KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Projekt współfinansowany ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

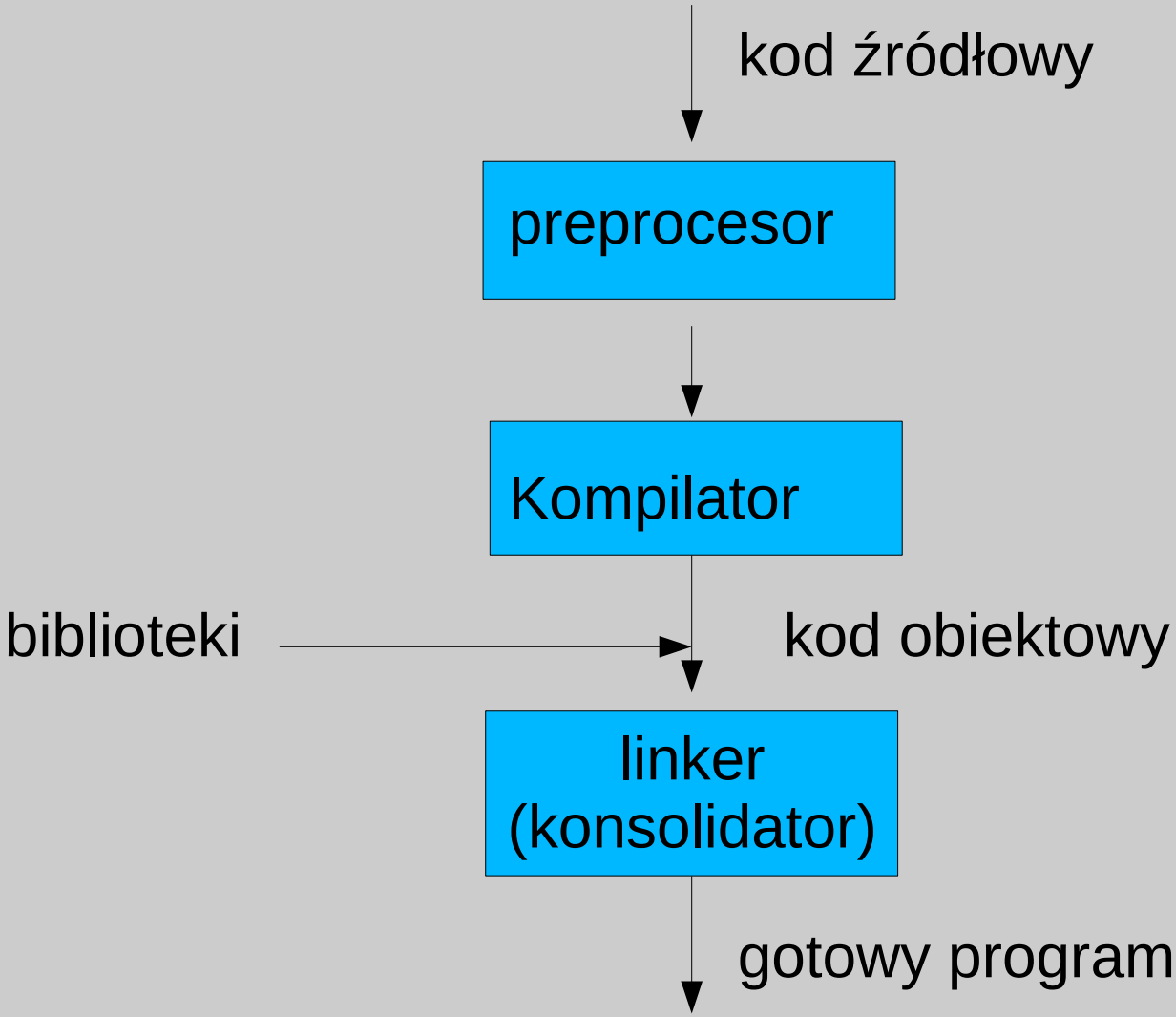
Języki Programowania

Literatura

Richard P. Feynman, Wykłady o obliczeniach
D. Harell, Rzecz o istocie informatyki
B. Kernighan, D Ritchie, Programowanie w języku C
S. Prata, Język C. Szkoła programowania.

