1. Języki niskiego i wysokiego poziomu

Interpreter

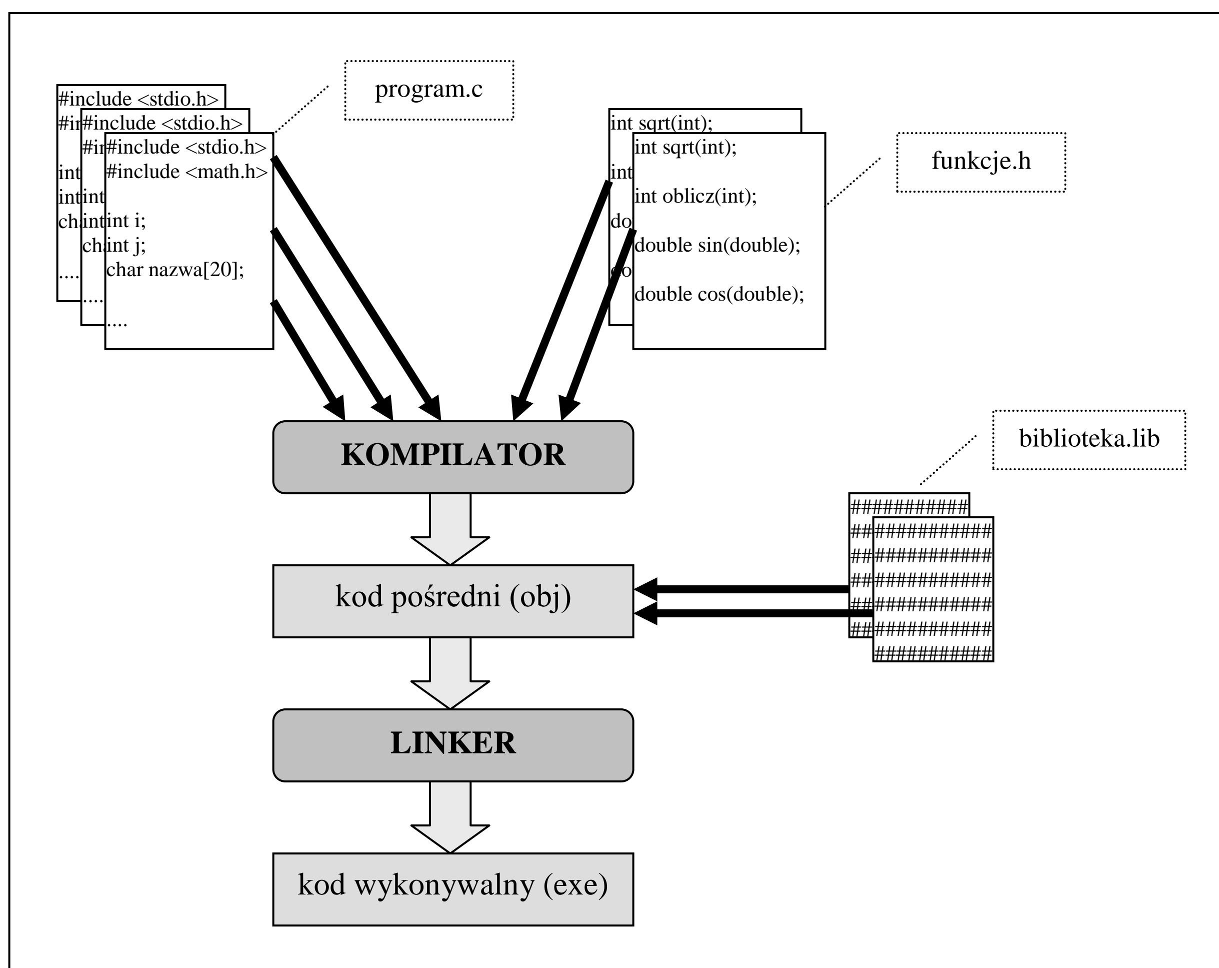
Kompilator

2. Język C

Powstał w 1972 roku, autor: Dennis Ritchie. Służył i służy przede wszystkim do programowania w systemie UNIX, istnieją liczne kompilatory dla innych środowisk.

