Język C, C++.

Krótka charakterystyka C i C++/

Języki C i C++ - uniwersalne, do celów ogólnych.

W językach tych powstają aplikacje dla różnych systemów operacyjnych, w tym m.in. dla Windows, Unix, Linux, MacOS, BeoS.

Język C został zdefiniowany w 1972 r. przez Denisa M Ritchie z Bell Laboratories w New Jersey.

W 1988 r. powstało opracowanie tzw. standardu ANSI lub "ANSI C".

Język C jest językiem ogólnego zastosowania. Język w miarę prostoty, niezbyt obszerny, szerokiego zastosowania. Charakteryzuje się także nowoczesnymi strukturami danych i bogatym zestawem operatorów. Niezbyt abstrakcyjny (wydumany). C nie jest językiem "bardzo wysokiego poziomu", nie jest nawet "duży", i nie został opracowany dla jakiejś szczególnej dziedziny zastosowań. Brak ograniczeń oraz ogólność powodują, że w wielu przypadkach jest wygodniejszy i bardziej sprawny od innych języków oprogramowania o pozornie większych możliwościach. Można zrobić w nim w miarę wszystko. Ścisły związek z UNIX - ci sami twórcy. Tysiące funkcji bibliotecznych.
Język C pierwotnie miał ułatwiać tworzenie oprogramowania systemowego - UNIX. UNIX z definicji wyposażony w kompilator C.
Język C można zakwalifikować jako język wysokiego poziomu (1:10 - źródło wynik). Można również wykonać operacje zastrzeżone dla niskiego poziomu - elementy programowania niskiego poziomu. Możliwość operowania adresami - operacje na adresach (np. wsk, *wsk, *(wsk+7)). Pozwala to wyeliminować wstawki w języku Asemblera.

Ogólna struktura programu w języku C

Cechą języka C/C+ jest możliwość budowy programu z wielu modułów. Modułem może byc każdy zbiór zawierający poprawny kod źródłowy.

Program w C zbudowany jest z funkcji.
Każda z nich może posiadać parametry oraz określony typ wartości. Aby można było wygenerować kod wynikowy programu (w DOS zbiór .EXE), w jednym i tylko w jednym z modułów musi się znaleźć funkcja **main(),** od której rozpocznie się wykonywanie programu.
Moduł zawierający funkcję main() nazywa się modułem głównym.

Najprostszy wykonywalny program C:

```
int main(void)
  {
    return 0;
  }
```

Program ten składa się tylko z bezparametrowej funkcji main, której wartość jest typu całkowitego (int). Ciało funkcji zbudowane jest z instrukcji, które powinny znaleźć się w bloku wyznaczonym nawiasami kwadratowymi. Funkcja ta zwraca za pomocą instrukcji return wartość zero.

Skompiluje się również program w postaci:
main() { }, ewentualnie w postaci void main() { } lub void main(void) { }

Ogólna struktura programu w C/C++

Nagłówek programu
#include (włączenia tekstowe)
#define stałe makroinstrukcje
Zmienne globalne
Prototypy funkcji
Funkcja **main**()
Funkcje pozostałe

W profesjonalnych programach w nagłówku powinna być podana nazwa programu, dane o autorze, prawa autorskie, przeznaczenie programu, data ostatniej aktualizacji programu, nazwa kompilatora, uwagi odnośnie kompilowania, linkowania, wykonania

Sekcja **#include** zawiera specyfikację włączanych plików bibliotecznych oraz własnych

Sekcja **#define** zawiera definicje stałych i makroinstrukcji

Następne sekcje to zmienne globalne oraz prototypy funkcji.
Muszą wystąpić przed ciałami funkcji, aby w dalszej części programu nie było odwołań do obiektów nieznanych kompilatorowi.
Jeżeli funkcje pozostałe umieszczone byłyby przed funkcją main() to specyfikowanie prototypów byłoby zbyteczne.

Włączenia tekstowe **#include** mogą w zasadzie wystąpić w dowolnym miejscu programu.
Zaleca się aby dłuższe funkcje lub grupy funkcji były umieszczane w osobnych plikach.
Istnieje wówczas włączenia tych funkcji przez inne programy.
Zaleca się parametryzację i uniwersalizację opracowywanych modułów.

Podstawowy fragment (element) programu w języku C

```
main()  /* klamra otwierająca - początek funkcji głównej
(jak begin w Pascalu)*/
{   /* tu zaczyna się treść programu - w funkcji main()*/
 /* treść */
}    /* klamra zamykająca - jak end w Pascalu*/
```

/* - rozpoczęcie komentarza, */ - koniec komentarza

W pisaniu programów rozróżniane są duże i małe litery - np. w **main()**

Język C/C++ ***nie ma instrukcji realizujących operacje we/wy*.**
Do tego celu służą funkcje bibliotek standardowych.
Użycie ich jest niezbędne, gdy trzeba pobrać dane od użytkownika i wysłać je na ekran.

Co potrzebne do programowania w C, C++

Do programowania w C potrzebne są

- komputer z SO Windows lub Linux,
- **kompilator języka C**,
- **linker** (zwykle jest z kompilatorem),
- przydatny tez **debuger**,
- **edytor tekstowy.**

Do tworzenia programów można wykorzystywać **zintegrowane środowiska programistyczne** jak:
firmy **Borland**: Turbo C, Borland C, Borland C++ **Builder**
Microsoft Visual C++,
Dev C++,
CodeBlocks
KDevelop (Linux) do KDE,
Eclipse

Do edycji można użyć dowolnego **edytora tekstowego** zapisującego pliki
w postaci czystego kodu ASCII, typu Notatnik, Edit
lub np**. Notepad ++, ConTEXT,** które pozwalają sprawdzać składnię
i podkolorowują odpowiednio tekst po wybraniu języka.

Do **kompilacji** można użyć środowiska programistycznego **Borland C++ Compiler 5.5 – program BCC32.exe.**

Kompilator ten jest darmowy, generuje 32-bitowy kod dla Windows, zawiera tylko niezbędne narzędzia uruchamiane z linii poleceń, jest prosty w instalacji i konfiguracji.

Najlepiej zainstalować w **C:\BCC55**.
W katalogu tym są podkatalogi: BIN, INCLUDE, LIB, HELP, EXAMPLES.

Pliki konfiguracyjne w katalogu C:\BCC55\BIN: **BCC32.cfg** i **ILINK32.cfg**

Zawartość plików

Plik **BCC32.cfg**

-I"c:\Bcc55\include"

-L"c:\Bcc55\lib"

lub w innej wersji

-5

-O2

-I"c:\Bcc55\include"

-L"c:\Bcc55\lib"

Plik **ILINK32.cfg** - składa się z jednej linii

-L"c:\Bcc55\lib"

Można też dodać katalog **C:\BCC55\BIN** do ścieżki wyszukiwania
– dopisać w wierszu poleceń:
PATH=%PATH%;C:\BCC55\BIN

Kompilacja programów przy pomocy BCC32.EXE:

Bcc32 program.c lub bcc32 program cpp
albo użycie np. programu bachowego np. **bcc.bat** typu:

copy %1 c:\bcc5\bin

cd c:

cd \bcc55\bin

bcc32.exe %1

Kompilator **BCC32** rozróżnia język, w którym został napisany program kompilowany na podstawie rozszerzenia pliku źródłowego.
Pliki z rozszerzeniem **C** są traktowane jako pliki klasycznego języka **C** *(strukturalne).*
Pliki z rozszerzeniem **CPP** są traktowane jako programy w języki **C++**
(strukturalne i technika obiektowo orientowana).

Pisanie programów

Pierwszy przykładowy program

Przyjęło się, że pierwszy program napisany w dowolnym języku programowania, powinien wyświetlić tekst "**Hello World**!" (Witaj Świecie!).

Język C nie ma żadnych mechanizmów przeznaczonych do wprowadzania i wypisywania danych - trzeba zatem skorzystać z odpowiadających za to funkcji - w tym przypadku ***printf()***, zawartej w standardowej bibliotece C (ang. ***C Standard Library***)

Zawartość pliku hello.c – wersja w języku C

```
#include <stdio.h>
int main (void)
{
printf ("Hello World!");
return 0;
}
```

- **preprocesor** - wstępny przetwarzacz - uwzględnia dodatkowe rzeczy
#include <stdio.h> - dołączenie nagłówków z biblioteki standardowej (ścieżka domyślnych przeszukiwań)
#include "program.h" - dołączenie programu z katalogu aktualnego - podwójne cudzysłowy

W języku C deklaracje funkcji bibliotecznych zawarte są w ***plikach nagłówkowych*** posiadających najczęściej rozszerzenie ***.h***.
Przykład obrazuje, jak przy użyciu dyrektywy **#include** umieścimy w kodzie plik standardowej biblioteki C **stdio.h** (Standard Input/Output.Headerfile) zawierającą definicję funkcji ***printf***:

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

W nawiasach trójkątnych < > umieszcza się nazwy standardowych plików nagłówkowych.
Żeby włączyć inny plik nagłówkowy (np. własny), znajdujący się w katalogu z kodem programu, trzeba go wpisać w cudzysłów: **#include "mój_plik_nagłówkowy.h"**

Mamy więc funkcję ***printf(),*** jak i wiele innych do wprowadzania i wypisywania danych,

W programie definiujemy główną funkcję **main()**, uruchamianą przy starcie programu, zawierającą właściwy kod.
Definicja funkcji zawiera, oprócz nazwy i kodu, także typ wartości zwracanej i argumentów pobierany.
Typem zwracany przez funkcję jest ***int*** (*Integer*), czyli liczba całkowita (w przypadku ***main()*** będzie to kod wyjściowy programu).
W nawiasach umieszczane są *argumenty* funkcji, tutaj zapis ***void*** oznacza ich pominięcie.

Funkcja ***main()*** jako argumenty może pobierać parametry linii poleceń, z jakimi program został uruchomiony, i pełną ścieżkę do katalogu z programem.

Kod funkcji umieszcza się w nawiasach klamrowych ***{*** i ***}***.

Wszystkie polecenia kończymy średnikiem.
return; określa wartość jaką zwróci funkcja (tu program).
Liczba zero zwracana przez funkcję ***main()*** oznacza, że program zakończył się bez błędów; błędne zakończenie często (choć nie zawsze) określane jest przez liczbę jeden.

Funkcję ***main()*** kończymy nawiasem klamrowym zamykającym.

Wersja programu w **C++**

Zawartość pliku **hello.cpp** – wersja w języku C++

```
//  hello.cpp
#include <iostream.h>
// preprocessor - dołączenie biblioteki  systemowej do funkcji cout – input ootput stream
int main()
{
        cout << "Hello World!"; // wyświetlenie napisu
        return 0;
}
```

Pierwszy własny prosty program – edycja i kompilacja przy pomocy **BCC32**

Program p1.c

Uruchamiamy edytor, np. **EDIT** z parametrem nazwy pliku, czyli EDIT p1.c
lub notatnik (zapisujemy plik jako wszystkie p1.c, bez txt)

Wpisujemy treść pliku

```
#include <stdio.h>
void main(void)
{
printf("Nazwisko Imie: Pierwszy program w C");
}
```

Uruchamiamy kompilator: **BCC32 p1.c**
Utworzony zostanie plik **p1.exe** jeśli nie było błędów w treści pliku programu.

Korzystanie ze środowiska programistycznego Borlanda,
np. **Turbo C** – program **TC.EXE**
(**TC.EXE**, zwykle w katalogu C:\TC\BIN lub C:\Turbo\Bin)

Uruchamiamy **TC.EXE**.

Otwieramy plik p1.c – File, Open – wskazujemy plik
i kompilujemy program ALT C, Build lub **Alt F9.**
Utworzony zostanie plik p1.exe

Uruchamiamy program ALT R lub **CTRL F9.**

By zobaczyć wyniki, należy przejść do ekranu użytkownika: Alt W – Menu Windows, User Screen lub bezpośrednio: **Alt F5**.

Skierowanie wyników programu do pliku (zamiast domyślnie na ekran):

Uruchamiamy program w wierszu poleceń DOS, z parametrami przekierowania wyników, czyli

Nazwa_programu > wynik
lub **Nazwa_programu >> wynik,**
(> - założenie nowego pliku, ewentualne skasowanie starego, >> - dopisanie do istniejącego) np. p1.exe > w1.txt

Wyniki można obejrzeć w notatniku lub w DOS poleceniem type wynik .lub MORE wynik

Pomoc w TURBO C

Alt H

Np. **ALT H, Index**, wpisujemy szukane słowo (np.

Instrukcje podstawowe w C

Objaśnienia:

Instrukcja ***#include*** poleca kompilatorowi dołączyć do programu plik **stdio.h** (tzw. pseudonagłówkowy),
w którym zawarta jest m.in. definicja funkcji *printf*.

Bardzo często używa się też innych plików pseudonagłówkowych, np. **conio.h**, **math.h** itp.
Wszystkie pliki ***.h** zawarte w **nawiasach <>** znajdują się w podkatalogu **INCLUDE**.

Druga linijka z main zawiera rozpoczęcie funkcji głównej programu czyli **main().**

Nagłówek **void main()** to inaczej **void main(void)**
void przed mian oznacza, że funkcja nie zwraca żadnego wyniku. (void)w nawiasie po main oznacza brak parametrów programu.

Deklaracja funkcji **main** może mieć inne formy, np.

int main() - funkcja w wyniku swojego działania zwróci wartość typu *int* i nie pobiera żadnych parametrów przy rozpoczęciu działania.
int main(void) - inny zapis, ale znaczenie takie samo jak wyżej
main() - jeszcze inny zapis, ale znaczenie takie samo jak wyżej
float main() - w tym przypadku wynikiem działania funkcji będzie typ *float*.
int main(int argc, char *argv[])

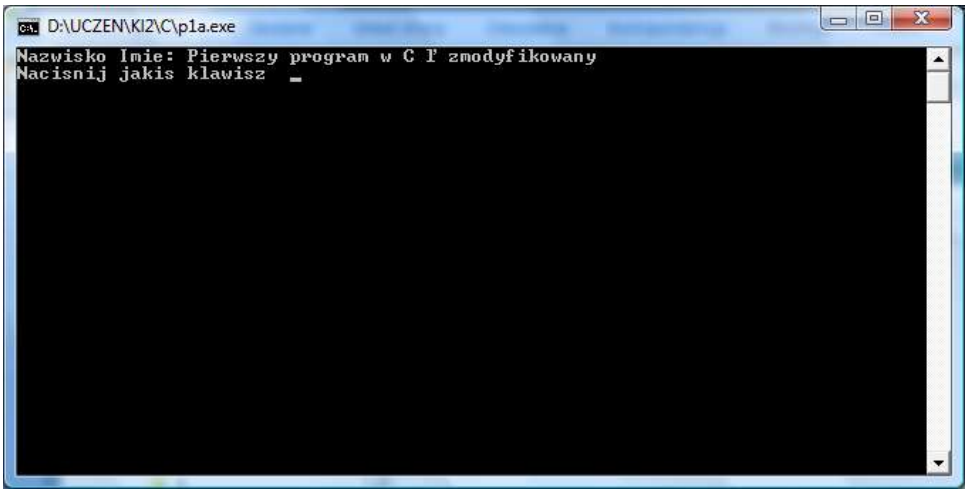
Po linii z **main()** zaczyna się nawiasem klamrowym otwierającym **{**
ciało (treść) funkcji głównej i kończy na końcu nawiasem zamykającym **}**
Wewnątrz funkcji głównej jest instrukcja **printf()**, powodująca wyświetlenie na ekranie tekstu, będącego jej parametrem (tekst w nawiasach w cudzysłowie).

Instrukcje kończymy średnikiem.
Po skompilowaniu czarny ekran i zaraz znika i dlatego trzeba dokonać modyfikacji programu.

Uruchamiamy edytor Edit lub notatnik, zapisujemy plik pod inną nazwą, np. p1a.c i modyfikujemy

```
//* program p1a.c  */
// to jest komentarz w jednej linii – używać powinno się tylko w C++
/* to jest też komentarz, może mieć kilka linii, dłuższy, ograniczony */
#include <stdio.h>  /* dołączenie pliku nagłówkowego z biblioteki systemowej
potrzebny do wywołania instrukcji printf   */
#include <conio.h>      /* dołączenie pliku conio.h do funkcji getch() i clrscr() */
int  main()       /* funkcja główna main() */
{  /*  początek main()*/
   clrscr();        /*  kasowanie ekranu */
   printf("Nazwisko Imie: Pierwszy program w C,  zmodyfikowany ");
   printf("\nNacisnij jakis klawisz  ");
   getch();          /*  czeka na jakiś klawisz */
   return 0;        /*  funkcja główna zwraca 0 przy pomyślnym wyniku zakończenia */
}          /*  koniec funkcji głównej main() */
```

Kompilacja: BCC32 p1a.c – w wyniku program skompilowany **p1a.exe**
Po uruchomieniu: p1a



W programie wprowadzono funkcje

clrscr() – kasowanie ekranu oraz
getch() – program czeka na naciśnięcie dowolnego klawisza.

Definicje tych funkcji są w pliku **conio.h**,
dlatego jest on dopisany w dyrektywie #include <**conio.h**>.

Zamiast **getch()** , można zastosować **getchar()** lub **while(!kbhit())**

\n to znak specjalny – powoduje przejście do nowej linii.
Podobnie **\t** – tabulacja, **\a** – sygnał dźwiękowy.