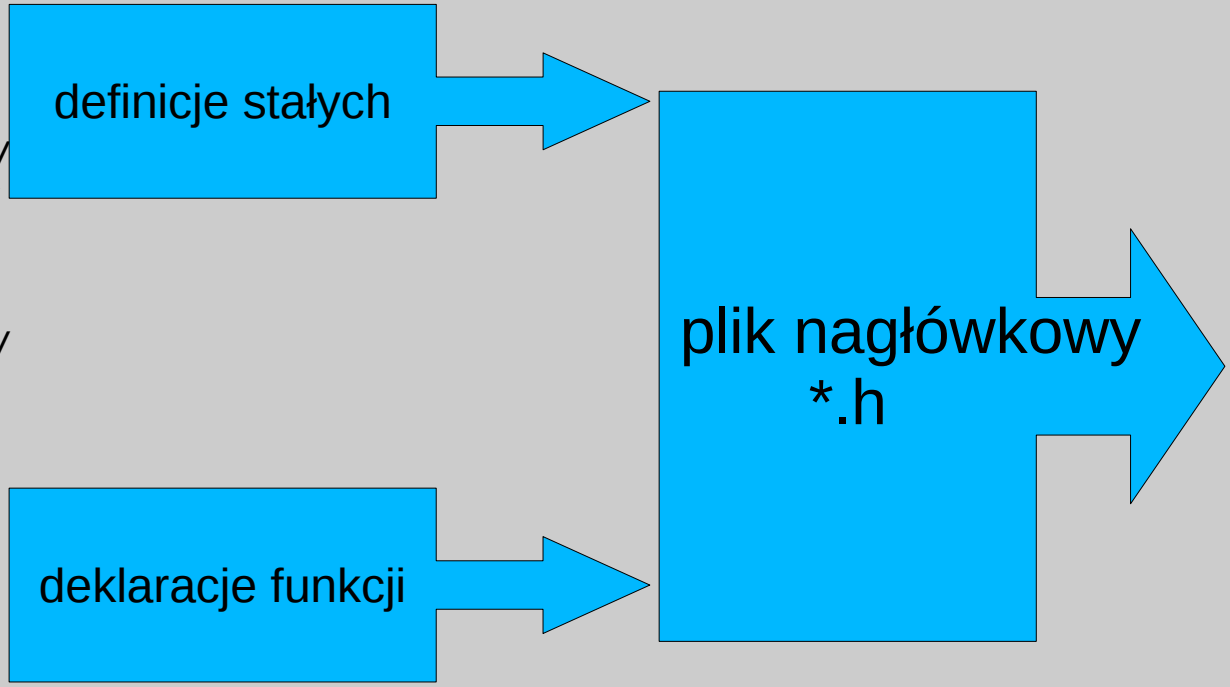
www.fizyka.umk.pl/~raad/jezyki.pdf

Model kompilacji w języku C



Struktura programu

```
/*dyrektywy preprocesora*/
/****************************************/
#include <stdio.h>
#include <stdlib.h>

#define MAX 100
/****************************************/
/* deklaracje zmiennych globalnych*/
int i,j;
float p;
/****************************************/
/*deklaracje lub definicje funkcji*/
void drukuj(char *);
int wartoscMax(int []);
float dodoawanie(float [],float []);
/****************************************/
/*funkcja główna*/
int main()
{
        return 0;
}
/*definicje funkcji (jeśli nie zostały wykonane przed main*/
void drukuj(char * napis)
{
}
```



Zmienne, typy i rozmiary

```
#include <stdio.h>
main()
{
        int i,j,k;

        i=10;j=20;
        k=i+j;
        printf("Wynik dodawania :%d",k);
}
```

Deklaracja zmiennej:

typ lista zminennych oddzielonych przecinkiem;

```
int i;
char k,l;
float tab1,tab2;
```

Każda zmienna może zostać zainicjalizowana podczas deklaracji

```
int i=0;
char k='a',l='w';
float tab1=1.0e-5;
```

Zmiennym zewnętrznym i statycznym, jeśli przypisanie nie zostało wykonane, przypisana jest wartość 0. Jeżeli zmienna jest lokalna i nie została zainicjalizowana to ma wartość niezdefiniowaną (jakieś śmieci)

Wszystkie zmienne muszą być zadeklarowane, nazwa zmiennej może składać się z liter i cyfr, pierwszy znak musi być literą. Duże i małe litery są odróżniane. Tak więc deklaracje
int k;
int K;
reprezentują dwie różne zmienne.
Można zadeklarować zmienną (stałą), która nie może ulec zmianie w trakcie działania programu (może być potrzebne np. przy definiowaniu stałych fizycznych czy matematycznych)

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

Tablice

Tablice czyli zbiór zmiennych tego samego typu

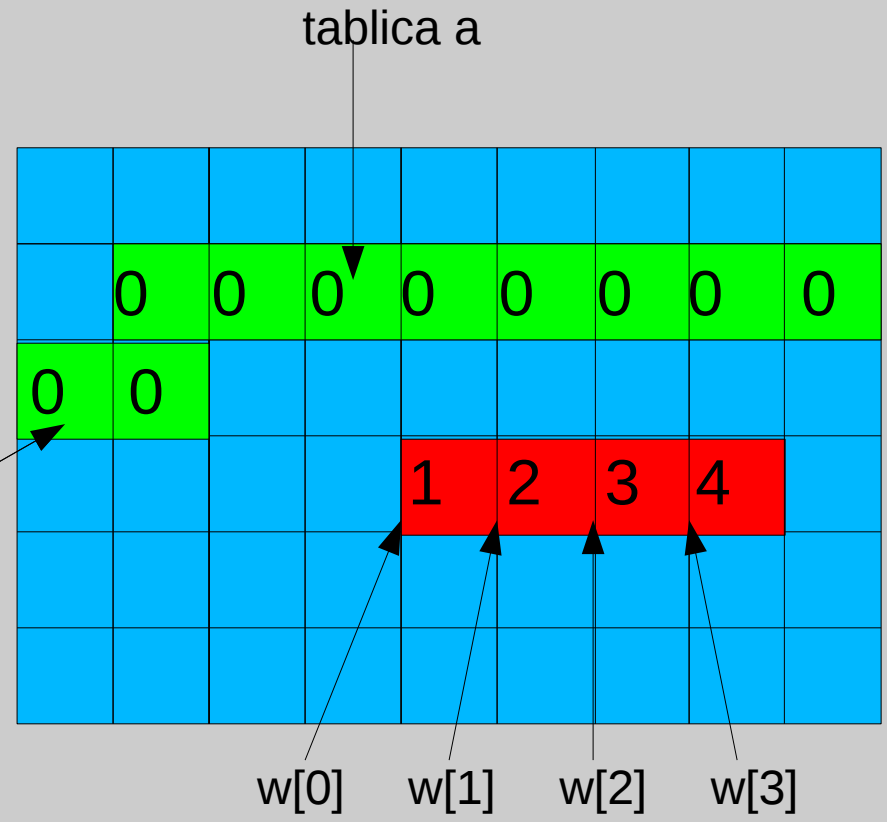
Deklaracja tablicy:

int a[10];
float x[20];
char b[15];
int p[3][3];

Deklaracja z inicjalizacją:

int w[4]={1,2,3,4};
int p[3][3]={{1,2,3},{3,1,1},{4,2,1}};

lub *int p[3][3]={1,2,3,3,1,1,4,2,1};*
int w[2][2]={0};



komórka pamięci
adres fizyczny : 0xFFEF
adres indeksowy: a[8]
wartość: 0

p[3][3]

p[0][0]	p[0][1]	p[0][2]
p[1][0]	p[1][1]	p[1][2]
p[2][0]	p[2][1]	p[2][2]

```
void main()
{
        int a[3][3]={{1,0,0},{0,1,0},{0,0,1}};

        printf("%d %d%d\n",a[0][0],a[0][1],a[0][2]);
        printf("%d %d%d\n",a[1][0],a[1][1],a[1][2]);
        printf("%d %d%d",a[2][0],a[2][1],a[2][2]);
}
```

100
010
000

Instrukcja warunkowa

```
if (warunek)
        instrukcja-1
else
        instrukcja-2
```

```
if (warunek)
{
        instrukcja-1
        .........
}
else
{
        instrukcja-2
        ..........
}
```

Warunek jest prawdziwy, wtedy gdy jest wartością różną od zera, może to być wartość jakiejś zmiennej, wynik relacji lub wynik operacji logicznej.

```
x=10;
if (x)
        instrukcja-1
```

```
if (x>=5)
        instrukcja-1
```

```
if (x>=5 && x<20)
        instrukcja-1
```

Formatowanie

styl K&R

```
if(warunek){

}
```

styl BSD

```
if(warunek)
{

}
```

```
main() {if(k==0) { printf("Napis");}}
```

```
main()
{
                if(k==0)                {
 printf("Napis");}}
```