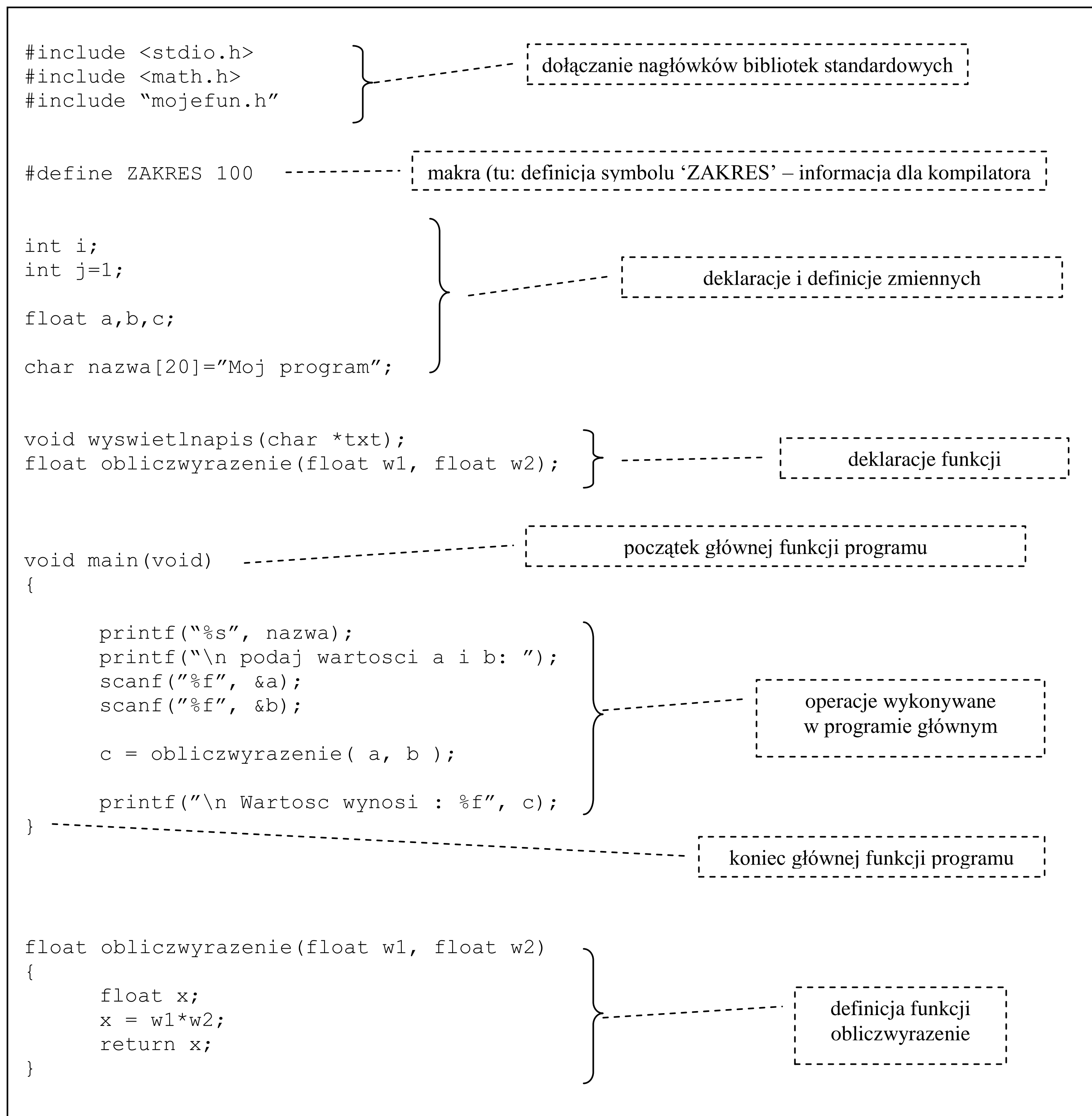
2.1. Środowisko kompilatora języka C

kod źródłowy, pliki nagłówkowe, kompilator, kod pośredni, biblioteki, linker, kod wynikowy
debugger, profiler



Środowisko = kompilator + pliki nagłówkowe + linker + biblioteki [+ zintegrowany edytor] [+ debugger] [+profiler]

2.2. Przykładowy prosty program w C



3. SKŁADNIA JĘZYKA C

3.1. Jednostki leksykalne

Słowa kluczowe

```
break switch case default if else for continue do while goto asm const
int float double char unsigned long struct union register volatile static extern typedef sizeof  …
```

Identyfikatory

- składają się z liter (alfabet łaciński!) lub cyfr lub znaku _
- zaczynają się od litery lub znaku _
- dowolna liczba znaków w identyfikatorze
- rozróżniane są duże i małe litery !!!

```
int zmienna, Zmienna, w1, w2;
int ObliczWyrazenie(int _x, int _y);
int _funkcjaPomocnicza(int i);
```

Literały:

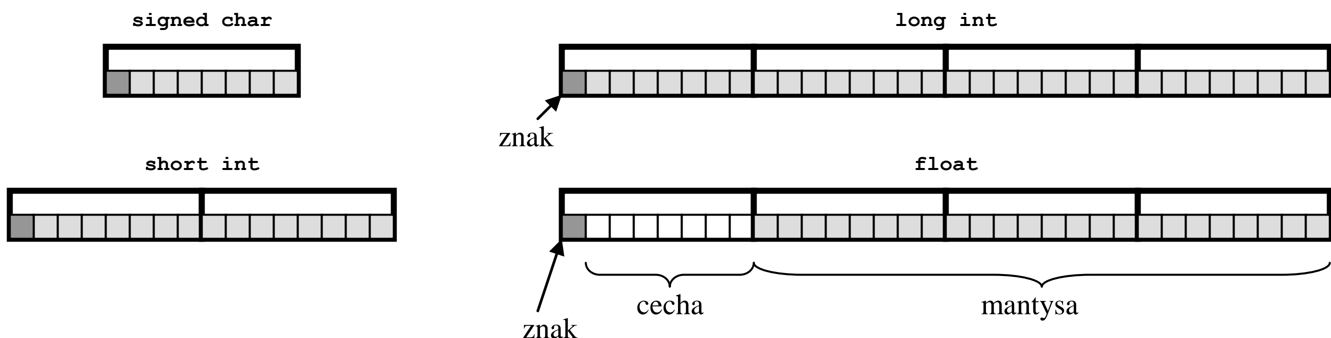
liczba: `1000    10003450L  0xFF20    0xff20    12.34     0.0   2.3E4     23000.00`
znak: `'A'   'a'   '\n'   '\t'   '\r'   '\b'   '\\'   '\''   '\"'   '\0'   '\76'`

3.2. Typy danych

Zmienne proste

typ	opis	rozmiar (B)*	zakres signed	zakres unsigned
char	zmienna znakowa np. ascii	1	-128 : 127	0 : 255
int	zmienna całkowita	2/4 (win32)		
short	zmienna całkowita krótka (int)	2	-32768 : 32767	0 : 65535
long	zmienna całkowita długa (int)	4		
float	zm. zmiennopozycyjna	4	$1.1 \times 10^{-38} : 3.4 \times 10^{38}$ (7 cyfr)	-
double	zm. zmiennopozycyjna długa	8	$2.2 \times 10^{-308} : 1.7 \times 10^{308}$ (15 cyfr)	-
long double	zm. zmiennopozycyjna długa	10	$: 1.2 \times 10^{4932}$ (19 cyfr)	-

* - rozmiar może się różnić w zależności od kompilatora (platforma sprzętowa, system)



