Wprowadzenie do programowania w językach C i C++

Cel: Znajomość podstawowych zasad programowania w języku C i C++.

Osiągnięcia: praktyczna znajomość elementów programowania w językach C i C++

Tematyka

Wprowadzenie do programowania w języku C/C++:

- struktura programu,
- nazwy i słowa zastrzeżone,
- podstawowe typy danych,
- struktura funkcji,
- stałe, zmienne,
- deklaracje,
- wyrażenia, instrukcje we/wy,
- operatory,
- instrukcje sterujące,
- preprocesor #,
- tablice,
- wskaźniki.
- Wejście, wyjście w C++.

Literatura i materiały pomocnicze:

- **Język ANSI C**. Brian W. Kernighan, Dennis M. Ritchie. WNT 1997
- **Wprowadzenie do programowania w języku C**. Roman K. Konieczny
- **Języki C i C++. Twój pierwszy program.** Alan R. Neibauer.
- **Wygraj z C++.** Kris Jamsa.
- **Język C++. Bjorne Stroustroup.**
- **Nauka programowania dla początkujacych C++.** Anna Struzińska-Walczak, Krzysztof Walczak
- **Symfonia C++.** Jerzy Grębosz.
- **Podstawy C++.** Lippman.
- **Od C do C++Buildera w 48 godzin**. Adam Majczak
- **Ćwiczenia z języka C**. Claude Dellanoy.
- **Ćwiczenia z języka C++.** Claude Dellanoy.

Charakterystyka ogólna języka **C**

- Język C został zdefiniowany w 1972 r. przez Dennisa M **Ritchie** z Bell Laboratories w New Jersey.
- Pierwowzorem był język B, zdefiniowany 2 lata wcześniej przez Kena Thompsona. Język B był adaptacją języka BCPL dla pierwszej wersji instalacji SO UNIX.
- Język C jest językiem ogólnego zastosowania. Język w miarę prostoty, niezbyt obszerny. Charakteryzuje się także nowoczesnymi strukturami danych i bogatym zestawem operatorów.
- 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 - cechy, wersje

- 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.
- **Wersje języka C**:
  - Wersja 1 - zgodna ze standardem pierwotnym K & R
  - Wersja 2 - zgodna z ANSI 1989
- Wersje wg firm
  - Borland C oparty na ANSI C,
  - Microsoft C,
  - Watcom C,
  - Unix C,
  - C++
- W języku C podprogramami są funkcje.
- Zmienne lokalne i globalne.
- Łączy cechy wysokiego z możliwościami języka niskiego poziomu. Język w wysokim stopniu przenośny.
- Elastyczność - łagodniej kontroluje zgodność typów - kontrowersyjne, niebezpieczne.
  Zwięzłość - w Pascalu np. 10 linii, w C - kilka linii.
  Ze zwięzłością nie należy przesadzać - trudniej wykryć błąd.

WPROWADZENIE DO JĘZYKA C

Ogólna struktura programu w języku C

- Cechą języka **C/C+** jest możliwość budowy programu z **wielu modułów**. **Modułem** może być każdy zbiór zawierający poprawny kod źródłowy. Nie są potrzebne jak w Pascalu części opisowe i implementacyjne modułów.
- 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.
- Skompilują się również programy w postaci:
  main() { }
  void main() { }
  void main(void) { }

Ogólna **struktura** programu w C / C++

Nagłówek programu / komentarze */*

#include *(włączenia tekstowe - preprocesor)*

#define stałe makroinstrukcje

Zmienne globalne

Prototypy funkcji

Funkcja main()

Funkcje pozostałe

```
/* Przykładowy program P02c.c */
#include <stdio.h> /* preprocesor - załączenie pliku bibliotecznego stdio.h */
#include <conio.h>  /* załącz. pliku do clrscr() i getch() */
#define PI 3.14159  /* definicja PI */
#define P(a) a*a  /* definicja funkcji P(a) jako kwadrat liczby a */
int main(void)  /* funkcja główna */
{   /* klamra otwierająca  funkcji głównej*/
float r; /* deklaracja zmiennej rzeczywistej r */
clrscr(); /* czyszczenie ekranu */
puts("Obliczenie pola kola"); /* wyświetlenie łańcucha */
printf("Podaj promien kola: "); /* wyswietl. napisu */
scanf("%f",&r); /* wczytanie adresu promienia r */
printf("PI=%f\n",PI); /* wyświetlenie PI */
printf("Pole kola o promieniu %f = %10.4f\n", r, PI*P(r)); /*wydruk format.: r (całkowita. szer. 10, po kropce 2), pole, po kropce 4 miejsca), \n – nowa linia */
printf("Obwod = %6.3f \n",2*PI*r); /* wydruk obwodu */
puts("Nacisnij cos"); /* wyświetlenie łańcucha */
getch(); /* Czekanie na naciśniecie klawisza */
return 0;  /* funkcja main() zwraca 0 * - ostatnia przed  } */
}   /* klamra zamykająca  funkcji głównej */
```

Uwagi do **struktury** programu w C

- 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.

Przykład: program daneos.c

```
/* Program daneos.c  - dane osobowe*/
#include <stdio.h>
int main(void)
{
char imie[20]; /* zmienna imie - lancuch 20 znakow */
   int i;  / * zmienna calkowita */
printf("\nPodaj swoje imie ");
gets(imie); /* wczytanie imienia */
puts("Ile masz lat? ");
scanf("%d",&i); /* wczytanie wieku – liczba całkowita */
printf("\n%s ma %d lat.",imie, i); /*wydruk imienia i lat*/
return 0;
}
```

Przykłady prostych programów w **C** i **C++**

```c
/* W języku C*/
/* Prog1.c - Program napisany w
     pliku źródłowym Prog1.c */
#include <stdio.h>
int main()
{
printf("Dzień dobry\n");
     /* printf - drukuj - fragment
     biblioteki standardowej */
/* \n - znak nowego wiersza */
return 0;
}
```

```c
// W języku C++
// Prog1.cpp -  w pliku zrodl. Prog1a.cpp
#include <iostream.h>
int main ()
{
cout << ("Dzień dobry\n”);
return 0;
}
```

Preprocesor

- **preprocesor** - wstępny przetwarzacz - uwzględnia dodatkowe rzeczy np.

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

Przykład 2

```
/* Prog2.c lub Prog2.cpp */
#include <stdio.h>
void main()
{
printf(„Program , ");
printf(„drugi, nacisnij jakis znak ");
printf("\n");
getchar();
/* getchar wczytuje pojedynczy znak z klawiatury */
 }
```

Kompilacja programu w **C, C++**

- Język C/C++ wymaga **kompilacji**, w wyniku czego tworzy się plik wykonywalny, binarny, z rozszerzeniem EXE w systemach DOS, Windows.
- Nie ma interpreterów do języka C. Skompilować program wystarczy raz.
- Nie są ważne odstępy w programie. Muszą być całe słowa. Mogą być znaki spacji, nowego wiersza, tabulacji - nieważne ile. Słowo **inlcude** musi być w jednej linii.
- Ważne jest **projektowanie**. *Jedna godzina poświęcona na projektowanie pozwala zaoszczędzić kilka godzin programowania*.

