, że standardowo parametry wejściowe funkcji zachowują się jak zmienne lokalne — ich zmiana nie jest odzwierciedlona „na zewnątrz".

Przykład 6. Rozważmy następujący przykład — zamianę (ang. *swap*) dwóch zmiennych.

```
void zamien(int x, int y) {
    int t = x;
    x = y;
    y = t;
}

int main() {
    int n = 1, m = 2;
    zamien(n, m); // przekazanie parametrów przez wartość
                  // (skopiowanie wartości)
    cout << n << ", " << m << endl;
    return 0;
}
```

Wynikiem działania tego fragmentu programu jest, rzecz jasna, napis `1, 2` na ekranie. Funkcja `zamien()` z punktu widzenia `main()` nie robi zupełnie nic. ■

W pewnych szczególnych przypadkach uzasadnione jest zatem przekazywanie parametrów w inny sposób — *przez referencję* (odniesienie). Dokonuje się tego poprzez dodanie znaku `&` bezpośrednio po nazwie typu zmiennej na liście parametrów funkcji.

Przekazanie parametrów przez referencję umożliwia nadanie bezpośredniego dostępu (również do zapisu) do zmiennych lokalnych funkcji wywołującej (być może pod inną nazwą). Obiekty te *nie są kopiowane*; udostępniane są takie, jakie są.

Co ważne, w taki sposób można tylko przekazać zmienną! Próba przekazania wartości zwracanej przez jakąś inną funkcję, stałej albo złożonego wyrażenia arytmetycznego zakończy się niepowodzeniem.

Przykład 6 (cd.). Tym samym dopiero teraz możemy przedstawić prawidłową wersję funkcji `zamien()`.

```
void zamien(int& x, int& y) {
    int t = x;
    x = y;
    y = t;
}

int main() {
    int n = 1, m = 2;
    zamien(n, m); // przekazanie parametrów przez referencję
    cout << n << ", " << m << endl;
    return 0;
}
```

Zmienna `n` widoczna jest tutaj pod nazwą `x`. Jest to jednak de facto ta sama zmienna — są to dwa różne odniesienia do tej samej, współdzielonej komórki pamięci.

Uzyskujemy wreszcie oczekiwany wynik: `2, 1`. ■

Istnieje jeszcze jedno ważne zastosowanie zmiennych przekazywanych przez referencję. Może ono służyć do obejścia ograniczenia związanego z tym, że funkcja za pomocą instrukcji `return` może zwrócić co najwyżej wartość jednego, ustalonego typu.

Przykład 7. Funkcja zwracająca część całkowitą ilorazu *oraz* resztę z dzielenia dwóch liczb.

```
void iloreszta(int x, int y, int& iloraz, int& reszta) {
    iloraz = x / y;
    reszta = x % y;
}

int main() {
    int n = 7, m = 2;   // wejście
    int i, r;           /* zmienne, które wykorzystamy
                           do przekazania wyniku */
    iloreszta(n, m, i, r);
    cout << n << "=" << i << "*" << m << "+" << r;
    return 0;
}
```

Wynikiem tego programu będzie więc `7=3*2+1`. ■

3.3.6. Argumenty domyślne funkcji

Argumenty domyślne to argumenty, których pominięcie przy wywołaniu funkcji powoduje, że zostaje im przypisana pewna z góry ustalona wartość.

Parametry z wartościami domyślnymi mogą być tylko przekazywane przez wartość. Ponadto, mogą się one pojawić jedynie na końcu listy parametrów (choć może być ich wiele).

