

Michał Matlak

Język C/C++ i obliczenia numeryczne

Krótkie wprowadzenie



Poznaj zalety języka C/C++!

- Początki programowania, czyli jak zacząć działać bez dogłębnej znajomości tematu
- Struktura prostych programów, czyli czego na pewno nie wolno Ci pominąć
- Obliczenia numeryczne, czyli jak możesz zastosować w praktyce swoją świeżo zdobytą wiedzę

Helion 

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Opieka redakcyjna: Ewelina Burska

Projekt okładki: Studio Gravite/Olsztyn
Obarek, Pokoński, Pazdrijowski, Zaprucki

Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock.

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

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/jcconu>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-2152-6

Copyright © Helion 2016

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	5
Rozdział 1. Szybki start	7
Rozdział 2. Rodzaje wielkości w języku C/C++ i ich deklaracja	11
Rozdział 3. Deklaracja tablic	17
Rozdział 4. Operacje na zadeklarowanych wielkościach i funkcje standardowe	19
Rozdział 5. Instrukcje warunkowe i sterowanie pracą programu	23
Rozdział 6. Automatyzacja obliczeń	31
Rozdział 7. Architektura programu i pierwsze programy	39
Rozdział 8. Operacje wyprowadzania wyników	51
Rozdział 9. Opis przykładowych programów do nauki programowania	57
Rozdział 10. Wskaźniki, tablice, funkcje, struktury, przeładowanie operatora, liczby zespolone	63
Rozdział 11. Przestrzenie nazw	77
Zakończenie	81
Dodatek A Cztery programy przykładowe w oparciu o rozdziały 9. i 10.	83
Literatura	89
Skorowidz	91

Rozdział 10.

Wskaźniki, tablice, funkcje, struktury, przeładowanie operatora, liczby zespolone

Każdej wielkości, np. x , zadeklarowanej w programie przypisany jest w pamięci komputera adres, który oznacza się jako $\&x$. Adres ten, czyli numer komórki w pamięci komputera, nosi nazwę wskaźnika do wielkości x . Znając wskaźnik, np. a , możemy postawić pytanie: co kryje się pod adresem a . Odpowiedzi udzieli nam tzw. operacja dereferencji (nazywana też operacją wyłuskiwania), którą zapisuje się jako $*a$. Jeśli więc $a = \&x$, to $x = *a$. Innymi słowy, x jest tym samym co $*(\&x)$. Zatem gwiazdka umieszczona przed symbolem wskaźnika a „demaskuje” zawartość komórki pamięci o adresie a . Zmiennej $*a$ możemy dalej używać w programie zamiast $x = (*a)$, np. zamiast pisać w programie $y = x * x$; możemy użyć alternatywy $y = (*a) * (*a)$; gdzie symbol $*$ między nawiasami oznacza zwykle mnożenie. Jeśli w programie chcemy używać wskaźnika a , to musimy go zadeklarować:

typ $*a$;

W deklaracji tej $*a$ musi być tego samego typu co wielkość, którą wskazuje, tzn. jeśli a ma wskazać wielkość x typu `double`, to $*a$ musi być zadeklarowane jako `double *a`; (a samodzielnie nie deklarujemy). Aby oswoić się z wprowadzonymi pojęciami, rozpatrzmy przykład 19. Oto on:

Przykład 19

Poniższy program jest bardzo krótki. Polecenie `printf()` poniżej wyświetli na ekranie stałą $x = 2.971828$, jej adres w komputerze, a następnie wyświetli 2 razy wartość zmiennej x , zapisaną w równoważnych postaciach jako $*a$ i $*(\&x)$.

```
////////////////////////////////////  
#include <stdio.h>  
#include <math.h>  
////////////////////////////////////
```

```

const double x = 2.971828;
/////////////////////////////////////////////////////////////////
double *a;
/////////////////////////////////////////////////////////////////
int main()
{
    a = &x;                // Poznajemy adres zmiennej x
    printf("x = %f  a = %d  *a = %f  *(&x) = %f\n", x, a, *a, *(&x));
    getchar();             // To pokazuje, że *a i *(&x) to jest to samo co x
    return 0;
}