Skompilowanie programu Prog1.c lub prog1.cpp

- **w DOS, Windows**:
  - W **Borland C**:
    bcc Nazwa_programu,
    np. bcc Prog1.c ==> Prog1.exe
    lub Compile albo Make
  - W **Turbo C**: **tcc** Nazwa_programu;
    Plik konfiguracyjny **Turboc.cfg**, np. -IC:\TC\INCLUDE -LC:\TC\LIB
  - przy pomocy **BCC**:
    BCC32 Nazwa_programu
    pliki konfiguracyjne:
    Bcc32.cfg, (np. -I"c:\Bcc55\include" -L"c:\Bcc55\lib")
    Link32.cfg, (np. -L"c:\Bcc55\lib„)
- **w UNIX**: **cc** Nazwa_programu –o Nazwa_programu_wynikowego
  - **cc** Prog1.c -o Prog1
  - lub
    gcc prog1.c -o Prog1

Proste przykład obliczeniowe - obliczenie objętości **walca**

```
/*****************************************/
/* Program p1.c */
/* Obliczanie objętości walca (wariant 1) */
/*--------------------------------------*/
#include <stdio.h>
main()
{ float promien, wysokosc, objetosc;
promien = 3.3;
wysokosc = 44.4;
objetosc = 3.1415926 * promien * promien *
    wysokosc; printf("Objetosc walca = %f",
    objetosc); }
/************************************/
```

Efektem wykonania będzie wydruk na ekranie:

Objętość walca=1519.010254

```
/*************************************************/ /*
Program p2.c */
/* Obliczanie objętości walca (wizytówka) */
/*-----------------------------------------------*/
/* Autor: Roman Konieczny */
/* Język programowania: C */
/* Komputer: IBM PC / AT (kompat.) */
/* Kompilator: Turbo C 2.0 */
/* Data ostatniej kompilacji: 30.03.1992 */
/*-----------------------------------------------*/
#include <stdio.h>
main()
{ float promien, wysokosc, objetosc;
promien = 3.3; wysokosc = 44.4;
objetosc = 3.1415926 * promien * promien *
wysokosc;
printf("Objetosc walca = %f", objetosc); }
/*************************************************/
```

Inne wersje programu na obliczenie **objętości walca**

```
/* Ten przykład pozbawiony
      komentarzy ale czytelny: */
/* p4.c */
#include <stdio.h>
main()
{ float promien, wysokosc, objetosc;
promien = 3.3; wysokosc = 44.4;
objetosc = 3.1415926 * promien *
      promien * wysokosc;
 printf("Objetosc walca = %f",
      objetosc);
}
```

```
/* Ten przykład jest bardzo zwięzły w
zapisie ale mało czytelny: */
/*p5.c*/
#include<stdio.h>
main() {float r,h,v;r=3.3;h=44.4;
v=3.1415926*r*r*h; printf("V=%f",v);}
```

Wniosek - brak jak i nadmiar komentarzy nie służy czytelności programu.

Stawianie klamer { } w programie

- *W praktyce stosowane są 2* ***sposoby umiejscawiana klamer*** *{ },*
  ograniczających sekwencje instrukcji.

1)

```
main()
{
   /* sekwencja instrukcji */
}
```

2)

```
main() {
   /* sekwencja instrukcji */
   }
```

- Na czytelność programu wpływa stosowanie **wcięć**.

Przeliczenie stopni Celsjusza na Fahrenheita

```
/* Program sz2.cpp - komentarz
     Przelicza stopnie Fahrenheita na
     Celsjusza */
#include <stdio.h>
void main()
/* Początek funkcji głównej main - nic nie
     zwraca*/
{
/* Definicje zmiennych */
int tf=0; /* typ integer - temp.
     Fahrenheita*/
float tc; /* typ float - temp Celsjusza */
while (tf <=100) /* pętla - iteracja tf
     używana też do sterowania
     przebiegiem pętli */
  {
     tc=(5*(tf-32))/9.0;
     printf("%4d %6.1f \n",tf,tc);
     tf=tf+10;
  }; /* while */
} /* main */
```

```
/* Program sz2a.cpp - bez zmiennej tc
Przelicza stopnie Fahrenheita na Celsjusza
*/

#include <stdio.h>

void main() /* Początek funkcji głównej main
- nic nie zwraca*/

{

/* Definicje zmiennych */

int tf=0; /* typ integer - temp. Fahrenheita*/

while (tf <=100) /* pętla - iteracja */

      { printf("%4d %6.1f \n",tf, 5*(tf-32)/9.0);
      tf=tf+10; }; /* while */

} /* main */
```

Wersja programu przeliczenia temperatury z "Język ANSI C" str. 31/32

```
#include <stdio.h>
#include <conio.h>
main()
{
 float fahr, celsius;
 int lower, upper, step;
 lower =0; /* dolna granica temp */
 upper=300; /* górna granica temp */
 step=20; /* krok */
 fahr=lower;
 clrscr(); /* czyszczenie ekranu */
 printf("Zamiana stopni Fahrenheita na Celsiusza\n");
 printf(" F Cels\n");
 while (fahr<=upper)
 {
   celsius=(5.0/9.0)*(fahr-32.0);
   printf("%3.0f %6.1f\n",fahr,celsius);
   fahr=fahr+step;
 }
 getch(); /* getch reads a single character directly from the keyboard, without echoing to the screen */
 return 0;
}
```

Formaty wydruku:

%d, %i - **liczba całkowita**

%x - liczba w formacie **16** bez znaku

%f, %e, %g - **liczba** rzeczyswita - tryb zmiennopozycyjny

%s - **string** – łańcuch znaków

%c - **znak**

%o - ósemkowo

PODSTAWOWE ELEMENTY JĘZYKA C

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 być litera.
- Znak podkreślenia traktowany jest jako litera.
- W języku C **rozróżniane** są duże i małe litery w identyfikatorach.

Przykład użycia różnych nazw:

```
/*********************************************************/
/*  Program p6.c */
/* Obliczanie objętości walca (rozróżnianie 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);
} /*********************************************************/
```

- Efektem wykonania programu są 2 wydruki:
- Objętość **walca** = 1519.010254 OBJETOSC **WALCA** = 6283.185059
- Użycie odstępu w nazwie jest niedozwolone.
- Niektóre implementacje rozpoznają w nazwie do 8 znaków, inne więcej (do 32)

Słowa zastrzeżone - kluczowe

- Są to słowa o szczególnym znaczeniu dla języka, których nie wolno używać programiście np. jako nazw zmiennych.
  - **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**
- W C++ dodano następujące słowa kluczowe:
  - **catch, cin, class, cout, delete, friend, inline, new, operator, private, protected**
- 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 – 2 bajty
  - **float** - typ zmiennopozycyjne pojedynczej precyzji – 4 bajty
  - **double** - typ zmiennopozycyjny podwójnej precyzji – 8 bajtów
- Dodatkowo występuje kilka **kwalifikatorów** stosowanych z tymi podstawowymi typami. Kwalifikatory **short** i **long** odnoszą się do obiektów całkowitych

Tabela typów

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 znaku	0...65535	2 bajty
long (long signed int)	liczba całkowita długa	2147483648... 2147483647	4 bajty
unsigned long (long unsigned int)	liczba całkowita długa 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

Inne typy danych

- Szczegóły zależne są od konkretnej implementacji kompilatora języka C.
- Są jeszcze inne typy danych jak:
  - **void** (typ funkcji nie zwracającej wartości),
  - **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.

Zmienne, stałe, deklaracje, wyrażenia

- **Zmienne** (deklarujemy np. int ilosc, nr; float cena;)
  i **stałe** (np. #define PI 3.14) – deklarowane na początku lub wewnątrz funkcji, np. const float PI=3.14;)
  są podstawowymi obiektami danych, jakimi posługuje się program.
  Deklaracje wprowadzają potrzebne zmienne oraz ustalają ich typy i ewentualnie wartości początkowe.
- **Operatory** (np.. +, -, *, /, %, ||, &&, !) określają co należy z nimi zrobić.
- **Wyrażenia** wiążą zmienne i stałe, tworząc nowe wartości.

Stałe w C

- W C występują 4 rodzaje stałych:
  - stałe całkowitoliczbowe,
  - stałe rzeczywiste,
  - stałe znakowe oraz **łańcuchy** znaków.
- Wartość 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 **0**111 **0**777 -0777 +0222
Stałe szestnastkowe	**0..9 a..f A..F + -**	Pierwszymi znakami muszą być **0x** lub **0X**	**0x** 0X1234 **0X**AFDEC 0xffff

Inicjowanie zmiennych typów int i long

- W celu zainicjowania zmiennych typów **int** i **long** po znaku równości podajemy całkowitą stałą liczbową.
  Przykłady:
  - int l=100; */* stała całkowita */*
  - unsigned k=121; */* stała całkowita unsigned – bez znaku */*
  - int la = 0x2ffa; */* stała całkowita szestnastkowa */*
  - long i = 25L; */* stała całkowita typu long */*
  - long unsigned z = 1000lu; */* lub 1000UL - unsigned long – bez całkowita długa znaku */*
- Stała całkowita, jak np. 1234 jest obiektem typu int.
- W stałej typu long na końcu występuje litera **l** lub **L**, jak 123456789**L**
- Stała całkowita nie mieszcząca się w zakresie int jest traktowana jako stała typu long.
- W stałych typu unsigned na końcu występuje **u** lub **U**
- Końcówka ul lub **UL** oznacza stałą typu unsigned long.