Oto kilka przykładów prawidłowych deklaracji funkcji:

- `void f(int x=3);`
- `void f(int x=3, int y=2, int z=5);`
- `void f(int x, int y=3, int z=2);`
- `void f(int x, int y, int z=2);`
- `void f(int& x, int y=2);`

A oto nieprawidłowe deklaracje funkcji:

- `void f(int x, int y=3, int z);`
- `void f(int x=3, int y);`
- `void f(int x, int& y=2);`

Przykład 8. Dla ilustracji rozważmy funkcję wyznaczającą pierwiastek liczby rzeczywistej, domyślnie o podstawie 2.

```
#include <cmath>

double pierwiastek(double x, double p=2)
{ // pierwiastek, domyślnie kwadratowy
    assert(p >= 1);
    return pow(x, 1.0/p);
}
```

```
8
9  int main(void)
10 {
11     cout << pierwiastek(10) << endl;       // 3.162278
12     cout << pierwiastek(10, 2.0) << endl; // 3.162278
13     cout << pierwiastek(10, 3.0) << endl; // 2.154435
14     return 0;
15 }
```

■

3.3.7. Przeciążanie funkcji

W języku C++ można nadawać te same nazwy (identyfikatory) wielu funkcjom pod warunkiem, że różnią się one co najmniej typem lub liczbą parametrów. Jest to tzw. *przeciążanie* funkcji (ang. *function overloading*). Funkcje takie mogą mieć różne przeciwdziedziny — nie jest to jednak warunek dostateczny pozwalający na rozróżnienie funkcji przeciążonych.

Przeciążanie ma sens, gdy funkcje wykonują podobne (w sensie funkcjonalności) czynności, jednakże na danych różnego typu.

Przykład poprawnych deklaracji:

```
int    modul(int x);
double modul(double x);
```

Przykłady niepoprawnych deklaracji:

```
void f();
int  f(int x, int y=2);
int  f(int x); // Błąd! – szczególny przypadek powyższego
```

oraz:

```
bool g(int x, int y);
char g(int x, int y);   // Błąd! – nie różnią się argumentami
```

3.4. Przegląd funkcji z biblioteki języka C

Elementem standardu języka jest też wiele przydatnych (gotowych do natychmiastowego użycia) funkcji zawartych w tzw. bibliotece standardowej. Poniżej omówimy te, które interesować nas będą najbardziej.

3.4.1. Funkcje matematyczne

Biblioteka `<cmath>` udostępnia wybrane funkcje matematyczne[4]. Ich przegląd zamieszczamy w tab. 3.1, 3.2 i 3.3.

Dla przypomnienia, funkcja „sufit” określona jest jako

$$\lceil x \rceil = \min\{i \in \mathbb{Z} : i \geqslant x\},$$

a funkcja „podłoga” zaś jako

$$\lfloor x \rfloor = \max\{i \in \mathbb{Z} : i \leqslant x\}.$$

[4]Pełna dokumentacja biblioteki `<cmath>` w języku angielskim dostępna jest do pobrania ze strony `http://www.cplusplus.com/reference/clibrary/cmath/`.

Tab. 3.1. Funkcje trygonometryczne dostępne w bibliotece <cmath>

Deklaracja	Znaczenie
`double cos(double);`	cosinus (argument w rad.)
`double sin(double);`	sinus (argument w rad.)
`double tan(double);`	tangens (argument w rad.)
`double acos(double);`	arcus cosinus (wynik w rad.)
`double asin(double);`	arcus sinus (wynik w rad.)
`double atan(double);`	arcus tangens (wynik w rad.)
`double atan2(double y, double x);`	arcus tangens `y/x` (wynik w rad.)

Tab. 3.2. Funkcja wykładnicza, logarytm, potęgowanie w bibliotece <cmath>

Deklaracja	Znaczenie
`double exp(double);`	funkcja wykładnicza
`double log(double);`	logarytm naturalny
`double log10(double);`	logarytm dziesiętny
`double sqrt(double);`	pierwiastek kwadratowy
`double pow(double x, double y);`	x^y

Przykład 2 (cd.). Przypomnijmy definicję funkcji sindod. Niech

$$\mathsf{sindod}: \quad \mathbb{R} \quad \to \quad \mathbb{R}$$

będzie funkcją taką, że

$$\mathsf{sindod}(u) := \begin{cases} \sin(u) & \text{dla } \sin(u) \geqslant 0, \\ 0 & \text{w p.p..} \end{cases}$$