```

Znamy już pojęcie tablicy oraz sposób, w jaki deklarujemy ją w programie. I tu wskaźniki okazują się bardzo pożyteczne, ponieważ zawartość całej tablicy można łatwo odczytać, używając jednego tylko wskaźnika. Aby to przedstawić, zadeklarujemy tablicę jednowymiarową np. jako `double tab[10]`. Nie wspominaliśmy jeszcze o tym, że samo słowo `tab`, jako nazwa tablicy, stanowi równocześnie wskaźnik do jej pierwszego elementu, czyli jego adres. Innymi słowy, `tab = &tab[0]`. Przyjęto tu konwencję, że:

```
tab[i] = *(tab + i) (93)
```

Wobec tego mamy tu prosty sposób dotarcia do zawartości każdego elementu tablicy, znając jedynie wskaźnik do niej, czyli `tab`. Dla dwuwymiarowych tablic kwadratowych, zadeklarowanych np. jako `double xtab[m][m]`, obowiązuje wzór:

```
xtab[i][j] = *(b + m * i + j) (94)
```

gdzie `b = xtab = &xtab[0][0]`. Zademonstrujemy to na poniższym przykładzie.

Przykład 20

Rozpatrujemy tu dwie tablice o nazwach `tab` (tablica jednowymiarowa o 4 elementach) i `xtab` (tablica dwuwymiarowa o 4 elementach). Przypisując symbolom `a` i `b` wskaźniki (adresy) do tych tablic, wyświetlimy elementy każdej z tablic na dwa sposoby. W sposobie pierwszym nie używamy wskaźników. W sposobie drugim używamy pisowni wskaźnikowej, zdefiniowanej wyżej. Wpisując ten program do komputera, można się przekonać, że obydwa te sposoby w zastosowaniu do tablicy `tab[4]` dają ten sam wynik, co można stwierdzić, porównując dwa pierwsze wydruki przy użyciu `printf()`. To samo dotyczy wyświetlenia elementów tablicy `xtab[2][2]` przy użyciu kolejnych dwóch poleceń `printf()`. Sprawdźmy też, że `xtab[i][j] = *(b + m * i + j)`, gdzie w naszym przykładzie `m = 2` i `b = xtab`.

```

/////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <math.h>
/////////////////////////////////////////////////////////////////
double tab[4];
double xtab[2][2];
/////////////////////////////////////////////////////////////////
int main()
{
    double *a, *b;
    tab[0] = 2.718282;
    tab[1] = 3.141579;
    tab[2] = 0.434294;
    tab[3] = 23.140693;
}

```

```

xtab[0][0] = 0.612151;
xtab[0][1] = -8.123456;
xtab[1][0] = 2.977519;
xtab[1][1] = 0.379143;

a = tab;
printf("tab[0] = %f tab[1] = %f tab[2] = %f tab[3] = %f\n",
tab[0], tab[1], tab[2], tab[3]);
printf("tab[0] = %f tab[1] = %f tab[2] = %f tab[3] = %f\n",
*a, *(a + 1), *(a + 2), *(a + 3));

b = xtab;
printf("xtab[0][0] = %f xtab[0][1] = %f xtab[1][0] = %f xtab[1][1] = %f\n",
xtab[0][0], xtab[0][1], xtab[1][0], xtab[1][1]);
printf("xtab[0][0] = %f xtab[0][1] = %f xtab[1][0] = %f xtab[1][1] = %f\n",
*b, *(b + 1), *(b + 2), *(b + 3));
// Zgodnie z powyższym wzorem xtab[i][j] = *(b + m * i + j) a dla m = 2 poszczególne elementy
// są równe:
// xtab[0][0] = *(b + 2 * 0 + 0) = *b, xtab[0][1] = *(b + 2 * 0 + 1) = *(b + 1),
// xtab[1][0] = *(b + 2 * 1 + 0) = *(b + 2) oraz xtab[1][1] = *(b + 2 * 1 + 1) = *(b + 3)
getchar();
return 0;
}

```

Poniżej prezentujemy inny sposób inicjowania tablic. Zamiast deklaracji dwóch tablic tuż przed `main` jako `double tab[4]`; i `double xtab[2][2]`; można wpisać od razu deklarację z przypisaniem wartości, czyli:

```

double tab[4] = {2.718282, 3.141579, 0.434294, 23.140693};
double xtab[2][2] = {0.612151, -8.123456, 2.977519, 0.379143};

```

gdzie w przypadku tablic `xtab[n][n]` wpisujemy najpierw elementy pierwszego wiersza, następnie drugiego wiersza itd. Używając powyższej deklaracji z przypisaniem, należy w `main` wykreślić 8 zbędnych wierszy zawartych pomiędzy liniami `double *a, *b;` oraz `a = tab;`.