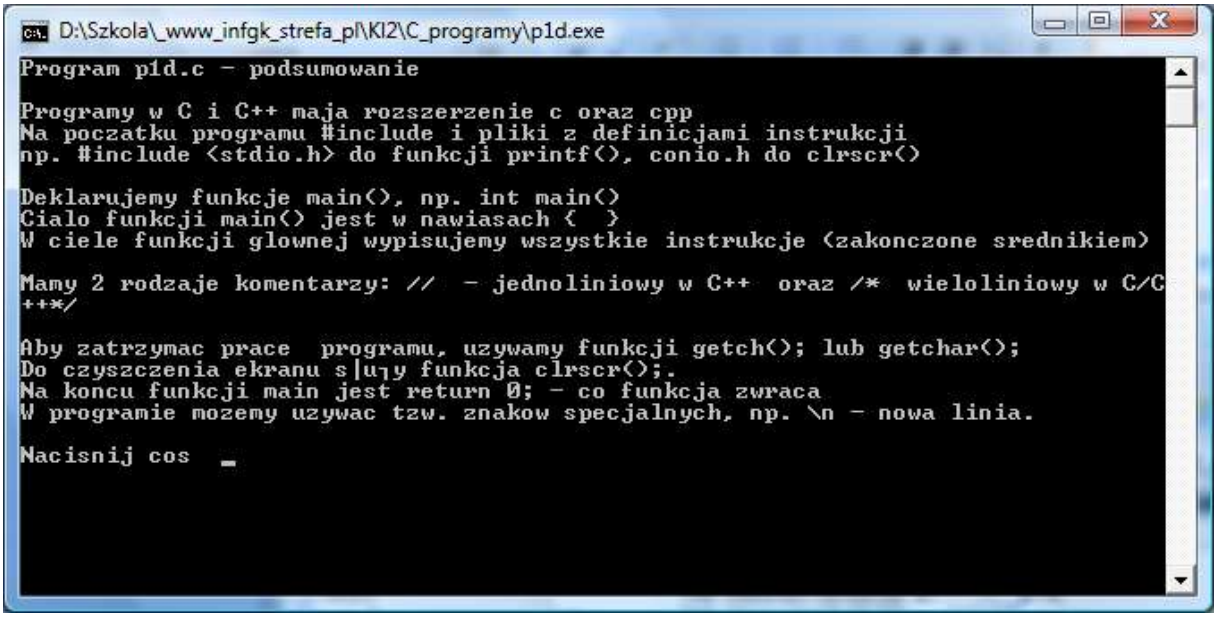
Komentarze – oznaczone // - jednolinijkowy – w języku C++
oraz oznaczone **/* komentarz */** - dłuższe, mogą zawierać wiele linii – w języku C.
Kompilator nie czyta komentarzy, stosuje się dla wyjaśnienia programu

Program p1b.c bez komentarzy

```
#include <stdio.h>
#include <conio.h>
int  main()
{
  clrscr();
  printf("Nazwisko Imie: Pierwszy program w C,  zmodyfikowany ");
  printf("\nNacisnij jakis klawisz  ");
  getch();
  return 0;
}
```

```
// Program, który podsumowuje dotychczasowy materiał
/* program p1d.c  */
#include <stdio.h>
#include <conio.h>
int  main()
{
  clrscr();
  printf("Program p1d.c - podsumowanie\n");
  printf("\nProgramy w C i C++ maja rozszerzenie c oraz cpp\n");
  printf("Na poczatku programu #include i pliki z definicjami instrukcji\n");
  printf("np. #include <stdio.h> do funkcji printf(), conio.h do clrscr()\n");
  printf("\nDeklarujemy funkcje main(), np. int main() \n");
  printf("Ciało funkcji main() jest w nawiasach {  }\n");
  printf("W ciele funkcji głównej wypisujemy wszystkie instrukcje (zakonczone  srednikiem) \n");
  printf("Mamy 2 rodzaje komentarzy: //  - jednoliniowy oraz /*  wieloliniowy */\n");
  printf("\nAby zatrzymać prace  programu, używamy funkcji getch(); lub getchar(); \n");
  printf("Do czyszczenia ekranu służy funkcja clrscr();. \n");
  printf("Na końcu funkcji main ()  jest return 0; - co funkcja zwraca\n");
  printf("W programie możemy używać tzw. znaków specjalnych, np. \\n - nowa linia. \n");
  printf("\nNacisnij cos  ");
  getchar();
  return 0;
}
```

Po skompilowaniu: bcc32 p1d.c otrzymujemy p1d.exe.
Po uruchomieniu pokaże się ekran programu:



Podsumowanie 1:

1. Mamy 2 rodzaje komentarzy i możemy w nich pisać uwagi, pomijane przez kompilator.
2. Aby zatrzymać pracę programu, używamy funkcji ***getch();***.
3. Do czyszczenia ekranu służy funkcja ***clrscr();.***
4. Na końcu funkcji **main** jest zwykle ***return 0;***.
5. W tekście możemy używać tzw. znaków specjalnych, np. przejście do następnej linii ***\n.***

Przykłady prostych programów

```
/* Program objw1.c  -  Obliczanie objętości walca   (startowy)     */
/*-------------------------------------------------------*/

#include <stdio.h>

void main()

{
   float promien, wysokosc, objetosc;

   promien  = 3.3;
   wysokosc = 44.4;
   objetosc = 3.1415926 * promien * promien * wysokosc;
   printf("Objetosc walca = %f", objetosc);
}
/*********************************************************/
```

```
/* Program daneos1.c  - dane osobowe*/
#include <stdio.h>
int main(void)
/* funkcja glowna, brak argumentow, zwraca typ calkowity int */
{
        char imie[20]; /* deklaracja tablicy znakow - łańcuch – tekst na 19 znaków
        int i;  /* deklaracja zmiennej całkowitej i
```

```
        printf("\nPodaj swoje imie ");  // wydruk napisu na ekran,\n - nowa linia
        gets(imie);   // wprowadzenie imienia
        puts("Ile masz lat? "); // wydruk napisu – funkcja puts()
        scanf("%d",&i);  // wprowadzenie lat
        printf("\n%s ma %d lat.",imie, i); // wyświetlenie imienia i lat
        return 0;  /* funkcja glowna zwraca 0 - pomyślny koniec */
}
```

Uwaga! Po uruchomieniu Alt F5 – ekran wyników.

Obliczenie pola prostokąta – 3 programy

```
/* ---------- polepros1.c  - boki a i b w programie ---------- */
void main(void)
{
float a, b;    // deklarujemy zmienne przechowujące boki prostokąta
  float pole; // deklarujemy zmienną zawierającą wynik obliczeń

  a = 5; b =10;    // przypisujemy im wartości
  pole = a * b; // obliczamy pole prostokąta (tu równe 50)
  printf("Pole = %f",pole);
  getch();
}
```

```
/*  ----------polepros2.c  - z funkcja  ----------/*
#include <stdio.h>
#include <conio.h>

float PoleProstokata(float bok1, float bok2)
{
  // w tym miejscu bok1 jest równy 5,
  // natomiast b jest równe 10

  float wynik;

  wynik = bok1 * bok2;
  return wynik;
}

void main(void)
{
  float a, b, p;

  a = 5; b = 10;
  p= PoleProstokata(a, b);
  printf("Pole prostokat o bokach %f i %f = %f ", a, b, p);
  getch();
}
```

```
/*------------------------------------------------------------------------*/
/* ppros3.c - z funkcja */
#include <stdio.h>
#include <conio.h>

 float PoleProstokata(float bok1, float bok2); /* zapowiedz funkcji */

/* funkcja glowna */
int main(void)
{
  float a, b, p;
  puts("Obliczenie pola prostokata o bokach a i b ");
  printf("Podaj a i b oddzielone spacja: ");
  scanf("%f %f",&a, &b);
  /* wywolanie funkcji na obliczenie pola */
  p= PoleProstokata(a, b);
  printf("Pole prostokat o bokach %f i %f = %f ", a, b, p);
  getch();
  return 0;
}

float PoleProstokata(float bok1, float bok2)
{
  // w tym miejscu bok1 jest równy a,
  // natomiast bok2 jest równy b
  float wynik;
  wynik = bok1 * bok2;
  return wynik;
}



* ----------ppros4.c - z funkcja i wprowadzaniem danych ----------*/
#include <stdio.h>
#include <conio.h>

float PoleProstokata(float bok1, float bok2); /* zapowiedz funkcji */

/* funckja glowna */
int main()
{
  float a, b, p;
  puts("Obliczenie pola prostokata o bokach a i b ");
  printf("Wprowadz a   ");
```

```
  scanf("%f", &a);  /* wprowadzenia a */
  printf("Wprowadz b   ");
  scanf("%f", &b); /* wprowadzenia b */
  p= PoleProstokata(a, b); /* wywolanie funkcji z parametrami a i b */
  printf("Pole prostokat o bokach %f i %f = %10.2f \n", a, b, p);
  printf("\nNacisnij cos ");
  getch(); /* czeka na znak */
  return 0;
}

/* Funkcja - definicja */
float PoleProstokata(float bok1, float bok2)
{
  /* w tym miejscu bok1 jest równy a,
   natomiast bok2 jest równy b */
  float wynik;
  wynik = bok1 * bok2;
  return wynik;
}
```

```
// Program czas.cpp w jezyku C++ - wyświetlenie czasu – komentarz w C++
#include <iostream.h>
  #include <time.h>
  main()
  {
   int a;
   time_t czas;
   do {
    cout << "1 - Wyswietl aktualny czas" << endl;
    cout << "2 - Zakoncz program" << endl;
    cout << "Twoj wybor?";
    cin >> a;
    if (a == 1)
    {
     time(&czas);
     cout << ctime(&czas);
    }
   }while (a != 2);
   cout << "Do zobaczenia" << endl;
  }
```

Podstawowe elementy języka C

(każdego praktycznie języka programowania)

- **zestaw znaków**
- **nazwy i słowa zastrzeżone**
- **typy danych**
- **stałe**
- **zmienne i tablice**
- **deklaracje**
- **wyrażenia**
- **instrukcje**

Zestaw znaków C:

- litery małe i duże języka łacińskiego (angielskiego)
- cyfry 0..9
- znaki specjalne: ! * + \ " < # (= | { > %) ~ ; } / ^ - [: , ? & _] ' . oraz znak odstępu (spacja)

Nazwy i słowa zastrzeżone (kluczowe, zarezerwowane)

Nazwy służą do identyfikowania elementów programu (stałych, zmiennych, funkcji, typów danych, itd.). Nazwa składa się z ciągu liter i cyfr, z tym, że pierwszym znakiem musi byc litera. Znak podkreślenia traktowany jest jako litera.

W języku C rozróżniane są duże i małe litery w identyfikatorach.

Przykład użycia różnych nazw:

```
/*********************************************************/
/*  Program  objwalc.c                                   */
/*  Obliczanie objetosci walca (rozroznianie nazwy /1)*/
/*-------------------------------------------------------*/

#include <stdio.h>

main()

{
   float promien, wysokosc, objetosc;
   float PROMIEN, WYSOKOSC, OBJETOSC;

   promien  = 3.3;    PROMIEN  = 10.;
   wysokosc = 44.4;   WYSOKOSC = 20.;

   objetosc = 3.1415926 * promien * promien * wysokosc;
   printf("\nObjetosc walca = %f", objetosc);

   OBJETOSC = 3.1415926 * PROMIEN * PROMIEN * WYSOKOSC;
   printf("\nOBJETOSC WALCA = %f", OBJETOSC);
}
/*********************************************************/
```

Użycie odstępu w nazwie jest niedozwolone. Niektóre implementacje rozpoznają w nazwie do 8 znaków, inne więcej (do 32)

Słowa zastrzeżone

Są to słowa o szczególnym znaczeniu dla języka, których nie wolno używać programiście np. jako nazw zmiennych.

Standardowy zestaw znaków zastrzeżonych języka C:

auto	break	case	char	const	continue	default	do	double	else	enum	extern	float	for
goto	if	int	long	register	return	short	signed	sizeof	static	struct	switch	typedef	union
unsigned	void	volatile	while										

Niektóre kompilatory mają niektóre lub cześć z następujących słów kluczowych

ada	asm	entry	far	fortran	huge	near	pascal

Niektóre kompilatory mogą mieć też rozpoznawać inne słowa zastrzeżone.

Podstawowe typy danych i rozmiary danych

W języku C występuje tylko kilka podstawowych typów danych:

- char - jeden bajt, zdolny pomieścić 1 znak
- int - typ całkowity
- float - typ zmiennopozycyjne pojedynczej precyzji
- double - typ zmiennopozycyjny podwójnej precyzji

Dodatkowo występuje kilka kwalifikatorów stosowanych z tymi podstawowymi typami. Kwalifikatory **short** i **long** odnoszą się do obiektów całkowitych

Typ	Opis	Zakres wartości	Reprezentacja
char (signed char)	pojedynczy znak (np. litera)	-128...127	1 bajt
unsigned char	znak (bez znaku - dodatni)	0...255	1 bajt
short	liczba całkowita krótka	-32768...32767	2 bajty
unsigned short	liczba krótka bez znaku	0...65535	2 bajty
int (signed int)	liczba całkowita	-32768...32767	2 bajty
unsigned int	całkowita bez	0...65535	2 bajty

	znaku		
long (long signed int)	liczba całkowita długa	-2147483648...2147483647	4 bajty
unsigned long (long unsigned int)	j.w. bez znaku	0...4294967295	4 bajty
float	l. rzeczywista	-3.4E-38..-3.4E-38,0,3.4E-38..3..3.4E38	4 bajty
double	l. rzeczywista podwójna	-1.7E308..-1.7E-308,0,1.7E-308..1.7E308	8 bajtów
long double	l. rzecz. długa podwójna	-1E4932..-3.4E-4932,0,3.4E-4932..1.1E4932	10 bajtów

Szczegóły zależne są od konkretnej implementacji kompilatora języka C.

Są jeszcze inne typy danych jak: **void** (typ funkcji nie zwracającej wartosci), **enum** (typ wyliczeniowy) oraz **typ wskaźnikowy**. Typ wyliczeniowy (enum) reprezentowany jest na 2 bajtach, wskaźnikowy na 2 lub 4 (w zależności od modelu pamięci), void nie jest reprezentowany.

TYPY, OPERATORY, WYRAŻENIA

Zmienne i stałe są podstawowymi obiektami danych, jakimi posługuje się program.
Deklaracje wprowadzają potrzebne zmienne oraz ustalają ich typy i ewentualnie wartości początkowe.
Operatory określają co należy z nimi zrobić.
Wyrażenia wiążą zmienne i stałe, tworząc nowe wartości.

Stałe

W C występują *4 rodzaje stałych*: stałe całkowitoliczbowe, stałe rzeczywiste, stałe znakowe oraz łańcuchy znaków. Wartosć stałej numerycznej nie może wykraczać poza dopuszczalne granice.

Stałe całkowitoliczbowe

Nazwa	Dozwolony zestaw znaków	Uwagi	Przykłady
Stałe dziesiętne	0..9 + -	Jeśli więcej niż 1 znak, pierwszym nie może być 0	0 1 876 -122 +909
Stałe ósemkowe	0..7 + -	Pierwszą cyfrą musi być 0	0 0111 0777 -0777 +0222
Stałe szesnastkowe	0..9 a..f A..F + -	Pierwszymi znakami muszą być 0x lub 0X	0x 0X1234 0XAFDEC 0xffff

W celu zainicjowania zmiennych typów **int** i **long** po znaku równości podajemy całkowitą stałą liczbową. Przykłady:

```
int l=100;
unsigned k=121;

int la = 0x2ffa;
long i = 25L; /* long */
long unsigned z = 1000lu; /* lub 1000UL */
```

Stała **całkowita**, jak np. 1234 jest obiektem typu **int**.
W stałej typu **long** na końcu występuje litera **l** lub **L**, jak 123456789L.
Stała całkowita nie mieszcząca się w zakresie int jest traktowana jako stała typu **long**.
W stałych typu **unsigned** na końcu występuje u lub **U**,
a końcówka ul lub **UL** oznacza stałą typu **unsigned long**.

Stałe rzeczywiste, zwane **zmiennoprzecinkowymi** reprezentują liczby dziesiętne.
Dozwolony zestaw znaków: **0..9 . + - E e** (E lub e reprezentuje podstawę systemu tj. 10)
*Uwagi: 1.2*10^-3 można zapisać 1.2E-3 lub 1.2e-3.*

Stałe zmiennopozycyjne zawierają albo kropkę dziesiętną (np. 123.4),
albo wykładnik **e** (np. 1e-2) albo jedno i drugie.
Typem stałej zmiennopozycyjnej jest double, chyba, że końcówka stanowi inaczej.
Występująca na końcu litera f lub **F** oznacza obiekt typu **float**,
a litera l lub **L** - typu **long double**.

Przykłady: 0. 2. 0.2 876.543 13.13E13 2.4e-5 2e8

Deklaracja i inicjalizacja zmiennych rzeczywistych - przykłady::

```
float s=123.16e10; /* 123.16*10^16 */
double x=10.;
long double x=.12398;
double xy=-123.45;
```

Stałe znakowe

Stała znakowa jest liczbą całkowitą; taką stałą tworzy jeden znak ujęty w apostrofy, np. 'x'.
Są to więc pojedyncze znaki zamknięte dwoma apostrofami.
Zestaw dowolnych widocznych znaków ASCII.
Wartością jest wartość kodu znaku w maszynowym zbiorze znaków.
Np. wartością stałej '0' jest liczba 48 - liczba nie mająca nic wspólnego z numeryczną wartością 0.