Stałe rzeczywiste

- **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** lu **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 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).
- Zestaw dozwolonych znaków: wszystkie „widoczne „ znali ASCII,
  np. : 'A' '#‘ ' ‘ */* (‘ ‘ to spacja) */*
- Przykład deklaracji: 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’

Sekwencje specjalne c.d.

- 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‘, np. definicja: char lf='\n';
- Ogólny format takiego zapisu:

  \ooo - dla systemu ósemkowego (o - cyfra 8-wa)

  \xhh - dla systemu szestnastkowego (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 null 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 napisu:

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\fAndrzeh\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++„

Napisy mogą być sklejane podczas kompilacji programu:

"ahoj," "przygodo" jest równoznaczne "ahoj, przygodo".

Ta możliwość jest użyteczna przy dzieleniu długich napisów na wiersze w programie źródłowym.

Typ **wyliczeniowy**, stała wyliczeniowa: **enum**

- **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.

- Przykład:
  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};

  /* (luty jest 2-gi itd.) */

Przykłady z **enum – tryb wyliczeniowy**

```c
/* Program w C */
#include <stdio.h>
#include <stdlib.h>
enum srodekTransportu {SAMOCHOD, TRAMWAJ, AUTOBUS,
        ROWER, NOGI};
void tankuj (enum srodekTransportu pojazd
)
{
   if ( pojazd == SAMOCHOD )
      printf("Samochod zatankowany do pelna!\n");
   else
      printf("Przeciez nie jade samochodem, wiec co mam
        zatankowac?\n");
   return;
}

int main ()
{
 enum srodekTransportu sposob;
 sposob = rand()%5;  /* losowanie sposobu dotarcia do pracy */
   switch (sposob)
   {
      case SAMOCHOD:
       printf("Pojade dzis sobie samochodem!\n");break;
      case AUTOBUS: case TRAMWAJ:
      printf("Skorzystam dzis z transportu publicznego!\n"); break;
      case ROWER:
      printf("Pojade dzis sobie rowerem!\n");  break;
      default:
      printf("Pojde na piechote!\n"); break;
   }
   tankuj(sposob);
   getchar();
   return 0;
}
```

```c
// C++
#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <conio.h>
 enum srodekTransportu {SAMOCHOD, TRAMWAJ, AUTOBUS, ROWER, NOGI};
void tankuj (srodekTransportu pojazd)
{
    if ( pojazd == SAMOCHOD )
   cout<<"Samochod zatankowany do pelna!"<<endl;
    else
cout<<"Przeciez nie jade samochodem, wiec co mam zatankowac?"<<endl;
   return;
}
int main ()
{
   srodekTransportu sposob;     sposob = SAMOCHOD;    /* wybranie
sposobu dotarcia do pracy */
   clrscr();
   switch (sposob)
   {
      case SAMOCHOD:   cout<<"Pojade dzis sobie samochodem!"<<endl;
              break;
      case AUTOBUS: case TRAMWAJ:
        cout<<"Skorzystam dzis z transportu publicznego!"<<endl;     break;
      case ROWER:
       cout<<"Pojade dzis sobie rowerem!"<<endl;    break;
      default:
              cout<<"Pojde na piechote!"<<endl;     break;
   }
   tankuj(sposob);
   getch();
   return 0;
}
```

Stała symboliczna - #define

- **Stała symboliczna** to stała reprezentowana przez nazwę, jak w przypadku zmiennych. Jednak w przeciwieństwie do zmiennych, po inicjalizacji stałej, jej wartość nie może być zmieniona.
- Stała symboliczna jest nazwą przedstawiającą inną stałą - numeryczną, znakową lub tekstową.
- Definicję stałej symbolicznej umożliwiają:
  instrukcja – dyrektywa **#define** – w języku **C**
  lub modyfikator **const** - w języku **C++.**
- #define NAZWA_STALEJ WARTOSC
  gdzie NAZWA_STALEJ jest nazwą stałej symbolicznej, a WARTOSC to tekst związany z tą nazwą łańcuchem znaków

  Np. #define znak 'A' /* bez średnika */
- Kompilatory **C++** oraz niektóre kompilatory C zapewniają deklarowanie stałych przy pomocy **const**.
  Korzystając z **const** deklaruje się stałą, określa jej typ i przypisuje wartość.
  Modyfikator const umieszczamy jednak wewnątrz funkcji, a nie jak dyrektywę #define przez main().
  Przykład: main() { const int DZIECI=8; char INIT='C'; …}

  To samo osiągniemy deklarując:

  #define DZIECI 8

  #define INIT 'C'

Przykłady z **#define**: **makrodefinicje** proste:

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"
    /* rozwinięcie daje tekst "Andrzej + Zaleski " */
#define WCZYTAJ_IOSTREAM_H    #include <iostream.h>
```

#define - makrodefinicje **parametryczne**

- **#define identyfikator(idPar1, idPar2,...) ciąg_jedn_leksykalnych**

  Przykłady:

```
#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 objętości walca  ( #define )            */
/*-------------------------------------------------------*/
#include <stdio.h>

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

main()
{
   float promien, wysokosc, objetosc;
   promien  = PROMIEN;
   wysokosc = WYSOKOSC;
   objetosc = PI * promien * promien * wysokosc;
   WYNIK;
}
/*******************************************************/
```

Wynik programu:
Objętość 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.
- 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.
  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 należy zwrócić uwagę na 2 zasadnicze rzeczy:
  - **nadanie wartości początkowej zmiennym**
  - **oszacowanie zakresu zmienności**

Przykład z zastosowaniem zmiennych – objętość walca - błędy

```
/*  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  przekroczony dopuszczalny zakres int  32767 */
   printf("\ni = %i ", i);
}
```

Efekt wykonania:

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 automatyczne i zewnętrzne, deklaracja, nadawanie wartości początkowych

- **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 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;

Zmienne – nadawanie wartości

- 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 początkową **zero**.
- Zmienne automatyczne bez jawnie określonej wartości początkowej 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 z **deklaracjami zmiennych**. Niektórym nadano wartości początkowe.

```
main()
{
 /* Komentarz  (znaczenie slow kluczowych)
     typy danych:
char     - znak (bajt),
int      - liczba calkowita,
float    - liczba rzeczywista,
double   - liczba rzeczywista podwojna,
modyfikatory:
short   - krotki,  long  - dlugi,
signed  - ze znakiem,  unsigned - bez znaku      */

/* --- typy znakowe ------------------------------- */
char  c1  ='k';
signed char   c2  ='\123';
unsigned char  c3  ='\xAB';

/* --- tablice znaków ----------------------------- */
 char   tekst[]   = "Katowice";
unsigned char   tekst[30] = "Taki sobie tekst to jest!";
```

W deklaracji typu text[]="... " kompilator sam sobie dobiera długość pola tablicy, dbając też, by ostatnim znakiem był **null** (\0).
W przypadku deklaracji explicite, np. text[20]="...", gdy łańcuch będzie dłuższy od 20 to nastąpi jego obcięcie, a gdy będzie krótszy, to niewykorzystane miejsca zostaną wypełnione albo znakami pustymi albo po znaku \0 będą wartości przypadkowe (nieistotne).