Zasięg zmiennych lokalnych ograniczony jest do obiektu (np. funkcji lub `main`), w którym zostały one zadeklarowane. Dlatego często zachodzi potrzeba przekazywania ich z obiektu do obiektu, np. z `main` do funkcji itp. Wiemy już, że do funkcji można przekazywać parametry przez ich wartość (porównaj (66) i przykład 16.). Innym, często używanym sposobem jest przekazywanie parametrów lokalnych przez wskaźniki. Jest to szczególnie wygodne i często używane w praktyce w przypadku tablic. Rozpatrzmy w tym celu przykład 21a i b, który to ilustruje.

Przykład 21

- a) W przykładzie tym w celu zachowania prostoty rozpatrujemy funkcję jednoparametrową `funk(double *y)`, do której parametr przekazujemy nie przez wartość, lecz przez (adres) wskaźnik `y`. Sama funkcja wywoływana jest w `main` za pomocą polecenia `z = funk(&x);`, co oznacza, że zamiast `*y` zostanie tam wstawione `*y = *(&x) = x`. Funkcja `funk` oblicza zatem wyrażenie `1. + 2. * cos(*y)` dla `*y = x` i zwraca jego wartość za pomocą polecenia `return` wynik do zmiennej `z` w `main`, gdzie następnie wyświetlany jest wynik. A oto sam program:

```

////////////////////////////////////
#include <stdio.h>
#include <math.h>
////////////////////////////////////
double funk(double *y);
////////////////////////////////////
int main()
{
    const double x = 3.;
    double z;
    z = funk(&x);
    printf("x = %f   wynik = %f\n", x, z);
    getchar();
    return 0;
}
////////////////////////////////////
double funk(double *y)  // Tu *y będzie równe *(&x) = x po wywołaniu w main z = funk(&x);
{
    double wynik;        //Opcjonalnie można to skrócić do: return 1. + 2. * cos(*y);
    wynik = 1. + 2. * cos(*y);
    return wynik;
}
////////////////////////////////////

```

- b)** Przykład ten, będący rozwinięciem poprzedniego, pokazuje, jak za pomocą jednego wskaźnika przekazać do funkcji wiele argumentów zadeklarowanych lokalnie jako tablica. W tym celu współczynniki trójkianu $w[0]x^2 + w[1]x + w[2]$ deklarujemy lokalnie w `main` jako tablicę `w[3]`, a wartości trójkianu obliczamy w funkcji zadeklarowanej jako `double funk(double xx, double *y);`, przekazując do niej argument trójkianu przez jego wartość `x`, a tablicę współczynników `w[3]` przez jej wskaźnik, czyli nazwę `w`. Wywołanie tej funkcji w `main` (w celu obliczenia wartości trójkianu) ma zatem postać `z = funk(x, w);`, gdzie `z` jest wartością trójkianu odpowiadającą `x`. Sugeruje to, że przy wywołaniu funkcji `double funk(double xx, double *y)` podstawiamy do niej `xx = x` i `y = w`, co oznacza, że wskaźnik `y` staje się teraz w środku funkcji `funk` nazwą tablicy `w`. Dzięki temu współczynniki trójkianu w funkcji `funk`, występujące tam w postaci `*y`, `*(y + 1)`, `*(y + 2)` — mogące też wystąpić równoważnie (porównaj **(93)** jako `y[0]`, `y[1]`, `y[2]` — zostaną tam wykorzystane jako `*w`, `*(w + 1)`, `*(w + 2)` (czyli równoważnie jako `w[0]`, `w[1]`, `w[2]`), co umożliwi obliczenie wartości trójkianu. Polecenie `return wynik;` przekazuje tę wartość do zmiennej `z` w `main` jako wartość zwracaną. W następnej linii w `main` drukowane są `x` oraz `z`, czyli argument i poszukiwana wartość trójkianu. Pokazuje to poniższy program:

```

////////////////////////////////////
#include <stdio.h>
#include <math.h>
////////////////////////////////////
double funk(double xx, double *y);
////////////////////////////////////
int main()
{
    double w[3];          // Lub double w[3] = {1., 2., 3.}; ale wtedy 3 wiersze tuż po double z należy
                          //skreślić
    const double x = 3.;
    double z;

