WPROWADZENIE DO C++

Michał Matlak

Helion

Redaktor prowadzący: Małgorzata Kulik

Projekt okładki: Studio Gravite / Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki
Grafika na okładce została wykorzystana za zgodą Shutterstock.com

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

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

ISBN: 978-83-283-9175-8

Printed in Poland.

Spis treści

Rozdział 6.

Automatyzacja obliczeń

Załóżmy, że mamy dodać do siebie 1000 liczb, które znajdują się w pamięci komputera, zadeklarowane np. jako tablica:

```
double aa[1000];                                                    (82)
```

Aby to zrobić za pomocą narzędzi znanych do tej pory, musielibyśmy się nieźle natrudzić. Trudność tę można pokonać, wprowadzając pojęcie **pętli**, czyli fragmentu programu, który jest wielokrotnie powtarzany. Problem rozwiązuje tu następujące polecenie pętli:

```
for (wartość początkowa; warunek sterujący; sposób powtarzania)     (83)
{
   instrukcje;
}
dalsze instrukcje;
```

Polecenie to powoduje, że obszar `instrukcje;` (pomiędzy nawiasami { oraz }) będzie powtarzany tyle razy, ile wynika z założonej `wartości początkowej`, użytego tu `warunku sterującego` i `sposobu powtarzania`. Najlepiej zrozumieć działanie instrukcji `for` na przykładzie.

Załóżmy, że w tablicy (zadeklarowanej np. jako `double aa[1000]`) znajduje się 1000 liczb (`aa[0], aa[1], ..., aa[999]`). Listing 6.1 przedstawia program, który obliczy sumę tych liczb.

LISTING 6.1. *Działanie pętli typu for*

```
#include <iostream>                                                 (84)
#include <iomanip>
#include <cmath>
#include <cstdio>

using namespace std;

char s=' ';
```

```
int i;
double aa[1000];
double x;

int main()
{

aa[0]=12.43;       //Należy tu wpisać wszystkie liczby
aa[1]=0.125;
.
.
.
aa[999]=121.11;
//////////////////////////////////////
x = aa[0];
for (i = 1; i < 1000; i=i+1)
{
   x = x + aa[i];              //Lub x+=aa[i];
}
//////////////////////////////////////
cout<<fixed;
cout<<"suma liczb wynosi"<<s<<s<<x<<endl;

getchar();
return 0;
}
```

Przypomnijmy jeszcze, co oznacza użyty wyżej zapis `x = x + aa[i]` lub `i = i + 1`. Znak = nie jest tu matematycznym znakiem równości, tylko znakiem podstawienia, który mówi: najpierw oblicz prawą stronę po znaku =, a następnie wstaw do zmiennej po lewej stronie, przed znakiem =. Po tej uwadze możemy już omawiać, jak działa użyta wyżej pętla `for`. Nawias `(i = 1; i < 1000; i = i + 1)` w pętli `for` generuje dla zmiennej `i < 1000` następujący ciąg wartości: `i = 1, 2, …, 999` (stąd `i` nazywa się **licznikiem pętli**). W pierwszej iteracji pętli (dla `i = 1` oraz `x = aa[0]`) do `x` dodane zostanie `aa[1]` i umieszczone znowu w `x`. Czyli po pierwszym cyklu: `x = aa[0] + aa[1]`. Teraz do `i = 1` zostanie dodane 1 (bo `i = i + 1`). Druga iteracja rozpoczyna się zatem z wartością `i = 2` oraz `x = aa[0] + aa[1]`. Ponieważ ogólnie `x = x + aa[i]` oraz po pierwszej iteracji `i = 2`, po drugiej iteracji otrzymamy w wyniku `x = aa[0] + aa[1] + aa[2]` itd., aż w końcu w ostatniej iteracji otrzymamy `i = 999`, czyli do `x` dodane zostanie `aa[999]`. Po ostatnim przebiegu pętli w `x` pojawi się zatem cała interesująca nas suma `aa[0] + aa[1] + ... + aa[999]`. Dlatego `x` nosi nazwę **akumulatora**. Kolejne powiększenie `i` o 1 da w wyniku `i = 1000`, a więc warunek `i < 1000` przestanie być spełniony. Pętla została zatem ukończona i komputer przeskoczy do wykonania poleceń wydruku. Widzimy więc, że kod