Formatowanie

styl K&R

```
if(warunek){

}
```

styl BSD

```
if(warunek)
{

}
```

```
main() {if(k==0) { printf("Napis");}}
```

```
main()
{
                if(k==0)                {
 printf("Napis");}}
```

Zagnieżdżanie warunków

```
if(warunek-1)
{
        if(warunek-2)
                if(warunek-3)
                        instrukcja-1
                else
                        instrukcja-2
}
else
        instrukcja-3
```

```
if(warunek-1)
{
        if(warunek-2)
                instrukcja-1
        else
                if(warunek-3)
                        instrukcja-2
                else
                        instrukcja-3
}
else
        instrukcja-4
```

Operator ?:

$expr_1$? $expr_2$: $expr_3$

Najpierw sprawdzany jest warunek $expr_1$. Jeżeli jest niezerowy (prawdziwy) , wówczas wykonywany jest $expr_2$ i to jest wynikiem tego warunku, w przeciwnym razie wykonywany jest $expr_3$ i on jest wynikiem warunku.
Przykład:

```
z=(a>b) ? a : b; /* z=max(a,b); nawiasy w wyrażeniu (a>b) nie są konieczne*/
```

Zapis tego samego za pomocą if:

```
if(a>b)
        z=a;
else
        z=b;
```

jeszcze inny zapis:

```
z=b;
if(a>b)
        z=a;
```

Instrukcja switch

```
switch(wyrażenie)
{
        case wartość-1:
                        instrukcja-1;
                        instrukcja-2;
                        break;
        case wartość-2:
                        instrukcja-1;
                        instrukcja-2;
                        break;
        .....................................
        default:
                        instrukcja-3;
                        instrukcja-4;


}
```

Zastosowanie instrukcji switch

```
#include <stdio.h>
void main()
{
        char c;
        c=getchar();
        switch(c)
        {
                case ' ':
                        printf("Wcisnieto spacje\n");
                        break;
                case '0':
                case '1':
                case '2':
                case '3':
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9':
                        printf("Wcisnieto cyfre\n");
                        break;
                default:
                        printf("Niewlasciwy znak\n");

        }
}
```

Pętla for

Składnia:

```
for(wyrażenie1;wyrażenie2;wyrażenie3)
        instrukcja;
```

```
for(wyrażenie1;wyrażenie2;wyrażenie3)
{
        instrukcja1;
        instrukcja2;
        .....................
}
```

W ogólności wszystkie 3 parametry instrukcji for są wyrażeniami, najczęściej jednak wyrażenie1 i wyrażenie3 są wyrażeniami przypisania, natomiast wyrażenie2 jest wyrażeniem relacyjnym.
Każde z wyrażeń może zostać pominięte, ale średniki muszą pozostać, czyli w skrajnym przypadku może to wyglądać tak:

```
for(;;);/*średnik w tym miejscu oznacza,że jest to koniec pętli więc żadna instrukcja nie będzie wykonywana*/
```

będzie to pętla, która nigdy się nie kończy!

Przykłady

```
/*Program wypisujacy liczby od 0 do
100, kazda liczba w odobnej linii */

#include <stdio.h>

void main()
{
        int i;


        for(i=0;i<=100;i++)
        {
                printf("i=%d\n",i):
        }
}
```

```
/*Program oblicza nastepujacy szereg
0+1+2+3+....+100 */

#include <stdio.h>

void main()
{
        int i,suma;
        /*sume mozna wyzerowac podczas dekla
        racji lub przed petla for*/
        for(i=0,suma=0;i<=100;i++)
        {
                suma+=i;
        }
        printf("Suma od 0-100 wynosi: %d",suma);
}
```

```
/*Obliczanie wartosci x^n, gdzie x=10, n=5;*/
#include <stdio.h>

void main()
{
	int i;
	float x,pow;

	x=10;
	pow=1;
	for(i=0;i<5;i++)
		pow*=x;

	printf("%f do potegi 5 wynosi: %f\n",x,pow);
}
```

while()

while(wyrażenie)

instrukcja;

Pętla powtarzana jest tak długo jak długo wyrażenie jest prawdziwe

```
#include <stdio.h>

void main()
{
        char a;

        while((a=getchar())!='k')
        {
                printf("nacisnieto : %c",a);
        }
}
```

do while()

```
do
        instrukcja;
while(wyrażenie);
```

Pętla wykonywana jest co najmniej raz tak długo jak długo wyrażenie jest prawdziwe (różne od zera)

```
#include <stdio.h>

void main()
{
        char a;

        do
        {
                a=getchar();
                printf("nacisnieto : %c",a);
        }
        while(a!='k');
}
```

break i continue

Są to instrukcje pozwalające przerwać pętle (break), lub ominąć fragment pętli (continue).

```
for(i=0;i<10;i++)
{
        sum+=t[i];
        if(sum>20)
                break;
}
```

```
for(i=0;i<10;i++)
{
        if(t[i]<0)
                continue;
        sum+=t[i];
}
```

```
for(i=0;i<10;i++)
{
        for(j=0;j<20;j++)
        {
                for(n=0;n<10;n++)
                {
                        if(tab[i][j][n]>20)
                                break;
                }
        }
}
```

GOTO

Instrukcja goto pozwala przejść z dowolnego miejsca w programie w dowolne inne (oznaczone etykietą), czyli np pozwala wyjść z zagnieżdżonych pętli.

```
for(i=0;i<10;i++)
{
        for(j=0;j<20;j++)
        {
                for(n=0;n<10;n++)
                {
                        if(tab[i][j][n]>20)
                                goto KONIEC_P;
                }
        }
}
KONIEC_P:
....
```

Z uwagi na małą przejrzystość kodu, w którym używany jest GOTO istnieje nie pisana zasada aby tej instrukcji nie używać lub też robić to tylko w ostateczności

Funkcje

Funkcje służą do zamknięcia fragmentu kodu, często powtarzanego, lub oddzielenia fragmentów kodu stanowiącego logiczną całość.
Składnia:

```
zwracany typ nazwa_funkcji(deklaracja parametrów, jeżeli są potrzebne)
{
        deklaracje zmiennych;

        instrukcje
}
```

Wartość zwracana przez funkcje jest to wartość występująca po instrukcji return. Po return może nastąpić dowolne wyrażenie lub nie występować nic.

```
void nicNieRob(){}
void nicNieRob(){return;}
```

```
int suma(int i)
{
        int j,sum;
        for(j=0,sum=0;j<=i;j++)
        {
                sum+=i;
        }
        return sum;
}
```