```

```

w[0] = 1.;
w[1] = 2.;
w[2] = 3.;
z = funk(x, w);
printf("x = %f  wynik = %f\n", x, z);
getchar();
return 0;
}
////////////////////////////////////////////////////
double funk(double xx, double *y)
{
    double wynik;
    wynik=((*y)*xx+(*(y+1)))*xx+(*(y+2)); //Lub też alternatywnie: wynik = (y[0] *xx +
                                           //y[1]) *xx + y[2];
    return (wynik);
}

```

Jeśli nasz program zawiera tablice i funkcje, to przekazania elementów tablicy (zadeklarowanej lokalnie) do jakiejś funkcji można dokonać jeszcze prościej, niż pokazano to w przykładzie 21b. W tym celu należy użyć zmodyfikowanej definicji funkcji (66), wprowadzając do niej typ i nazwę danej tablicy (czyli jej wskaźnik). W pierwszej linii (66) istniejący nawias uzupełniamy w następujący sposób:

```

typ nazwa(typ p1, typ p2, ..., typ pn, typ nazwa tablicy [], ...) (95a)

```

a pozostała część definicji (66) pozostaje bez zmian. Występujący po nazwie (wskaźniku) tablicy nawias [] pozostawiamy pusty, gdyż komputer sam rozpozna (po deklaracji tablicy), ile elementów posiada tablica o danej nazwie. Jeśli do przekazania jest wiele tablic, należy je wszystkie wpisać po kolei w nawiasie w (95a), używając rozdzielającego przecinka między poszczególnymi segmentami określającymi daną tablicę. Funkcję (95a) wywołujemy w programie, pisząc np. w odpowiedniej linii polecenie:

```

z = nazwa (p1, p2, ..., pn, nazwa tablicy, ...); (95b)

```

Funkcja, do której przekazujemy tablicę, może być również typu void, tzn. może nie zwracać żadnej wartości. Przypadek ten pokażemy niżej w przykładzie 25., a teraz przedstawimy bardzo prosty i krótki program (nawiązujący do przykładu 21b), który w praktyce ilustruje przekazywanie tablic w funkcjach za pomocą (95a i b).

Przykład 22

Przedstawiamy tu, zgodnie z definicją (95a), funkcję zadeklarowaną jako `double calc` ↪ `(double x, double a[]);`, która oblicza wartość trójmianu kwadratowego $w[0]x^2 + w[1]x + w[2]$, gdzie współczynniki trójmianu zadeklarowane są lokalnie w `main` w postaci tablicy `w[3]`. Sytuacja jest tu o wiele prostsza niż w przykładzie 21b, gdyż polecenie w `main` w postaci: `result = calc(arg, w);` od razu przekazuje do funkcji `calc` całą tablicę `w[3]`. Zatem funkcja `calc`, po jej wywołaniu, będzie działała w taki sposób, jakby `x` zostało zamienione na `arg` oraz tablica `a` na tablicę `w`. Następnie obliczona zostaje wartość samego trójmianu. Polecenie `return result;` przekazuje z kolei wartość trójmianu jako wartość zwracaną do `main` i umieszcza ją w zmiennej `wynik`. W kolejnej linii następuje wydruk argumentu `arg` i obliczonej wartości trójmianu `wynik`. Spójrzmy więc na poniższy program:

```

/////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <math.h>
/////////////////////////////////////////////////////////////////
const double arg = 10.; //Argument trójkątny
/////////////////////////////////////////////////////////////////
double calc(double x, double a[]);
/////////////////////////////////////////////////////////////////
int main()
{
    double w[3];                //Lub double w[] = {1., 2., 3.}
    double wynik;
    w[0] = 1.;
    w[1] = 2.;
    w[2] = 3.;
    wynik = calc(arg, w);
    printf("arg = %f  wynik = %f\n", arg, wynik);
    getchar(); //Zamiast poprzednich dwóch linii można też napisać:
    return 0; //printf("arg = %f  wynik = %f\n", arg, calc(arg, w));
}
/////////////////////////////////////////////////////////////////
double calc(double x, double a[])
{
    double result;
    result = (a[0] * x + a[1]) * x + a[2];
    return result; //Lub return (a[0] * x + a[1]) * x + a[2];
}