programu dodawania 1000 liczb przy użyciu pętli **(83)** jest bardzo krótki. W pętli **(84)** możemy stosować skrótowy zapis zgodnie z tabelą 4.3 jako opcję, czyli zamiast `x = x + aa[i];` można użyć skrótowego zapisu `x += aa[i]`. Tego skrótu często używa się w praktyce, natomiast zapisu `i += 1` (zamiast `i = i + 1`) raczej nie, chociaż jest on poprawny. Wprowadzono także skrótowe zapisy `++i` oraz `i++`, oznaczające inkrementację (powiększenie) indeksu `i` o wartość 1. Skrót `++i` oznacza, że do `i` należy dodać 1, zanim zmienna `i` zostanie użyta. Natomiast `i++` używamy wtedy, gdy `i` ma być powiększone o 1 w kolejnej operacji tam, gdzie się ono pojawi. Skróty `++i` oraz `i++` oznaczają zatem co innego i należy przy ich używaniu zachować pewną ostrożność. W niektórych sytuacjach mogą być jednak równoważne, np. w **(84)** możemy użyć w instrukcji `for` zarówno skrótu `++i`, jak i `i++`, ponieważ w obu wypadkach indeks `i` generowany w pętli będzie przebiegał przez ten sam zbiór wartości (`i = 1, 2,.., 999`). W podobny sposób działa dekrementacja — `i` oraz `i`—, czyli obniżanie indeksu `i` o wartość 1. Do używania zapisu skrótowego należy się zatem powoli przyzwyczajać, ponieważ jest on często stosowany w praktyce.

Jeśli chcielibyśmy obliczyć np. sumę `aa[0] + aa[2] + ... + aa[998]`, to należałoby w programie dokonać zmiany: `for(i = 1; i < 1000; i = i + 1)` na `for(i = 2; i < 1000; i = i + 2)`. Podobnie zatem, jak wspomniano wyżej, możemy używać skrótowej notacji `++i` oraz `i += 2` zamiast odpowiednio `i = i + 1` oraz `i = i + 2`.

A oto kolejny przykład. W rozdziale 2. omawialiśmy skrótowo możliwość użycia w praktyce biblioteki `<vector>`. Teraz, przy okazji omawiania polecenia pętli typu `for`, pokażemy, jak w bieżącym programie można łatwo rozszerzyć wymiar przestrzeni `vector` (zwiększyć liczbę składowych), jeśli zaistnieje taka konieczność. Założona bowiem na początku liczba składowych wektora może w trakcie działania programu okazać się niewystarczająca. Do tego celu służy operacja `push_back()`, której działanie poznamy w listingu 6.2. Zadeklarowano tu zmienną `wektor(i), (i=0, 1, 2, 3)`, której wymiar będziemy mogli podwyższyć w programie. Podwyższymy najpierw zadeklarowany wymiar równy 4 (`wektor(4)`) o 1 przy użyciu polecenia `wektor.push_back(3);` i następnie wprowadzimy do nowej zmiennej `wektor[4]` liczbę, aby sprawdzić za pomocą wydruku, czy rzeczywiście w tej zmiennej znajduje się wstawiona liczba. Uzyskaliśmy więc ten sam efekt, jak gdybyśmy od samego początku użyli deklaracji `wektor(5)` zamiast `wektor(4)`. Aby z kolei zwiększyć wymiar do 16 (`wektor(15)`), użyta została poniżej pętla typu `for`. Ponawiamy próbę sprawdzenia, umieszczając np. w zmiennej `wektor[12]` liczbę, która następnie, podobnie jak wyżej, zostaje wydrukowana tak, jak być powinno. Jest to potwierdzenie, że rozszerzona przestrzeń jest gotowa do użycia.