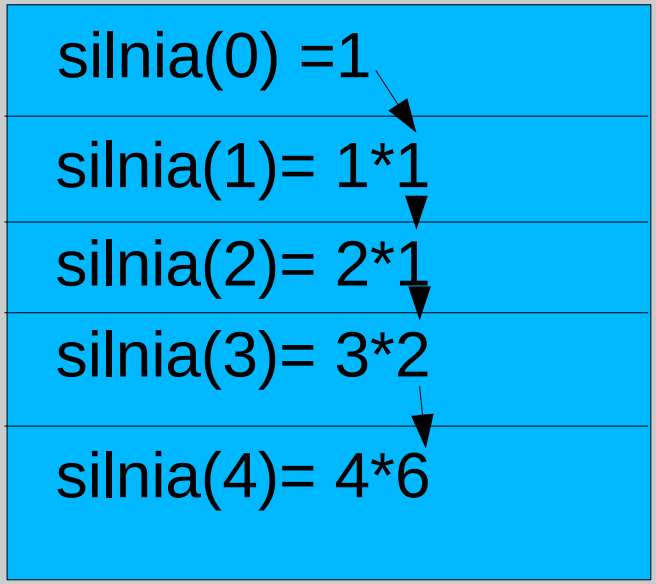
```
#include <stdio.h>

int suma(int od, int koniec)
{
	int i,sum;
	for(i=od,sum=0;i<=koniec;i++)
	{
		sum+=i;
	}
	return sum;
}

void main()
{
	int res;

	res=suma(5,20);
	printf("Rezultat sumowania: %d",res);
}
```

```
int maxElement(int tab[],int s)
{
	int i,x=0;/*lepiej x=tab[0]*/

	for(i=0;i<s;i++)
	{
		if(tab[i]>x)
			x=tab[i];
	}

	return x;
}
```

Funkcje rekurencyjne

Funkcja rekurencyjna - funkcja wywołująca samą siebie (w sposób pośredni lub bezpośredni)

```
int silnia(int i)
{
	int res;

	if(i>0)
		res=i*silnia(i-1);
	else
		res=1;

	return res;
}
void main()
{
	printf("Silnia :%d\n",silnia(5));
}
```



STOS

Rozwiązanie rekurencyjne mogą być wolniejsze od iteracyjnych, zbyt wiele wywołań może prowadzić do przepełnienia stosu, są za to bardziej zwięzłe (zajmują mniej linii kodu) i często łatwiejsze w implementacji. Rekurencje jest szczególnie przydatna przy tworzeniu pewnych typów struktur danych np. drzewa, listy.

Preprocesor

- Ładowanie plików nagłówkowych
- Deklaracja stałych
- Rozwijanie makr
- Warunkowa kompilacja
- Kontrola linii
- Diagnostyka

Pliki nagłówkowe

Każda linia w pliku źródłowym zawierająca:
#include “nazwa_pliku”
lub
#include <nazwa_pliku>
jest zastąpiona zawartością pliku o podanej nazwie, jeżeli nazwa pliku znajduje się w cudzysłowie, to plik ten poszukiwany jest w tym samym miejscu co plik źródłowy, jeżeli nie zostanie tam znaleziony, lub nazwa pliku <> to plik poszukiwany jest na podstawie pewnych reguł zależnych od implementacji.
Dołączony plik również może zawierać #include.
#include powinno być używane tylko do dołączania plików nagłówkowych (*.h), nigdy *.c chociaż jest to możliwe.
Pliki nagłówkowe są to pliki zawierające deklaracje zmiennych, stałych, typów oraz funkcji.

UWAGA! Jeżeli plik nagłówkowy pojawi się 2 razy to dwukrotnie zostanie załadowany.

Deklaracja stałych

Stałą można zadeklarować używając słowa kluczowego const jak również:
#define LICZBA 100
#define ZNAK 'a'
Jeszcze inny sposób definiowania stałych

```
enum {TAK,NIE}; /*pierwsza wartość wynosi 0*/
enum  miesiac {STYCZEN=1,LUTY,MARZEC, KWIECIEŃ, MAJ, CZERWIEC,
                LIPIEC,SIERPIEN,WRZESIEN,PAZDZIERNIK,LISTOPAD,
                GRUDZIEN};
```

```
#include <stdio.h>
#define MAX 100

void main()
{
        int x=MAX;
}
```

```
#include <stdio.h>
enum {TAK,NIE};

void main()
{
        int x=TAK;
}
```

Makra

- Definicja makra kończy się wraz z końcem linii, w której znajduje się #define. Można to zmienić używając znaku \

```
#define NUMBERS 1, \
                2, \
                3
```

- Makra rozwijane są sekwencyjnie

```
#define BUFSIZE 1020
#define TABLESIZE BUFSIZE
#undef BUFSIZE
#define BUFSIZE 37
```

Instrukcje preprocesora

Tworzenie makr:
#define nazwa podmieniany tekst
np.

#define FOREVER for(;;)

Można zdefiniować makro

#define MAX(A,B) ((A)>(B) ? (A):(B))

Co jeśli ktoś wywoła to makro tak MAX(X++,Y++); ???

#define MY_ABS(A) ((A)>0 ?(A):-(A))

```
#include <stdio.h>
#define MY_ABS(A) (A)>0 ? (A):-(A))

void main()
{
        x=-10;
        printf(“abs=%d”,MY_ABS(x));
}
```

Makra jak funkcje

Po nazwie makra umieszczane są nawiasy, jeżeli po nazwie zostanie umieszczona spacja to makro nie jest już makrem-funkcją

```
#define lang_init ()    c_init()
    lang_init()
        ==> () c_init()()
```

Makro-funkcje mogą mieć argumenty

```
#define MY_ABS(A) ((A)>0 ? (A):(-A)) /* Nie działa poprawnie*/

MY_ABS(x+3)  /* (-10+3)>0 ? (-10+3): (--10+3)   !!!!*/

#define SQUARE(A) A*A /*również niepoprawne */

SQUARE(x+1) /* -10+1*-10+1*/

#define SQUARE(A) (A)*(A)
```

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))

    min (min (a, b), c)
     ((((a) < (b) ? (a) : (b))) < (c)
        ? (((a) < (b) ? (a) : (b)))
          : (c))

          min(, b)       ==> ((   ) < (b) ? (   ) : (b))
          min(a, )       ==> ((a  ) < ( ) ? (a  ) : ( ))
          min(,)         ==> ((   ) < ( ) ? (   ) : ( ))
          min((,),)      ==> (((,)) < ( ) ? ((,)) : ( ))