```

W języku C/C++ istnieje szczególnie przydatna możliwość wywoływania funkcji z funkcji w taki sposób, że parametrem (argumentem) funkcji wywołującej może być nazwa funkcji wywoływanej. Najłatwiej jest to zademonstrować na przykładzie. W tym celu wykorzystamy tu przykład 18., gdzie funkcję podcałkową `fcn`, która zwraca wartość, zadeklarowaliśmy jako `double fcn(double arg)`. Analogicznie do tablic samo słowo `fcn`, oznaczające nazwę funkcji, jest równocześnie wskaźnikiem (adresem) do tej funkcji. Ale po nazwie `fcn` (a również po każdej innej nazwie funkcji) występuje jeszcze dodatkowo nawias `()`. Przyjęto więc konwencję, aby wartość zwracaną przez funkcję `fcn` oznaczać albo przez `fcn()`, albo też przez `(*fcn)()`. Tak więc funkcję `fcn` z przykładu 18. można wywołać w programie na dwa sposoby: w znany nam już sposób, pisząc w jakiejś linii polecenie np. `y = fcn(x)`; lub też pisząc `y = (*fcn)(x)`. Oba te sposoby wywołania są całkowicie równoważne, bo dają w wyniku to samo `y` jako wartość zwracaną. Zarówno jeden, jak i drugi sposób realizuje w zasadzie wywołanie funkcji za pomocą jej wskaźnika (adresu), ponieważ wskaźnik ten (tu: nazwa `fcn`) w obu przypadkach został użyty w sposób jawny. Jak widać, również w przypadku funkcji takie pojęcia jak „wskaźnik”, „adres” czy „nazwa” można uznać za synonimy. Przykład 23., oparty na wspomnianym wyżej przykładzie 18., pokazuje w sposób konkretny, jak można wywołać funkcję z funkcji za pomocą nazwy funkcji wywoływanej. Przypomnijmy najpierw, że centralną rolę w przykładzie 18. pełni trójpłatowa funkcja `simpson`, zadeklarowana jako `double simpson(double a, double b, int m)`. Nowum polega na tym, że funkcję tę zastąpimy w przykładzie 23. czteropłatową funkcją, którą zadeklarujemy jako `double simpson(double a, double b, int m, double fcn(double x))`; a samą funkcję umieścimy w programie po `main`. Opcjonalnie możemy tu zastąpić `fcn`

przez (*fcn) w całej funkcji `simpson`. Widzimy teraz, że nowa funkcja `simpson` wywołuje funkcję `fcn` przy użyciu jej nazwy, która stała się argumentem funkcji `simpson`. Ale argument (parametr) każdej funkcji można przy jej wywoływaniu zamienić na jakiś inny. Nic więc nie stoi na przeszkodzie, aby funkcję całkującą wywołać w `main` za pomocą polecenia: `calka = simpson(a, b, m, funk);`. Ale wtedy musimy również naszą aktualną funkcję podcałkową o wybranej przez nas nazwie `funk` zadeklarować przed `main` jako `double funk(double x);`, a samą funkcję `funk` umieścić po `main`. Wywołanie funkcji `simpson` w programie jako `calka = simpson(a, b, m, funk);` spowoduje, że funkcja `simpson` będzie działać teraz dokładnie tak, jakby jej program został od razu napisany przy użyciu nazwy `funk` zamiast wcześniejszej nazwy `fcn`. Oznacza to, że w samej funkcji `simpson` wartość zwracana przez `fcn` zostanie automatycznie zastąpiona wartością zwracaną przez `funk` (porównaj też przykład 23.). Widzimy więc, że mamy tu teraz sytuację bardziej komfortową niż w przykładzie 18., bo funkcję podcałkową, która współpracuje z procedurą całkującą, możemy teraz nazwać, jak chcemy (np. `funk`). Niezwykle ważny jest też fakt, że tak zaprogramowana funkcja `simpson`, umożliwiająca użycie dowolnej nazwy dla funkcji podcałkowej, będzie funkcjonować również i wtedy, gdy funkcja `simpson` zostanie najpierw skompilowana do pliku o określonej nazwie, a następnie (jako plik w języku wewnętrznym) dołączona do programu właściwego w procesie kompilacji i konsolidacji. W przykładzie 23. omówimy teraz szczegółowo to, co należy zmienić w programie z przykładu 18., aby uzyskać pożądaną swobodę w wyborze nazwy funkcji podcałkowej (przejście od wspomnianej **wersji 1** do bardziej użytecznej **wersji 2**).