LISTING 6.2. *Powiększanie wymiaru przestrzeni przy użyciu operacji push_back()*

```
#include <iostream>                                                    (85)
#include <vector>
#include <cstdio>

using namespace std;

int i;

int main()
{
vektor <double> wektor(4);        //Tu są nawiasy ()

wektor[0]=-0.1;
wektor[1]=0.1;        //Tu są nawiasy []
wektor[2]=0.2;
wektor[3]=-0.3;

wektor.push_back(3);        //Rozszerzenie wymiaru o 1
cout<<fixed;
wektor[4]=0.123456;        //I sprawdzenie, czy to działa
cout<<"wektor[4]="<<wektor[4]<<endl;

getchar();        //Nacisnąć Enter

for(i=0;i<=10;i++)
{
wektor.push_back(5+i);
}

wektor[12]=-0.456789;
cout<<"wektor[12]="<<wektor[12];

getchar()
return 0;
}
```

Aby jeszcze lepiej zrozumieć funkcjonowanie pętli `for`, obliczymy zamiast sumy (jak w **(84)**) iloczyn n kolejnych liczb, czyli $n!$ (n silnia), gdzie $n! = 1 \cdot 2 \cdot \ldots \cdot n$. Listing 6.3 przedstawia przykład takiego programu.

LISTING 6.3. *Użycie pętli typu for do obliczania iloczynu*

```
#include <iostream>                                                    (86)
#include <cmath>
#include <iomanip>
#include <cstdio>
```

```
using namespace std;

char s=' ';

const int n=10;
int i, k = 1;

int main()
{

for(i = 2; i <= n; ++i)
{
   k = k * i;        // Tabela 4.3 pozwala wprowadzić skrót k *= i;
}

cout<<"n! wynosi"<<s<<s<<k;

getchar();
return -0;
}
```

Zadeklarowaliśmy tu, że `n=10`, czyli obliczamy 10!. Zastosowaliśmy też skrót `++i`, zamiast pisać `i = i + 1`. Startujemy w pętli od `k = 1` oraz `i = 2`. Po pierwszym przebiegu `k = 2 = 1·2`, po drugim (`i = 3`) `k = 1·2·3` itd., aż osiągniemy (`i = 10`) `k = 1·2·...·10 = 10!`. Kolejne `++i` powiększy `i` o 1, dając w wyniku `i = 11`, czyli warunek `i <= n` (`n = 10`) przestanie być spełniony, co oznacza zakończenie wykonywania pętli. Po wyjściu z pętli wynik będzie się znajdował w zmiennej (akumulatorze) `k`, a komputer wydrukuje wynik `10!`.

W listingach 6.1 – 6.3 poznaliśmy polecenie pętli `for` **(83)**, która obejmuje przypadek, gdy liczba powtórzeń jest znana. Bardzo często jednak liczby powtórzeń nie da się z góry przewidzieć. Wtedy używamy innych poleceń, które same mogą decydować o liczbie potrzebnych powtórzeń. Są dwa takie standardowe polecenia (z pozoru do siebie podobne, ale jednak różne), które organizują powtarzanie (pętlę) w taki właśnie sposób. Pierwszym z nich jest polecenie `do...while` (czyli z ang. „wykonuj…dopóki"). Polecenie to ma następującą postać:

(87)

```
do
{
   instrukcje do wykonania;
}
while(warunek);       //Tutaj jest ;
następne instrukcje;
```

Działanie tego polecenia polega na wielokrotnym powtarzaniu sekwencji poleceń zawartej między nawiasami { oraz } w taki sposób, że za każdym razem sprawdzany jest warunek następujący po słowie `while`. Pętla **(87)** będzie wykonywana tak długo, dopóki warunek ten będzie spełniony. W przeciwnym razie komputer przejdzie do wykonania poleceń `następne instrukcje;`. Zauważmy przy tym, że:

- Pętla będzie zawsze wykonana jeden raz, nawet wtedy, gdy warunek następujący po słowie `while` nie będzie spełniony.
- Jeśli warunek zawsze będzie spełniony (bo mamy np. błąd w programie lub w naszych wzorach), komputer będzie wykonywał niekończącą się pętlę bez jakiegokolwiek rezultatu. W takiej sytuacji należy nacisnąć kombinację klawiszy *Ctrl+Break*. Działanie programu ulegnie wtedy przerwaniu.

A oto druga możliwość, czyli polecenie `while`, którego schemat jest następujący:

```
while(warunek)
{                                                          (88)
   instrukcje do wykonania;
}
dalsze instrukcje;
```

Jeśli warunek po słowie `while` będzie spełniony, to będą wykonywane kolejne instrukcje zawarte w obszarze pomiędzy dwoma nawiasami { oraz } aż do momentu, gdy warunek następujący po `while` przestanie być spełniony. Wówczas komputer przejdzie do wykonywania poleceń `dalsze instrukcje;`. Gdyby w bloku była do wykonania tylko jedna instrukcja, znaki { oraz } możemy pominąć. Polecenie **(88)** działa więc nieco inaczej niż poprzednie **(87)**. Musimy jednak i tu zachować pewną ostrożność, ponieważ:

- Jeżeli warunek po słowie `while` nie będzie spełniony, to blok pomiędzy{ i } programu nie będzie już wykonywany i nastąpi przeskok do poleceń `dalsze instrukcje;`.
- Jeśli `warunek` zawsze będzie spełniony, to komputer będzie wykonywał niekończącą się pętlę, którą można zatrzymać tylko przez naciśnięcie wspomnianej wyżej kombinacji klawiszy *Ctrl+Break*.

Zasadnicza różnica pomiędzy poleceniami **(87)** a **(88)** tkwi więc w tym, że w przypadku niespełnienia warunku w **(87)** komputer i tak jeden raz wykona pętlę, podczas gdy w poleceniu **(88)** już nie. Świadomość tego faktu może być czasem pomocna przy układaniu programu. Działanie poleceń `do...while` oraz `while` omówimy na konkretnych przykładach. Oto pierwszy z nich:

Załóżmy, że dwie liczby `a` i `b` typu `double` zostały już wcześniej obliczone w programie. Naszym celem jest teraz np. tablicowanie funkcji `y = exp(sin(x))` z krokiem 0.01 w przedziale od `a` do `b` (lub od `b` do `a` w zależności od tego, która z tych liczb jest większa). Komputer „wie", ile wynosi `a` oraz ile wynosi `b`, ale my tego nie wiemy, bo zresztą same wartości `a` oraz `b` mogą zależeć jeszcze od innych parametrów wejściowych i mogą stale się w programie zmieniać. Nie możemy zatem zatrzymywać komputera za każdym razem, aby stwierdzić, ile komputer musi wykonać w pętli powtórzeń, tablicując funkcję z ustalonym krokiem 0.01. Niech więc komputer zadecyduje sam, co i jak ma robić, ale my musimy mu w tym pomóc, pisząc w tym celu program przedstawiony na listingu 6.4.