    #define foo(x) x, "x"
          foo(bar)       ==> bar, "x"
```

Makro traktowane jako funkcja

```
#define SKIP_SPACES(p, limit)  \
    { char *lim = (limit);         \
      while (p < lim) {            \
        if (*p++ != ' ') {         \
          p--; break; }}}

if (*p != 0)
      SKIP_SPACES (p, lim); //BŁĄD!!!
    else …
```

Rozwiązanie

```
#define SKIP_SPACES(p, limit)     \
    do { char *lim = (limit);         \
         while (p < lim) {            \
           if (*p++ != ' ') {         \
             p--; break; }}}          \
    while (0)
```

Podwójne wywołania

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
        next = min (x + y, foo (z));
next = ((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

Kolejne makro

```
#define SWAP(a, b)  a ^= b; b ^= a; a ^= b;
```

To wywołanie działa poprawnie

```
Swap(x,y);
```

Teraz działanie niepoprawne

```
if(x>5)
     Swap(x,y);
```

Lepiej zdefiniować makro tak

```
#define SWAP(a, b)  {a ^= b; b ^= a; a ^= b;}
```

Niestety w tym przypadku też nie będzie działać

```
Z=10;
X=5;
Y=6;

if(x>5)
     Swap(x,y);
Else
     Swap(x,z);
```

Makro powinno więc być napisane tak:

```
#define SWAP(a, b)  do{a ^= b; b ^= a; a ^= b;} while(0)
```

Zamiana argumentu na string

Żeby potraktować argument makra jako string trzeba użyć znaku #

```
#define str(s) #s
#define foo 4
    str(foo)
        ==> „foo”
#define xstr(s) str(s)
    xstr(s)
        ==>xstr(4)
        ==>str(4)
        ==>”4”
```

Predefiniowane makra

- __DATE__ łańcuch znaków zawierający aktualną datę
- __FILE__ łańcuch znaków zawierający nazwę pliku
- __LINE__ liczba całkowita reprezentująca numer linii kodu
- __TIME__ łańcuch znaków zawierający aktualną godzinę

Preprocesor i wyrażenia warunkowe

Instrukcja:

```
#if stała (wyrażenie całkowite)

#endif
```

```
#if stała (wyrażenie całkowite)

#else

#endif
```

Inne słowa kluczowe które mogą być użyte #ifndef, #elif
Zastosowanie np. kompilacja warunkowa, definiowanie plików nagłówkowych

```
#if SYSTEM == BSD
        #define TIME “time.h”
#elif SYSTEM == MSDOS
        #define TIME “tim.h”
#endif

#include TIME
```

Diagnostyka

- Dyrektywa #error

```
#ifdef __WINDOWS__
    #error "Nie można kompilować pod WINDOWS”
#endif
```

- Dyrektywa #warning

Podobnie jak #error ale nie przerywa preprocesingu

Zmienne globalne

Zmienne globalne – zmienne deklarowane poza funkcją “widziane” przez wszystkie funkcje, globalnie dostępne.
Zmienne lokalne są dostępne wewnątrz bloku czy funkcji poza przestają istnieć. Czas “życia” zmiennych lokalnych jest równy czasowi działania programu, zmiana zmiennej globalnej w jednej funkcji będzie widziana przez wszystkie pozostałe.

definicja zmiennej globalnej

int x;
float tab[10];

deklaracja zmiennej globalnej

extern int x;
extern float tab[];

Deklaracja opisuje właściwości zmiennej (np. typ) definicja z dodatkowo wymusza rezerwację pamięci dla tej zmiennej

Zmienne statyczne i rejestrowe

Zmienna lub funkcja statyczna jest deklarowana przy użyciu słowa kluczowego *static.*
Sposób deklaracji:

static int x;

Użycie słowa static do zmiennej lokalnej oznacza, że choć zmienna jest lokalna to jej czas życia (jak i przechowywana wartość) jest taki jak zmiennej globalnej. Ale dostęp do tej zmiennej jest tylko w obszarze życia zmiennej lokalnej.
Użycie *static* dla zmiennej globalnej powoduje że zmienna ta nie będzie widziana przez funkcje w innym pliku źródłowym, podobnie jeżeli użyje się słowa *static* w deklaracji funkcji.
Deklaracja zmiennej ze słowem kluczowym *register* daje sygnał kompilatorowi aby jeśli to możliwe przechowywał tą zmienną w rejestrze porcesora. Kompilator może to jednak zignorować!

register int x;

Zastosowanie static

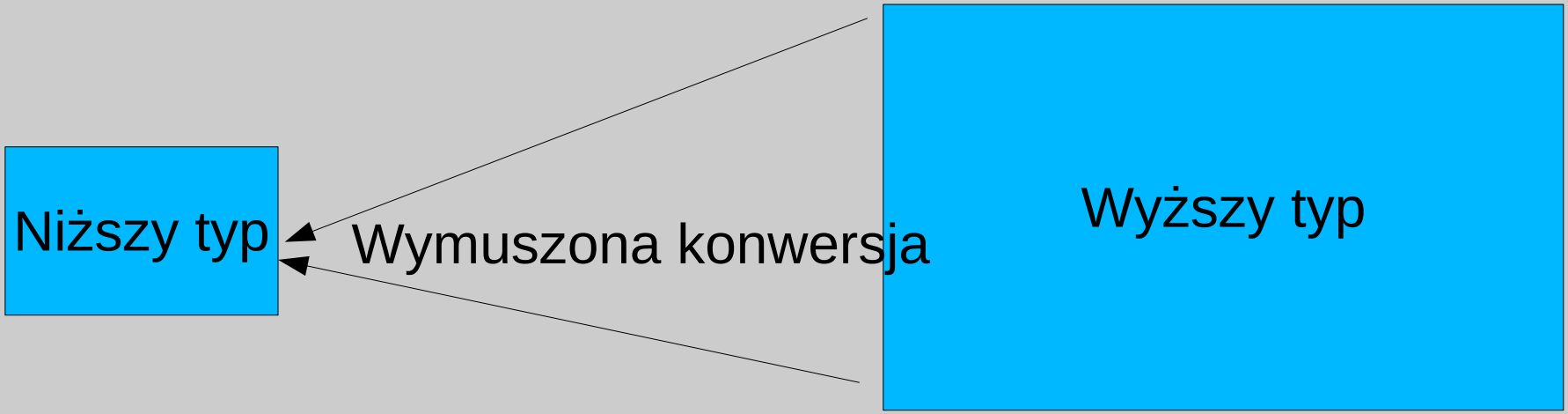
```
float poleOkregu(float r)
{
        static const float pi=3.14;

        return pi*r*r;
}


void ff()
{
        static int k=0;
        int c=0;

        c++;
        k++;
        printf("%d %d",c,k);
}
```

```
void printTr(int szer)
{
        register int i,j;
        const char znak='o';

        for(i=0;i<szer;i++)
        {
                for(j=0;j<i;j++)
                        printf(" ");
                for(j=0;j<szer-i*2;j++)
                                printf("%c",znak);
                for(j=0;j<i;j++)
                        printf(" ");
                printf("\n");
        }
}
```