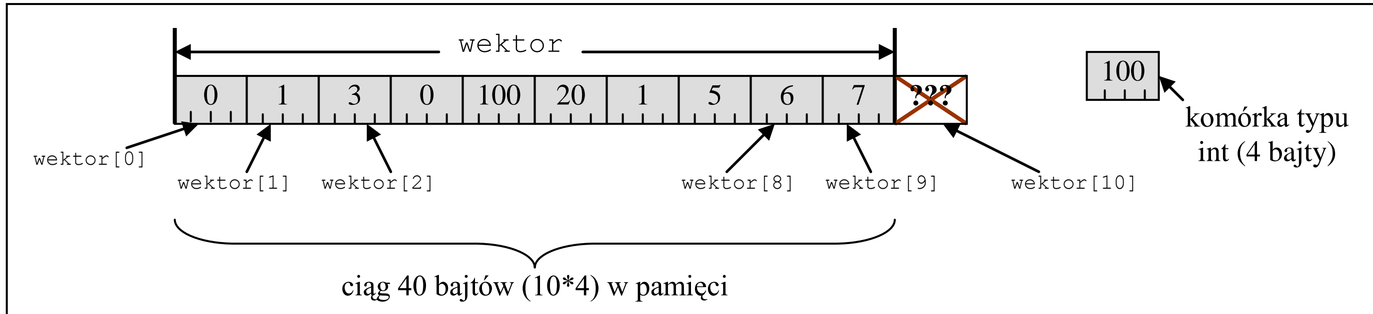
$$\texttt{liczba = mantysa * } 2^{\texttt{cecha}}$$

Deklaracja zmiennej : informacja o typie i nazwie zmiennej
Definicja zmiennej : nadanie zmiennej wartości

Tablice zmiennych

`int wektor[10];` // deklaracja tablicy jednowymiarowej typu int

`int wektor[10] = {0,1,3,0,100,20,1,5,6,7};` // deklaracja i definicja tablicy jednowymiarowej typu int

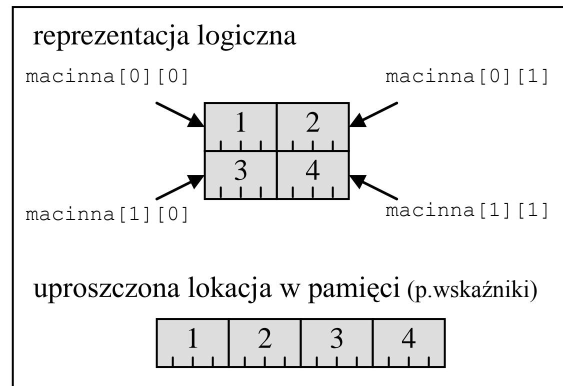


Dostęp do zmiennych:

```
wektor[0] = 1;
wektor[2] = wektor[1] + wektor[0];
```

Tablice wielowymiarowe

```
int macierz[5][5];
int macinna[2][2]= { {1,2}, {3, 4}};
int szescian[2][2][2];
```



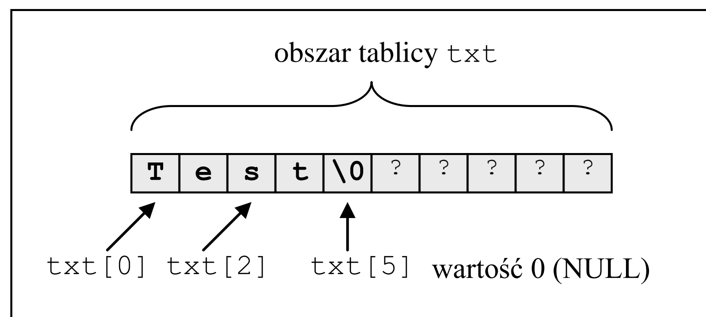
Łańcuchy tekstowe (stringi)

String : ciąg zmiennych w tablicy typu char, zakończony wartością 0 (zero).
Identyfikacja stringu – nazwa tabeli – jest jednocześnie wskaźnikiem na pierwszy znak stringu.
Wiele operacji na stringach jest zdefiniowane a bibliotece <string.h>

```
char tekst[20];
char tekst1[20]="test";
char tekst2[]="przykład";
char *tekst3="przykład 2";
```

Programista musi kontrolować dostęp do tablicy.
Nie jest dozwolone wykroczenie poza jej obszar.

```
char txt[10]="Test";
```



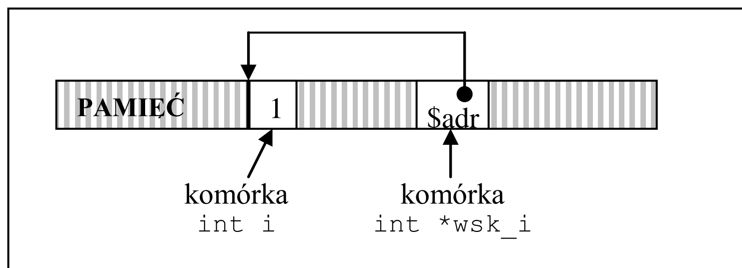
`printf( "%s", txt ); ->` **Test**

`printf( "%c %c", txt[0], txt[2] ); ->` **T s**

Wskaźniki

Wskaźnik może wskazywać dowolny obiekt w pamięci: zmienną, inny wskaźnik, strukturę, funkcję.

```
int i=1, j=2;
int *wsk_i=i;
int *wsk_x;
wsk_x = &j;

int **wskwsk;
wskwsk = &wsk_x;
```