LISTING 6.4. *Użycie pętli typu do...while*

```
#include <iostream>                                                    (89)
#include <cmath>
#inxclude <iomanip>
#include <cstdio>

using namespace std;

const double a=-10.;
const double krok=0.01;
const double b=10.;

char s=' ';
double a1, b1,x, y;

int main()
{

a1=a;
b1=b;

if(a>b)
{
    a1=b;
    b1=a;
}

cout<<fixed;
x=a1;

do
{
    y=exp(sin(x));
    cout<<"x="<<x<<s<<s<<"y="<<y<<endl<<endl;
    x=x+krok;
```

```
}
while(x<b1);

getchar();
return 0;
}
```

Na samym początku deklarujemy zmienne użyte w programie i za pomocą instrukcji `if` korygujemy przedział zmienności `[a, b]` na `[a1, b1]`, jeśli `a > b`. Teraz `a1 <= b1` i możemy już rozpocząć tablicowanie naszej funkcji w przedziale `[a1, b1]` z krokiem 0.01. Tablicowanie rozpoczynamy od wartości początkowej argumentu `x = a1`, z którym wchodzimy do pętli `do...while`, drukując kolejno argument i obliczoną wartość funkcji. Polecenie `x = x + 0.01` zwiększa teraz `x` o 0.01. Jeśli spełniony jest warunek `x <= b1` w `while`, to komputer znowu wykona pętlę itd. W końcu `x` stanie się większe od `b1` i komputer, kończąc pętlę, zatrzyma się na poleceniu `getchar();`.

Jako ciekawostkę podamy jeszcze informację, że krok, z jakim tablicujemy funkcję, podawany jako 0.01, jest w istocie nieco inny (może to być np. wartość 0.009999999776482582, różniąca się nieco od 0.01 — zależy to od konkretnego kompilatora). Jest to liczba, z którą faktycznie pracuje komputer, popełniając narastające błędy zaokrąglenia. Uwaga ta dotyczy każdej innej zmiennej zmiennopozycyjnej.

W listingu 6.5 pokażemy teraz działanie polecenia `while` (porównaj **(88)**). Przedstawia to program z poprzedniego listingu po zastosowaniu definicji pętli `while` **(88)**.

LISTING 6.5. *Działanie pętli typu while*

```
#include <iostream>                                                    (90)
#include <cmath>
#include <iomanip>
#include <stdio.h>

using namespace std;

const double a=-10.;
const double krok=0.5;
const double b=10.;


char s=' ';
double a1, b1,x, y;

int main()
{
```

```
    a1=a;
    b1=b;

if(a>b)
{
    a1=b;
    b1=a;
}

cout<<fixed;
x=a1;

while(x<b1)
{
    y=exp(sin(x));
    cout<<"x="<<x<<s<<s<<"y="<<y<<endl<<endl;
    x=x+krok;
}
getchar();
return 0;
}
```

Pętla w tym fragmencie programu przebiega również pomiędzy nawiasami { oraz }. Dopóki spełniony będzie warunek `x <= b1` w `while`, dla każdego `x` będzie obliczana funkcja `y = exp(sin(x))`, a potem każdorazowo nastąpi wyświetlenie jednej linii tabeli (czyli `x` i `y`) oraz powiększenie `x` o 0.01. Proces ten skończy się, gdy `x` stanie się większe od `b1` i nastąpi przeskok do polecenia `getchar();`, które zatrzyma program w celu umożliwienia obejrzenia wyników.

Polecenia typu `do...while` i `while` mogą być również użyte wszędzie tam, gdzie liczba powtórzeń jest łatwa do ustalenia. Spójrzmy więc na listing 6.6. Zastosujemy tu równolegle polecenia `do...while` oraz `while` do tego samego problemu, który omawialiśmy przy okazji listingu 6.1. Oto odpowiedni fragment programu z listingu 6.1, zaznaczony za pomocą dwóch rzędów ukośników, który po poniższej modyfikacji sumuje 1000 liczb umieszczonych w tablicy zadeklarowanej jako `double aa[1000]` przy użyciu pętli `do...while`.

LISTING 6.6. *Użycie pętli do...while dla ustalonej liczby powtórzeń*

```
i = 1;                                                    (91)

x = aa[0];
do
{
   x = x + aa[i];          //Lub x+=aa[i];
```

```
   i = i + 1;                    //Lub ++i;
}
while(i < 1000);
```

W przykładzie z listingu 6.7 użyjemy polecenia `while`.