```
oooooooooo
 oooooooo
  oooooo
   oooo
    oo
```

Rzutowanie (casting)

•int i=10,x;
•float b=2.3556;
•
•x=i+b;
•Gdy operator działa na zmiennych o dwóch różnych typach, są one konwertowane do wspólnego typu na podstawie kilku prostych reguł, dla typów nie zawierających unsigned są one następujące:
• Jeżeli jeden z operatorów jest typu long double, to drugi zostanie przekształcony do long double.
• W przeciwnym razie jeżeli jeden z operatorów jest double, drugi zostanie przekształcony do double
• W przeciwnym razie jeżeli jeden z operatorów jest float, drugi zostanie przekształcony do float
• W przeciwnym razie przekształć char i short w int.
• Następnie jeżeli jedna z wartości jest long, przekształć drugą na long.
•
•Konwersja może zostać wymuszona poprzez użycie operatora rzutowania:
• (*nazwa typu*) *wyrażenie*
•*wyrażenie* jest konwertowane do typu podanego w nawiasach
•




```
#include <stdio.h>
main() {
   int sum = 17, count = 5;
   double mean;
   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

```
#include <stdio.h>
main() {
   int  i = 17;
   char c = 'c'; /* ascii value is 99 */
   int sum;
   sum = i + c;
   printf("Value of sum : %d\n", sum );
}
```

Wskaźniki

Wskaźnik to zmienna, która wskazuje na adres innej zmiennej.
Deklaracja wskaźnika:

*typ *nazwaZmiennej;*

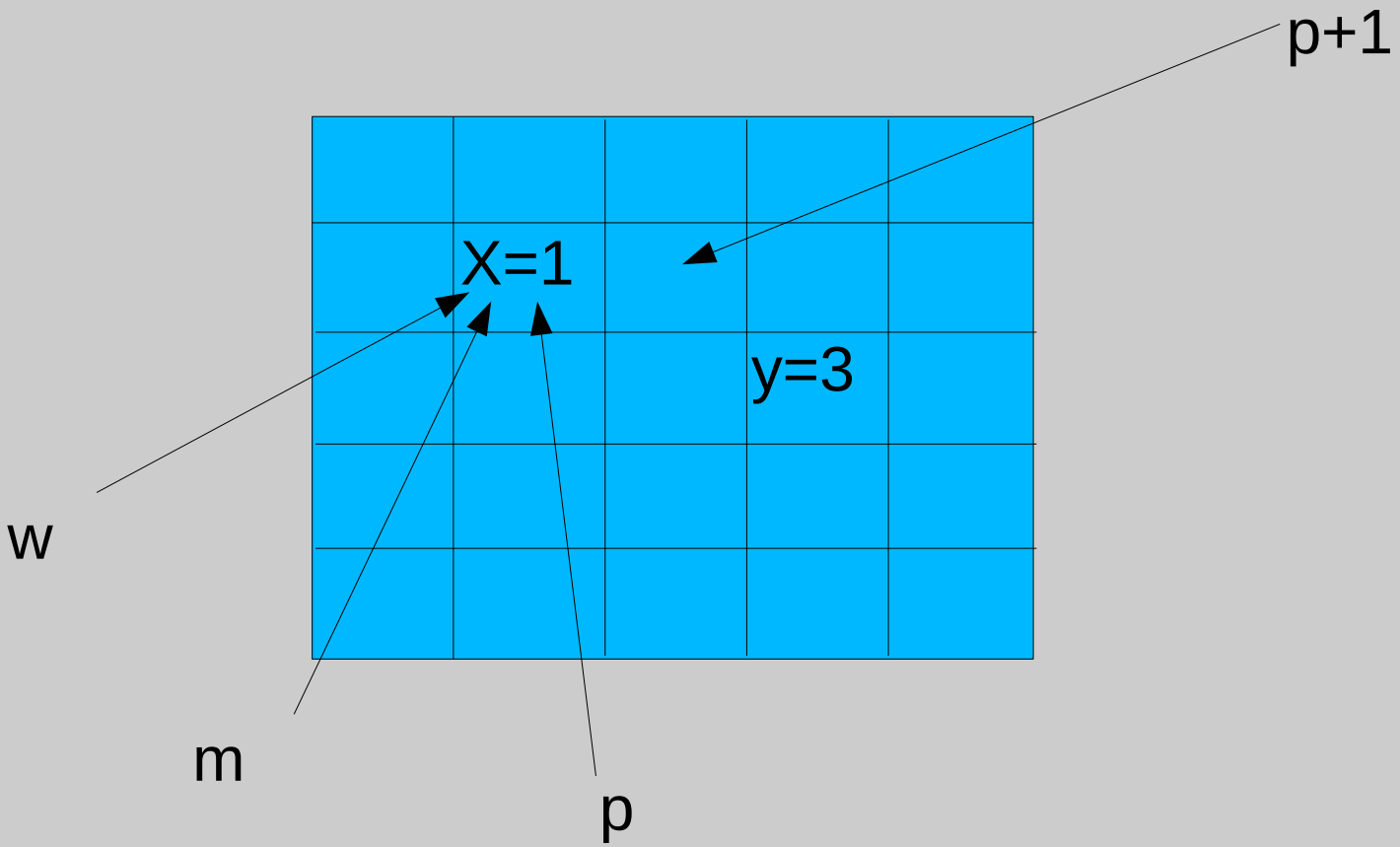
np.