```
/* --- typy liczb całkowitych ---------------------- */
  int                   i1  = 0;
  signed int            i2  =-32767;
  unsigned int          i3;
  signed                i4  = 32767;
  unsigned              i5  = 32767;
  short                 i6  = 3;
  signed short          i7  =-3;
  unsigned short        i8  = 44;
  short int             i9  = 22;
  signed short int      i10 =-555;
  unsigned short int    i11 = 1;
  long                  i12 = 11111;
  signed long           i13 =-1;
  unsigned long         i14;
  long int              i15 = 22222;
  signed long int       i16;
  unsigned long int     i17;

  /* --- typy liczb rzeczywistych -------------------- */
  float                 r1 =-3.4E38;
  signed float          r2 = 3.4E38;
  unsigned float        r3 = 5.5E38;
  long float            r4 = 1.7E308;
  double                r5 = 1.7e308;
  long double           r6 = 1.1e4932;
  float                 r7 = 1234.56789;

  /*  dalsza ( robocza ) cześć programu ... */
}
```

Operatory

- Operatory łączą stałe i zmienne.
- Kolejność wykonywania działań określają nawiasy i priorytet operatorów.
- **Operatory**
  - **Arytmetyczne**:
    Są operatory dwuargumentowe – wykonują działania na 2 argumentach, np. x-y
    Istnieją **też 2 operatory** jednoargumentowe **+** oraz **–** służące do określania znaku liczby. Operator **–** służy ponadto do zmiany znalu wartości zmiennej.
  - Operatory **relacyjne dwuargumentowe**: <, <=, >, >=, =>, !=
  - Operatory **logiczne**: !, &&, |
  - **Bitowe operatory logiczne**: &, |, ~, <<, >>,
  - Operatory **zwiększania** ++ i **zmniejszania** **--**, np. x = ++aa; x=aa+;
  - **Wieloznakowe operatory przypisania**:
    =+ (a=+b; ➔ a=a+b;) , -=, *=, /=, %=, MM=, &=, !=, ~=

Operatory arytmetyczne

Wyrażenia 2-argumentowe: +, -, *, /, %

+ dodawanie

- odejmowanie

***** mnożenie

/ dzielenie

% reszta z dzielenia liczb całkowitych - operator dzielenia modulo

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

Operatory relacyjne i operatory logiczne

- Operatory **relacji**:
  > >= < <= *(mają ten sam priorytet)*
- Operatory **przyrównania**:
- **==** *(równe)*
- **!=** *(różne)* *(priorytet niższy)*
- Operatory **logiczne**:
- **&&** *(and - i)*
- **||** *(or - lub)*

Wyrażenia

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

```
3.2             /* samodzielny element – liczba */
PI              /* samodzielny element – stała*/
a=a+b;
x=y;
x<ALFA;
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**:

  1) 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); }

  2) Instrukcje sterujące, warunkowe: if (wyrażenie) instrukcja; if (warunek) instrukcja; If (wyrażenie) {lista_instrukcji1} else {lista_instrukcji2}

  switch (w1) {case etyk1: instr1; case etyk2: instr2; … default: instr_awaryjne; }

  Pętle: while (wyrażenie) instrukcja; while (wyrażenie) {lista_instr}

  do instrukcja; while (wyrazenie); do {lista_instr} while (wyrażenie);

  for (wyrażenie1; wyrażenie2; wyrazenie3) instrukcja;

  for (wyr1; wyr2; wyr3 {lista_instr}

  break; continue; goto etykieta; exit;

  3) Instrukcje wywołania funkcji, np. k=funkcja1(x,y); znak=getchar();

  printf(…); scanf(…); puts(…)

Przykład instrukcji **złożonej** (grupującej)

```
#include <stdio.h>
#define PI          3.1415926
#define PROMIEN     3.3
#define WYSOKOSC    44.4
#define KRYTERIUM 1500.0
#define WSP         2.2
#define WZOR_OBJETOSC  PI * promien * promien * wysokosc
main()
{
  float   promien, wysokosc, objetosc;
  promien  = PROMIEN;
  wysokosc = WYSOKOSC;
  objetosc = WZOR_OBJETOSC;
  printf("Objetosc walca = %f", objetosc);

  if (objetosc > KRYTERIUM)
    {
      float  objetosc,  promien = PROMIEN * WSP;
      objetosc = WZOR_OBJETOSC;
      printf("\nObjetosc walca = %f  (lokalna)", objetosc);
    }

  printf("\nObjetosc walca = %f", objetosc);
}
```

Efekt wykonania programu:

Objetosc walca = 1519.010254
Objetosc walca = 7352.010742 (lokalna)
Objetosc walca = 1519.010254

Instrukcje sterujące: warunkowe, pętle …

- C jest językiem imperatywnym - oznacza to, że instrukcje wykonują się jedna po drugiej w takiej kolejności w jakiej są napisane.
- Aby móc zmienić kolejność wykonywania instrukcji potrzebne są instrukcje sterujące.
- Instrukcje warunkowe:
- **if** (wyrażenie) instrukcja; if (warunek) instrukcja;
- **If** (wyrażenie) {lista_instrukcji1} **else** {lista_instrukcji2}
- **switch** (w1) {case etyk1: instr1; **case** etyk2: instr2; … default: instr_awaryjne; }
- Pętle:

while (wyrażenie) instrukcja; while (wyrażenie) {lista_instr}

do instrukcja; while (wyrazenie); do {lista_instr} while (wyrażenie);

for (wyrażenie1; wyrażenie2; wyrazenie3) instrukcja;

for (wyr1; wyr2; wyr3 {lista_instr}

- Instrukcje

– break; continue; goto etykieta; exit;

Instrukcja **if**, **if…else**

Instrukcja **if** służy do podejmowania decyzji w programie.

Najprostsza postać: **if (wyrażenie) instrukcja;**

Jeżeli wartość wyrażenia jest różna od zera, wówczas jest wykonywana instrukcja, a w przeciwnym przypadku nie.

Użycie instrukcji if :

```
if (wyrażenie)
    {
  /* blok wykonany, jeśli wyrażenie jest prawdziwe */
}
/* dalsze instrukcje */
```

Instrukcja if może także przyjąć postać: **if (wyrażenie) instrukcja1; else instrukcja 2;**

Jest wtedy możliwość reakcji na nieprawdziwość wyrażenia - wtedy należy zastosować słowo kluczowe else:

```
if (wyrażenie)
    {
  /* blok wykonany, jeśli wyrażenie jest prawdziwe */
} else {
  /* blok wykonany, jeśli wyrażenie jest nieprawdziwe */
}
/* dalsze instrukcje */
```

Przykłady z **if** w C i C++

/* przyk1.c */

```
#include <stdio.h>
 int main ()
{
  int a, b;
  a = 3;
  b = 5;
  if (a==b)  /* jeśli a równa się b */
  {
    printf ("a jest równe b\n");
  }
       else  /* w przeciwnym razie */
  {
    printf ("a nie jest równe b\n");
  }
  return 0;
}
```

/* prz1.cpp */

```
#include <iostream.h>

 int main ()
{
  int a, b;
  a = 3;
  b = 5;
  if (a==b)  /* jeśli a równa się b */
  {
    cout << "C++  a jest równe b\n";
  }
 else  /* w przeciwnym razie */
  {
    cout <<"C++  a nie jest równe b\n";
  }
  return 0;
}
```

```
// Program p_if1.cpp
#include <iostream.h>
int x, y;
void main(void)
{
x=1;
if (x) y=10; // przypisanie zostanie dokonane
cout << " x = " << x << " y= " << y << '\n';
x=0;
if (x) y=5; // przypisanie nie zostanie dokonane
cout << " x = " << x << "  y = " << y;
}
```

Organizacja obliczeń cyklicznych. Pętle while, do while, for

- Podstawową umiejętnością, jaką musi posiadać programista, jest organizacja obliczeń cyklicznych.
- Obliczenia cykliczne w C i C++ wykonuje się przy pomocy instrukcji **while**, **do while** oraz **for**.
- Pętle:

  while (wyrażenie) instrukcja;

  while (wyrażenie) {lista_instr}

  do instrukcja; while (wyrazenie);

  do {lista_instr} while (wyrażenie);

  for (wyrażenie1; wyrażenie2; wyrazenie3) instrukcja;

  for (wyr1; wyr2; wyr3 {lista_instr}

Instrukcja **while**

Często zdarza się, że nasz program musi wielokrotnie powtarzać ten sam ciąg instrukcji.