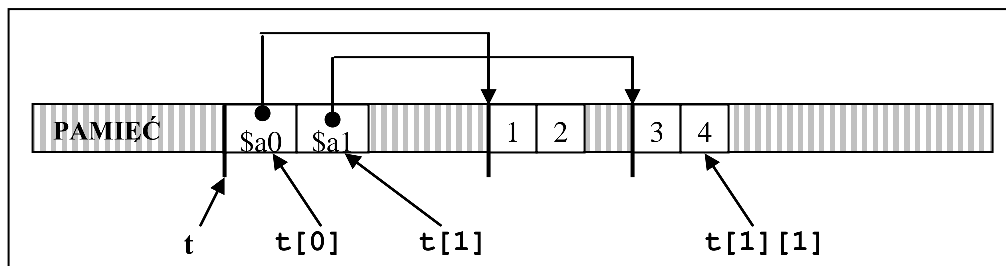
& - operator pobrania adresu

Nazwa tablicy jest wskaźnikiem jej pierwszego (zerowego!) elementu.

```
int wek[10], *w, a;
w = &wek[0];
a = wek[0];          a = *w;
a = wek[1];          a = *(w+1); // obliczane automatycznie – int ma 4 bajty!!!!
```

Nazwa tablicy dwuwymiarowej jest wskaźnikiem do tablicy wskaźników do tablic jednowymiarowych.

```
int tab[2][2] = { {1,2}, {3, 4}};
int **t;
t = .... (inicjalizacja tablic)
t[0][0] = tab[0][0];
t = tab (konieczna konwersja)
```



void * - wskaźnik ogólnego typu do pamięci

Struktury danych i unie

```
struct student{
   int wiek;
   int nr_indexu;
   char imie[20];
   char nazwisko[20];
};

struct student prymus = {24, 80876, ”Alfred”, ”Nobel” } , grupa[30];
struct student *pytany;
```

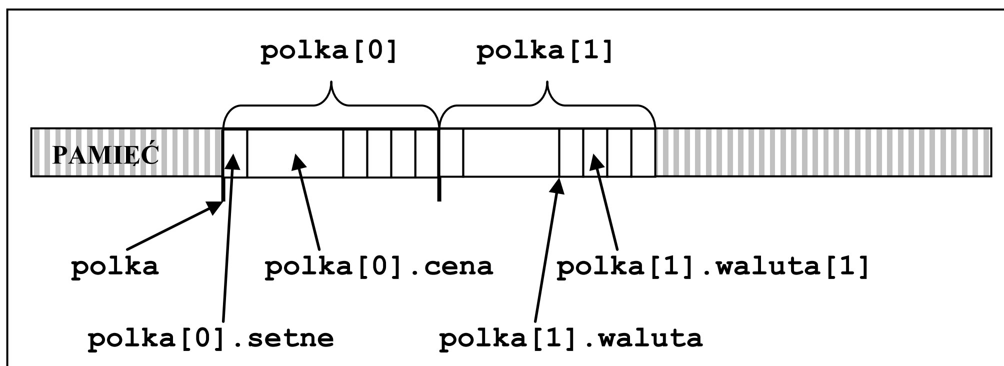
Dostęp 'kropkowy' do składników obiektu i dostęp poprzez wskaźnik:

```
prymus.wiek = 64;
printf(” %s %s ”, prymus.imie, prymus.nazwisko)

pytany = & grupa[0];
printf(” %s %s %d ”, pytany->imie, pytany->nazwisko, pytany->wiek);

printf(” %d ”, sizeof(struct student));     ->
```

48

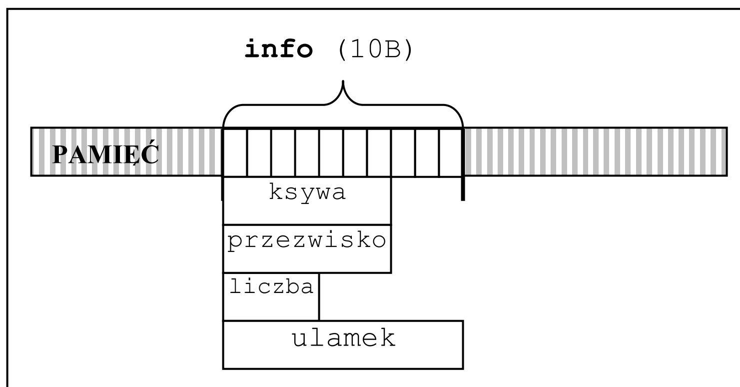
Rozlokowanie struktury w pamięci; unie

```
struct Towar{
  char setne;
  int cena;
  char waluta[4];
};

struct Towar polka[2];

polka[0].cena =  10;
polka[1].cena =  20;
polka[0].setne = 0;
```



Unia: struktura, w której wszystkie pola zaczynają się od tego samego miejsca pamięci -> rozmiar unii ~ najdłuższy element

```
union jednadana{
  char ksywa[7];
  char przezwisko[7];
  int liczba;
  long double ulamek
};

union jednadana info;
```



Operator **const**

```
const float pi = 3.14;
```

Wartości zmiennej typu **const** nie można już zmienić w programie.

3.3. Wyrażenia

Operatory

Operator przypisania:

`x` = *L-VALUE (l-wyrażenie)*

Inne operatory przypisania (wraz z działaniem):

```
+=    -=    *=    /=    %=    >>=    <<=    &=    ^=    |=
```

`x += 8;` jest równoważne `x = x + 8;`

Operatory arytmetyczne:

```
+ - * / % >> << ++ --
```

`x++` jest równoważne `x = x + 1`

Operatory logiczne:

```
! ~ & ^ |  == != > <
```

Operator wyliczeniowy (połączeniowy)

```
w1, w2, w3

x = (y+2, y-5, 5, 10);
```

Operator wyboru

```
warunek ? jeśli_spelniony : jeśli_nie_spelniony

maximum = (a > b)? a : b ;
```

Operator konwersji

```
char z =20;
int a;

a = (int) z;
```

Priorytety operatorów

W tabeli zamieszczono operatory według ich malejącego priorytetu

	Operatory
max	( ) [ ] -> .
	! ~ ++ -- - (konwersja) *wsk &zm
	* / %
	+ -
	<< >>
	< <= > >=
	== !=
	&
	^
	|
	&&
	||
	?:
min	= += -= *= /= %= >>= <<= &= ^= |=
	,