*char *p;*

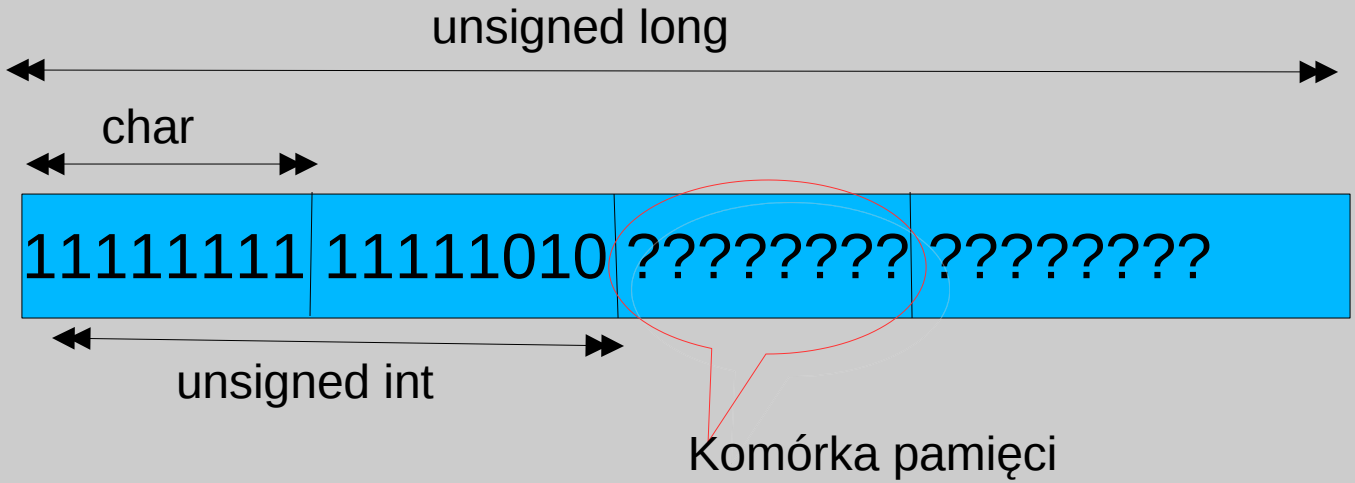
Do pobrania adresu służy operator &

```
int x=1,y=3;
int *p;

p=&x;/* adres zmiennej x przypisany do p, p wskazuje na x*/
y=*p;
```

Użycie * (operator dereferencji) na zmiennej wskaźnikowej powoduje uzyskanie dostępu do obiektu wskazywanego przez wskaźnik.



p*=*w*=*m*=*&x;
**p++; /*Jaką wartość ma x ?*/*
****p*=****p+5;***



*++*p;/*Inkrementacja wartości wskazywanej komórki*/*
p++; /***Inkrementacja adresu , dlaczego???*/*
*(*p)++;/*Inkrementacja wartości wskazywanej komórki*/*

Stałe wskaźniki

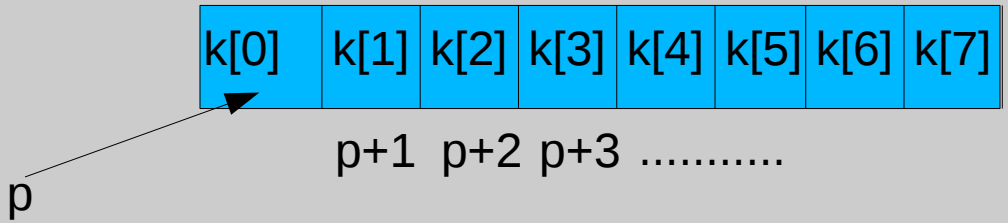
```
 const int *x;
int * const y;

const int *x=&z;
int * const y=&z;

 *x = 1;  /* ŹLE*/
 y = x;   /*ŹLE */
```

Wskaźniki i tablice

Tablice i wskaźniki są silnie ze sobą powiązane, dowolna operacja, którą wykonuje sie poprzez indeksowanie tablicy daje się również zrobić za pomocą wskaźników. Wskaźniki natomiast są szybsze.

*int k[8],*p;*

*p=&k[0]; /*można również p=k */*



k[0] -> *p
k[1] -> *(p+1)
k[2] -> *(p+2)
.
.
.

k -> p
k+1 -> p+1
k+2 -> p+2
.
.
.

Uwaga! wskaźnik jest zmienną, natomiast nazwa tablicy nie, więc operacje typu p++, p=k, są dozwolone, ale k++, nie!

Arytmetyka adresowa

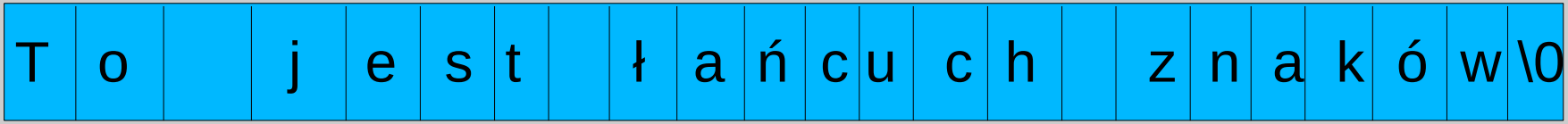
Wskaźnik może zostać zainicjalizowany tak jak każda inna zmienna, wartość inicjalizacji powinna być 0 (C gwarantuje, że 0 nie jest poprawnym adresem dla danych, zamiast 0 używa się NULL stałej specjalnie przeznaczonej dla wskaźników), jeśli wskaźnik nie wskazuje na nic), lub adres wcześniej zaalokowanego obszaru pamięci.

```
int *p=0;

int x=4;
int *p=&x;
```

Wskaźniki mogą być porównywane (operatory == != < >= itd), ale tylko wtedy gdy wskazują na tą samą tablice, w przeciwnym razie wynik tych operacji jest niezdefiniowany. Można natomiast dowolne wskaźniki porównywać z 0.
Do wskaźnika dowolnego typu można dodawać lub odejmować liczbę całkowitą. Odpowiednie przesunięcie zależne od typu (rozmiaru) zmiennej, na którą wskazuje wskaźnik wykonane jest automatycznie.

Łańcuchy

“ To jest napis” - jest to stała łańcuchowa, która wewnętrznie jest reprezentowana przez tablice. Łańcuch znaków musi być zakończony znakiem '\0'.