Pewne znaki niegraficzne mogą być reprezentowane w stałych znakowych i napisowych przez sekwencje specjalne, takie jak **\n** (znak nowego wiersza).

Przykłady: 'A' '#' ' '

```
char a='a';
```

Znaki z zestawu ASCII o kodach **0 do 31** są znakami sterującymi, niewidocznymi na wydrukach.
Znaki o kodach **0...127** są jednakowe dla wszystkich komputerów bazujących na kodzie ASCII.
Znaki o kodach **128...255** (kod rozszerzony ASCII) mogą się różnić na różnych komputerach.

Escape-sekwencje - sekwencje specjalne

Niektóre znaki "niedrukowalne" mogą być przedstawione w postaci tzw. escape-sekwencji, np. znak nowej linii jako sekwencja \n.
Pierwszym znakiem tej sekwencji jest **backslash** (\).
Sekwencja specjalna wygląda jak 2 znaki, ale reprezentuje tylko jeden znak

Sekwencja znaków	Wartość ASCII	Znaczenie
\a	7	Sygnał dźwiękowy (BEL)
\b	8	Cofniecie o 1 znak (BS)
\t	9	Tabulacja pozioma (HT)
\v	11	Tabulacja pionowa (VT)
\n	10	Nowa lina (LF)
\f	12	Nowa strona (FF)
\r	13	Powrót karetki (CR)
\"	34	Cudzysłów
\'	39	Apostrof
\?	63	Znak zapytania
\\	92	Backslash
\0	0	Znak pusty (null) - nie jest równoważne stałej znakowej '0'

Dowolny znak przedstawiony w postaci escape-sekwencji może być podany jako kod liczbowy ósemkowy lub szesnastkowy.

Np. litera K, wartość 10-na ASCII = 75 (wzorzec bitowy dla 1 bajtu = 01001011) ma odpowiednie escape-sekwencje:
\113 jako liczba ósemkowa 01 001 011
\x4B jako liczba szesnastkowa 0100 1011.

Przykłady tak zapisanych stałych znakowych: '\101', '\7', '\x20', '\xAB', '\x2c'
```
char lf='\n';
```

Ogólny format takiego zapisu:

- \ooo - dla systemu ósemkowego (o - cyfra 8-wa)
- \xhh - dla systemu szesnastkowego (h - cyfra 16-wa)

Stałe napisowe - napisy, łańcuchy znaków (stałe łańcuchowe)

Stała napisowa lub napis jest ciągiem złożonym z zera lub więcej znaków, zawartym między znakami cudzysłowu, np. "Jestem napisem".
Stała łańcuchowa składa się z ciągu o dowolnej liczbie znaków.
Ciąg ten musi być ograniczony znakami cudzysłowu.
Przykłady:

```
"Wynik = "
" + 2 mln $"
"Linia nr 1\nLinia nr2"
""
"A + B =  "
```

Łańcuchy mogą zawierać escape-sekwencje.
Łańcuchem pustym są same cudzysłowy.
Łańcuch w sposób niejawny jest zakończony znakiem nul czyli **\0**.
Dlatego np. stała znakowa 'K' nie jest równoważna łańcuchowi "K".

Łańcuch znaków (napis) można traktować jako tablicę złożoną ze znaków, uzupełnioną na końcu znakiem **'\0'**
Taka reprezentacja oznacza, że praktycznie nie ma ograniczenia dotyczącego długości tekstów. Programy muszą jednak badać cały tekst, by określić jego długość, np. **strlen(s)** ze standardowej biblioteki.

Przykład:
Napis "Katowice" , który może być zadeklarowany jako tablica jednowymiarowa, której elementami są znaki, np.
char napis[] = "Katowice"

Nr elementu	1	2	3	4	5	6	7	8	9
Napis	**K**	**a**	**t**	**o**	**w**	**i**	**c**	**e**	**\0**
Wartość indeksu	0	1	2	3	4	5	6	7	8

Deklaracja i inicjalizacja łańcuchów - przykłady:

```
char s[10]="\n\fAndrzej\x60";
char str[20]={'\n','\f', 'A', 'n', 'd', 'r', 'z', 'e', 'j', '\x60', '\0'};
char *str1 = "Andrzej Zalewski autor podręcznika" "Programowanie w języku C/C++"
```

Stałe wyliczeniowe (enumeration constant)

Stałe wyliczeniowe tworzą zbiór stałych o określonym zakresie wartości.
Wyliczenie jest listą wartości całkowitych, np.
enum boolean {NO, YES};
Pierwsza nazwa na liście wyliczenia ma wartość 0, następna 1 itd., chyba że nastąpi jawnie podana wartość.

Przykłady:

```
enum KOLOR {CZERWONY, NIEBIESKI, ZIELONY, BIAŁY, CZARNY}
```

KOLOR staje się nazwą wyliczenia, powstaje nowy typ.
Do CZERWONY zostaje przypisana wartość 0, do niebieski 1, zielony 2 itd.
Każda stała wyliczeniowa ma wartość całkowitą.
Jeśli specjalnie się nie określi, to pierwsza stała przyjmie wartość 0, druga 1 itd.

```
enum KOLOR {red=100, blue, green=500, white, black=700};
```

red przyjmie wartość 100, blue 101, green 500, white 501, black 700

```
enum escapes{BELL='\a', BACKSPACE='\b', TAB='\t', NEWLINE='\n', VTAB='\v',
RETURN='\r'};
enum months {JAN=1, FEB, MAR, APR , MAY, JUL, AUG, SEP, OCT, NOV, DEC};
```

Stałe symboliczne - makrodefinicje

Stała symboliczna jest nazwą przedstawiającą inną stałą - numeryczną, znakową lub tekstową.
Definicję stałej symbolicznej umożliwia instrukcja **#define**:

#define NAZWA tekst

gdzie NAZWA jest nazwą stałej symbolicznej, a tekst jest związanym z tą nazwą łańcuchem znaków

Przykłady:

Makrodefinicje proste:

#define identyfikator <ciąg-jednostek-leksykalnych>

```
#define PI 3.14159
#define TRUE 1
#define FALSE 0
#define NAPIS1 Siemianowice
#define IMIE "Andrzej"
// (puts(IMIE) rozwija w tekst puts("Andrzej")
#define IMIE_I_NAZWISKO IMIE+"Zalewski"
#define WCZYTAJ_IOSTREAM_H #include <iostream.h>
```

Makrodefinicje parametryczne

```
#define identyfikator(idPar1, idPar2,...) ciąg_jedn_leksykalnych
```

```
Np.
#define ILORAZ(a,b) ((a)/(b))
//- makrodefinicja ILORAZ - parametry w nawiasach!
#define SUMA(a,b) ((a)+(b))
```

W trakcie kompilacji nazwy stałych symbolicznych są zastąpione przez odpowiadające im łańcuchy znaków.
Ułatwia to parametryzację programu, a także umożliwia zastępowanie często niewygodnych w pisaniu sekwencji programu, tworzenie makrodefinicji, tworzenie własnych dialektów języka, czy nawet języków bezkompilatorowych
(na bazie kompilatora C).

Przykład z walcem, z zastosowaniem pseudoinstrukcji **#define**

```
/*******************************************************/
/*  Program  p8.c                                      */
/*  Obliczanie objetosci walca        ( #define )     */
/*-----------------------------------------------------*/

#include <stdio.h>

#define PI         3.1415926
#define PROMIEN    3.3
#define WYSOKOSC  44.4
#define WYNIK     printf("Objetosc walca = %f", objetosc)

main()

{
   float promien, wysokosc, objetosc;

   promien  = PROMIEN;
   wysokosc = WYSOKOSC;
   objetosc = PI * promien * promien * wysokosc;
   WYNIK;
}
/*******************************************************/
```

Wynik programu:
Objetosc walca = 1519.010254

Dzięki **#define** program jest bardziej czytelny. Stała PI może być użyta wielokrotnie, wartości danych są widoczne na początku.

Użycie dużych liter jako nazwy stałej symbolicznej nie jest obowiązkowe ale zalecane.

Zmienne

Zmienną nazywamy nazwę (identyfikator) reprezentującą określony typ danych.
Zmienna jest to pewien fragment pamięci o ustalonym rozmiarze, który posiada własny identyfikator (nazwę) oraz może przechowywać pewną wartość, zależną od typu zmiennej.

W chwili rozpoczęcia pracy programu zmienna powinna posiadać nadaną wartość początkową (nie powinna być przypadkowa).
W trakcie pracy programu wartości zmiennych ulegają zwykle zmianom.
Należy przewidzieć zakres zmienności tych zmiennych.

W języku C wszystkie zmienne muszą być zadeklarowane przed ich użyciem, zwykle na początku funkcji przed pierwszą wykonywaną instrukcją.

Deklaracja zapowiada właściwości zmiennych.
Deklaracja składa się ona z nazwy **typu** i **listy zmiennych**, jak np.
int fahr, celsius;

W języku C oprócz podstawowych typów: **char, short, int, long, float, double** występują także **tablice, struktury, unie, wskaźniki** do tych obiektów oraz **funkcje** zwracające ich wartości.

W przypadku zmiennych nalezy zwrócić uwagę na 2 zasadnicze rzeczy:

- nadanie wartości początkowej
- oszacowanie zakresu zmienności

```
/*******************************************************/
/*  Program  p9.c                                      */
/*  Obliczanie objetosci walca     ( zmienne )         */
/*-----------------------------------------------------*/

#include <stdio.h>

#define PI        3.1415926
#define PROMIEN   3.3
#define WYNIK    printf("Objetosc walca = %f", objetosc)

main()

{
   float promien, wysokosc, objetosc;
   int i;

   promien  = PROMIEN;
   objetosc = PI * promien * promien * wysokosc;   /* uwaga */
   WYNIK;
   i=40000;                                        /* uwaga */
   printf("\ni = %i ", i);
}
/*******************************************************/
```

Efekt wykonania:

```
Objetosc walca = 0.000000
i = -25536
Objetosc walca = 0.000000
i = -25536
```

Objętość walca jest równa 0, bo nie została zainicjalizowana zmienna 'wysokosc' - przez domniemanie kompilator przyjął 0. Na wydruku i widać, że nastąpiło przekroczenie dopuszczalnego zakresu zmiennej 'i' typu int.

Zmienne mogą być **automatyczne** oraz zmienne **zewnętrzne**.

Zmienne automatyczne pojawiają się i znikają razem z wywołaniem funkcji.
Zmienne zewnętrzne to zmienne **globalne**, dostępne przez nazwę w dowolnej funkcji programu.

Zmienna zewnętrzna musi być zdefiniowana dokładnie jeden raz na zewnątrz wszystkich funkcji - definicja przydziela jej pamięć.
Taka zmienna musi być też zadeklarowana w każdej funkcji, która chce mieć do niej dostęp.
Można to zrobić albo przez jawną deklarację extern, albo niejawnie przez kontekst.
Jeśli definicja zmiennej zewnętrznej pojawia się w pliku źródłowym przed użyciem zmiennej w konkretnej funkcji, to nie ma potrzeby umieszczania w tej funkcji deklaracji extern.

Deklaracje

Deklaracje umożliwiają wyspecyfikowanie grup zmiennych określonego typu.

Większość kompilatorów dopuszcza nadanie wartości początkowej zmiennej w deklaracji (inicjalizację).

Wszystkie zmienne muszą być zadeklarowane przed użyciem.
W deklaracji określa się typ, a następnie wymienia jedną lub kilka zmiennych tego typu, np.

```
int lower, upper;
char c, line[1000];
```

W **deklaracjach** można nadawać zmiennym wartości początkowe, np.

```
char esc='\';
int i=0;
float eps=1.0e-5;
int limit=MAXLINE+1;
```

Jeśli zmienna nie jest automatyczna, to jej wartość początkową nadaje się tylko raz - jakby przed rozpoczęciem programu; jej inicjatorem musi być wyrażenie stałe.
Zmiennym automatycznym jawnie określone wartości początkowe nadaje się za każdym razem przy wywołaniu funkcji lub przy wejściu do zawierającego je bloku.
Zmiennym zewnętrznym i statycznym przez domniemanie nadaje się wartość pocz. zero.
Zmienne automatyczne bez jawnie określonej wartości pocz. mają wartości przypadkowe (śmiecie).

Kwalifikator **const** mówi, że wartość zmiennej będzie stała.

```
const double e=2.71828182;
const char mas[]="Uwaga:";
```

Deklarację **const** można stosować również do tablicowych paramentów funkcji, by wskazać, że funkcja nie może zmieniać tablicy:

int strlen(const char[]);

Przykład programu – deklaracje i działania podstawowe

```
// Program deklar1.cpp
void main(void)
{
  // Deklaracja używanych zmiennych
  int   a, b, c;
/* Deklaracja trzech zmiennych typu int (całkowita).
Można zadeklarować kilka zmiennych tego samego typu w jednej linijce.
Wystarczy je rozdzielić przecinkiem.*/
 float r = 5.3;
/* Deklaracja zmiennej rzeczywistej typu float z przypisaniem wartości
początkowej */
/* --------------------- właściwy kod programu */
 a = 5; b = 3; // Przypisujemy zmiennym a i b wartości
 c = a + b;
/* Dodanie zmiennych a oraz b i wpisanie wyniku do zmiennej c.
 Zmienna c jest teraz równa 8. */
  c = a - b;
/* Odjęcie zmiennej b od a i wpisanie wyniku do zmiennej c. Zmienna c jest
teraz równa 2. */
  c = a * b;
/* Pomnożenie zmiennej a przez b i wpisanie wyniku do c.
Zmienna c jest teraz równa 15; */
  c++;       // Zwiększenie zmiennej c o 1. Teraz jest ona równa 16.
  ++c;      // Zwiększenie zmiennej c o 1. Teraz jest ona równa 17.
  --c;         // Zmniejszenie zmiennej c o 1. Teraz jest ona równa 16.
  c--;         // Zmniejszenie zmiennej c o 1. Teraz jest ona równa 15.
  c = a % b;
/* Wpisanie do c reszty z dzielenia a przez b. Zmienna c jest równa 2. */
  r = a / b;
/* Podzielenie zmiennej a przez b i wpisanie wyniku do r.
Zmienna r jest teraz równa 1. */
  r = a;
// Przypisanie wartości zmiennej a do zmiennej r. Teraz r jest  równe 5.
  r = r / b;
// Podzielenie zmiennej r przez b. Teraz r jest równe 1.666667
  /* ======= nowy blok programu ========== */
    {
     // Deklaracja używanych w bloku zmiennych
        int x=5;
 /*Deklaracja zmiennej lokalnej dla tego bloku zmienna typu całkowitego.
  Zmiennej nie możemy wykorzystywać poza obrębem tego bloku */
    int     r=7;
/* Deklarujemy wewnątrz tego bloku zmienną lokalną o takiej samej nazwie
jak zmienna występująca bloku nadrzędnym, jednak o innym typie (wcześniej
był to float). */
    // ------- Kod bloku -----------
    x += r;
/* Dodajemy do zmiennej x wartość zmiennej r. Teraz zmienna x równa 12.  W
przypadku, gdy zmienna lokalna ma taką samą nazwę jak zmienna występująca w
bloku nadrzędnym używana jest zmienna lokalna. */
    x += a;
 /*Dodajemy do zmiennej x wartość zmiennej a. Teraz zmienna x jest równa
17. Wewnątrz tego bloku możemy używać zmiennych należących zarówno do tego
bloku programu, jak i zmiennych zadeklarowanych w blokach nadrzędnych (tu
zmiennej a) */
  }
}
```

Wyrażenia

Wszystko co zwraca wartość jest wyrażeniem.
Wyrażeniem może być samodzielny element, np. liczba, stała, zmienna.
Może to być też kombinacja w/w elementów połączonych znakami operatorów,
np. arytmetycznych lub logicznych.

Przykłady:

- 3.2
- PI
- a=a+b;
- x=y;
- x<LAFA;
- k == m;
- y = x = a+b;
- x != y
- b[i-2] = a[j+1];

Instrukcje

Instrukcje są to fragmenty tekstu programu (zwykle linie), które powodują jakąś czynność komputera w trakcie wykonywania programu.

Instrukcje można podzielić na grupy:

Instrukcje obliczające wartości wyrażeń,
np. a=b+c;

Każda taka instrukcja musi być zakończona średnikiem

Instrukcje grupujące (złożone) - ograniczone klamrami { },
np.

```
{
  a=3;
  b=2.2;
  p=a*b;
  printf("Pole=%f",p);
}
```

Instrukcje sterujące, np. while, if...

Instrukcje wywołania funkcji
Np.

```
int funkcja_d(int a, int b, float c);
int k;
k=funkcja_d(5, 6, 17.33);
```

Przykład programu z użyciem zmiennych - prosty kalkulator.

Program, który poprosi nas o wpisanie dwóch liczb i wykona na nich podstawowe działania matematyczne: zsumuje je, odejmie, pomnoży i podzieli.