Przykład 23

Aby przejść od **wersji 1** do **wersji 2**, należy w przykładzie 18. (tuż przed `main`) usunąć poprzednią deklarację i zastąpić ją nową (w poniższej linii deklarujemy jednocześnie dwie funkcje, czyli `simpson` i `funk`, oddzielone przecinkiem!):

```
double simpson(double a, double b, int m, double fcn(double x)), funk(double x);
```

Zauważmy przy tym, że funkcja podcałkowa nazywa się teraz `funk` i dlatego nazwę funkcji podcałkowej na końcu przykładu 18. należy zamienić na `double funk(double x)`. Następnie w `main` zamiast polecenia wywołującego funkcję całkującą wpisujemy `calka = simpson(a, b, m, funk);`. Kolejna zmiana dotyczy samej procedury całkującej, w której należy zastąpić starą nazwę (o trzech argumentach) nową (o czterech argumentach), a mianowicie:

```
////////////////////////////////////Procedura całkująca////////////////////////////////////
double simpson(double a, double b, int m, double fcn(double x))
{
    // Bez zmian
}
////////////////////////////////////
```

gdzie samo wnętrze tej procedury jest identyczne jak w przykładzie 18. Wywołując w programie funkcję całkującą `calka = simpson(a, b, m, funk);`, powodujemy, że występujące wewnątrz nowej funkcji `simpson` wywołania `fcn(a)`, `fcn(b)` i `fcn(x)` (porównaj wnętrze procedury w przykładzie 18.) będą funkcjonowały identycznie jak odpowiednio `funk(a)`, `funk(b)` i `funk(x)`, bo `funk` zastępuje teraz wszędzie `fcn`. W listingu A.1 umieszczonym w dodatku A podajemy pełny tekst ulepszonej **wersji 2**, aby (przez porównanie z **wersją 1**) najłatwiej można było prześledzić wszystkie zmiany dokonane przy przejściu od **wersji 1** do **wersji 2**.

Skorowidz

A

adres, 63
 funkcji, 68
 tablicy, 64
 zmiennej, 63
akumulator, 32
architektura programu, 39
argumenty funkcji, 45, 46
automatyzacja obliczeń, 31

B

biblioteki specjalistyczne, 39
biblioteki standardowe, 39
 <complex>, 76, 78
 <float.h>, 45, 80
 <iostream>, 79
 <limits.h>, 45, 80
 <math.h>, 44
 <stdio.h>, 44
bisekcja, 60, 85
blok instrukcji, 38
break, 27, 37

C

całkowanie metodą
 Gaussa – Legende’a, 59, 84
 Simpsona, 57, 83
case, 27
<complex>, 76, 78
const, 14
cout, 79

D

definicja (deklaracja)
 funkcji, 45, 46
 stałej, 14

 struktury, 72, 73
 tablicy, 17
 wskaźnika, 63
 zmiennej, 14
dekrementacja, 32
dereferencja, 63
double, 13
drukowanie wyników, 52, 53, 79
dyrektywa
 #define, 14, 48
 #include, 39
dzielenie liczb całkowitych, 21

E

else, 24
else if, 24

F

fclose, 54
float, 13
<float.h>, 45, 80
fopen, 53
for, 31
fprintf, 54
funkcje
 z przeladowanym operatorem, 73 - 75
 zwracające wartość, 45, 46
funkcje typu
 inline, 48
 main, 48
 standardowe, 19, 20
 void, 46, 48

G

getch, 35, 36
getchar, 35, 36
goto, 36, 37

I

#include, 39, 40
inkrementacja, 32
inline, 48, 49
instrukcje warunkowe, 23, 37
 do...while, 33
 if...else if, 24, 25
 if, 23
 if...else, 24
 switch, 27
 while, 33

K

klucz do przestrzeni nazw, 77
kropka dziesiętna, 9, 11

L

liczby
 całkowite, 12
 zespolone, 73
 zmiennopozycyjne, 13
licznik pętli, 32
<limits.h>, 45, 80