Oto przykładowy program wyznaczający wartość tej funkcji w różnych punktach.

```
#include <iostream>
#include <cmath>
using namespace std;

double sindod(double);

int main()
{
    cout << sindod(-3.14159*0.5) << endl;
    cout << sindod(+3.14159*0.5) << endl;
    return 0;
}


double sindod(double u)
{
    double s = sin(u); // funkcja z biblioteki <cmath>
    if (s >= 0) return s;
    else        return 0.0;
}
```

■

Tab. 3.3. Funkcje dodatkowe w bibliotece <cmath>

Deklaracja	Znaczenie
`double fabs(double);`	wartość bezwzględna
`double ceil(double);`	„sufit”
`double floor(double);`	„podłoga”

3.4.2. Liczby pseudolosowe

W bibliotece <cstdlib> znajduje się funkcja służąca do generowania liczb pseudolosowych.

Generator należy zainicjować przed użyciem funkcją `void srand(int z)`, gdzie `z>1` to tzw. *ziarno*. Jedno ziarno generuje zawsze ten sam ciąg liczb. Można także użyć aktualnego czasu systemowego do zainicjowania generatora. Dzięki temu podczas każdego kolejnego uruchomienia programu otrzymamy inny ciąg.

```
#include <cstdlib>
#include <ctime> // tu znajduje się funkcja time()

//...
srand(time(0)); // za każdym razem inne liczby
//...
```

Funkcja `int rand();` generuje (za pomocą czysto algebraicznych metod) pseudolosowe liczby całkowite z rozkładu dyskretnego jednostajnego określonego na zbiorze

$$\{0, 1, \ldots, \texttt{RAND\_MAX} - 1\}.$$

Jednostajność oznacza, że prawdopodobieństwo „wylosowania” każdej z liczb jest takie samo. `RAND_MAX` jest stałą zdefiniowaną w bibliotece <cstdlib>.

Jeśli chcemy uzyskać liczbę np. ze zbioru $\{1, 2, 3\}$, możemy napisać[5]:

```
cout << (rand() % 3) + 1;
```

Aby z kolei uzyskać liczbę rzeczywistą z przedziału $[0, 1)$, piszemy

```
cout << ((double)rand() / (double)RAND_MAX);
```

Przykład 9. Korzystając z własnoręcznie napisanej funkcji generującej liczbę rzeczywistą z przedziału $[0, 1)$ łatwo napisać funkcję „losującą” liczbę ze zbioru $\{a, a+1, \ldots, b\}$, gdzie $a, b \in \mathbb{Z}$.

Oto przykładowy program.

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <cmath>
using namespace std;

double los01()
{
    return ((double)rand() / (double)RAND_MAX);
}

```

[5] Podana metoda nie cechuje się zbyt dobrymi własnościami statystycznymi. Szczegóły poznamy jednak dopiero na laboratoriach ze statystyki matematycznej w semestrze VI.

```
12 int losAB(int a, int b)
13 {
14     double ab = los01()*(b-a+1)+a; // liczba rzeczywista
15                                    // z przedziału [a,b+1)
16     return (int)(floor(ab));       // ,,podłoga'' z ab
17 }
18
19 int main()
20 {
21     srand(time(0)); // zainicjuj generator
22     for (int i=0; i<5; i++) // 5 razy...
23        cout << losAB(1,10) << " ";
24     cout << endl;
25     return 0;
26 }
```

Przykładowy wynik na ekranie:
`4 3 8 2 1`
a po chwili:
`4 6 10 4 2`

■

Informacja

W bibliotece <cstdlib> znajduje się także funkcja `void exit (int status);`, której wywołanie powoduje natychmiastowe zakończenie programu. Funkcja ta działa podobnie jak wywołanie instrukcji `return` w `main()`, jednakże można z niej korzystać (z powodzeniem) w dowolnym miejscu programu.

3.4.3. Asercje

Biblioteka <cassert> udostępnia funkcję o następującej deklaracji:

```
void assert(bool);
```

Funkcja `assert()` umożliwia sprawdzenie dowolnego warunku logicznego. Jeśli nie jest on spełniony, nastąpi zakończenie programu. W przeciwnym wypadku nic się nie stanie.

Taka funkcja może być szczególnie przydatna przy testowaniu programu. Zabezpiecza ona m.in. przed danymi, które teoretycznie nie powinny się w danym miejscu pojawić.

Przykład 10. Dla przykładu rozpatrzmy „bezpieczną” funkcję wyznaczającą pierwiastek z nieujemnej liczby rzeczywistej.