Przykład:

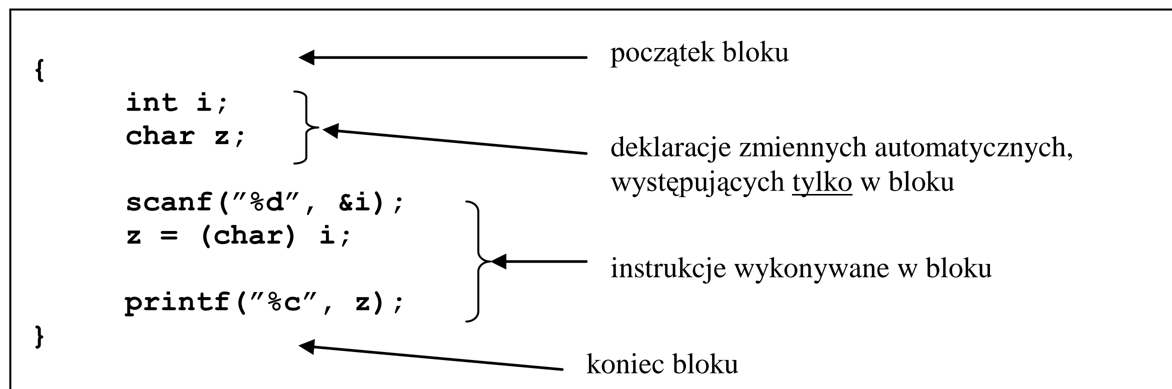
a + b * c   **a + (b * c)**   **(a + b) * c**

3.4. Instrukcje

Instrukcja pusta ;

W języku C każda instrukcja kończy się znakiem **;**

Grupowanie instrukcji



Zmienne globalne: widziane przez wszystkie funkcje
Zmienne automatyczne: tworzone przy wejściu do bloku i niszczone po wyjściu z niego

Instrukcja przypisania

```
z = 20;
x = 2*(a+b);
```

(Uwaga! Istnieje różnica między x++ a ++x)
x++ postinkrementacja, ++x preinkrementacja

int x = 1, y = 1, a, b;

a = x++; (*x* wyniesie 2, *a* wyniesie 1)
b = ++y; (*y* wyniesie 2, *b* wyniesie 2)

Instrukcje inkrementacji i dekrementacji:

```
a++;    b--;    ++c;    --d;
```

Instrukcja warunkowa **if else**

```
if (warunek)
  zrob_jeśli_spelniony();
else
  zrob_jeśli_nie_spelniony();
```

```
if (warunek)
{
      op1();
      op2();
}
else
{
      op3();
      op4();
}
```

Instrukcja wyboru **switch**

```
switch ( zmienna )
{
  case przypadek_1:
        instr_1();
        instr_2();
        break;
  case przypadek_2:
        instr_3();
  case 3:
        instr_4();
        break;
  default:
        instr_5();
}
```

Instrukcja pętli **for**

```
for( inicjacja_poczatkowa ; warunek_zakonczenia ; wykonywanie_kroku )
```

```
int i;

for( i=0 ; i<10 ; i++)
       printf("%d", i);
```

```
for( i=0 ; i<10 ; i=i+1)
{
     instr_1();
}
```

pętla będzie wykonywana dla kolejnych wartości i: 0, 1, 2, ..., 9

```
for ( ; i<10; )
{
  ...
}
```

```
for ( ; ; )
{
     ...
}
```

iteracja w dół:

```
for( i = 10 ; i > 0 ; i-- )
       printf( "%d", i );
```

iteracja np. o skoku geometrycznym:

```
for( i = 1 ; i < 1000 ; i = i*2 )
       printf( "%d", i );
```

Pętla **while**

```
while( warunek )
  czynnosc();

while( warunek )
{
  instr_1();
  instr_2();
  …
}
```

```
while( i <100 )
     i++;
```

```
while ( 1 )
{
  … pętla nieskończona
}
```

Pętla **do-while**

```
do
{
  instr_1();
  instr_2();
}while( warunek )
```

Pętla do-while wykona się co najmniej jeden raz !

`continue` – ponowne rozpoczęcie pętli
`break` – zakończenie wykonywania pętli (wyjście)

Przykład:

```
int i, j=60, k;

for(int i=1 ; i < 100 ; i++)
{
        if(i == 1)
              continue;

        k = j/i;

        if(i>j)
              break;
}
```

3.5. Funkcje

Deklaracja funkcji:

```
typ_zmiennej_zwracanej  nazwa_funkcji (typ_par_1 nazwa_par_1, typ_par_2  nazwa_par_2, ...);
```

Definicja funkcji

```
typ_zmiennej_zwracanej  nazwa_funkcji (typ_par_1 nazwa_par_1, typ_par_2  nazwa_par_2, ...)
{
  ... instrukcje
  return wartosc_zwracana;
}
```

Przykład:

```
/* deklaracja i definicja funkcji o jednym parametrze, niezwracającej nic*/

void wyswietl_tekst(char *tekst)
{
        printf(”%s”, tekst);
}

/* deklaracja i definicja funkcji o dwóch parametach, zwracającej ich iloczyn*/

int iloczyn(int a, int b)
{
        int c;
        c = a*b;
        return c;
}
```

Wywołanie funkcji:

```
char text_nr1[20]=”testowy”;
int obliczenia, x=10, y=20;

wyswietl_tekst ( text_nr1 );
wyswietl_tekst ( ”cos tam cos tam” );
wyswietl_tekst ( text_nr1 );

obliczenia = iloczyn ( x, y );
```

Przekazywanie parametrów może odbywać się:
- przez wartość
- przez wskaźnik
- przez referencję (tylko C++)

Przekazywanie przez wartość: na czas wywoływania funkcji tworzona jest kopia parametru – wstawiona do funkcji wartość nie jest zmieniana

Przekazywanie przez wskaźnik: do funkcji przekazywany jest wskaźnik zmiennej, nie może on być zmieniany, ale zmienna wskazywana przez niego – tak.