Wersja programu do wczytania 2 liczb a i b typu całkowitego:

```
//Program kalk1a.c
// Wczytuje 2 liczby całkowite I wyświetla na ekranie
#include <stdio.h>
#include <conio.h>
Int main ()
{
        int a,b; //deklaracja zmiennych a i b
        clrscr();
        printf("Podaj liczbe a: ");
        scanf("%d",&a); // wczytanie liczby a
        printf("Podaj liczbe b: ");
        scanf("%d",&b);  // wczytanie liczby b
        printf("Wpisane liczby a i b : %d, %d.\n",a,b);
        printf("\nNacisnij Enter ");
        getch(); // czekanie na Enter
        return 0 ;
}
```

Typ *int* (integer) oznacza, że zmienna może przyjmować wartości całkowite z zakresu **+/- 32767**. Jeżeli zamierzamy używać w programie dużych liczb wtedy zamiast typu *int* używamy typu *long*. Kompilator już wie, że w kodzie programu będą pojawiać się zmienne *a* i *b* i wie ile pamięci musi zarezerwować na ich przechowanie.

Następnie wypisujemy na ekranie komunikat, który prosi nas o wpisanie liczby.
Program zatrzyma się w tym miejscu i będzie czekał dotąd, aż napiszemy liczbę i zatwierdzimy je ENTERem.
Funkcja ***scanf()*** służy do odczytu wprowadzonych danych z klawiatury
Wzorzec konwersji określa typ zmiennej, którą wpiszemy z klawiatury lub wypiszemy na ekranie.

Poniższa tabela przedstawia deklaracje zmiennych określonego typu, odpowiednie dla nich wzorce konwersji oraz przykłady ich użycia:

Typ zmiennej	**Deklaracja**	**Wzorzec**	**Postać funkcji *scanf()***	**Postać funkcji *printf()***
liczba **całkowita**	int A	**%d**	*scanf("%d",&A);*	*printf("Liczba A wynosi: %d",A);*
liczba **rzeczywista**	float B	**%f**	*scanf("%f",&B);*	*printf("Liczba B wynosi: %f",B);*
ciąg znaków	char *C	**%s**	*scanf("%s",&C);*	*printf("Łańcuch ma postać: %s",C);*
pojedynczy znak	char D	**%c**	*scanf("%c",&D);*	*printf("Znak D to: %c",D);*

Program zmodyfikowany, wykonujący podstawowe obliczenia matematyczne

```
/* Program kalk1.c */
#include <stdio.h>
#include <conio.h>
int main() /* funkcja glowna */
{
int a,b; /* deklaracja zmiennych całkowitych a i b */
int suma,roznica,iloczyn;
float iloraz; /* deklaracja zmiennej rzeczywistej */
clrscr();  /* kasowanie ekranu */
```

```
printf("Prosty kalkulator\n");
printf("\nPodaj liczbe a: ");
scanf("%d",&a);  /* wczytanie liczby a */
printf("Podaj liczbe b: ");
scanf("%d",&b); /* wczytanie liczby b */
/* obliczenia */
suma=a+b;
roznica=a-b;
iloczyn=a*b;
iloraz=(float)a/(float)b; /* operator rzutowania w dzieleniu */
/* Wydruk wynikow */
printf("\nWyniki dzialan:\n");
printf("\nSuma:    %d  ",suma);
printf("\nRoznica: %d  ",roznica);
printf("\nIloczyn: %d  ",iloczyn);
printf("\nIloraz:  %f  ",iloraz);

getch(); /* czeka na naciśniecie klawisza */
return 0 ;
}
```

Zmienne **typów liczbowych** mogą przyjmować wartości tylko z określonych przedziałów liczbowych.

typ zmiennej	zakres
int	-32 768 do 32 767
long	-2 147 483 648 do 2 147 483 647
float	3,4* 10^(-38) do 3,4*10^(38)
double	1,7* 10^(-308) do 1,7*10^(308)
long double	3,4* 10^(-4932) do 3,4*10^(4932)

Podsumownie 2:

1. Aby wczytać liczbę należy użyć funkcji ***scanf*** w postaci:
   scanf("wzorzec",&zmienna);

2. Aby wypisać wczytaną w ten sposób liczbę należy użyć funkcji ***printf***,
   która służy do wypisywania komunikatów.
   Postać funkcji: ***printf("Komunikat wzorzec",zmienna);***

3. W funkcji ***scanf*** zawsze przed nazwą zmiennej używamy znaku ***&***,
   a nie robimy tego przy używaniu funkcji ***printf***.

4. Zmienna służy do przechowania danych, których wartość ustala się w trakcie działania programu i może być zmieniana.

5. Każda zmienna musi być zadeklarowana przed jej użyciem jako zmienna odpowiedniego typu: ***int***, ***float*** itp.

6. Do wypisywania komunikatów służy funkcja ***printf***, a do wczytywania zmiennych funkcja ***scanf***.

7. Do poprawnego użycia obu funkcji należy znać podstawowe wzorce konwersji: ***%d, %f, %s***.

Operacje na znakach i łańcuchach znaków

Typy znakowe: - deklaracja char zmienna.
pojedynczy znak: char znak;
łańcuch znaków (napis, słowo) – char *napis lub char napis[n];

Wzorzec konwersji przy wyświetleniu lub wczytywaniu zmiennej typu *char* to
%c dla pojedynczego znaku (łańcucha jednoznakowego)
%s dla łańcucha dłuższego niż 1 znak

Przykład:

```
/* znaki1.c */
#include <stdio.h>
#include <conio.h>
#include <string.h>
main (void)
{
char znak1,znak2,znak3; //deklaracja znaków
char *slowo1; // *slowo2;
char slowo2[20];
clrscr();  // czyszczenie ekranu
znak1='a';
znak2=102; // znak w postaci kodu dziesiętnego ASCII – litera f
slowo1="Imie"; // wpisujemy własne imię

printf("Podaj znak3: "); scanf("%c",&znak3); // podajemy jakiś znak

printf("\nPodaj slowo2: "); scanf("%s",slowo2); // wpisujemy słowo

printf("\nZmienne zawieraja znaki: ");
printf("znak1 (a), znak2 (102), znak3 (wprowadzony): %c %c %c
",znak1,znak2,znak3);
printf("\noraz slowa: ");
printf("slowo1, slowo2: %s %s  ",slowo1, slowo2);

getch();
return 0 ;
}
```

Przypisać wartość zmiennej jednoznakowej możemy na kilka sposobów:

- w apostrofach: *zmienna='a';*
- poprzez przypisanie kodu znaku: *zmienna=97;*
- poprzez wczytanie znaku z klawiatury funkcją *scanf()*: *scanf("%c",&zmienna);*

Przypisanie wartości do zmiennej dla łańcucha dłuższego niż jeden znak odbywa się podobnie jak w pierwszym przypadku, z tą jednak różnicą, że zamiast apostrofów (') należy używać cudzysłowia (

Program, który podaje kod wciśniętego przez nas klawisza:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main (void)
{
char znak;
int kod;
clrscr();

printf("Wciśnij znak na klawiaturze: \n");
scanf("%c",&znak);
printf("Kod wciśniętego znaku to: %d \n",znak);

printf("Podaj kod znaku: \n");
scanf("%d",&kod);
printf("Znak o podanym kodzie to: %c \n",kod);

getch();
return 0 ;
}
```

Przykładowa zawartość ekranu po wykonaniu programu:

```
Wciśnij znak na klawiaturze:
a
Kod wciśniętego znaku to: 97
Podaj kod znaku: 100
Znak o podanym kodzie to: d
```

Do odczytania długości łańcucha służy funkcja ***strlen()*.**

Przykład:

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
main (void)
{
int dlugosc;
char *lancuch;
clrscr();

lancuch="Adam Nowak";
dlugosc=strlen(lancuch);
printf("Lancuch '%s' ma: %d znaków \n",lancuch, dlugosc);

printf("Pierwsza litera łańcucha to: %c \n",lancuch[0]);
printf("Ostatnia litera łańcucha to: %c \n",lancuch[dlugosc-1]);

getch();
return 0 ;
}
```

W języku C w wielu przypadkach liczenie zaczyna się od zera, a nie od jedynki.
Dlatego na pierwszy znak łańcucha wskazuje odwołanie: *lancuch[0]*, na drugi: *lancuch[1]* itd. Ostatni znak łańcucha jest to zawsze znak *\0*.

Inne funkcje operujące na łańcuchach.
M. Inn. to
strcat() - łączy dwa łańcuchy,
strcmp() - porównuje dwa łańcuchy rozróżniając małe i duże litery,
strlwr() i ***strupr()*** - zamienia w danym łańcuchu duże litery na małe i odwrotnie,
strrev() - odwraca kolejność znaków w łańcuchu,
strset() - wypełnia łańcuch danym znakiem.

```
/* znaki5.c - operacje na tekstach */
#include <stdio.h>
#include <conio.h>
#include <string.h>
main (void)
{
char *lancuch1, *lancuch2, *lancuch3;
char znakwyp='x';  // znak wypełniający
clrscr();
puts("Operacje tekstowe\n");

lancuch1="Janusz";
lancuch2="Kowalski";

printf("Lancuch1 to: %s \n",lancuch1);
printf("Lancuch2 to: %s \n",lancuch2);

printf("\nZmieniamy duze litery na male: \n");
strlwr(lancuch1);
printf("Lancuch1 wyglada teraz tak: %s \n",lancuch1);

printf("\nZmieniamy male litery na duze: \n");
strupr(lancuch2);
printf("Lancuch2 wyglada teraz tak: %s \n",lancuch2);

printf("\nLaczymy dwa lancuchy: \n" );
lancuch3=strcat(lancuch1,lancuch2);
printf("Lancuch3 wyglada teraz tak: %s \n",lancuch3);

printf("\nOdwracamy kolejnosc znakow w lancuchu: \n" );
strrev(lancuch3);
printf("Lancuch3 wyglada teraz tak: %s \n",lancuch3);

printf("\nWypelniamy lancuch znakiem 'x':\n" );
strset(lancuch3,znakwyp);
printf("Lancuch3 wyglada teraz tak: %s \n",lancuch3);

getch();
return 0 ;
}
```

Podsumowanie 3:

1. Zmienne liczbowe mogą zawierać się w pewnych zakresach, których nie można przekraczać.
2. Deklaracja zmiennej znakowej: ***char znak;***
   a zmiennej łańcuchowej: ***char *slowo;***
3. Wartość zmiennej znakowej można przypisać w programie poprzez umieszczenie znaku w apostrofach lub przez napisanie jego kodu.
4. Wartość zmiennej łańcuchowej można przypisać w programie poprzez umieszczenie napisu w cudzysłowie.
5. Zmienne znakowe i łańcuchowe można wczytywać z klawiatury używając funkcji ***scanf()***
   i odpowiednich wzorców konwersji**:** ***%s*** dla ciągu znaków i ***%c*** dla pojedynczego znaku.
6. Każdy znak posiada swój kod ASCII.
7. Kod ASCII mają również znaki nie przedstawione na klawiaturze komputera. np. ß, ö, à.
8. Łańcuch, który wygląda jak liczba nie liczbą.
   Istnieją funkcje, które potrafią przekonwertować łańcuch liczbowy do postaci liczby.
9. Mając dany łańcuch, możemy odczytać dowolny jego znak używając nawiasów kwadratowych. Pierwszy wpisany znak ma numer 0, a nie 1.
10. Każdy ciąg kończy znak **'\0'.**
11. Długość łańcucha można ograniczyć przy deklaracji, np.: ***char slowo[10];***
12. Łańcuchy można ze sobą porównywać, łączyć, odwracać w nich kolejność liter, zmieniać małe litery na duże i odwrotnie, a także przeszukiwać, kopiować na siebie itp.
    Nazwy funkcji, które to wykonują zawsze zaczynają się na '**str**' (z angielskiego: string).

Wejście i wyjście programu

Do podstawowych funkcji języka C, umożliwiających komunikację z otoczeniem należą:

- dla operacji **wejścia**: **getchar, gets, scanf;**
- dla operacji **wyjścia: putchar, puts, printf**

W **DOS**, wyniki wysyłane na ekran mogą być przy pomocy znaku potoku wysłane do pliku lub na drukarkę.
Np.
p11 > Wynik.txt (do pliku)
p11 > PRN (na drukarkę)

Funkcja putchar: int putchar(int)

Funkcja wysyła pojedynczy znak na zewnątrz (do standardowego strumienia wyjściowego **stdout**, standardowo na ekran)
Funkcja (makro) zwraca wartość wypisanego znaku lub EOF (jako sygnał błędu).

Przykład:

```
/*******************************************************/
/*  Program  p12.c                                     */
/*                              ( putchar )            */
```

```
/*---------------------------------------------------------*/

#include <stdio.h>

#define SPACJA 32

main()
{
  char napis[] = "ABCDEFGHIJKLMNOPQRS...";
  int znak = 73;

  /* wyprowadzanie pojedynczych znakow
     przy pomocy funkcji  putchar   */

  putchar('\n');     putchar(znak);     putchar(SPACJA);
  putchar(55);       putchar(SPACJA);   putchar('Z');
  putchar(SPACJA);   putchar('\066');   putchar(SPACJA);
  putchar('\x3A');   putchar('\n');     putchar(napis[0]);
  putchar(SPACJA);   putchar(napis[4]); putchar(SPACJA);
  putchar(znak+'\066'-52);              putchar('\n');
}
/*********************************************************/
```

Wynik:

```
I 7 Z 6 :
A E K
```

Funkcja puts: int puts(const char *s);

Wysyła łańcuch s do standardowego strumienia wyjściowego (stdout) i dołącza znak końca wiersza.
W przypadku powodzenia operacji wartość jest nieujemna, w przeciwnym wypadku EOF.

```
/*********************************************************/
/*  Program  p13.c                                       */
/*                              ( puts )                 */
/*-------------------------------------------------------*/

#include <stdio.h>

#define NOWA_LINIA putchar('\n')

main()
{
  char napis[] = "ABCDEFGHIJKLMNOPQRS...";

  /* wyprowadzanie pojedynczych linii tekstu
     przy pomocy funkcji  puts            */

  NOWA_LINIA; puts(napis); NOWA_LINIA;
  puts("To jest praktyczna funkcja");
  puts("\066\067\x2B\x2A itd.");
}
/*********************************************************/
```

Wynik:

```
ABCDEFGHIJKLMNOPQRS...

To jest praktyczna funkcja
67+* itd.
```

Funkcja printf()

Wyprowadza wynik przetwarzania w różnych formatach.

printf (łańcuch, lista argumentów) lub inaczej
printf(ciag_formatujący, lista parametrów);

Ciąg formatujący jest zwykłym ciągiem znaków do wyświetlenia na ekranie.
Jednak niektóre znaki mają funkcję specjalną i nie zostaną one po prostu wyświetlone.
Takim właśnie znakiem jest znak **% .**
Gdy funkcja **printf**() go napotka to wie, że po nim wystąpi określenie rodzaju argumentu i formatu jego wyświetlenia na ekranie.

Ogólnie **ciąg formatujący** ma zapis :

% [flagi] [szerokość] [precyzja] [modyfiktor wielkości] typ_parametru

Tylko "typ_parametru" musi wystąpić po znaku % ,
natomiast parametry podane w nawiasach kwadratowym są opcjonalne i może ich w ogóle nie być (tak jest w przedstawionym przykładzie).