LISTING 6.7. *Użycie pętli typu while dla ustalonej liczby powtórzeń*

```
i = 1;                                                    (92)

x = aa[0];
while(i < 1000)
{
   x = x + aa[i];          //Lub x+=aa[i];
   i = i + 1;              //Lub ++i;
}
```

Jak widać, w każdym z przypadków **(91)** i **(92)** odpowiedni fragment pętli wykonał (znaną z góry) liczbę dodawań równą 999 (zaczynając od wartości początkowych indeksu `i = 1` oraz akumulatora `x = aa[0]`).

Instrukcje warunkowe typu `if` **(77a, b)**, `switch` **(80a, b)** wraz z instrukcjami powodującymi automatyzację obliczeń i organizującymi pętle, jak instrukcje `for` **(83)**, `do...while` **(87)** czy `while` **(88)**, należą do podstawowych narzędzi programowania. Z praktycznych powodów wygodnie jest opuścić jakąś pętlę wcześniej, niż przewiduje to organizacja programu. Do tego celu służy polecenie skoku bezwarunkowego `goto etykieta;`. Komputer przerywa wtedy działanie pętli, przeskakuje bezpośrednio do instrukcji znajdującej się tuż za linią, w której umieszczono znacznik (`etykieta:`), i wykonuje kolejno dalsze instrukcje. Inny sposób na opuszczenie pętli to użycie polecenia `break` (porównaj **(80a, b)**). Działanie poleceń `goto` (z ang. „idź do") oraz `break` jest jednak odmienne. Zademonstrujemy to na przykładzie pokazującym sposób opuszczania pętli typu `do...while` przez oba te polecenia.

W instrukcji `do...while` w celu pokazania różnicy umieścimy polecenia `goto` i `break` zamiennie w tej samej linii:

```
do                                                        (93)
{
instrukcje;

goto skok;    lub    break;
}
```

```
while (warunek);
   następne instrukcje 1;
.
   instrukcje;
.
skok:
   następne instrukcje 2;
```

Załóżmy, że w przestrzeni między nawiasami { oraz } znajduje się polecenie `goto skok;`. Program po dojściu w pętli do polecenia `goto skok;` opuści bezwarunkowo pętlę i przejdzie do wykonywania polecenia znajdującego się bezpośrednio po etykiecie `skok:`, czyli wykona polecenia `następne instrukcje 2;`. Załóżmy teraz, że w pętli mamy polecenie `break;` zamiast `goto skok;`. Program przeskoczy wtedy bezwarunkowo do najbliższej instrukcji poza pętlą, czyli rozpocznie wykonywanie poleceń `następne instrukcje 1;`. Polecenia `break;` można użyć do opuszczenia tylko jednej pętli w przypadku pętli zagnieżdżonych.

Należy tu jeszcze dodać, że skok bezwarunkowy można zrealizować tylko wtedy, gdy zarówno `goto`, jak i etykieta znajdują się wewnątrz jednej i tej samej funkcji (jest to blok instrukcji posiadający własną nazwę — porównaj rozdział 7.). Ponieważ nazwy etykiet obowiązują jedynie w ramach danej funkcji (tzn. mają charakter lokalny), niemożliwe jest zrealizowanie przeskoku za pomocą `goto` pomiędzy funkcjami. Ale oznacza to, że tej samej nazwy etykiety dla `goto` można używać w różnych funkcjach.

Polecenia `goto` można użyć w programie do utworzenia niekończącej się pętli, służącej do wyświetlania wyników pośrednich. Pokażemy to na kolejnym przykładzie.