Ł

łańcuchy symboli, 12

M

main, 48
makroinstrukcje, 48
<math.h>, 44
metoda
 bisekcji, 60, 85
 Gaussa – Legendre’a, 59, 84
 Newtona, 60, 61, 86
 Simpsona, 57, 83
miejsce zerowe funkcji, 60

N

napisy i ich umieszczanie, 51
nazwy
 stałych, 11–14
 struktur, 71, 72
 zmiennych, 11–14

O

obliczanie całki, 57, 59, 83, 84
operacje
 arytmetyczne, 19
 logiczne, 26
 porównywanie liczb, 26
 przypisania, 14
operatory, 21
 przeładowania, 73 - 75
otwarcie pliku, 53

P

parametry funkcji, 45, 46
pętla
 do...while, 33
 for, 31
 while, 33
pliki nagłówkowe, 39
precyzja
 podwójna, 13
 pojedyncza, 13
printf, 52
przekazywanie
 parametrów, 45
 tablic, 67
 wskaźników, 65, 66
przestrzenie nazw, 77
puts, 47, 51

R

reszta z dzielenia, 21
return, 40, 45
równanie kwadratowe, 7

S

schemat programu, 39, 40
słowa kluczowe, 12
stała, 14
<stdio.h>, 44

sterowanie pracą programu, 23
sterowanie wydrukiem, 51 - 55
struktury, 72, 73
switch, 27
symbole
 //, 7
 /*...*/ , 41

Ś

średnik, 10

T

tablice, 17
typy danych
 całkowite, 12
 char, 12
 łańcuchowe, 12
 zmiennopozycyjne, 13

U

using namespace std, 78
utrata dokładności, 41

V

void, 46, 48

W

warunek, 25, 26
wektory, 17, 72
wskaźniki, 63
wyluskiwanie, 63
wyrażenie logiczne, 26
wywoływanie
 funkcji, 46
 funkcji z funkcji, 68, 69

Z

zakresy wielkości, 12, 13, 45, 80
zamknięcie pliku, 54
zapisywanie do pliku, 53, 54
zatrzymanie
 pętli, 33, 34
 programu, 35, 36
zmiennie
 globalne, 40, 47
 lokalne, 47
 łańcuchowe, 12
zwracanie wartości, 45, 46

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

Język C i jego następca C++ towarzyszą programistom komputerowym od dawna. Mimo to zainteresowanie tymi językami wcale nie maleje, lecz rośnie – i język C++ nadal się rozwija. Przyczyna jest prosta: nie ma on sobie równych, jeśli chodzi o szybkość działania kodu wynikowego i wszechstronne możliwości zastosowania w różnych dziedzinach. Jednak do nauki tego języka trzeba się trochę przyłożyć. Jeśli przerażają Cię opasłe tomy opisujące C/C++ w najdrobniejszych szczegółach, sięgnij po książkę, która pomoże Ci szybko nauczyć się podstaw programowania w tym języku.

W tym krótkim, treściwym podręczniku pokazano sposób konstruowania programów obliczeniowych na prostych, przejrzystych przykładach. Dzięki temu także Ty możesz szybko napisać swój pierwszy program, który ułatwi Ci pracę i odpowie na Twoje konkretne potrzeby. Dowiesz się stąd, jak używać poszczególnych elementów programowania w C/C++ jako przygotowania do pracy z gotowymi bibliotekami numerycznymi oraz jak powinna wyglądać struktura Twojego programu i jak zaimplementować w nim różne funkcje. Sprawdzisz, jak efektywnie uczyć się programowania i na co koniecznie należy zwracać uwagę. Zobacz sam!

- Szybki start
- Rodzaje wielkości w języku C/C++ i ich deklaracja
- Deklaracja tablic
- Operacje na zadeklarowanych wielkościach
- Instrukcje warunkowe i sterowanie pracą komputera
- Automatyzacja obliczeń
- Architektura programu i pierwsze programy
- Operacje wyprowadzania wyników
- Opis przykładowych programów do nauki programowania
- Wskaźniki, tablice, funkcje, struktury i liczby zespolone
- Cztery przykładowe programy

C/C++ to przepustka do programowania dla każdego!



księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
<http://helion.pl>

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>

sięgnij po **WIĘCEJ**



KOD KORZYŚCI

ISBN 978-83-283-2152-6



9 788328 321526

Informatyka w najlepszym wydaniu

cena: 24,90 zł