Przykład:

```
/*********************************************************/
/*  Program  p14.c                                       */
/*                              ( printf  /1/ )          */
/*-------------------------------------------------------*/

#include <stdio.h>

main()
{
  int  k = 21101;  /* liczba dziesietna */

  printf("\nk(_10) = %i   k(_8) = %o    k(_16) = %X", k, k, k);


}
/*********************************************************/
```

Wynik:

```
k(_10) = 21101   k(_8) = 51155    k(_16) = 526D
```

Formaty realizowane przez funkcję printf (znaki typu w łańcuchach formatujących)

Typ danych - znak	Argument	Format wyjściowy

typu w formacie	wejściowy	
Liczba		
%d, %i	**int**	liczba całkowita ze znakiem
%u	unsigned int	l. całowita. ze bez znaku
%o	int	ósemkowa
%x	int	szesnastkowa bez znaku, małe litery a..f
%X	int	j.w. będą duże litery A..F
%f	**float**/double	l. zmiennoprzecinkowa: [-]nnnn.mmmmm
%e	float/double	j.w. w postaci [-].nnnne[+-]mmm
%E	-"-	j.w. ze znakiem E
%G	-"-	jak %f lub %E, mniejsza liczba znaków
ZNAK lub ŁAŃCUCH		
%c	**char** (znak)	pojedynczy znak
%s	char * - wskaźnik **łańcucha**	łańcuch znaków, aż do napotkania bajtu zerowego \0
WSKAŹNIK		
%n	int *	Liczba dotychczas wysłanych znaków
%p	pointer - wskaźnik	argument w postaci wskaźnika, co zależy od modelu pamięci (segm:offs lub offs)- liczba 16-wa

Najczęściej stosowane typy parametrów

%d	zmienna typu **int** (ze znakiem)
%u	zmienna typu **int** (bez znaku)
%c	zmienna typu **char** (litera)
%f	zmienna typu **float** (rzeczywista)
%x	zmienna typu **int** (bez znaku) wyświetlana w postaci szesnastkowej
%o	zmienna typu **int** (bez znaku) wyświetlana w postaci ósemkowej
%s	ciąg znaków
%p	wskaźnik

Przykłady:

```
/*********************************************************/
```

```
/*  Program  p15.c                                          */
/*                              ( printf  /2/ )             */
/*-------------------------------------------------------*/

#include <stdio.h>

main()
{
  float  a,b,c;

    a=153.67789;    b=2.33E-2;   c = a * b;

    printf(" %f  razy  %E  wynosi:  %f ", a,b,c);
}
/*******************************************************/
Wynik:  153.677887  razy  2.330000E-02  wynosi:  3.580695


/*******************************************************/
/*  Program  p17.c             ( Turbo C 2.0 )          */
/*                             ( printf  /4/ )          */
/*-------------------------------------------------------*/

#include <stdio.h>

main()  /* zagadnienie dokladnosci */
{
  double  a=153.67789;
  int     k=22799;
  int     m=-500;
  int     *p;     /* wskaznik */
  char    c='M';
  char    napis[]="Lepsze C od B lub P    ";

  p=&k;
  printf("\n format  %%d   %d", k);
  printf("\n format  %%i   %i", m);
  printf("\n format  %%u   %u", k);
  printf("\n format  %%o   %o", k);
  printf("\n format  %%x   %x", k);
  printf("\n format  %%X   %X", k);
  printf("\n format  %%f   %f", a);
  printf("\n format  %%e   %e", a);
  printf("\n format  %%E   %E", a);
  printf("\n format  %%g   %g", a);
  printf("\n format  %%G   %G", a);
  printf("\n format  %%c   %c", c);
  printf("\n format  %%s   %s", napis);
  printf("\n format  %%p   %p", *p);
  printf("\n ----------------------");
  printf("\n "); printf(napis);
}
/*******************************************************/
Wynik:
 format  %d   22799
 format  %i   -500
 format  %u   22799
 format  %o   54417
 format  %x   590f
 format  %X   590F
 format  %f   153.677890
```

```
format  %e   1.536779e+02
format  %E   1.536779E+02
format  %g   153.678
format  %G   153.678
format  %c   M
format  %s   Lepsze C od B lub P
format  %p   654C:590F
-----------------------
Lepsze C od B lub P
```

Użyty w łańcuchach formatu zapis %% reprezentuje pojedynczy "widzialny" znak procentu.

Ogólny zapis ciągu formatującego i objaśnienia

% [flagi] [szerokość] [precyzja] [modyfiktor wielkości] typ_parametru

Opis flag:

-	wyrównuje liczbę do lewej (normalnie byłaby wyrównana do prawej)
+	liczba zawsze zaczyna się od znaku "+" (dla dodatnich) lub "-" (dla ujemnych), normalnie znak jest wyświetlany tylko dla liczb ujemnych

Parametr szerokość:

n	gdzie n jest liczbą określającą ile znaków zostanie wyświetlonych. Jeśli n jest większe od szerokości liczby to zostanie ona uzupełniona spacjami. Jeśli jest mniejsze to liczba nie zostanie ucięta.
0n	gdzie n jest liczbą określającą ile znaków zostanie wyświetlonych. Jeśli n jest większe od szerokości liczby to zostanie ona uzupełniona zerami. Jeśli jest mniejsze to liczba nie zostanie ucięta.

Parametr precyzja:
Parametr ten zawsze zaczyna się od kropki, a następnie podajemy ilość liczb do wyświetlenia po przecinku (tak jak to było przy szerokości.

Modyfikator wielkości:

l	określa, że parametr jest typu long (np. long int to %ld)
h	określa, że parametr jest typu short (np. short int to %hd)

```
#include <stdio.h>

void main(void)
{
  float         f = 0.521;
  int           i = -123;
  unsigned int  u = 24;
  char          c = 'A';
```

```
  printf("Zmienna f = %f, a zmienna i jest rowna %d.\n", f, i);
  printf("Zmienna c = %c, a zmienna u jest rowna %u.\n", c, u);
  printf("Zmienna u w zapisie szesnastkowym jest rowna %x, \n", u);
  printf("natomiast w zapisie osemkowym jest rowna %o.", u);
}
```

Wyniki:

```
Zmienna f = 0.521000, a zmienna i jest rowna -123.
Zmienna c = A, a zmienna u jest rowna 24.
Zmienna u w zapisie szesnastkowym jest rowna 18,
natomiast w zapisie osemkowym jest rowna 30.
```

Przykład użycia dla zmiennej w poniższym programie (modyfikacja programy powyżej).

```
/* program printf2.c */
#include <stdio.h>

void main(void)
{
  float f = 0.521;

  printf("Zmienna f = %6.3f\n", f);
  printf("Zmienna f = %-6.3f\n", f);
  printf("Zmienna f = %06.3f\n", f);
  printf("Zmienna f = %+6.3f\n", f);

  getch();
}
```

Wynik

```
Zmienna f =  0.521
Zmienna f = 0.521
Zmienna f = 00.521
Zmienna f = +0.521
```

Objaśnienia

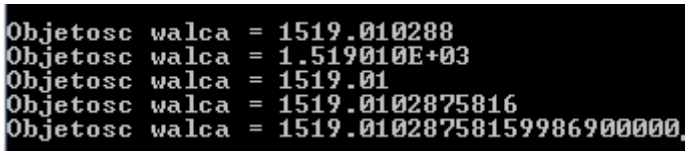
"[%6.3f]"	wyświetli się "[0.521]". Pamiętaj, że liczba sześć oznacza szerokość całej liczby, a nie tylko części przed przecinkiem. Ponieważ szerokość liczby jest równa pięć, to została dodana jedna spacja _przed_ liczbą.
"[%-6.3f]"	wyświetli się "[0.521]". Jak wyżej, tylko spacja została dodana _po_ liczbie (wyrównanie do lewej).
"[%06.3f]"	wyświetli się "[00.521]". Czyli zamiast spacji, zostało dodane zero.
"[%+6.3f]"	wyświetli się "[+0.521]". Oczywiście dla *f* równego -0.521 wyświetli się znak minus, nie plus.

Przykład formatowania liczb:

```
/*******************************************************/
/*  Program  p21.c                                     */
/*  Obliczanie objetosci walca   ( wieksza dokladnosc ) */
/*-----------------------------------------------------*/
```

```c
#include <stdio.h>

#define PI           3.1415926
#define PROMIEN      3.3
#define WYSOKOSC    44.4

main()
{
   double promien, wysokosc, objetosc;

   promien  = PROMIEN;
   wysokosc = WYSOKOSC;
   objetosc = PI * promien * promien * wysokosc;
   printf("\nObjetosc walca = %f", objetosc);
   printf("\nObjetosc walca = %E", objetosc);
   printf("\nObjetosc walca = %g", objetosc);
   printf("\nObjetosc walca = %15.10f", objetosc);
   printf("\nObjetosc walca = %25.20f", objetosc);

}
/*********************************************************/
```

Wynik:



Funkcja getchar

int getchar(void)

Funkcja odczytuje znak ze standardowego strumienia wejściowego (stdin) i zwraca go po dokonaniu konwersji bez rozszerzenia znakowego.
Funkcja przy każdym wywołaniu podaje następny znak z wejścia lub EOF, gdy napotkała koniec pliku. W przypadku błędu lub końca zbioru wartością funkcji jest EOF. Stała symboliczna EOF jest zdefiniowana w nagłówku <stdio.h>.
Jej wartoscią jest na ogół -1, ale e testach należy używać EOF.
Zadaniem funkcji jest wprowadzenie pojedynczego znaku do pamięci komputera.

Syntaktycznie postać funkcji jest następująca:
c = getchar(); gdzie c jest typu char.

Po każdym wywołaniu funkcja getchar pobiera ze strumienia znaków następny znak i wraca z jego zawartością. Zawartością zmiennej jest następny znak z wejścia.

Przykłady:

```c
#include <stdio.h>
```

```
/* przepisz wejscie na wyjście, wersja 1 */
main()
{
 int c;
 c = getchar(); /* przeczytaj znak */

 while (c != EOF) /* dopóki znak nie jest znakiem konca pliku (Ctrl Z) */
  {
        putchar(c);  /* wypisz przeczytany znak */
        c=getchar(); /* przeczytaj nastepny znak */
  }
}
```

TYPY ZMIENNYCH I WZORCE KONWERSJI

Zmienne służą do tego, aby zapamiętać sobie tymczasową wartość (którą potem można zmienić w każdej chwili według potrzeby).
Dane w programie umieszczane są w postaci tzw. **zmiennych.**
Zmienna jest to pewien fragment pamięci o ustalonym rozmiarze, który posiada własny identyfikator (nazwę) oraz może przechowywać pewną wartość, zależną od typu zmiennej.

Deklaracja zmiennych

typ nazwa_zmiennej;

Oto deklaracja zmiennej o nazwie "wiek" typu "int" czyli liczby całkowitej:
int wiek;
Zmiennej w momencie zadeklarowania można od razu przypisać wartość (inicjalizować):
int wiek = 16;

W języku C zmienne deklaruje się na samym początku bloku (czyli przed pierwszą instrukcją).

Język C nie inicjalizuje zmiennych lokalnych.
Oznacza to, że w nowo zadeklarowanej zmiennej znajdują się śmieci - to, co wcześniej zawierał przydzielony zmiennej fragment
pamięci.
Aby uniknąć ciężkich do wykrycia błędów, dobrze jest inicjalizować (przypisywać wartość) wszystkie zmienne w momencie zadeklarowania.

Zasięg zmiennej

Zmienne globalne i lokalne

Zmienne globalne - obejmujące zasięgiem cały program
Zmienne mogą być dostępne dla wszystkich funkcji programu — nazywamy je wtedy **zmiennymi globalnymi**.
Deklaruje się je przed wszystkimi funkcjami programu:
Przykład

```
#include <stdio.h>
int a,b; /* nasze zmienne globalne */

void func1 ()
```

```
{
        /* instrukcje */
        a=3;
/       * dalsze instrukcje */
}

int main ()
{
        b=3;
        a=2;
        return 0;
}
```

Zmienne globalne, jeśli programista nie przypisze im innej wartości podczas definiowania, są inicjalizowane wartością 0.

Zmienne lokalne – o zasięgu obejmującym pewien blok.
Zmienne, które funkcja deklaruje do "własnych potrzeb" nazywamy **zmiennymi lokalnymi**.
Nasuwa się pytanie: "czy będzie błędem nazwanie tą samą nazwą zmiennej globalnej i lokalnej?". Otóż odpowiedź może być zaskakująca: nie. Natomiast w danej funkcji da się używać tylko jej zmiennej lokalnej. Tej konstrukcji należy, z wiadomych względów, unikać.

```
int a=1; /* zmienna globalna */
int main()
{
int a=2; /* to już zmienna lokalna */
printf("%d", a); /* wypisze 2 */
}
```

Czas życia zmiennych

Czas życia zmiennych to czas od momentu przydzielenia dla zmiennej miejsca w pamięci (stworzenie obiektu) do momentu zwolnienia miejsca w pamięci (likwidacja obiektu).
Zakres ważności to część programu, w której nazwa znana jest kompilatorowi.

Przykład

```
main()
{
int a = 10;
        { /* otwarcie lokalnego bloku */
                int b = 10;
                printf("%d %d", a, b);
        } /* zamknięcie lokalnego bloku, zmienna b jest usuwana */
printf("%d %d", a, b); /* BŁĄD: b juz nie istnieje */
} /* tu usuwana jest zmienna a */
```

Zdefiniowano dwie zmienne typu int.
Zarówno a i b istnieją przez cały program (czas życia).
Nazwa zmiennej a jest znana kompilatorowi przez cały program.
Nazwa zmiennej b jest znana tylko w lokalnym bloku, dlatego nastąpi błąd w ostatniej instrukcji.

Stałe

Stała pozostanie taka sam w trakcie przebiegu programu.
Nadaje się jej wartość w czasie pisania programu a nie w czasie działania.
Stała, różni się od zmiennej tym, że nie można jej przypisać innej wartości w trakcie

działania programu.

Stałą deklaruje się z użyciem słowa kluczowego **const**:
const typ nazwa_stałej=wartość;

Dobrze jest używać stałych w programie, ponieważ unikniemy wtedy przypadkowych pomyłek a kompilator może często zoptymalizować ich użycie (np. od razu podstawiając ich wartość do kodu).

Przykład:
```
const  int  WARTOSC_POCZATKOWA=5;
int  i=WARTOSC_POCZATKOWA;
WARTOSC_POCZATKOWA=4; /* tu kompilator zaprotestuje */
int  j=WARTOSC_POCZATKOWA;
```

Stałe symboliczne

Stała symboliczna jest nazwą zastępującą ciąg znaków
#define NAZWA tekst

Np.
```
#define PI  3.1415926
#define MIEJSCOWOSC Sosnowiec
#define WYNIK printf(("Pole=%d\f %”,pole1)
#define WZOR1 (a*b)
```

Typy zmiennych:

Określając typ zmiennej przekazuje się kompilatorowi informację, ile pamięci trzeba zarezerwować dla zmiennej, a także w jaki sposób wykonywać na nim operacje.

Jeśli potrzebujemy w pewnym miejscu programu innego typu danych to stosujemy rzutowanie.
W takim wypadku stosujemy **konwersję** (**rzutowanie**) jednej zmiennej na inną zmienną

Istnieją **wbudowane** i **zdefiniowane przez użytkownika** typy danych.

W języku C wyróżniamy następujące podstawowe typy zmiennych.

- **char** – typ znakowy - jednobajtowe liczby całkowite, służy do przechowywania znaków;
- **int**- liczby całkowite - typ całkowity, o długości domyślnej dla danej architektury komputera;
- **float** – liczby rzeczywiste - typ zmiennopozycyjny (zwany również zmiennoprzecinkowym), reprezentujący liczby rzeczywiste (4 bajty);
- **double** – liczby rzeczywiste - typ zmiennopozycyjny podwójnej precyzji (8 bajtów);

- ***short*** - liczby całkowite krótkie
- ***long*** - liczby całkowite długie
- ***long double*** - liczby zmiennoprzecinkowe podwójnej precyzji długie

Typ **int** przeznaczony jest do liczb **całkowitych.**

Liczby te możemy zapisać na kilka sposobów:

System dziesiętny: np. **21; 13; 45; 156**

System ósemkowy (oktalny): np. **010; 027**. (Cyfry 0 ..7)
Cyfra 8 nie jest dozwolona.
Jeżeli chcemy użyć takiego zapisu musimy zacząć liczbę od **0**.

System szesnastkowy (heksadecymalny), np. **0x10, 0xff.**
Aby użyć takiego systemu musimy poprzedzić liczbę ciągiem 0x.
Wielkość znaków w takich literałach nie ma znaczenia **.**

Typ **float** dla liczb zmiennoprzecinkowych

Ten typ oznacza liczby zmiennoprzecinkowe czyli ułamki.
Istnieją dwa sposoby zapisu:

- System dziesiętny, np. **10.256**, 3.14 (z kropką dziesiętna)
- System "naukowy" – wykładniczy, np. **6e2** (czyli $6 * 10^2$)
  3.4e-3 (czyli $3.4 * 10^{(-3}$–

Typ **double** dla liczb zmiennoprzecinkowych

Double - czyli "podwójny" - oznacza liczby zmiennoprzecinkowe podwójnej precyzji. Oznacza to, że liczba taka zajmuje zazwyczaj w pamięci dwa razy więcej miejsca niż float (np. 64 bity wobec 32 dla float), ale ma też dwa razy lepszą dokładność.

Domyślnie ułamki wpisane w kodzie są typu double. Możemy to zmienić dodając na końcu literę "f":
1.4f (znaczy float), 1.4 (znaczy double)

Typ **char** dla znaków

Jest to typ znakowy, umożliwiający zapis znaków ASCII.
Może też być traktowany jako liczba z zakresu 0..255.
Znaki zapisujemy w pojedynczych cudzysłowach (czasami nazywanymi apostrofami),
by odróżnić je od łańcuchów tekstowych (pisanych w podwójnych cudzysłowach).

Np. **‘A’, ‘a’, ‘!’, ‘$’**

Pojedynczy cudzysłów ' zapisujemy **' \ ' '**
a null (czyli zero, które między innymi kończy napisy) tak: **' \ 0 '**

Inne **znaki specjalne**

- '\a' - alarm (sygnał akustyczny terminala)
- '\b' - backspace (usuwa poprzedzający znak)
- '\f' - wysuniecie strony (np. w drukarce)
- '\r' - powrót kursora (karetki) do początku wiersza
- **'\n'** - znak nowego wiersza
- '\"' - cudzysłów
- '\'' - apostrof

- '\\' - ukośnik wsteczny (backslash)
- **'\t'** - tabulacja pozioma
- '\v' - tabulacja pionowa
- '\?' - znak zapytania (pytajnik)
- **'\ooo'** - liczba zapisana w systemie oktalnym (ósemkowym), gdzie 'ooo' należy zastąpić trzycyfrową liczbą w tym systemie
- **'\xhh'** - liczba zapisana w systemie heksadecymalnym (szesnastkowym), gdzie 'hh' należy zastąpić dwucyfrową liczbą w tym systemie, np. '\xAFF'

Specyfikatory – signed, unsigned, short, long

słowa kluczowe, które postawione przy typie danych zmieniają jego znaczenie.
signed – liczba ze znakiem i **unsigned** – nieujemna (bez znaku)

---

short i long są wskazówkami dla kompilatora, by zarezerwował dla danego typu mniej (odpowiednio - więcej) pamięci.
Mogą być zastosowane do dwóch typów: **int** i **double** (tylko long), mając różne znaczenie.
Jeśli przy short lub long nie napiszemy, o jaki typ nam chodzi, kompilator przyjmie wartość domyślną czyli int.