```
1  #include <cassert>
2  #include <cmath>
3
4  double pierwiastek(double x)
5  {
6      // Dane: x ⩾ 0
7      // Wynik: √x
8
9      assert(x>=0); // jeśli nie, to błąd – zakończenie programu
10     return sqrt(x);
11 }
```

■

Ciekawostka

Sprawdzanie warunków przez wszystkie funkcje `assert()` można wyłączyć globalnie za pomocą dyrektywy

```
#define NDEBUG
```

umieszczonej na początku pliku źródłowego bądź nagłówkowego.

3.5. Ćwiczenia

Zadanie 3.1. Wyraź następujące instrukcje dane w pseudokodzie za pomocą instrukcji języka C++ korzystających z pętli `for`.

1. **dla** $i = 0, 1, \ldots, n-1$ wypisz i (dla pewnego $n \in \mathbb{N}$),
2. **dla** $i = n, n-1, \ldots, 0$ wypisz i (dla pewnego $n \in \mathbb{N}$),
3. **dla** $j = 1, 3, \ldots, 2k-1$ wypisz j (dla pewnego $k \in \mathbb{N}$),
4. **dla** $i = 1, 2, 4, 7, \ldots, n$ wypisz i (dla pewnego $n \in \mathbb{N}$),
5. **dla** $j = 1, 2, 4, 8, 16, \ldots n$ wypisz j (dla pewnego $n \in \mathbb{N}$),
6. **dla** $j = 1, 2, 4, 8, 16, \ldots, 2^k$ wypisz j (dla pewnego $k \in \mathbb{N}$),
7. **dla** $x = a, a+\delta, a+2\delta, \ldots, b$ wypisz x (dla pewnych $a, b, \delta \in \mathbb{R}, a < b, \delta > 0$).

Zadanie 3.2. Napisz kod w języku C++, który spośród liczb $1, 2, \ldots, 100$ wypisze:

1. wszystkie podzielne przez 7, tzn. $7, 14, 21, \ldots$,
2. wszystkie podzielne przez 2 lecz niepodzielne przez 5, tzn. $2, 4, 6, 8, 12, \ldots$,
3. co drugą podzielną przez 5 lub podzielną przez 7, tzn. $5, 10, 15, 21, 28, \ldots$

Zadanie 3.3. Napisz funkcję `max()`, która zwraca maksimum danych $a, b, c \in \mathbb{N}$.

Zadanie 3.4. Napisz funkcję `med()`, która znajduje medianę (wartość środkową) trzech liczb rzeczywistych, np. $\texttt{med}(4, 2, 7) = 4$ i $\texttt{med}(1, 2, 3) = 2$.

Zadanie 3.5. [MD] Napisz funkcję wyznaczającą zadaną (całkowitą) potęgę danej liczby rzeczywistej. Przykład wywołania: `potega(0.5,2)` zwróci 0.25

Zadanie 3.6. Napisz funkcję, która dla danego $n \in \mathbb{N}$ oblicza $\max\{k \in \mathbb{N}_0 : 2^k \leqslant n\}$.

Zadanie 3.7. Napisz funkcję, która zwraca przybliżenie wartości liczby π za pomocą wzoru $\pi \simeq 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots\right)$ na podstawie n (parametr funkcji) pierwszych elementów tego szeregu liczbowego.

Zadanie 3.8. Napisz funkcję, która zwraca przybliżenie wartości liczby e za pomocą wzoru $e \simeq 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$, gdzie $k! = 1 \times 2 \times \cdots \times k$. Sumuj kolejne wyrazy tego szeregu, dopóki różnica pomiędzy kolejnymi składnikami będzie mniejsza niż `eps` (parametr funkcji, wartość domyślna: 10^{-9}).

Zadanie 3.9. Napisz funkcje, które posłużą do wyznaczenia wartości następujących wyrażeń.

1. 2^n dla danego $n \in \mathbb{N}$,
2. $\sum_{i=1}^{n} i$ dla danego $n \in \mathbb{N}$,
3. $\sum_{i=1}^{n} \frac{1}{i!}$ dla danego $n \in \mathbb{N}$,
4. $\prod_{i=1}^{n} \frac{i}{i+1}$ dla danego $n \in \mathbb{N}$,
5. $e^x \simeq \sum_{n=0}^{m} \frac{x^n}{n!}$ dla danego $x \in \mathbb{R}$ i $m \in \mathbb{N}$,
6. $\ln(1+x) \simeq \sum_{n=1}^{m} \frac{(-1)^{n+1}}{n} x^n$ dla danego $x \in [-1, 1]$ i $m \in \mathbb{N}$,
7. $\sin x \simeq \sum_{n=0}^{m} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$ dla danego $x \in \mathbb{R}$ i $m \in \mathbb{N}$,