Aby *nie przepisywać wiele razy tego samego kodu* można skorzystać z tzw. **pętli**. Pętla wykonuje się dotąd, dopóki prawdziwy jest warunek – jak długo wartość warunek jest różna od zera.

Postać instrukcji while:

```
while (wyrażenie) instrukcja;
```

lub inaczej

```
while (warunek)
{
  /* instrukcje do wykonania w pętli */
}
/* dalsze instrukcje */
```

Najpierw jest obliczana wartość wyrażenia (warunek) i jeżeli ma ono wartość różną od zera (prawda) to jest wykonywana instrukcja, która może być instrukcją złożoną. Instrukcja może się nigdy nie wykonać, jeśli przy pierwszym obliczeniu wyrażenie (warunek) miało wartość zero (fałsz).

Przykłady z while

/* while1.c */

```
/* obliczenie kwadratów liczb od 1 do 10 */
#include <stdio.h>
void main ()
 {
   int i = 1;
   while (i <= 10)
   {  /*  początek while - dopóki i nie przekracza 10 */
     printf ("%d\n", i*i); /* wypisz i*i na
     ekran*/
     i=i+1;  /* lub ++i;  - zwiększamy i o jeden*/
    }   /* while */
  getchar();
}
```

// while2.cpp

```
// Program oblicza sumę i iloczyn liczb zakończonych liczbą 0
#include <iostream.h>

void main (void)
 {
   int suma=0, iloczyn=1, liczba;
   char znak;
   cout << "Podaj liczbe, 0 - koniec ";
   cin >> liczba;

   while (liczba != 0)
   {
    suma    += liczba; // suma = suma + liczba;
    iloczyn *= liczba; // iloczyn = iloczyn*liczba;
    cout << "Podaj liczbe, 0 - koniec ";
    cin >> liczba;
   }  // while

 cout << "\nSuma liczb wynosi " << suma << endl;
   cout << "Iloczyn liczb wynosi " << iloczyn << endl;
   cout << "\nWprowadz jakis znak "; cin >> znak;
}
```

Instrukcja do..while

- Pętle **while** (oraz for) mają jeden zasadniczy mankament - może się zdarzyć, że nie wykonają się ani razu.
  Aby mieć pewność, że nasza pętla będzie miała co najmniej jeden przebieg musimy zastosować pętlę **do while**.
- Składnia instrukcji **do while**

do instrukcja; while (wyrażenie);

- Instrukcja (może być złożona) jest wykonywana tak długo, jak długo wartość wyrażenia jest rózna od zera (prawda). W przypadku, gdy wartość ta będzie 0 (fałsz) to wykonywanie instrukcji kończy się. Wynika stąd, że instrukcja jest zawsze wykonywana co najmniej jeden raz i to jest podstawowa różnica z instrukcją while.
- Po uwzględnieniu, że instrukcja jest złożona (wiele instrukcji w pętli) składnia **do while** wygląda następująco:

```
do
{
  /* instrukcje do wykonania w pętli */
} while (warunek);
/* dalsze instrukcje */
```

Podsumowując, zasadniczą różnicą pętli **do while** jest to, iż sprawdza ona warunek pod koniec swojego przebiegu.

To właśnie ta cecha decyduje o tym, że pętla wykona się co najmniej raz

Przykłady funkcji do while

```
/* program dowhile1.c */
/* liczy szesciany liczb od 1 do 10 */

#include <stdio.h>

 void main ()
 {
   int a = 1;
   puts("Program liczy szesciany liczb od 1 do 10");

   do
   {
     printf ("Liczba %5d,  szescian liczby %7d\n", a,
       a*a*a);
     ++a; /* a=a+1; */
   } while (a <= 10);

  puts("\nNacisnij Enter  ");
  getchar();
 }
```

```
/* W tej pętli zamiast bloku instrukcji
zastosowano pojedynczą instrukcję */
/* program dowhile1a.c */

#include <stdio.h>
void main ()
 {
   int a = 1;
   do printf ("%d\n", a*a*a); while (++a <= 10);
   getchar();
 }
```

Instrukcja **for**

Od instrukcji while czasami wygodniejsza jest instrukcja for. Umożliwia ona wpisanie ustawiania zmiennej, sprawdzania warunku i inkrementowania zmiennej w jednej linijce co często zwiększa czytelność kodu.

Instrukcję for stosuje się, gdy można z góry określić liczbę wykonań pętli. Stosuje się zwłaszcza do wykonywania operacji na tablicach

Postać instrukcji for: **for (wyrażenie1; wyrażenie2; wyrażenie3) instrukcja;**

Powyższa instrukcja jest równoważna konstrukcji:

```
wyrazenie1;
while (wyrazenie2)
{
Instrukcja;
Wyrazenie3;
}
```

Jeśli instrukcja jest złożona (więcej niż jedna to zapisujemy następująco:

```
for (wyrażenie1; wyrażenie2; wyrażenie3)
{
  /* instrukcje do wykonania w pętli */
 }   /* dalsze instrukcje */
```

Pętla for różni się od tego typu pętli, znanych w innych językach programowania.

Opis co oznaczają poszczególne wyrażenia:

wyrażenie1 - jest to instrukcja, która będzie wykonana przed pierwszym przebiegiem pętli.

Zwykle jest to inicjalizacja zmiennej, która będzie służyła jako "licznik" przebiegów pętli.

wyrażenie2 - jest warunkiem zakończenia pętli. Pętla wykonuje się tak długo, jak prawdziwy jest ten warunek.

wyrażenie3 - jest to instrukcja, która wykonywana będzie po każdym przejściu pętli. Zamieszczone są tu instrukcje, które zwiększają licznik o odpowiednią wartość.

Przykłady z for

```
/* program For1a.c  */
#include <stdio.h>

void main ()
{
 int i;

 for(i=1; i<=10; ++i)
 {
  printf("%3d", i);
 }
 printf("\n");

 for(i=1; i<=10; ++i)
  printf("%3d", i);

 printf("\n");

 for(i=1; i<=10; printf("%3d", i++ ) );
  printf("\n\n");

 for(i=10; i>=1; printf("%3d", i-- ) );
 printf("\n");
 getchar();
 }
```

```
/* Program for2.c  srednia arytmet */
#include <stdio.h>
#include <conio.h>
main()
{
int n;
double suma;
float liczba;
clrscr();
printf("Srednia wprowadzonych liczb\n");
printf("Wprowadz kolejne liczby, Ctrl Z lub F6 - koniec\n");;
for(n=0, suma=0;   scanf("%f",&liczba)==1;  suma += liczba,  n++);
if(n==0)         printf("ciag pusty\n");
else
printf("liczba elementow = %5d  srednia = %lf\n",n,suma/n);getch();
return 0;
}
```

```
//  Program  co_piata.cpp
#include  <iostream.h>
#include <conio.h>
void main (void)
{
 clrscr();
 int licznik;

 for (licznik=1; licznik <=100; licznik +=5)    cout << licznik << ' ' ;

  getch();
}
```

Instrukcja **switch**

Jeśli zachodzi konieczność podjęcia decyzji w sytuacjach *wielowariantowych* – istnieje wiele możliwych dróg wyboru, wówczas stosujemy instrukcję **switch**.

Stosuje się *aby ograniczyć wielokrotne stosowanie instrukcji if.*

switch (w1) {case etyk1: instr1; **case** etyk2: instr2; … default: instr_awaryjne; }

Użycie instrukcji **switch**:

```
 switch (wyrażenie)
{
   case wartość1: /* instrukcje, jeśli wyrażenie == wartość1 */
     break;
   case wartość2: /* instrukcje, jeśli wyrażenie == wartość2 */
     break;
   /* ... */
   default: /* instrukcje, jeśli żaden z wcześniejszych warunków nie został spełniony */
     break;
 }
```

Należy pamiętać o użyciu **break** po zakończeniu listy instrukcji następujących po case

Jeśli tego nie zrobimy, program przejdzie do wykonywania instrukcji z następnego case.

Może mieć to fatalne skutki.