Przykłady

```
signed char a;     /* zmienna a przyjmuje wartości od -128 do 127 */
unsigned char b;   /* zmienna b przyjmuje wartości od 0 do 255    */
unsigned short c;
unsigned long int d;
```

Operatory i wyrażenia

Do wykonania obliczeń zmieniających dane w informacje, niezbędne są operatory. Operator to symbol mówiący komputerowi, jak ma przetwarzać dane.

Operatory, z punktu widzenia liczby argumentów dzielimy na dwuargumentowe i jednoargumentowe.

a op b - op jest operatorem 2-argumentowym, np. a+b; a*b; a/b; a%b;

op a - op jest operatorem jednoargumentowym, np. -a; -(a+b); !a

Można je pogrupować wg cech funkcjonalnych na grupy:

- operatory arytmetyczne
- operatory porównania - relacyjne

- operatory logiczne
- operatory bitowe
- operatory przypisania
- operatory unarne
- operatory rozmiaru
- operatory konwersji
- operator warunkowy
- operator przecinkowy
- operatory wskazywania

Operatory

Najczęściej używane operatory:

+, -, *, /, =	dodawanie, odejmowanie, mnożenie, dzielenie, operator przypisania
%	reszta z dzielenia (odpowiada operacji modulo)
++, --	inkrementacja i dekrementacja
!	negacja
<, >	mniejsze, większe
<=, >=	nie większe, nie mniejsze
&&, ||	i, lub (logicznie)
==	równe
+=, -=, *=, /=	połączenie operatorów przypisania i innych

Operatory arytmetyczne

Operatory arytmetyczne można podzielić na 3 grupy:

- operatory inkrementacji i dekrementacji: --Op, ++Op, Op--, Op++
- operatory addytywne: "+" i "-"
- operatory multiplikatywne: "*', "/", "%"

Operatory te używane są do wykonywania działań matematycznych.

W języku C występuje pięć dwuargumentowych operatorów arytmetycznych:

- dodawanie **+**
- odejmowanie **-**
- mnożenie *****
- dzielenie **/**
- modulo - reszta z dzielenia **%** określona tylko dla liczb całkowitych (*dzielenie modulo*).

Brak jest operatora potęgowania (realizuje to funkcja biblioteczna pow)

Jeżeli dzielimy zmienne lub liczby całkowite przy pomocy operatora /,
to wynik zawsze będzie liczbą całkowitą.
Z operatora **%** korzystamy do obliczenia reszty z dzielenia liczb całkowitych.

Kolejność działań jest identyczna jak w matematyce.

Dzielenie całkowite obcina część ułamkową wyniku.

Przykład:
```
int x,y, a=5, b=2;
x=a/b;              /* w wyniku x otrzyma wartość 2 */
y=a%b;              /* y => 1  - reszta z dzielenia */
```

x % y daje w wyniku resztę dzielenia x przez y.
Operatora **%** nie można stosować do danych typu **float** i **double**

Należy pamiętać, że (w pewnym uproszczeniu) wynik operacji jest typu takiego jak *największy* z argumentów.
Oznacza to, że operacja wykonana na dwóch liczbach całkowitych nadal ma typ całkowity nawet jeżeli wynik przypiszemy do zmiennej rzeczywistej.

Przykłady:

```
float a = 7 / 2;
printf("%f\n", a);
```
wypisze `3.0`, a nie `3.5`.

Odnosi się to nie tylko do dzielenia, ale także mnożenia, np.:
```
float a = 1000 * 1000 * 1000 * 1000 * 1000 * 1000;
printf("%f\n", a);
```

prawdopodobnie da o wiele mniejszy wynik niż byśmy się spodziewali.

Aby wymusić obliczenia rzeczywiste należy zmienić typ jednego z argumentów na liczbę rzeczywistą po prostu zmieniając literał lub korzystając z rzutowania, np.:

```
float a = 7.0 / 2; /* wcześniejszy zapis: float a = 7 / 2; */
float b = (float)1000 * 1000 * 1000 * 1000 * 1000 * 1000;
printf("%f\n", a);
printf("%f\n", b);
```

Przykład programu do wydania reszty - ilość banknotów 20-, 10-, 5- i 1- dolarowych.

```
/* change.c */
#include <stdio.h>
main()
{
int amount,twenties, tens, fives, ones, r20, r10;
printf("Wprowadz kwote do wydania: "); /* Wprowadzenie kwoty do wydania */
scanf("%d", &amount);
twenties= amount/20; /* banknoty 20-dolarowe */
r20=amount % 20;  /* r20 reszta pozostała po 20-ch*/
```

```
tens= r20/10; /* ilość banknotów 10-dolarowych */
r10=r20 % 10;  /* r10 reprezentuje resztę po 10-ch */
fives= r10 / 5; /* banknoty 5-dolarowe */
ones = r10 % 5; /* reszta - dolarówki */
putchar('\n');
printf("By przekazac %d wydaj banknoty: \n", amount);
printf("%d dwudziestki\n", twenties);
printf("%d dziesiatki\n", tens);
printf("%d piatki \n", fives);
printf("%d pojedyncze (s)\n", ones);
}
```

Zwykle po obu stronach jest ten sam typ danych (int lub float).
Można też użyć różnych typów numerycznych po lewej i prawej stronie działania.
Wyświetlona wartość będzie zależała od typu zmiennej po lewej stronie.

Przykład:

```
#include <stdio.h>
void main()
{
 int t;
 float c, s, r;
 c=56.09;
 s=4.98;
 t=c+s; // calkowite
 r=c+s; // rzeczywiste
 printf ("Razem - (int): %d  (float): %.2f",t, r);
}
```

Wynik: Razem - (int): 61 (float): 61.07

Operatory porównania i relacji

W języku C występują następujące **operatory porównania**:

- równe ==
- różne !=
- mniejsze <
- większe >
- mniejsze lub równe <=
- większe lub równe >=

Wykonują one odpowiednie porównanie swoich argumentów i zwracają jedynkę jeżeli warunek jest spełniony lub zero jeżeli nie jest.

Relacje i operatory logiczne
Operatory relacji: > >= < <= (mają ten sam priorytet)
Operatory przyrównania: == (równe) != (różne) (priorytet niższy)

Przykłady wyrażeń z operatorami porównania
k=1; m=2; n=3;

Wyrażenie	Opis	Wartość
m<n	prawda	1
(k+m)>=n	prawda	1
(m-k)>n	fałsz	0
n>=3	prawda	1
k != m	prawda	1
k == 2	fałsz	0
(m+2)<=k	fałsz	0

Operatory logiczne:
&& logiczne AND - iloczyn
|| logiczne OR - alternatywa
! logiczne NOT - negacja

Wyrażenie ma wartość logiczną 0 wtedy i tylko wtedy, gdy jest równe 0 (jest "fałszywe").
W przeciwnym wypadku, gdy wyrażenie jest różne od zera ma wartość logiczną 1 (jest "prawdziwe").
Operatory logiczne w wyniku dają zawsze albo 0 albo 1.

```
printf("koniunkcja: %d\n", 18 && 19); // wynik 1 bo 18 i 19 <> 0
printf("alternatywa: %d\n", 'a' || 'b'); // alternatywa – wynik 1
printf("negacja: %d\n", !20); // negacja – wynik 0
```

Operatory logiczne z operatorami porównania umożliwiają budowę bardziej skomplikowanych wyrażeń logicznych, np.
k=1; m=2; n=3;

Wyrażenie	Opis	Wartość
m<n	prawda	1
((k+m)>=n) && (m == 1)	fałsz (1 i 0)	0
((m-k)>n) || (m == 2)	Prawda (0 lub 1)	1
!(m+2)<=k	prawda (NOT 0)	1

Operatory przypisania

Instrukcją przypisania jest trójka elementów zapisana jako:

identyfikator operator_przypisnia wyrażenie

Identyfikator to np. nazwa zmiennej.
Wyrażenie – dowolne wyrażenia

Operator	Przykład zapisu	Zapis równoważny
=	a = b	a = b
+=	a += b	a = a + b
-=	a -= b	a = a - b

*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>=b	a = a>>b
&=	a &= b	a = a & b
^=	a ^= b	a = a ^ b
|=	a |= b	a = a | b

Operator **przypisania** **=** , przypisuje wartość prawego argumentu lewemu argumentowi,

np.:

```
int a = 6, b;
b = a;
printf("%d\n", b); /* wypisze 6 */
```

Operator ten ma łączność prawostronną tzn. obliczanie przypisań następuje z prawa na lewo i zwraca on przypisaną wartość, dzięki czemu może być użyty kaskadowo:

```
int a, b, c;
a = b = c = 2;
printf("%d %d %d\n", a, b, c);  /* wypisze "2 2 2" */
```

Instrukcja przypisania
Identyfikator operator przypisania wyrażenie

Zapis skrócony postaci `a #= b;`,

gdzie # jest jednym z operatorów: **+, -, *, /, &, |, ^, << , >>.**
Ogólnie zapis
`a #= b;`
jest równoważny zapisowi
`a = a # b;`
np.:

```
int a = 1;
a += 5;      /* to samo, co a = a + 5;       */
a /= a + 2; /* to samo, co a = a / (a + 2); */
a %= 2;      /* to samo, co a = a % 2;       */
```

Przykłady zapisu skróconego i tradycyjnego

Język C/C++	Sposób tradycyjny (też w C/C++)
a+=10;	a=a+10;
b-=30;	b=b-30;
a*=20;	a=a*20;
b/=5;	b=b/5;

Przykłady:

double x, y, x;
x = y = z = 1.01;

c += d + e; // równoważne: c = c + d + e;

Przykład programu wykorzystującego operatory przypisania:

```
/*********************************************************/
/*  Program  p32.c                                       */
/*                        ( operatory przypisania )            */
/*-------------------------------------------------------*/

#include <stdio.h>

main()
{
  int    k = 5,  m = 3,  i;
  printf("\n wartosci poczatkowe:  k=%i, m=%i, i=?",k,m);
  printf("\n----------------------------------------");
  printf("\n  operacja         wynik  ");
  printf("\n----------------------------------------");
  i = m;
  printf("\n ( i = m   )       i = %i ", i);
  i += k;
  printf("\n ( i += k  )       i = %i ", i);
  i -= k;
  printf("\n ( i -= k  )       i = %i ", i);
  i *= k;
  printf("\n ( i *= k  )       i = %i ", i);
  i /= k;
  printf("\n ( i /= k  )       i = %i ", i);
  i <<= 3;
  printf("\n ( i <<= 3 )       i = %i ", i);
  i >>= 2;
  printf("\n ( i >>= 2 )       i = %i ", i);
  i &= k;
  printf("\n ( i &= k  )       i = %i ", i);
  i ^= k;
  printf("\n ( i ^= k  )       i = %i ", i);
  i |= k;
  printf("\n ( i |= k  )       i = %i ", i);
  i %= k;
  printf("\n ( i %%= k  )       i = %i ", i);
  printf("\n----------------------------------------");
}
/*********************************************************/
```

Wydruk wyników:

Operatory unarne (jednoargumentowe)

W grupie tej występują 3 operatory:

- ++ zwany operatorem inkrementacji (zwiększania)
- -- zwany operatorem dekrementacji (zmniejszania)
- - zwany minusem jednoargumentowym oraz +

Operatory ++ i -- są szczególnie charakterystyczne dla języka C.
Przykładami zapisu mogą być: ++i, --i, i++, i--.

Operatory unarne – inkrementacji **++** , dekrementacji **--**.
minus jednoargumentowy (np. a=-b)

Inkrementacja i dekrementacja

Aby skrócić zapis wprowadzono dodatkowe operatory:
inkrementacji **++** i dekrementacji **- -** ,
które dodatkowo mogą być **prefiksowe** lub **postfiksowe**.

W rezultacie są cztery operatory:

- pre-inkrementacja ("++i"),
- post-inkrementacja ("i++"),
- pre-dekrementacja ("--i") i
- post-dekrementacja ("i--").

Inkrementacja jest to proces zwiększenia zmiennej o 1,
a **dekrementacja** zmniejszenia o 1.

Zapisowi **i++** lub **++i** odpowiada równoważny zapis **i = i + 1**

Zapisowi **i--** lub **--i** odpowiada **i = i-1.**

Należy pamiętać, że wyrażenia ++i oraz i++ są różne.

Operacja: ++i zwiększa wartość zmiennej i o 1 przed jej użyciem,
natomiast i++ zwiększa wartość zmiennej i o 1 po użyciu jej poprzedniej wartości.

Operatory zwiększania lub zmniejszania o 1 jest wykonywana przez większość procesorów w jednym rozkazie, dlatego w języku C są oddzielne operatory pozwalające na wykonanie tych operacji.

Operator	**Dekrementacja**	**Inkrementacja**
przedrostkowy	--Op	++Op
przyrostkowy	Op--	Op++

Operand Op musi być typu arytmetycznego lub wskaźnikowego i jednocześnie musi stanowić l-wartość (zmieniana jest więc wartość zmiennej).

Operatory **pre-** zwracają nową wartość argumentu,
natomiast **post**- starą wartość argumentu.

```
int a, b, c;
a = 3;
b = a--; /* po operacji b=3 a=2 */
c = --b; /* po operacji b=2 c=2 */
```

Przykład program z zastosowaniem operatorów unarnych

```
/*********************************************************/
/*  Program  p33.c                                       */
/*                          ( operatory unarne )         */
/*-------------------------------------------------------*/

#include <stdio.h>

main()
{
  int   i=5 ;

  printf("\n----------------");
  printf("\n  i = %i ",   i  );
  printf("\n  i = %i ", ++i  );
  printf("\n  i = %i ",   i  );
  printf("\n  i = %i ",   i++);
  printf("\n  i = %i ",   i  );
  printf("\n  i = %i ", --i  );
  printf("\n  i = %i ",   i  );
  printf("\n  i = %i ",   i--);
  printf("\n  i = %i ",   i  );
  printf("\n  i = %i ",  -i  );
  printf("\n----------------");
}
/*********************************************************/
```

Instrukcja **printf("\n i = %i", ++i);** jest równoważna sekwencji instrukcji:
i=i+1; printf("\n i = %i",i);
Instrukcja **printf("\n = %%i", i++);** jest równoważna sekwencji instrukcji:
printf("\n i =%i",i); i =i+1;.

Czasami (szczególnie w C++) użycie operatorów stawianych za argumentem jest nieco mniej efektywne (bo kompilator musi stworzyć nową zmienną by przechować wartość tymczasową).

*Dwuargumentowe operatory + i - mają ten sam priorytet, niższy od ***, /, %,***
który z kolei jest niższy od priorytetu jednoargumentowych operatorów + i -.
Operatory arytmetyczne są lewostronnie łączne.

Rzutowanie, operator konwersji

Zadaniem **rzutowania** jest konwersja danej jednego typu na daną innego typu.
Konwersja może być niejawna (domyślna konwersja przyjęta przez kompilator) lub jawna (podana explicite przez programistę).

Przykłady konwersji niejawnej:

```
int i = 42.7;                /* konwersja z double do int */
float f = i;                 /* konwersja z int do float */
double d = f;                /* konwersja z float do double */
unsigned u = i;              /* konwersja z int do unsigned int */
f = 4.2;                     /* konwersja z double do float */
i = d;                       /* konwersja z double do int */
```

Operator konwersji (rzutowania) pozwala na jawne przekształcenie typów danych

Zapis operatora konwersji
(nazwa typu) wyrażenie

Wyrażenie jest przekształcane wg reguł dla typu określonego przez nazwę typu.

Operator rzutowania służy do jawnego wymuszenia konwersji np.:

Przykłady:
int a;
a=2;
(double)a; /* *typ operandu zmienia się na typ podany w nawiasie - tu double* */

```
double d = 3.14;
int pi = (int)d  // utrata precyzji - pi=3
```

Przykład programu z operatorami konwersji:

```
/*********************************************************/
/*  Program p35.c                                        */
/*                        ( operator konwersji )         */
/*-------------------------------------------------------*/

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

main()
{
   char                c = 'k';
   int                 i = 70;
   float               f = 345.6789;

   printf("\n Wartosci poczatkowe:  ");
   printf("\n c = \'%c\';  i = %i;  f = %8.4f;", c ,i, f);
   printf("\n-------------------------------------");
   printf("\n Konwersja           Wynik           ");
   printf("\n-------------------------------------");
   printf("\n (int)    c          %i ", (int)    c      );
   printf("\n (float)  c          %f ", (float)  c      );
   printf("\n (double) i          %f ", (double) i      );
   printf("\n (char)   i          %c ", (char)   i      );
   printf("\n (double) i+2000     %f ", (double) i+2000 );
   printf("\n (int)    f          %i ", (int)    f      );
   printf("\n sqrt((double) c)    %f ", sqrt((double) c));
   printf("\n-------------------------------------");
}
/*********************************************************/
```

Wynik

```
Wartosci poczatkowe:
c = 'k';  i = 70;  f = 345.6789;
-------------------------------------
Konwersja           Wynik
-------------------------------------
(int)    c          107
(float)  c          107.000000
(double) i          70.000000
(char)   i          F
(double) i+2000     2070.000000
(int)    f          345
sqrt((double) c)    10.344080
-------------------------------------
```

Operator rozmiaru - sizeof

Operator sizeof może być używany w jeden ze sposobów:

- sizeof wyrażenie (albo sizeof Op) - rozmiar pamięci zajmowany przez operand Op (wyrażenie)
- sizeof (identyfikator-typu) - rozmiar w bajtach typu określonego przez podany identyfikator

Rezultatem jest liczba całkowita, określająca rozmiar argumentu w bajtach.