8. $\cos x \simeq \sum_{n=0}^{m} \frac{(-1)^n}{(2n)!} x^{2n}$ dla danego $x \in \mathbb{R}$ i $m \in \mathbb{N}$,
9. $\arcsin x \simeq \sum_{n=0}^{m} \frac{(2n)!}{4^n (n!)^2 (2n+1)} x^{2n+1}$ dla danego $x \in (-1, 1)$ i $m \in \mathbb{N}$.

Zadanie 3.10. Napisz funkcję `odle()`, która przyjmuje jako parametr współrzędne rzeczywiste dwóch punktów w $\mathbb{R}^2$ (struktura `Punkt` składająca się z dwóch wartości typu `double`) $\mathbf{p}, \mathbf{r}$ i zwraca ich odległość euklidesową równą $\sqrt{(p_x - r_x)^2 + (p_y - r_y)^2}$.

Wskazówka. Skorzystaj z funkcji `sqrt()` z biblioteki `<cmath>`.

Zadanie 3.11. Napisz funkcję `odlsup()`, która przyjmuje jako parametr współrzędne rzeczywiste dwóch punktów w $\mathbb{R}^2$ $\mathbf{p}, \mathbf{r}$ i zwraca ich odległość w metryce supremum: $\max\{|p_x - r_x|, |p_y - r_y|\}$.

Zadanie 3.12. Dane trzy punkty w $\mathbb{R}^2$: $\mathbf{a} = (a_x, a_y)$, $\mathbf{b} = (b_x, b_y)$, $\mathbf{c} = (c_x, c_y)$. Napisz funkcję, która wyznaczy kwadrat promienia okręgu przechodzącego przez $\mathbf{a}, \mathbf{b}$ i $\mathbf{c}$ (3 parametry będące strukturami o nazwie `Punkt`), określony wzorem

$$r^2 = \frac{|\mathbf{a}-\mathbf{c}|^2 \, |\mathbf{b}-\mathbf{c}|^2 \, |\mathbf{a}-\mathbf{b}|^2}{4 \, |(\mathbf{a}-\mathbf{c}) \times (\mathbf{b}-\mathbf{c})|^2},$$

gdzie $|\mathbf{a}-\mathbf{b}| = \sqrt{(a_x-b_x)^2 + (a_y-b_y)^2}$ oraz $\mathbf{a} \times \mathbf{b} = a_x b_y - a_y b_x$.

Zadanie 3.13. Napisz funkcję, która dla danych $a, b, c, d, e, f \in \mathbb{R}$ rozwiąże układ dwóch równań liniowych względem niewiadomych $x, y \in \mathbb{R}$:

$$\begin{cases} ax + by &= c, \\ dx + ey &= f. \end{cases}$$

Wynik zwróć w postaci struktury o dwóch polach typu `double`. Jeśli układ jest sprzeczny bądź nieoznaczony, ustaw wartość pól na `NaN` (funkcja `nan()` w bibliotece `<cmath>`).

⋆ **Zadanie 3.14.** Dane są liczby rzeczywiste $x_1, \ldots, x_4, y_1, \ldots, y_4 \in \mathbb{R}$ (dwie czteroelementowe struktury). Napisz funkcję, która sprawdzi, korzystając z jak najmniejszej liczby warunków logicznych, czy prostokąty $[x_1, x_2] \times [y_1, y_2]$ oraz $[x_3, x_4] \times [y_3, y_4]$ mają część wspólną (tj. przecinają się).

Zadanie 3.15. Napisz funkcję implementującą algorytm Euklidesa do wyznaczania największego wspólnego dzielnika dla danych dwóch liczb całkowitych nieujemnych a, b. Poprawnie identyfikuj wszystkie możliwe przypadki danych wejściowych, w tym m.in. $a < b$, $b < a$, $a < 0$, $b < 0$.

Zadanie 3.16. Napisz funkcję, która sprawdza, czy dana liczba naturalna jest liczbą pierwszą (`true`), czy też liczbą złożoną (`false`).

Zadanie 3.17. [MD] Napisz funkcję wyznaczającą liczbę dzielników zadanej liczby naturalnej. Przykład wywołania: `ile_dzielnikow(10)` zwróci wartość 4.

Zadanie 3.18. [MD] Napisz funkcję, która wyznaczy liczbę liczb pierwszych w zadanym zbiorze $\{a, a+1, \ldots, b\}$, gdzie $a < b$ — parametry funkcji. Użyj funkcji napisanej w zad. 3.17. Przykład wywołania: `ile_pierwszych(1,5)` zwróci wartość 3.

Zadanie 3.19. Korzystając ze wzoru na przybliżoną wartość funkcji sin podanego w zad. 3.9, utwórz program, który wydrukuje tablice przybliżonych wartości $\sin x$ dla $x = \frac{k}{n}\pi$, $k = 0, 1, \ldots, n$ i pewnego n, np. $n = 10$. Wynik niech będzie postaci podobnej do poniższej.