Przykłady **switch** – w C i C++

/* program switch1.c */

```
#include <stdio.h>

 void main ()
 {  /* main */
   unsigned int dzieci, przyp=0;    float podatek=1000.0;
   printf("Podaj ilosc dzieci "); scanf("%u",&dzieci);

   switch (dzieci)
  {
     case  0: break; /* brak dzieci - czyli brak ulgi */
     case  1:  /* ulga  2% */
      {
       podatek = podatek - (podatek/100* 2);  przyp=1;      break;
      }
     case  2:  /* ulga  5% */
     {
       podatek = podatek - (podatek/100* 5);  przyp=2;      break;
      }
     default:  /* ulga 10% */
     {
       podatek = podatek - (podatek/100*10);  przyp=3;    break;
     }
   }  /* switch */

   printf ("Ilosc dzieci %u. Przypadek %u.  Do zapłaty: %f\n",
         dzieci, przyp,  podatek);  /* wydruk */
    getchar();
}  /* main */
```

/ program switch2.cpp */*

```
#include <iostream.h>

 void main ()
 { //main
   unsigned int dzieci, przyp=0;    float podatek=1000.0; char znak;
   cout <<"Podaj ilosc dzieci ";
   cin >> dzieci;  // Wprowadzenie ilosci dzieci

   switch (dzieci)
    {  //switch
     case  0: break; /* brak dzieci - czyli brak ulgi */
     case  1: // ulga  2%
      {  // case 1
       podatek = podatek - (podatek/100* 2); przyp=1;    break;
      } //case 1
     case  2: /* ulga  5% */
     {
       podatek = podatek - (podatek/100* 5);  przyp=2;     break;
      }
     default:  /* ulga 10% */
     {
       podatek = podatek - (podatek/100*10);   przyp=3;    break;
     }
   }  // switch

 cout << "Ilosc dzieci "<< dzieci << ".  Przypadek " <<   przyp << ".
Podatek " << podatek;
  cout << '\n' << "Wprowdz jakis znak "; cin >> znak;

 }  // main
```

Wejście i **wyjście** programu

- Do podstawowych funkcji języka C, umożliwiających komunikację z otoczeniem (wprowadzanie danych, wyprowadzanie wyników) 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. Gdy program ma nazwę P11.exe

  p11 > Wynik.txt /* do pliku */

  p11 > PRN /* na drukarkę */

Funkcja **putchar**: int putchar(int)

- **Funkcja** wysyła na zewnątrz (do standardowego strumienia wyjściowego **stdout**, standardowo na ekran0 **pojedynczy znak**
  Funkcja (makro) zwraca wartość wypisanego znaku lub EOF (jako sygnał błędu).
- Przykład:

```
/*  Program  p12.c      */
#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.

```
/* Przykład */
#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)

Int printf(const char *format,...)

```
/* Przykład: */
/*  Program  p14.c  */
#include <stdio.h>
main()
{
  int  k = 21101;    /* liczba dziesiętna */
  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 typu w formacie	**Argument wejściowy**	**Format wyjściowy**
Liczba		
%d, %i	int	**liczba całkowita ze znakiem**
%u	unsigned int	**liczba całkowita ze bez znaku**
%o	int	**ósemkowa**
%x	int	**szestnastkowa** 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

Przykład z printf

```
/*  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

Przykład programu z funkcją printf

```
#include <stdio.h>
void main()
{
  char napis[] = "ABCDEFGHIJKLMNOPQRS...";
  int  znak = 73;
printf("\n%c %c %c %c %c \n%c %c %c\n",
   znak, 55, 'Z', '\066', '\x3A', napis[0], napis[4], znak+'\066'-52);

printf("\n%s\nTo jest praktyczna funkcja\n%c%c%c%c itd.",
   napis, '\066', '\067', '\x2B', '\x2A');
}
```

Wyniki:

```
I 7 Z 6 :
A E K

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

/* Program - Przykład z printf() */

```
#include <stdio.h>
void 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);
}
```

Wyniki

```
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   590F
 ----------------------
 Lepsze C od B lub P
```

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

Pełny zapis instrukcji formatu printf:

- **% [Pole znaku] [Szerokość] [.Dokładność] [Modyfikator] Typ danych**
- **% [flagi] [szerokość] [.precyzja] [F | N | h\ l | L] znak-typu**
- Wszystkie sekwencje formatujące zaczynają się od znaku **%.**
  Kolejne elementy określają:
  - **Znaki flag** - dodatkowe możliwości (np. wyrównanie lewostronne)
  - **Szerokość** - liczba wysyłanych znaków -całkowita szerokość pola na zapis liczby
  - **Precyzja** (dokładność) - liczba cyfr znaczących - określa ilość miejsc dziesiętnych po kropce dziesiętnej. Może wystąpić *. Specyfikator precyzji - rozpoczyna się znakiem kropki, który oddziela go od pozostałych elementów sekwencji formatującej.
  - **Modyfikator** rozmiaru argumentów - zawierają dodatkową informację nt. rozmiaru argumentu i są wykorzystywane w połączeniu ze znakami typu w celu wskazania faktycznego typu argumentu.
  - **Znaki typu** - określają typ argumentów
  - Zapisy w nawiasach kwadratowych są opcjonalne i nie muszą wystąpić
- **Pole znaku (znaki flag):**
  '+' '-‘ '#‘ ' ' (spacja) 0 (zero)
  - wyprowadzony ciąg znaków będzie wyrównany lewostronnie
  + znak ma być wydrukowany (- lub +)

Flagi w sekwencjach strujących, znak

Flagi w sekwencjach formatujących

Flaga	*Co określa*
-	Justuje lewostronnie wartość, umieszczając z prawej strony odpowiednią ilość spacji
+	Jeśli wyświetlana jest liczba ze znakiem, wówczas zostanie przed nią umieszczony znak + lub -, w zależności od wartości
spacja	Jeśli wartość jest nieujemna, zamiast plus wyświetlana będzie spacja, w przypadku wartości ujemnych -
#	Wpływ tego znaku zależy od tego, z jakim znakiem typu jest związany - tabela poniżej

Typ danych	**Reakcja, gdy występuje znak #**
c, s, d, i, u	Nie wpływa
%o	Przy wartościach dodatnich na początku zostanie dopisany znako 0
%x, %X	Wszystkie liczby będą przedstawione z 0x lub 0X
%e, %E, %f	Również w przypadku braku miejsc znaczacych dziesiętnych, po przecinku będzie kropka dzies.
%g, %G	Odpowiednio jak dla %e i %E, ale będą widoczne nieznaczące zera

Specyfikatory precyzji (dokładności) - zagadnienie dokładności obliczeń

Specyfikator dokładności	*Precyzja - znaczenie*
nie podano	**Domyślne precyzje**: - 1 dla znaków typu d, i, o, u, x, X - 6 dla typu e, E, f - wszystkie znaczące cyfry dla typu g i G
0	Dla znaków typu d, i, o, u, x, X przyjmowana jest precyzja d domyślna. Dla e, E, f nie jest drukowana kropka dziesiętna. Liczby całkowite o wartości zero nie będą drukowane
n	Drukowane będzie n znaków lub cyfr dziesiętnych. Może nastąpić obcięcie lub zaokrąglenie
*	Lista argumentów zawiera wartość określającą precyzję.

Specyfikator szerokości - określa minimalną szerokość dla wartości wyjściowej

Specyfikator szerokości pola	***Wpływ na formatowane dane***
n	Drukowanych jest przynajmniej n znaków. Jeśli wartość zajmuje mniej niż n znaków, umieszczone będą dodatkowe spacje, z lewej lub prawej strony (jeśli ustawiono odpowiednią flagę)
0n	Drukowanych jest przynajmniej n znaków. Jeśli znaków drukowanych jest mniej niż n to nastąpi wypełnienie zerami z lewej strony
*	Szerokość pola znajdzie się w argumencie poprzedzającym argument, z którym związana jest dana sekwencja

Modyfikatory rozmiaru argumentów

- zawierają dodatkową informację o typie argumentów wejściowych. Jedna z liter: F, N, h, l

Modyfikator	***Działanie***
F	Przekazanie wskaźnika typu **far**. Argument zostanie potraktowany jako daleki wskaźnik - używany ze znakami typu p, s, n
N	Przekazanie wskaźnika typu **near**. Argument będzie traktowany jako wskaźnik bliski. Nie stosować w modelu **huge**.
h	Oznacza przekazanie danej typu short int. Argument traktuje się jako short int dla znaków typu d, i, o, u, x, X
l	Oznacza przekazanie danej typu **long int** względnie **double**.
L	Argument traktowany jako **long double** dla znaków typu e, E, f, g, G

Przykład z formatami printf()