Przykłady programów:

```
include <stdio.h>

int main()
{
 printf("sizeof(short ) = %d\n", sizeof(short ));
 printf("sizeof(int   ) = %d\n", sizeof(int   ));
 printf("sizeof(long  ) = %d\n", sizeof(long  ));
 printf("sizeof(float ) = %d\n", sizeof(float ));
 printf("sizeof(double) = %d\n", sizeof(double));
 return 0;
}
```

Wynik

```
sizeof(short ) = 2
sizeof(int   ) = 4
sizeof(long  ) = 4
sizeof(float ) = 4
sizeof(double) = 8
```

Przykład 2 programu z użyciem sizeof:

```
/*********************************************************/
/*  Program p34.c              Turbo C  2.0              */
/*                          ( operator  sizeof )         */
/*-------------------------------------------------------*/
#include <stdio.h>
main()
{
   char                c1;
   char          tekst[]  = "Turbo C jak Turbo-Jetta",
                 napis[80]= "Jaki komputer taki pan ";
   int                 i1;
   short int           i2;
   long int            i3;
   float               r1;
   long float          r2;
   double              r3;
   long double          r4;
   int                *p;

   printf("\n Turbo C 2.0  (reprezentacja) ");
   printf("\n============================== ");
   printf("\n Typ         Liczba bajtow    ");
   printf("\n------------------------------ ");
   printf("\n char        :  %2i ", sizeof(char));
   printf("\n int         :  %2i ", sizeof(int));
```

```
   printf("\n short int   :  %2i ", sizeof(short int));
   printf("\n long  int   :  %2i ", sizeof(long  int));
   printf("\n float       :  %2i ", sizeof(float));
   printf("\n long float  :  %2i ", sizeof(long float));
   printf("\n double      :  %2i ", sizeof(double));
   printf("\n long double :  %2i ", sizeof(long double));
   printf("\n pointer     :  %2i ", sizeof p);
   printf("\n------------------------------ ");
   printf("\n tekst       :  %i ", sizeof tekst);
   printf("\n napis       :  %i ", sizeof napis);
   printf("\n zmienna r4  :  %i ", sizeof r4);
   printf("\n============================== ");
}
/*******************************************************/
```

Wynik

```
Turbo C 2.0  (reprezentacja)
==============================
Typ          Liczba bajtow
------------------------------
char         :   1
int          :   2
short int    :   2
long  int    :   4
float        :   4
long float   :   8
double       :   8
long double  :  10
pointer      :   2
------------------------------
tekst        :  24
napis        :  80
zmienna r4   :  10
==============================
```

Operator warunkowy ?

Wyrażenie z użyciem operatora warunkowego:

Op1 ? Op2: Op3;

wyrażenie_warunkowe ? wyrażenie_na_tak: wyrażenie_na_nie;

Operator warunkowy (?:) pozwala zapisać w bardziej zwartej formie pewne konstrukcje programowe wymagające użycia instrukcji if.

Np.
z = (a>b)? a:b; /* z=max(a,b) */

Przykłady:

Typowe zastosowanie - funkcja max

```
/* C++ - Operator warunkowy - funkacja max */
#include <iostream.h>
void main()
{
 int max(int, int);
```

```
 int a, b, m;
 char c;
 cout << "Program wyznacza max 2 liczb\n";
 cout << "Podaj 2 liczby calkowite oddzielone spacja: ";
 cin >> a >> b;
 m=max(a,b);
 cout << "Max liczb " << a << " i " << b << " wynosi " << m;
 cout << "\nNacisnij cos";
 cin >> c;
}
int max(int i, int j)
{
 return i > j? i: j;
}
```

Wynik

```
Program wyznacza max 2 liczb
Podaj 2 liczby calkowite oddzielone spacja: 5
3
Max liczb 5 i 3 wynosi 5
Nacisnij cos
```

Operator przecinkowy/wyliczeniowy , (koma)

Wyrażenie wyliczeniowe ma postać:

Op1, Op2, ..., Opn

gdzie Opi, (i=1..n) są operandami dowolnych typów.

Opracowanie tego wyrażenia polega na obliczeniu wartości kolejnych operandów, począwszy od Op1, a skończywszy na Opn. Typ i wartość całego wyrażenia określa operand Opn.

W C++ wynik wyrażenia wyliczeniowego jest l-wartością.

Poprawne jest więc wyrażenie: (a+=b, c-=d, c*=a)++.

Przykład:

```
int n=10, i=2, k=4, wynik;
wynik=(n-=i, k+=2, i*=5):
```

W wyniku obliczenia powyższego wyrażenia zmienne uzyskają następujące wartości:
n = n-i = 10-2=8
k = k+2 = 4+2=6
i = i*5 = 2*5=10 (ostatni operand)
wynik = wartość ostatniego operandu, czyli 10

Operator przecinkowy stosowany jest m.in. w instrukcjach for, np. do sterowania równoległego 2 indeksami, a także w wywołaniach funkcji.
Stosując wyrażenia wyliczeniowe w wywołaniach funkcji należy pamiętać o użyciu

nawiasów, np.:
func(i, (i+=k, k++), j); - funkcja otrzymuje tylko 3 parametry.

Przykład programu z operatorem przecinkowym:

```
/*************************************************************/
/*  Program  p37.c                                           */
/*                                ( operator przecinkowy )   */
/*-----------------------------------------------------------*/

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

main()
{
   int  i, j;

   printf("\n    i     j    sqrt(i)    sin(j)");
   printf("\n---------------------------------- ");
   for (i=0, j=10; j>=0; i++, j--)
     printf("\n   %2i    %2i    %6.3f    %6.3f",i,j,
            sqrt((double) i), sin((double) j) );     /* argument w
radianach */

   printf("\n---------------------------------- ");
}
/*************************************************************/
```

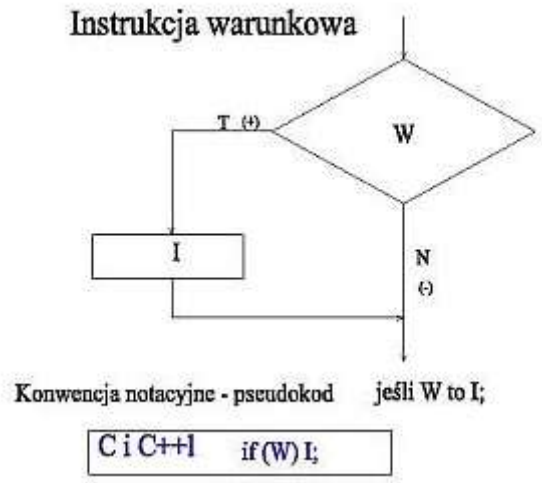
```
Po skompilowaniu i uruchomieniu programu ze skierowaniem wyników do pliku
Np. p37.exe > w.txt
otrzyma się wydruk
    i     j    sqrt(i)    sin(j)
-----------------------------------
    0    10      0.000    -0.544
    1     9      1.000     0.412
    2     8      1.414     0.989
    3     7      1.732     0.657
    4     6      2.000    -0.279
    5     5      2.236    -0.959
    6     4      2.449    -0.757
    7     3      2.646     0.141
    8     2      2.828     0.909
    9     1      3.000     0.841
   10     0      3.162     0.000
-----------------------------------
```

Instrukcje warunkowe

Instrukcja if .. else

jesli Warunek to Instrukcja;

if (W) I;

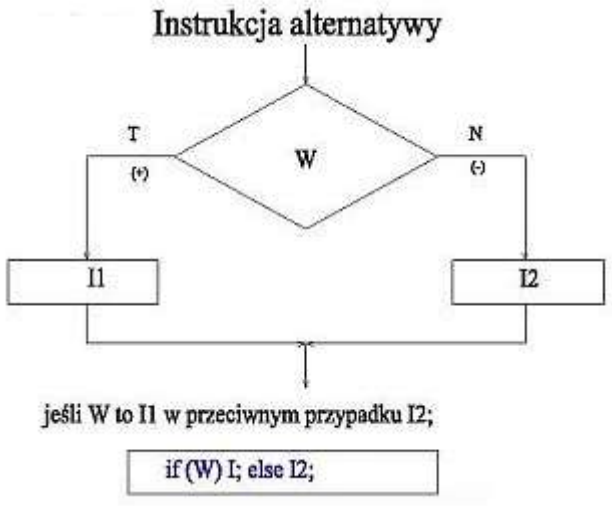



```
if( a > 10 ) {
  printf("Zmienna a jest wieksza od dziesięciu !\n");
  printf("Jest bowiem równa %d.", a);
} else {
  printf("Zmienna a jest mniejsza lub równa dziesięć !\n");
  if(a != 5) printf("Jednak nie jest równa pieć !");
}
```

```
// if1a.c – instrukcja if
#include <stdio.h>
#include <conio.h>
#include <math.h>
main (void)
{
float liczba1, x;
clrscr();
x=0;
printf("Podaj liczbe dodatnia: \n");
scanf("%f",&liczba1);

if (liczba1 >=0)
 x=sqrt(liczba1);
printf("Pierwiastek z %f = %f", liczba1, x);

getch();
return 0 ;
}
```

jeśli W to Instrukcja1 w przeciwnym przypadku Instrukcja2;

if (W) I1; else I2;



// program if1.c

```
#include <stdio.h>
#include <conio.h>
main (void)
{
int liczba1,liczba2;
clrscr();

printf("Podaj pierwszą liczbę: \n");
scanf("%d",&liczba1);

printf("Podaj drugą liczbę: \n");
scanf("%d",&liczba2);

if (liczba2==0)
      printf("Nie wolno dzielić przez 0!\n");
else
      printf("Wynik dzielenia: %f\n", (float) liczba1/liczba2);

getch();
return 0 ;
}
```

```
/* program if1a.c */
include <stdio.h>
#include <conio.h>

int main()
{
int a, b;
 clrscr();
 puts("Podaj 2 liczby calkowite ");
 printf("a = ");
 scanf("%i",&a);
 printf("b = ");
 scanf("%d",&b);


if( a > 10 )
 {
```

```
   printf("Zmienna a jest wieksza od dziesieciu !\n");
   printf("Jest bowiem rowna %d.", a);
  }
  else
  {
    printf("Zmienna a jest mniejsza lub rowna dziesiec !\n");

   if(a != 5) printf("Jednak nie jest rowna piec !");
  }

 printf("\n\nZmienna b = %d\n",b);
getch();
}
```

```
// program if1.cpp
#include <iostream.h>
int Latka;
int main() {
 cout << "Program dla doroslych\nIle ty masz lat?\n";
 cin >> Latka;
 if (Latka < 18) cout << "Spadaj, malolacie!\n";
 else cout << "Ty stary koniu! ;>\n";
}
```

Ogólna składnia if.. else

```
if (warunek_logiczny)
{
  instrukcje_do_wykonania
}
else if (inny_warunek_logiczny)
{
  instrukcje_do_wykonania
}
else
{
  instrukcje_do_wykonania
}
```

Po słowie kluczowym **if**, w nawiasie umieszcza się warunek.
Jeśli jest prawdziwy, program przechodzi do wykonywania kodu w klamrach.
Co jeśli warunek jest fałszywy? Wtedy program sprawdza kolejne warunki
(każdy następny to już nie **if**, lecz **else if**) aż do napotkania wartości prawdziwej, po której wykonywany jest kod w klamrach.
Jeśli wszystkie wartości będą fałszywe, program wykona kod umieszczony w klamrach po instrukcji **else**.

```
// Program if2.cpp C++
#include <iostream.h>

int main()
{
 int wiek = 0;
 char a;
 cout << "Podaj ile masz lat ";
 cin >> wiek;
```

```
  if (wiek > 18)
  {
    // Jeśli wprowadzona liczba jest większa od 18
    cout << "Jesteś już dorosły!" << endl;
  }
  else if (wiek < 18)
  {
    // Jeśli wprowadzona liczba jest mniejsza od 18
    cout << "Nie jesteś jeszcze dorosły!" << endl;
  }
  else
  {
    // Jeśli wprowadzona liczba jest równa 18
    cout << "Masz równo 18 lat więc jesteś już dorosły!" << endl;
  }

    cin >> a;  // wprowadzenie jakiegoś znaku

  return 0;
}
```

Z reguły jednak rozpatruje się zawsze kilka przypadków, a nie jeden lub dwa.
Jak to zrobić? Weźmy pod uwagę przypadek szukania pierwiastków dwumianu:

```
#include <stdio.h>
#include <conio.h>
main (void)
{
float A,B,C,delta;
clrscr();

printf("Podaj współczynnik A: \n");
scanf("%f",&A);
printf("Podaj współczynnik B: \n");
scanf("%f",&B);
printf("Podaj współczynnik C: \n");
scanf("%f",&C);

delta=b*b-4*a*c;

if (delta<0)
printf("Nie istnieją pierwiaski rzeczywiste!\n");
if (delta==0)
printf("Istnieje jeden pierwiastek rzeczywisty.\n");
if (delta>0)
printf("Istnieją dwa pierwiaski rzeczywiste.\n");

getch();
return 0 ;
}
```

Przy tak napisanym kodzie program sprawdzi wszystkie trzy podane warunki, niezależnie od tego, czy poprzedni warunek został spełniony czy też nie.
Gdyby jednak w stosownych miejscach umieścić instrukcję *else*, program sprawdziłby najpierw pierwszy warunek.
Gdyby nie był on spełniony program sprawdziłby drugi warunek, a kiedy ten również nie byłby spełniony program sprawdziłby trzeci warunek.

Poprawiony program:

```
#include <stdio.h>
#include <conio.h>
main (void)
{
float A,B,C,delta;
clrscr();

printf("Podaj współczynnik A: \n");
scanf("%f",&A);
printf("Podaj współczynnik B: \n");
scanf("%f",&B);
printf("Podaj współczynnik C: \n");
scanf("%f",&C);

delta=b*b-4*a*c;

if (delta<0)
printf("Nie istnieją pierwiaski rzeczywiste!\n");
else
if (delta==0)
printf("Istnieje jeden pierwiastek rzeczywisty.\n");
else
printf("Istnieją dwa pierwiaski rzeczywiste.\n");

getch();
return 0 ;
}
```

Jak widać, ostatnią instrukcję *if* można było usunąć, ponieważ nie istnieje więcej możliwych przypadków na znak delty niż trzy.
Warunek logiczny nie musi być warunkiem pojedynczym.
Można łączyć ze sobą kilka warunków w jedną całość zamiast dla każdego używać osobnej instrukcji *if*.

Na przykład, zamiast pisać:

```
if (warunek1)
  if (warunek2)
    printf("komunikat");
```

można napisać:

```
if ((warunek1)&&(warunek2)) printf("komunikat");
```

W obu przypadkach komunikat zostanie wyświetlony, jeśli obydwa warunki zostaną spełnione, jednakże w drugim przypadku kod jest bardziej elastyczny. Operator && (logiczne I) można zmienić do postaci || (logiczne lub):

```
if ((warunek1)||(warunek2)) printf("komunikat");
```

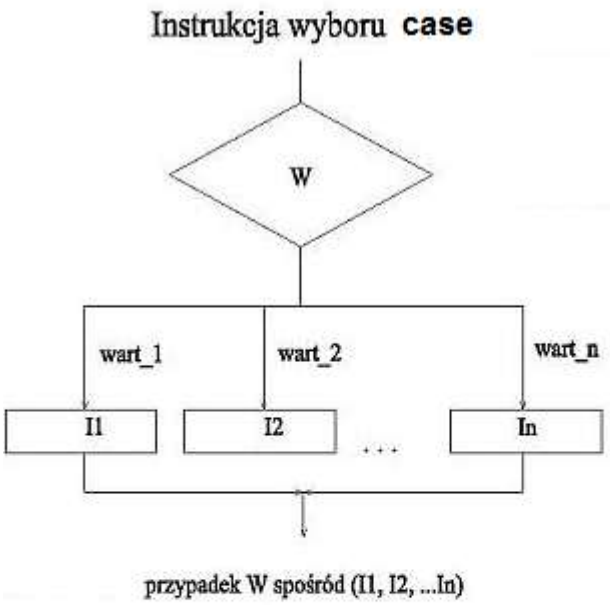
Wtedy wystarczy, aby tylko jeden warunek był spełniony, aby program wypisał komunikat. Wyrażenia takie można łączyć w bardziej skomplikowane, jednakże całość zawsze należy ująć w okrągły nawias.

Podsumowanie 4:

1. Jeżeli do rozpatrzenia mamy kilka przypadków, stosujemy instrukcję warunkową *if*.
2. Instrukcję zapisujemy: ***if (warunek) instrukcja1; else instrukcja2*.**
3. Dla więcej niż jednej instrukcji należy zgrupować je za pomocą nawiasów klamrowych.
4. W warunku logicznym instrukcji *if* zawsze stosujemy operator porównania ==, a nie przypisania =.
5. Jeśli jednocześnie powinno być sprawdzone kilka warunków, łączymy je za pomocą operatorów logicznych.