Umieśćmy gdzieś w programie następującą sekwencję:

```
cout("x="<<x<<s<<"y="<<y);                                    (94)
skok:
goto skok;
```

Tutaj, inaczej niż wcześniej, etykieta `skok:` umieszczona jest przed `goto skok;`. Oznacza to, że komputer wyświetli (wydrukuje) najpierw `x` i `y` z instrukcji `cout(...);` i będzie wykonywał dalej niekończącą się pętlę, zawartą pomiędzy znacznikiem `skok:` i poleceniem `goto skok;`. Wykonując pętlę, komputer będzie cały czas wyświetlał na ekranie `x` oraz `y`, ponieważ pomiędzy poleceniami `skok:` i `goto skok;` nie ma żadnych innych poleceń. Daje to podobny efekt jak wstrzymanie działania programu za pomocą polecenia `getchar();` lub `getch();`.

Wyjścia z niekończącej się pętli w **(94)** dokonujemy (podobnie jak w **(87)** i **(88)**) przez naciśnięcie klawiszy *Ctrl+Break*. Polecenia `goto etykieta;`, umożliwiającego wykonanie skoku bezwarunkowego, nie należy jednak nadużywać, gdyż może to negatywnie wpłynąć na czytelność programu i zwiększyć ryzyko popełnienia błędu. Większe bezpieczeństwo uzyskamy tu przez użycie takich poleceń jak `if` **(77a, b)**, `switch` **(80a, b)**, `for` **(83)**, `do...while` **(87)** i `while` **(88)**, które praktycznie eliminują potrzebę użycia `goto`, a do tego są jasne i przejrzyste. Paradoksalnie jednak odpowiednio dobrana kombinacja skoku bezwarunkowego `goto` oraz instrukcji warunkowej `if` pozwala z powodzeniem zastąpić polecenia typu `switch`, `for`, `do...while` i `while`. Sprawdzenie prawdziwości powyższego stwierdzenia pozostawiamy Czytelnikowi.

WPROWADZENIE DO C++

- **Najważniejsze definicje języka**
- **Inspirujące przykłady praktyczne**
- **Solidne podstawy języka C++**

Poznaj język C++ przy pomocy 68 przykładowych programów

Należący do języków ogólnego przeznaczenia C++ od lat pozostaje niezwykle popularny wśród programistów. I wciąż zdobywa nowych entuzjastów. Jeśli chcesz się przekonać, jak się pracuje z tym wszechstronnym i potężnym językiem, przygodę z nim koniecznie rozpocznij od tego podręcznika! Przeprowadzi Cię on krok po kroku przez najważniejsze zagadnienia i definicje związane z C++. Z pewnością docenisz to, że każde pojęcie, które należy opanować, zostało zilustrowane gotowym, działającym programem. Autorowi książki bowiem chodzi o to, by teorię przyswajać na podstawie praktyki — niejako przy okazji.

Praca z tym podręcznikiem pozwoli Ci się przekonać, jak dane definicje sprawdzają się w użyciu — będziesz je przekładać na własną aktywność jako programista. Przyjrzysz się takim związanym z C++ zagadnieniom jak stosowane w tym języku rodzaje wielkości, charakterystyczne dla niego operacje wejścia-wyjścia czy instrukcje warunkowe i sterowanie pracą programu. Poznasz operacje na zadeklarowanych wielkościach, działania na liczbach zespolonych i funkcje standardowe, dowiesz się też, w jaki sposób w C++ automatyzować obliczenia i czym charakteryzuje się architektura tego języka. Zdobędziesz wiedzę o jego strukturach, klasach i szablonach. A wszystko to od strony praktycznej i równocześnie z perspektywy stricte matematycznej — bo język C++ to matematyka w najpiękniejszym programistycznym wydaniu!

Prof. dr hab. Michał Matlak — fizyk teoretyk. Emerytowany pracownik Zakładu Fizyki Teoretycznej. Były kierownik studiów doktoranckich na Wydziale Matematyki, Fizyki i Chemii Uniwersytetu Śląskiego w Katowicach. Autor książki *Język C/C++ i obliczenia numeryczne. Krótkie wprowadzenie*.

Helion

helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-9175-8

9 788328 391758

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 57,00 zł