```
/* Program p18.c ( Turbo C 2.0 ) */
#include <stdio.h>
void main() /* wybrane formaty */
{
double a=153.67789, b=153. ;
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 %%#-+8d %#-+8d", k);
printf("\n format %%#-+8i %#-+8i", m);
printf("\n format %%#- 8u %#- 8u", k);
printf("\n format %%#-+8o %#-+8o", k);
printf("\n format %%#-+8x %#-+8x", k);
printf("\n format %%#-+8X %#-+8X", k);
printf("\n format %%#020.8f %#020.8f", a);
printf("\n format %%# 20.8e %# 20.8e", a);
printf("\n format %% +20.8E % +20.8E", b);
printf("\n format %% +20.8g % +20.8g", a);
printf("\n format %% +20.8G % +20.8G", b);
printf("\n format %% +20.8c % +20.8c", c);
printf("\n format %%.20s %.20s", napis);
printf("\n format %%24p %24p", *p);
printf("\n ---------------------------------------");
printf("\n%44.*s",30,napis);
m=printf("\n\n123456789 123456789 123456789
      123456789 1\n");
printf("\n\"Wynik\" poprzednio wykonanej instrukcji");
printf("\np r i n t f wynosi %i znaki \n", m);
}
```

Wyniki

```
format %#-+8d +22799
format %#-+8i -500
format %#- 8u  22799
format %#-+8o 054417
format %#-+8x 0x590f
format %#-+8X 0X590F
format %#020.8f 00000000153.67789000
format %# 20.8e       1.53677890e+02
format % +20.8E      +1.53000000E+02
format % +20.8g          +153.67789
format % +20.8G               +153
format % +20.8c                 M
format %.20s Lepsze C od B lub P
format %24p                   590F
---------------------------------------
                Lepsze C od B lub P

123456789 123456789 123456789 123456789 1

"Wynik" poprzednio wykonanej instrukcji
p r i n t f wynosi 44 znaki
```

Przykład p19c

```
/*  Program  p19.c              */
#include <stdio.h>
void main()  /* zapisy rownowazne,
     dynamiczny format */
  double  a=153.22;
  char    napis[]="Wszystko dobre co
     nie jest niedobre ";
  int     i,j;
  printf("\n%.9s", napis);
  printf("\n%.*s", 9, napis);
  i=9;   printf("\n%.*s", i, napis);
  i=14;  printf("\n%.*s", i, napis);
  i=21;  printf("\n%.*s", i, napis);
  printf("\n a = %#09.3f", a);
  printf("\n a = %#0*.3f", 9, a);
  printf("\n a = %#0*.*f", 9, 3, a);
  i=9; j=3;
  printf("\n a = %#0*.*f", i, j, a);
  i=20; j=8;
  printf("\n a = %#0*.*f", i, j, a);
}
```

Funkcje

- Funkcje można podzielić na biblioteczne, realizujące standardowe operacje oraz funkcje własne, opracowane przez użytkownika.
- W języku C nie wolno dokonywać zagnieżdżeń w funkcjach.
  Zaleca się by długość funkcji nie przekraczała 60 linii tekstu.
- Funkcje (ew. makrodefinicje) pozwalają wydatnie skrócić tekst programu, gdy pewne fragmenty się powtarzają.
- **Funkcja** to wyróżniony fragment programu, komunikujący się z pozostałą częścią programu w ściśle określony sposób. Do komunikacji służą **parametry**, w definicji funkcji zwane formalnymi, a przy wywołaniu funkcji parametrami aktualnymi.
  Funkcja może być wielokrotnie wywoływana z dowolnego miejsca programu.
- Najważniejsze korzyści ze stosowania funkcji:
  1. Program napisany z wykorzystaniem funkcji jest bardziej czytelny i zrozumiały
  2. Jest bezpośrednia odpowiedniość między algorytmem a programem
  3. Pewne powtarzające się fragmenty lub operacje mogą być wyodrębnione i *zapisane w postaci jednej funkcji*
  4. Podczas testowania i uruchamiania można oddzielnie testować poszczególne funkcje

Definiowanie funkcji

- Deklaracja funkcji zgodnie z zalecanym dla języka C przez ANSI

Typ_funkcji nazwa_funkcji (lista_parametrów_formalnych)

lub inaczej

Typ nazwa (deklracje parametrów)

{

Instrukcje

}

- Typ funkcji należy do zbioru typów dozwolonych w języku C. Nazwa funkcji analogicznie jak zmiennej.
- Lista parametrów formalnych określa zestaw par w postaci:

  Typ_parametru_formalnego nazwa_parametru_formalnego, które to pary oddzielone są przecinkami, np. int i, char c, float x

- Przykłady: Int **SumaKwadratow**(int n); void **funkcja_a**(void);

float **objetosc_walca** (float promien, float wysokosc)

{return (PI*promien*wysokosc); }

Podstawowe informacje o funkcji

- Ogólny zapis funkcji ma postać:

```
typ nazwa(argumenty)
{
  ciało funkcji
}
```

Typ określa, jaki typ danych zwraca ta funkcja, tzn. do zmiennej jakiego typu można przypisać wynik funkcji.

Funkcja może nie zwracać żadnego typu, wtedy jej typem jest void

- Do zwracania danej wartości przez funkcję służysłowo kluczowe return.
- **Nazwa** służy do identyfikowania funkcji w programie. Nazwa funkcji nie zawsze jednoznacznie identyfikuje funkcję.
- **Argumenty** określają, jakiego typu zmienne należy przekazać do funkcji przy jej wywoływaniu.
- W przypadku, kiedy nazwa nie identyfikuje jednoznacznie funkcji (istnieją dwie różne funkcje o tych samych nazwach), parametry tych funkcji muszą się różnić.
  Mamy wtedy do czynienia z *przeładowaniem nazwy funkcji* (patrz niżej).
- Argumenty funkcji mogą przyjmować wartości domyślne. Należy w takim przypadku po nazwie argumentu dodać znak równości i wartość domyślną.
- **Ciało funkcji** jest to kod, który zostanie wykonany po wywołaniu funkcji.

Przykłady funkcji

- Funkcja nie zwracająca wyniku (brak return), nie przyjmująca argumentów (typu void):

```
void f1()
{   cout << "tekst\n"; }
```

- Funkcja nie zwracająca wyniku, przyjmująca parametr bez wartości domyślnych:

```
void f2(char *tekst)
{   cout << tekst;  }
```

- Funkcje zwracająca wynik (return), przyjmująca parametry bez wartości domyślnych:

```
int f3(int a, int b)
{   return a + b; }

int f4(int a, int b)
{    int c = a + b;    return c; }
```

- Funkcja zwracająca wynik (return), przyjmująca parametry z wartością domyślną:

```
int f5(int a, int b = 0)
{    return a + b; }
```

-

Wywoływanie funkcji

- Aby wywołać w programie przygotowaną wcześniej funkcję, należy wpisać w odpowiednim miejscu jej nazwę,
  a w nawiasach okrągłych argumenty do tej funkcji.
  Np. int n=4; wynik=SumaKwadratow(n);
- Wywoływana funkcja musi zawsze być zadeklarowana przed miejscem, gdzie została wywołana.
  Można przed wywołaniem funkcji napisać tylko jej deklarację, zaś definicję (ciało) w dowolnym miejscu programu.
- Np. int SumaKwadratow (int n); *// prototyp funkcji*
- W językach C/C++ jeśli funkcja nie posiada argumentów należy ją wywoływać stawiając za jej nazwą pusty nawias,
  np. funk1()

Przykłady funkcji