Instrukcja wyboru case dla wielu warunków

switch – case – przypadek W spośród (I1, I2, … In)



C i C++

```
switch (W) {
case wart_1: I1;
case wart_2: I2;
case wart_n: In;
default Instrukcja_awaryjna
}
```

```
switch (W) {
case wart_1: I1;  break
case wart_2: I2;  break
case wart_n: In;  break
default Instrukcja_awaryjna; break
}
```

Instrukcja wyboru **switch** (przełącznik) to tzw. zwrotnica wielokierunkowa.
Instrukcja ta pozwala na zdefiniowanie działań dla różnych wyników jednego wyrażenia.
Jej konstrukcja pozwala na łatwiejsze od zwykłej pętli definiowanie przypadków (większej ich liczby).

Instrukcją decyzyjną **switch** zastąpić można wielokrotne wywoływanie instrukcji warunkowej **if** np. dla różnych wartości tej samej zmiennej – przykładowo, gdy zmienna może przyjąć 10 różnych wartości, a dla każdej z nich należy podjąć inne działanie.

Składnia:

```
switch(wyrażenie_kluczowe)
{
  case wartosc_1:
    instrukcje;
  break;

  case wartosc_2:
    instrukcje;
  break;

  // ...

  default:
    cout << "Błąd";

}
```

Wyrażenie najczęściej jest zmienną o określonej wartości.
Jeśli tą wartością jest wartość1, wykonywane są instrukcje następujące po odpowiedniej etykiecie case aż do następnej instrukcji przerywającej, z reguły **break** (instrukcja opuszczenia nie musi występować na zakończenie każdego bloku rozpoczętego przez **case** – wykonany zostanie wtedy kod następnych przypadków).
Przypadek default jest opcjonalny, określa instrukcje wykonywane, gdy wartość zmiennej nie jest równa żadnemu z wyszczególnionych przypadków.

Przykład 1

```
#include <iostream>

int main()
{
  int liczba;
  cout << "Podaj liczbe: ";
  cin >> liczba;
  cout << endl;

  switch(liczba)
  {
    case 2: cout << "Dwa"; break; // opuszczenie switch - case
    case 4: cout << "Cztery"; break;
    case 8: cout << "Osiem"; break;
    default: cout << "Iles tam :)";
  }

}
```

Przykład 2

```
#include <stdio.h>
#include <conio.h>
```

```
main (void)
{
char znak;
clrscr();

printf("Wciśnij cyfrę od 0 do 5\n");
scanf("%c",&znak);
switch(znak)
{
case '0': pritntf("nacisnąłeś klawisz 0");break; //przerwanie switch
case '1': pritntf("nacisnąłeś klawisz 1");break;
case '2': pritntf("nacisnąłeś klawisz 2");break;
case '3': pritntf("nacisnąłeś klawisz 3");break;
case '4': pritntf("nacisnąłeś klawisz 4");break;
case '5': pritntf("nacisnąłeś klawisz 5");break;
default: pritntf("nacisnąłeś jakiś inny klawisz"); // pozostałe przypadki
}

getch();
return 0 ;
}
```

Instrukcja skoku goto

Przy omawianiu instrukcji s*witch-case* należy wspomnieć również o instrukcji skoku: *goto*.
Jest to bardzo przydatna instrukcja przy skomplikowanych programach.
Powróćmy do naszego przykładu z instrukcją *case*.
Po naciśnięciu odpowiedniego klawisza zostaje wykonana odpowiednia instrukcja.
A gdybyśmy chcieli, żeby wykonana została nie jedna czy dwie instrukcje, a 20?
Można oczywiście wypisywać je jedna po drugiej, ale kod szybko straciłby na czytelności.
Można się więc posłużyć instrukcją skoku, aby program na chwilę przeszedł do innego miejsca kodu, wykonał to co jest tam napisane i z powrotem powrócił do miejsca początkowego.
Jak oznaczyć miejsce, do którego program ma wyskoczyć?
Służą do tego **etykiety**.

Zmodyfikujmy powyższy przykład:

```
#include <stdio.h>
#include <conio.h>
main (void)
{
char znak;
clrscr();

jeszcze_raz: //to jest etykieta o nazwie jeszcze_raz

printf("Wciśnij cyfrę od 0 do 5\n");
scanf("%c",&znak);

switch(znak)
{
case '0': goto koniec; break;
case '1': goto jeden; break;
case '2': printf("nacisnąłeś klawisz 2"); goto koniec; break;
case '3': printf("nacisnąłeś klawisz 3"); goto koniec; break;
case '4': printf("nacisnąłeś klawisz 4"); goto koniec;  break;
case '5': goto jeszcze_raz;  break;
```

```
default: printf("nacisnąłeś jakiś inny klawisz");
}

jeden: //to jest etykieta o nazwie jeden
{
printf("\nEtykieta 1 ");
getch();
}
koniec: //to jest etykieta o nazwie koniec
printf("\nEtykieta koniec ");

getch();
return 0 ;

}
```

Działanie tego programu jest bardzo proste:
wciśnij klawisz => jaki klawisz został wciśnięty? =>
jeśli 0 => idź do etykiety 'koniec' i zacznij wykonywać wszystko od tego miejsca;
jeśli 1 => idź do etykiety 'jeden' i zacznij wykonywać wszystko od tego miejsca;
jeśli 2 => napisz komunikat a potem idź do etykiety 'koniec' i zacznij wykonywać wszystko od tego miejsca;
jeśli 3 => napisz komunikat a potem idź do etykiety 'koniec' i zacznij wykonywać wszystko od tego miejsca;
jeśli 4 => napisz komunikat a potem idź do etykiety 'koniec' i zacznij wykonywać wszystko od tego miejsca;
jeśli 5 => idź do etykiety 'jeszcze_raz' i zacznij wykonywać wszystko od tego miejsca;

Podsumowanie 5:

1. Instrukcję *if* stosujemy dla mniejszej ilości warunków do sprawdzenia, lub dla bardziej skomplikowanych warunków (wtedy stosujemy operatory logiczne np. && lub ||).
2. Instrukcję *case* stosujemy dla dużej ilości prostych warunków.
3. Określenia *else* (w instrukcji *if*) i *default* (w instrukcji *case*) znaczą: "dla pozostałych przypadków"
4. Instrukcja *goto* jest to instukcja skoku do pewnego miejsca w kodzie programu.
5. Miejsce skoku należy oznaczyć odpowiednią etykietą.
6. Za pomocą *goto* można robić pętle programowe lub szybko skończyć jego działanie.

Iteracje

Pętle: Powtarzaj, Dopóki, Dla

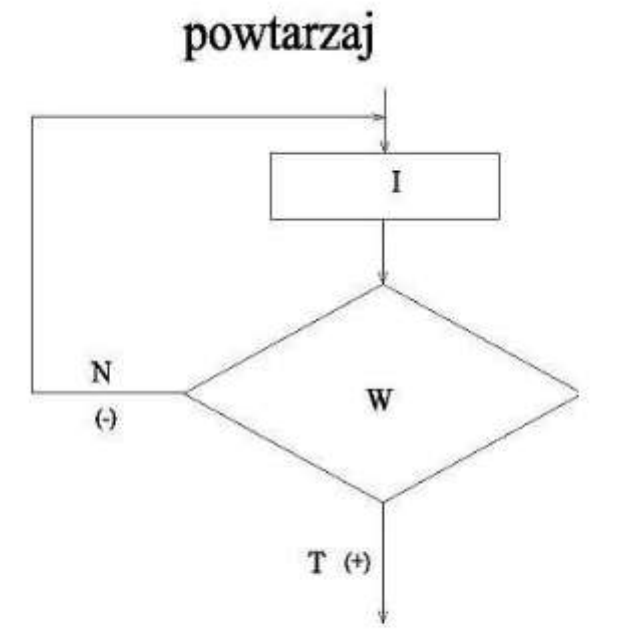
Powtarzaj

Pętla do while

Składnia instrukcji:

- **do instrukcja while (wyrażenie);**
- **do { lista instrukcji } while (wyrażenie);**

Instrukcja (może być złożona) jest wykonywana, jak długo (wyrażenie) jest różne od zera.
Instrukcja jest wykonana co najmniej raz - podstawowa różnica z while.



Pseudokod powtarzaj I aż do W;

C, C++

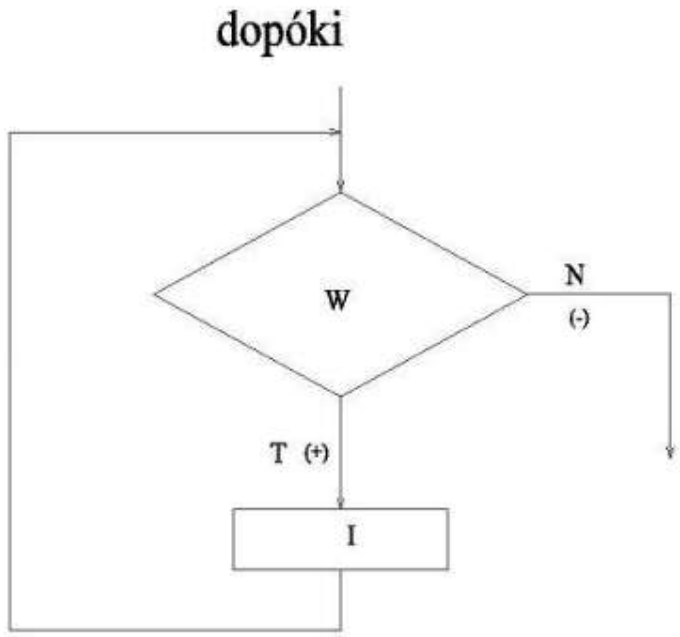
```
do
  I;
while (!W);
```

Pętla do while

```
/* program dowhile1.c */
include <stdio.h>
#include <conio.h>

void main(void)
{
  int licznik = 10;

  printf("Poczatek petli\n");

do
  {
    printf("Zmienna licznik = %d\n", licznik);
    licznik--;
  } while(licznik  != 0);

  printf("Koniec petli\n");
 getch();
}
```

```
Poczatek petli
Zmienna licznik = 10
Zmienna licznik = 9
Zmienna licznik = 8
Zmienna licznik = 7
Zmienna licznik = 6
Zmienna licznik = 5
Zmienna licznik = 4
Zmienna licznik = 3
Zmienna licznik = 2
Zmienna licznik = 1
Koniec petli
```

Pętla while



dopóki W wykonuj I;

---

while (W) I;

(warunek) *// dopóki warunek prawdziwy*
{ instrukcj1; instrukcja2; ... instrukcjaN;};
// wykonuj te instrukcje (listę instrukcji lub jedną instrukcję)

while (wyrażenie) instrukcja; // instrukcja pojedyncza (lub złożona)

```
while (wyrażenie)
  {
    lista instrukcji // instrukcja złożona (lista instrukcji)
  }
```

Z pętli while korzystamy, kiedy nie znana jest liczba powtórzeń i być może pętla ta wcale nie będzie wykonana.
Warunek jest testowany przed każdym, nawet pierwszym wykonaniem pętli,
Najpierw jest obliczana wartość wyrażenia (wyrażenie) i jeżeli ma ono wartość różną od zera (prawda), to jest wykonywana instrukcja, która może być instrukcją złożoną.
Instrukcja może nie być nigdy wykonana, jeżeli przy pierwszym obliczeniu wartości wyrażenia (wyrażenie) będzie ono miało wartość zero (fałsz).

Przykład - obliczenie sumy i iloczynu ciągu liczb zakończonych liczbą zero.

```
#include <iostream.h>

void main(void)
{
 int s, i, l, n; // suma, iloczyn ciagu liczb, liczba w ciagu

 s=0;
 i=1;
 n=0;

 cout << "Podaj liczby (zero kończy)\n ";
 cin >> l;

 while ( l!= 0) // while (l)
 {
  s += l;
  i *= l;
  n++;
  cin >> l;
 }

 cout << "Ilosc liczb: " << n << "\n";
 cout << "Suma liczb wynosi " << s << endl;
 cout << "Iloczyn liczb = " << i;
}
```

Wyniki:

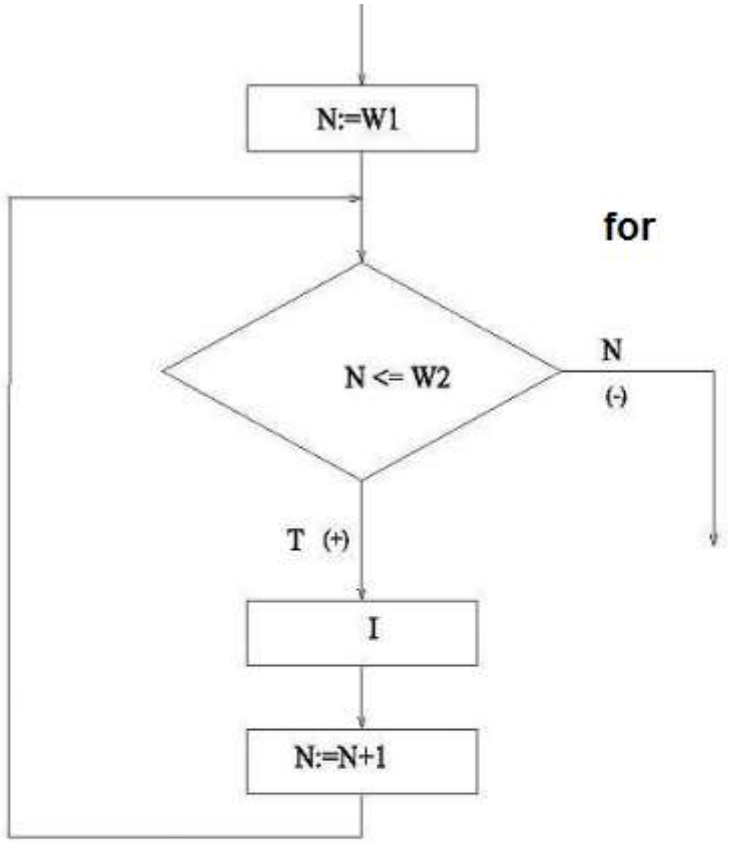
```
Podaj liczby (zero kończy)
1
2
3
4
0
Ilosc liczb: 4
Suma liczb wynosi 10
Iloczyn liczb = 24
```

Przykład

```
#include <stdio.h>
#include <conio.h>

void main(void)
{
 int licznik = 10;
```

```
 printf("Poczatek petli\n");

 while(licznik  != 0)
 {
  printf("Zmienna licznik = %d\n", licznik);
  licznik--;
 }
 printf("Koniec petli\n");
getch();

}
```

Instrukcja for



dla N:=W1 do W2 wykonuj I; Pseudokod

```
for (N:=W1; N<=W2; N++) I;
```

równoważna instrukcji C, C++

```
N=1;
while (N<W2)
{
I;
N=N+1;
}
```

Instrukcję for stosuje się w przypadkach, gdy z gór można określić liczbę wykonań pętli.
Jest ona chyba najczęściej używana.

Postać instrukcji for:

- **for (wyrażenie1; wyrażenie2; wyrażenie3) instrukcja;**
- **for (wyrażenie1; wyrażenie2; wyrażenie3) { lista instrukcji}**

Powyższe instrukcje for są równoważne konstrukcjom z while:

```
wyrażenie1;
while (wyrażenie2}
{
 instrukcja; // lub lista instrukcji
 wyrażenie3;}
```

Najpierw obliczana jest wartość wyrażenie1, co powoduje na ogół wyznaczenie wartości początkowej zmiennej sterującej.
Następnie jest obliczana wartość wyrazenie2 i jeżeli jest różna od zera (prawda) to jest wykonywana instrukcja (lista instrukcji) oraz wyrazenie3,
w którym na ogół następuje zmian wartości zmiennej sterującej pętli.

Przykłady:

1) Programy drukowania 5 liczb - od 1 do 5

```
// a) program drukowania 5 liczb
#include <iostream.h>

const int N=6;
void main(void)
{
 for (int i=1; i<N; i++)
  cout << i << endl;
}
```

Wydruk:

```
1
2
3
4
5
```

```
// b) program drukowania 5 liczb malejąco
#include <iostream.h>

const int N=6;
void main(void)
{
 for (int i=N; i>0; i--)
  cout << i << endl;
}
```

2) Tabliczka mnożenia

```
/* timestab.c */
main()
{
int row, column;
          puts("\t\tMy Handy Multiplication Table\n\n");
     for(row=1; row <=10;row++)
          {
                                        for(column=1; column<=10;column++)
                                        printf("%6d", row*column);
          putchar('\n');
          }
}
```

Wynik

```
My Handy Multiplication Table


     1     2     3     4     5     6     7     8     9    10
     2     4     6     8    10    12    14    16    18    20
     3     6     9    12    15    18    21    24    27    30
     4     8    12    16    20    24    28    32    36    40
     5    10    15    20    25    30    35    40    45    50
     6    12    18    24    30    36    42    48    54    60
     7    14    21    28    35    42    49    56    63    70
     8    16    24    32    40    48    56    64    72    80
     9    18    27    36    45    54    63    72    81    90
    10    20    30    40    50    60    70    80    90   100
```