Przykład:

```
void funkcja(int a, int *b)
{
  int c;
  c = a + *b;        /* b jest wskaźnikiem, wartość wskazywana przez b otrzymywana jest przez *b */
  a = c;
  *b = c;
}

int x = 10, y = 20;

funkcja( x ,  &y );
```

W efekcie wywoływania funkcji zmienna x nie zmieni wartości, natomiast zmienna y będzie miała wartość 30.

3.6 Opracowywanie wyrażeń warunkowych

```
int x = 10, y = 5, a;

if ( x > 5 && x < 15 )                  if ( x < -5 || x > 5 )
  ….                                         …

if ( x > 5 && y > 2 )                   if ( x > 5 || y > 2 )
  …                                          …

if ( x > 5 && x < 15 && y > 2)          if ( ( x < -5 || x > 5 ) && y > 2 )
  …                                          …

if ( sin(x) > 0.5 )
  …

while ( a -- )        <->       for ( a=a-1; a >= 0 ; a-- )
  …                               …

while ( -- a )        <->       for ( a=a-1; a > 0 ; a-- )
  …                               …
```

```
   if ( a == x )     <-porównanie    przypisanie ->    if ( a = x )
     …                                                    …
```

```
if( y = generator_losowy() )
  …
```

3.7 Poprawny styl pisania programów – wcięcia

```
#include <stdio.h>

float a, b, c = 10, delta;
float x1, x2;
int i, j;

float oblicz_delte( float _a, float _b, float _c );

void main( void )
{
        printf( "Wprowadz a:" );
        scanf( "%f", &a );

        …

        delta = oblicz_delte( a, b, c );

        if ( delta >= 0)
             printf( "Wielomian posiada pierwiastki rzeczywiste" );

        if ( delta == 0)
        {
             x1 = x2 = -b /( 2.0*a );
             printf( "obliczony pierwiastek: %f\n", x1 );
        }

        ....

        for ( i = 0 ; i < 10 ; i++ )
        {
             for ( j = 0 ; j < i ; j++ )
                  printf( "#" );

             printf( "\n" );
        }
}

float oblicz_delte( float _a, float _b, float _c )
{

        return ( b * b - 4.0 * a * c );
}
```

Komentarze:

```
/* komentarz w języku C */
/* wielolinijkowy komentarz
  w języku C */

// komentarz w języku C++ - do końca linijki
```

4.0 Przegląd wybranych funkcji w bibliotekach standardowych

Poniżej przedstawiono wybrane (użyteczne) funkcje

stdio.h – standard input / output

printf(formatowanie, zmienna1, zmienna2, …);

przykładowe formatowania (string):

```
printf( "Napis prosty");
printf( "Napis prosty ze zmienna int : %d", a);
printf( "Napis prosty ze zmienna int bez znaku: %u", b);
printf( "Napis prosty ze zmienna long: %l", c);
printf( "Napis prosty ze zmienna float : %f", d);
printf( "Napis prosty ze zmienna double : %lf", e);
printf( "Napis prosty ze zmienna w postaci szesnastkowej : %x", sz);
printf( "Napis prosty ze ze stringiem : %s", txt);

printf("Dwie zmienne: x= %d  y=%d",x, y);

printf( "Zmienna int wysw. co najmniej w 5 kratkach (dopelnianych spacja) : %5d", a);

printf( "Zmienna float wysw. co najmniej w 5 kratkach + kropka, z dokl. 2 miejsc po przecinku : %5.2f", d);

printf("Znaki specjalne: \n nowa linia \r poczatek linii \t odstep kolumnowy");
printf("Znaki specjalne: \b zmazanie ostatniego znaku  \a dzwonek");
```

scanf(formatowanie, adres_zmiennej1, adres_zmiennej2,....);

Formatowanie w `scanf` :

1. należy nie używać znaków innych niż potrzebnych do formatowania zmiennych (%d, %s itp.) – wyjątek : zmienne są oddzielone konkretnym separatorem (np. tabulacją) – wówczas można użyć np.: `"%d\t%d"`
2. %s wczytuje tylko jeden wyraz (do wystąpienia spacji, tabulacji lub nowej linii, ponowne wywołanie sczytuje następny wyraz z konsoli

char* gets(char *buffer);
pobiera z konsoli tekst zakończony znakiem nowej linii (klawisz enter)

```
sprinf( bufor, formatowanie, zmienna1, zmienna2,...);
sscanf( bufor, formatowanie, adr_zm1, adr_zm2,...);
```

sprintf działa podobnie jak printf , z tą różnicą, że tekst jest wpisywany do bufora a nie na ekran
sscanf działa analogicznie – z podanego tekstu w buforze 'wyłuskuje' wartości, które wpisuje pod adresy zmiennych

```
FILE *plik;
//...
fprinf( plik, formatowanie, zmienna1, zmienna2,...);
fscanf( plik, formatowanie, adr_zm1, adr_zm2,...);
```

fprintf i fscanf działają podobnie jak printf i scanf, z tą różnicą, że operacje nie odbywają się na konsoli, tylko na pliku.

```
char* fgets(  char *string,  int n, FILE *plik );
```
fgets pobiera jedną linię z pliku.

string.h - operacje na tekstach

`size_t strlen( const char *string );` –kopiuje podaje długość tekstu (w bajtach, nie liczy zera na koncu!)