```
x         sin(x)
0.0000000  0.0000000
0.3141593  0.3090170
...
3.1415927  0.0000000
```

Zadanie 3.20. [MŚN] Napisz funkcję, która przyjmuje jako parametr liczbę całkowitą i zwraca liczbę powstałą z zadanej liczby poprzez odwrócenie kolejności jej cyfr dziesiętnych. Zakładamy, że liczba jest maksymalnie siedmiocyfrowa. Na przykład dla 1475 wynikiem powinno być 5741.

⋆ **Zadanie 3.21.** [Euler#1] Napisz funkcję, która wyznaczy sumę wszystkich wielokrotności liczb 3 i 5 ze zbioru $\{1, 2, \ldots, u\}$, gdzie $u \in \mathbb{N}$ jest parametrem funkcji. Dla przykładu, dla $u = 10$ wielokrotności to 3,5,6,9, a ich suma jest równa 23.

⋆ **Zadanie 3.22.** [Euler#25] Wyrazy ciągu Fibonacciego $(F_n)_{n=0,1,\ldots}$ określone są wzorem $F_0 = F_1 = 1$ oraz $F_n = F_{n-1} + F_{n-2}$ dla $n \geqslant 2$. Napisz funkcję, która dla danego k zwróci takie i, że F_i jest najmniejszym elementem, który ma dokładnie k cyfr w zapisie dziesiętnym. Na przykład dla $k = 3$ jest to 12, bo $F_{12} = 144$, a $F_{11} = 89$. Uwaga: funkcja ma działać także np. dla $k = 1000$.

3.6. Wskazówki i odpowiedzi do ćwiczeń

Wskazówka do zadania 3.14. Spróbuj zrobić to zadanie niewprost, czyli sprawdzić, czy prostokąty nie mają części wspólnej. □

Wskazówka do zadania 3.16. Dla liczby k co najwyżej należy sprawdzić, czy dzieli się ona bez reszty przez każdą z liczb $2, 3, \ldots, \sqrt{k}$. □

Wskazówka do zadania 3.19. Do estetycznego sformatowania „tabelki" użyj narzędzi z biblioteki <iomanip>. □