```
// Funkcja dodaje 2 podane liczby
include <iostream.h>

int sum(int a, int b); // Deklaracja funkcji

int main()
{
   int a, b, c;
   cout << "Podaj 2 liczby ";
   cin >> a >> b;
   c = sum(a, b); // wywołanie funkcji
   cout << " Suma " << a << '+' << b <<
'=' << c << endl;
   cout << "Wprowadz jakis znak ";
   cin >> a;
    return 0;
}

// definicja funkcji
int sum(int a, int b)
{
   return a+b;
}
```

```
// Obliczenie silni- funkcja rekurencyjna
#include <iostream.h>
// definicja funkcji
int silnia(int n)
{
   if(n == 0) return 1;
   return silnia(n-1)*n;
}

void main()  // funkcja główna
{
   int a, c;
   cout << "Obliczenie silni\n";
   cout << "Wprowadz liczbe calkowita ";
   cin >> a;
// wydruk z wywołaniem funkcji
   cout << "Silnia " << a << '=' << silnia(a) << endl;
   getchar();
   getchar();
}
```

Tablice

- Tablica jest to struktura danych zawierająca uporządkowany zbiór obiektów tego samego typu i odpowiadająca matematycznemu pojęciu wektora (dla tablic jednowymiarowych), macierzy (dla tablic dwuwymiarowych) itp.
- W językach C, C++ elementy tablicy numerujemy od 0.
- Tablica to ciąg zmiennych jednego typu. Ciąg taki posiada jedną nazwę a do jego poszczególnych elementów odnosimy się przez numer (indeks).
- **Deklarowanie tablicy**
- **Typ_danych nazwa_tablicy[rozmiar];**
- Typ_danych nazwa_tablicy [lista_rozmiarow] = {lista_wartosci};
- Przykłady tablic: int tablica[20]; double tab[100]; int b[NMAX*MMAX];
- Int t[10][5];
- Podobnie jak przy deklaracji zmiennych, także tablicy możemy nadać wartości początkowe przy jej deklaracji.
  Odbywa się to przez umieszczenie wartości kolejnych elementów oddzielonych przecinkami wewnątrz nawiasów klamrowych:
- int tablica[3] = {1,2,3};

Przykłady z tablicami

```
include <stdio.h>
#define ROZMIAR 4
void main()
{
  int tab[ROZMIAR] = {3,6,8, 10};
  int i;
  printf ("Druk tablicy tab:\n");

  for (i=0; i<ROZMIAR; ++i) {
    printf ("Element numer %d = %d\n", i,
         tab[i]);
  }
getchar();
}
```

```
    #include <stdio.h>
    int main()
    {
      int tab[4] = {3,6,8, 10};
      int i;
      printf ("Druk tablicy tab:\n");

      for (i=0; i<(sizeof tab / sizeof *tab);
    ++i) {
        printf ("Element numer %d =
    %d\n", i, tab[i]);
      }
      getchar();
    }
```

```
#include <stdio.h>

main()

{
 char str[]="Alicja";
 for (int i=0; str[i] != '\0';
i++)
  printf("%c",str[i]);
  getchar();
 }
```

```
#include <iostream.h>
#include <conio.h>
// Lekcja 16

void wyp_tab(int tab[], int l)
{
                int i;
        cout << "\nWydruk " << l << " liczb\n";
                for (i=0; i<l; i++)
                 cout << tab[i] << ' ';
}


void czyt_l(int tab[], int l_el)
{
                int i;
        cout << "Podaj kolejno  " << l_el << " liczb\n";
                for (i=0; i<l_el; i++)
                {
                 cout << "Podaj liczbe " << i << ": ";
                 cin >> tab[i];
                 }
}
void main (void)
{
 clrscr();
 int liczby[3];
 czyt_l(liczby,3);
 wyp_tab(liczby,3);
 getch();

}
```

Wskaźniki w C

- W językach C i C++ można używać zmiennych wskaźnikowych, *których wartościami* są **wskaźniki** zawierające adres pewnego obiektu. Obiektem może być np. zmienna, struktura czy też obiekt pewnej klasy.
- **Wskaźnik** (ang. *pointer*) to specjalny rodzaj zmiennej, w której zapisany jest adres w pamięci komputera, tzn. wskaźnik **wskazuje** miejsce, gdzie zapisana jest jakaś informacja. Wskazywaną daną był inny wskaźnik do kolejnego miejsca w pamięci.
- By stworzyć wskaźnik do zmiennej i móc się nim posługiwać należy przypisać mu odpowiednią wartość (adres obiektu, na jaki ma wskazywać).
  Skąd mamy znać ten adres? Wystarczy zapytać nasz komputer, jaki adres przydzielił zmiennej, którą np. wcześniej gdzieś stworzyliśmy.
  Robi się to za pomocą operatora **&** (operatora pobrania adresu).
- Przykład

```
#include <stdio.h>
 int main (void)
 {
   int liczba = 80;
   printf("Zmienna znajduje sie pod adresem: %p, i przechowuje wartosc: %d\n",
         (void*)&liczba, liczba);
   getchar();
 }
```

Program ten wypisuje adres pamięci, pod którym znajduje się zmienna oraz wartość jaką kryje zmienna przechowywana pod owym adresem.

- Wynik: Zmienna znajduje sie pod adresem: 0012FF50, i przechowuje wartosc: 80

Wskaźniki – zmienne wskaźnikowe

- Wskaźników dotyczą 2 operatory wskazywania:

  Operator adresu &

  Operator adresowania pośredniego * , inaczej operator wyłuskania

- Aby móc zapisać gdzieś taki adres pamięci należy zadeklarować zmienną wskaźnikową.
  Robi się to poprzez dodanie ***** (gwiazdki) po typie na jaki zmienna ma wskazywać, np.:

```
int    *wskaznik1; char  *wskaznik2;  float   *wskaznik3;
```

Niech v będzie nazwą zmiennej, wartość wyrażenia &v podaje adres tej zmiennej. Wartość tę możemy przypisać zmiennej wskaźnikowej, np. pv. Możemy wtedy powiedzieć, że zmienna pv jest wskaźnikiem zmiennej v (wskazuje zmienną v).

Przykład z danymi powyżej

```
#include <stdio.h>

 int main (void)
 {
  int u=13, v, *pu, *pv;
  pu=&u;
  pv=&v;
  v=*pu;
  printf("\n &u = %p  &v=  %p",&u, &v);
  printf("\n &pu= %p  &pv= %p", &pu,&pv);
  printf("\n pu = %p   pv= %p",pu, pv);
  printf("\n u =%-12d  v = %d", u, v);

   getchar();
 }
```

Wyniki

```
&u = 0012FF50  &v=  0012FF4C
 &pu= 0012FF48  &pv= 0012FF44
 pu = 0012FF50   pv= 0012FF4C
 u =13           v = 13
```

Przykład programu z użyciem operatora wyłuskania

```
#include <stdio.h>

 int main (void)
 {
   int liczba = 80;
   int *wskaznik = &liczba;
   printf("Wartosc zmiennej: %d; jej adres: %p.\n", liczba, (void*)&liczba);
   printf("Adres zapisany we wskazniku: %p, wskazywana wartosc: %d.\n",
         (void*)wskaznik, *wskaznik);

   *wskaznik = 42;
   printf("Wartosc zmiennej: %d, wartosc wskazywana przez wskaznik: %p\n",
         liczba, *wskaznik);

   liczba = 0x42;
   printf("Wartosc zmiennej: %d, wartosc wskazywana przez wskaznik: %p\n",
         liczba, *wskaznik);

   getchar();
 }
```

Wyniki:

Wartosc zmiennej: 80; jej adres: 0012FF50.
Adres zapisany we wskazniku: 0012FF50, wskazywana wartosc: 80.
Wartosc zmiennej: 42, wartosc wskazywana przez wskaznik: 0000002A
Wartosc zmiennej: 66, wartosc wskazywana przez wskaznik: 00000042