`strcpy(char *przeznaczenie, char *zrodlo);` – kopiuje tekst ze źródła do miejsca przeznaczenia
`strcat(char *przeznaczenie, char *zrodlo);` – 'dokleja' tekst ze źródła do miejsca przeznaczenia

```
char tekst_1[100]="asdf";
char tekst_2[100]="nic";

printf("\n %s %s", tekst_1, tekst_2);

strcpy(tekst_2, tekst_1);
printf("\n %s %s", tekst_1, tekst_2);

strcpy(tekst_2, "nowy");
printf("\n %s %s", tekst_1, tekst_2);

strcat(tekst_2, tekst_1);
printf("\n %s %s", tekst_1, tekst_2);

printf("\n %d %d", strlen(tekst_1), strlen(tekst_2));
```

```
asdf nic

asdf asdf

asdf nowy

asdf nowyasdf

4 8
```

math.h

```
double sin( double x );
double cos( double x );
double tan( double x );

double asin( double x );
double acos( double x );
double atan( double x );

double sqrt( double x );

#define M_PI        3.14159265358979323846
#define M_E         2.7182818284590452354
```

conio.h – console input output

```
int getch();        // czeka na wciśnięcie klawisza, zwraca jego kod
int kbhit();        // podaje, czy został wciśnięty jakiś klawisz

int klawisz;

// z oczekiwaniem na wciśnięcie
klawisz = getch();

// bez zatrzymania
if( kbhit() !=  0  )
  klawisz = getch();
```

4.1 Operacje na plikach

biblioteka **stdio.h**
struktura `FILE`

```
fopen( char *nazwa_pliku , char *tryb_dostepu );
```

tryby dostępu: **"r"** , **"w"** , **"rt"** , **"wt"** , **"rb"** , **"wb"** , **"a"** , **"at"** , **"ab"**

```
fprintf(), fscanf()

char* fgets( char *lancuch, int n, FILE *uchwytpliku );
int fputc(int c, FILE * uchwytpliku);
size_t fwrite( void *bufor, size_t rozmiar, size_t liczba_blokow, FILE *uchwyt);
size_t fread( void *bufor, size_t rozmiar, size_t liczba_blokow, FILE *uchwyt );
fclose()
```

Otwieranie i czytanie z pliku

```
char buff[1000];
FILE * f = NULL;

f = fopen("notatka.txt", "rt");

if ( f != NULL )
{
        buff[0]=0;
        fgets( buff, 999, f );
        printf( "odczytano pierwsza linie: %s", buff );

        fclose ( f );
}
else
        prinf( "Plik notatka.txt nie zostal otworzony" );
```

Otwieranie i zapisywanie do pliku

```
char buff[1000]="asdfasdfasdfasdf";
FILE * f = NULL;

f = fopen("notatka.txt", "wt");

if ( f != NULL )
{
  fprintf(f, "Ten tekst i ten %s wpisywany jest do pliku", buff);
  fclose( f );
}
```

Operacje na bloku danych - odczyt

```
char buff[1000];
FILE * f = NULL;
int odczytano=0;

f = fopen("notatka.txt", "rb");
```

```
if ( f != NULL )
{
  odczytano = fread(buff, 1, 999, f );
  printf("Z pliku odczytano %d bajtow", odczytano);
  fclose( f );
}
```

Operacje na bloku danych - zapis

```
char buff[1000];
FILE * f = NULL;
int zapisano;

f = fopen("notatka.txt", "wb");

if ( f != NULL )
{
  zapisano = fwrite(buff, 1, 999, f );

  if(zapisano != 999)
       printf("Blad - zapisano tylko %d bajtow", zapisano);

  fclose( f );
}
```

5 Dynamiczny przydział pamięci

biblioteka **stdlib.h** lub **malloc.h**

```
void* malloc( size_t size );                      // przydziela blok pamięci o rozmiarze size
void* calloc( size_t num, size_t size );          // przydziela blok pamięci o rozmiarze size*num
void  free( void *memblock );                     // zwalnia blok pamięci
void* realloc( void *memblock, size_t size );// zmienia rozmiar bloku pamięci
```

Tablice dynamiczne:
- można tworzyć w trakcie działania programu,
- można podczas tworzenia wyznaczać dowolny ich rozmiar (o ile jest miejsce w pamięci),
- są oszczędne - nie trzeba przydzielać więcej pamięci, niż to jest w danej chwili potrzebne.

Tablice dynamiczne trzeba zwalniać, gdy są niepotrzebne (np. przy wyjściu z programu).

```
char *dlugi_tekst;

dlugi_tekst = (char*) malloc( 100 );

if( dlugi_tekst == NULL )
{
  printf( "blad!" );
  exit(0);
}

dlugi_tekst[0] = 0;
strcpy( dlugi_tekst, "asdf" );

free( dlugi_tekst );
```

```
// użytkownik zażądał wprowadzenia do bufora tekstu z dużego pliku, większego niż utworzony wcześniej bufor
int nowy_rozmiar;

nowy_rozmiar = 1000;
dlugi_tekst = realloc( dlugi_tekst, nowy_rozmiar );
```

Allokacja tablicy liczb (np. typu long)
(rozmiar zmiennej long – 4 Bajty)

```
long *tab = NULL;
int rozmiar = 100;

tab = (long*) malloc( rozmiar * sizeof(long) );
```
lub
```
tab = (long*) malloc( 100 * 4 );
```
lub
```
tab = (long*) calloc( rozmiar,  sizeof(long) );
```
lub
```
tab = (long*) calloc( 100,  4 );
```

Zwolnienie tablicy liczb (np. typu long)

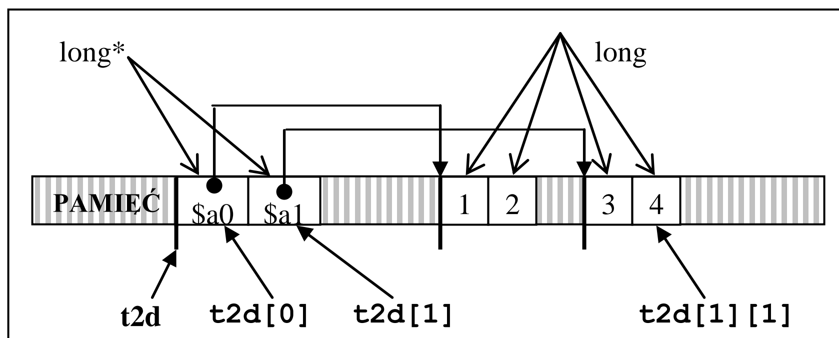
```
free( tab );
```

Allokacja tablicy dwuwymiarowej (np. typu long)

```
int n = 5, m = 4;  // wymiary tablicy
int i;

long **t2d = NULL;

// tworzenie tablicy wskaźników do tablic z liczbami long
// elementy tablicy są wskaźnikami long* do tablic zmiennych typu long
t2d = (long**) malloc ( n * sizeof( long*) );

// do każdego elementu w tablicy są wpisywane wskaźniki nowotworzonych tablic zmiennych typu long

for(i =0; i<n; i++ )
  t2d[i] = (long*) malloc ( m * sizeof( long ) );
```

Tablica jest już gotowa. Dostęp do tablicy:

```
t2d[1][1] = 5;
```



Zwalnianie tablicy dwuwymiarowej

```
// zwalniamy najpierw tablice zmiennych typu long, do  których mamy wskaźniki
for(i =0; i<n; i++ )
     free( t2d[i] );

// zwalniamy tablicę wskaźników do tablic zmiennych typu long
free( t2d );
```

Alokacja tablicy struktur

```
struct student{
  int wiek;
  int nr_indexu;
  char imie[20];
  char nazwisko[20];
};

struct student *grupa = NULL;

int liczba;

// na początek ma być w grupie 3 osoby

liczba = 3;
grupa = (struct student*) calloc( liczba, sizeof( struct student ) );

// dodanie nowej osoby do grupy

liczba++;
grupa = (struct student*) realloc(grupa, liczba);

// zwolnienie tablicy grupy

free( grupa );
```

6. Programy wielomodułowe w C/C++

Kod dużych projektów programistycznych dzieli się na wiele modułów, gdyż programowanie w jednym pliku źródłowym:
- sprawia problemy edycyjne;
- utrudnia szukanie błędu;
- znacznie wydłuża czas kompilacji (używane jest dużo zasobów procesora i pamięci);
- utrudnia dystrybucje;
- uniemożliwia współpracę w grupie programistycznej.

Aby rozbić jeden większy moduł na mniejsze, należy przenieść część kodu do innych (najlepiej powiązanych tematycznie nazwą) plików i utworzyć pliki nagłówkowe, zawierające niezbędne deklaracje.

Pliki źródłowe – rozszerzenia **`*.c, *.cpp`** – zawierają definicje funkcji, definicje zmiennych globalnych
Pliki nagłówkowe – rozszerzenia **`*.h, *.hpp`** - zawierają deklaracje funkcji, makrodefinicje (`#define ...`) oraz deklaracje zmiennych globalnych.

Przyjęte są następujące zasady:

1. plik źródłowy może odczytywać dowolny plik nagłówkowy (#include "mojnagl.h"),
2. definicja typu i funkcji nie może się powtórzyć w całym projekcie,
3. deklaracje zmiennych, typów i funkcji nie mogą się powtórzyć w obrębie jednego pliku źródłowego.

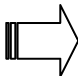
Do kompilacji modułu wystarczy deklaracja funkcji, które są w nim używane.
Do łączenia modułów w program wykonywalny *.exe potrzebne są moduły, w których są definicje wszystkich używanych funkcji.

Uwaga! Zmienne globalne należy definiować tylko w jednym pliku źródłowym. Ich deklaracje w plikach nagłówkowych lub innych źródłowych należy poprzedzić słowem kluczowym **`extern`** , gdyż w przeciwnym przypadku zostanie utworzone kilka jej kopii w pamięci i podczas łączenia zostanie zasygnalizowany błąd 'already defined' !

Przykład:

mojprog.c

```
#include <stdio.h>

int x, y;
int dx, dy;

int funkcja_1(int a)
{
	int b=10;
	…
	return a*b*dx;
}

int funkcja_2(int a)
{
	int b=100;
	…
	return a-b-dx;
}

int funkcja_3(int a)
{
	int b=1000;
	…
	return a+b+dy;
}

int funkcja_4(int a)
{
	int b=77;
	…
	return a%b+dy;
}


void main(void)
{
	int u, w;
	u = 1;
	w = 2;

	printf("start");

	x = funkcja_1(u) + funkcja_2(w);
	y = funkcja_3(u) + funkcja_4(w);
	...
}
```



mojefun.h

```
extern int x, y;
extern int dx, dy;

int funkcja_1(int a);
int funkcja_2(int a);
int funkcja_3(int a);
int funkcja_4(int a);
```

mojefun.c

```
int x, y;
int dx, dy;

int funkcja_1(int a)
{
	int b=10;
	…
	return a*b*dx;
}

int funkcja_2(int a)
{
	int b=100;
	…
	return a-b-dx;
}

int funkcja_3(int a)
{
	int b=1000;
	…
	return a+b+dy;
}

int funkcja_4(int a)
{
	int b=77;
	…
	return a%b+dy;
}
```

mojprog2.c

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

void main(void)
{
	int u, w;
	u = 1;
	w = 2;

	printf("start");

	x = funkcja_1(u) + funkcja_2(w);
	y = funkcja_3(u) + funkcja_4(w);
	...
}
```

W przypadku wielokrotnego dołączania tego samego fragmentu kodu można zabezpieczyć się makrodefinicjami (warunkwymi):

```
#ifndef DEFINICJA_MOJEJ_STRUKTURY
#define DEFINICJA_MOJEJ_STRUKTURY

struct moja_struktura{
  int x;
  int y;
};

#endif
```

Powyższy kod pozwala na zdefiniowanie struktury w nagłówku, który jest dołączany wielokrotnie do tego samego modułu bez powodowania błędu 'struct moja _struktura already defined'.