*char znaki[]=”łańcuch znaków” /*tablica*/*

znaki[0]='p';

*char *pZnaki=”łańcuch znaków” /*wskaźnik na stałą łańcuchową*/*

*pZnaki[0]='p'; /*źle, tak nie wolno*/*

Kopiowanie łańcuchów

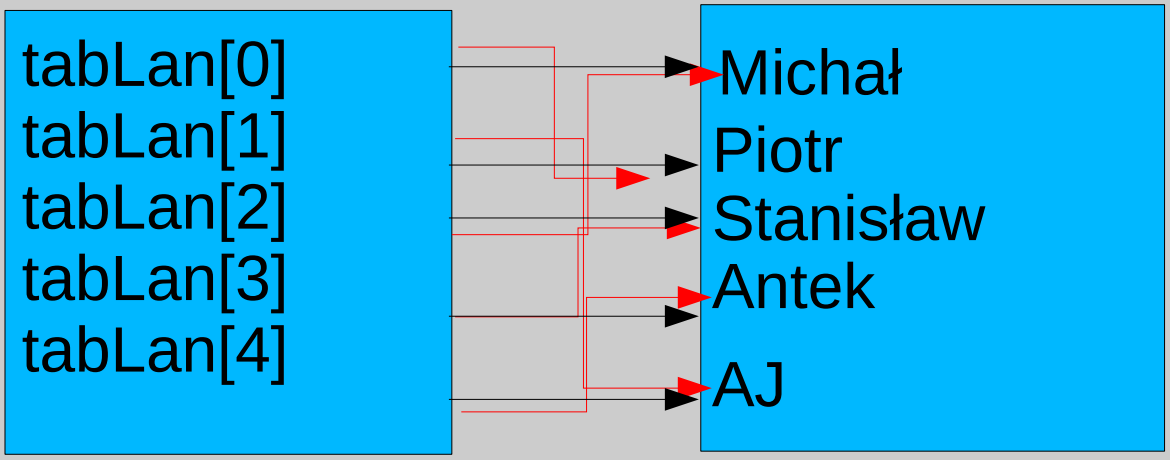
```
void strcpy(char *dest,char *source)
{
        int i=0;
        while((dest[i]=source[i])!='\0')
        {
                i++;
        }
}

void strcpy(char *dest,char *source)
{
        while((*dest=*source)!='\0')
        {
                dest++;
                source++;
        }
}

void strcpy(char *dest,char *source)
{
        while(*dest++=*source++);
}
```

Długość łańcucha

```
int strlen(char *s)
{
        char *p=s;

        while(*(p++));

        return p-s;
}

int strcmp(char *s,char *t)
{
        for(;*s==*t;s++,t++)
                if(*s=='\0')
                        return 0;

        return *s-*t;
}
```

Deklaracja:

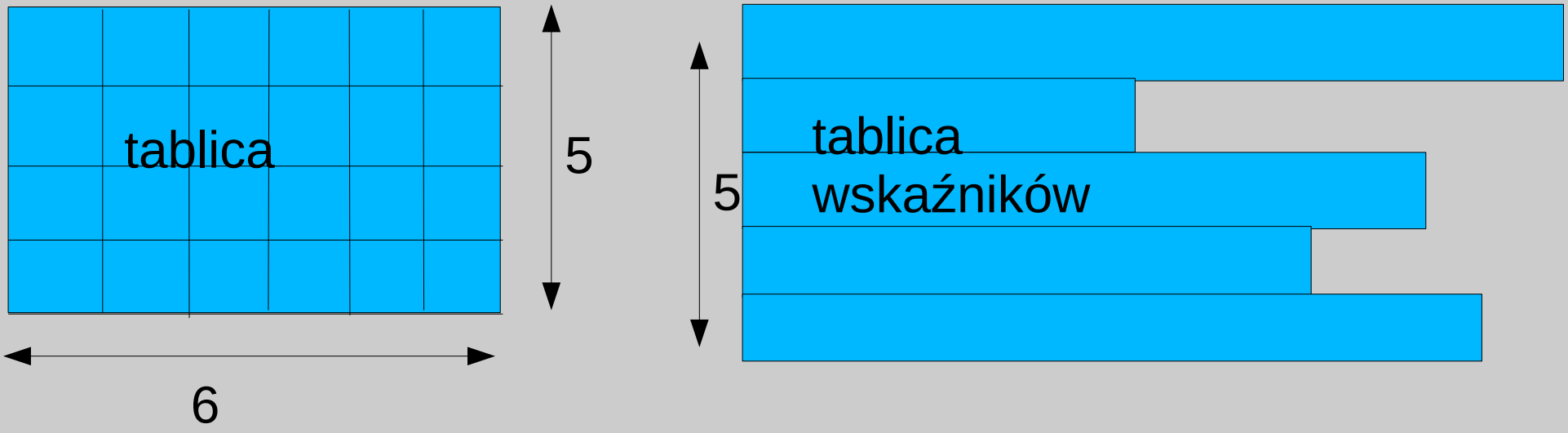
```
int *p[10];
char *tabLan[]={“Michał”,”Piotr”,”Stanisław”,”Antek”,”AJ”};
```



Wskaźniki i wielowymiarowe macierze

int tab[5][6];
*int *p[5]; /*Tablica wskaźników, uwaga!!! wskaźniki nie są zainicjalizowane*/*

*int (*p)[6]; /*wskaźnik na 6 elementową tablice*/*



```
int tab[3][2]={{1,2},{3,4},{5,6}};

int main()
{
        int *p;

        p=&tab[0][0];/* Lub p=(int *) tab */
        for(i=0;i<6;i++,p++)
                printf("%d ",*p);

        printf("\n");

        return 0;
}
```

Przekazywanie wartości do funkcji

C przekazuje parametry do funkcji przez wartości, co oznacza że wywoływana funkcja nie może zmienić wartości zmiennej, która jest parametrem funkcji.

```
void przestaw(int a,int b)/*taka deklaracja
daje działanie inne od spodziewanego*/
{
        int tmp;

        tmp=a;
        a=b;
        b=tmp;
}
```

```
void przestaw(int *a,int *b)
{
        int tmp;

        tmp=*a;
        *a=*b;
        *b=tmp;
}
```

Wywołanie:

```
int x=3,y=6;
przestaw(x,y);
```

```
int x=3,y=6;
przestaw(&x,&y);
```

Używając wskaźników dokonujemy przekazywania parametrów do funkcji przez zmienną a nie przez wartość!!

```
void printTab(int tab[],int k)
{
        int i;

        for(i=0;i<k;i++)
        {
                printf("%d ",tab[i]);
        }
}

void printTab(int *tab,int k)
{
        int i;

        for(i=0;i<k;i++,tab++)
        {
                printf("%d ",*tab);
        }
}

int tab[3][3]={{1,2,3},{2,3,4},{4,5,6}};

void main()
{
        printTab(&tab[1][0],3);
}
```

```
int sumEl(int *tab,int s)
{
	int *tmp,sum;

	tmp=tab;
	/*Najpierw drukujemy tablice*/
	for(i=0;i<s;i++,tmp++)
		printf(”%d ”,*tmp);

	/*Teraz sumujemy wszystkie elementy*/
	for(sum=0;tab!=tmp;tab++)
		sum+=*tab;

	return sum;
}
```

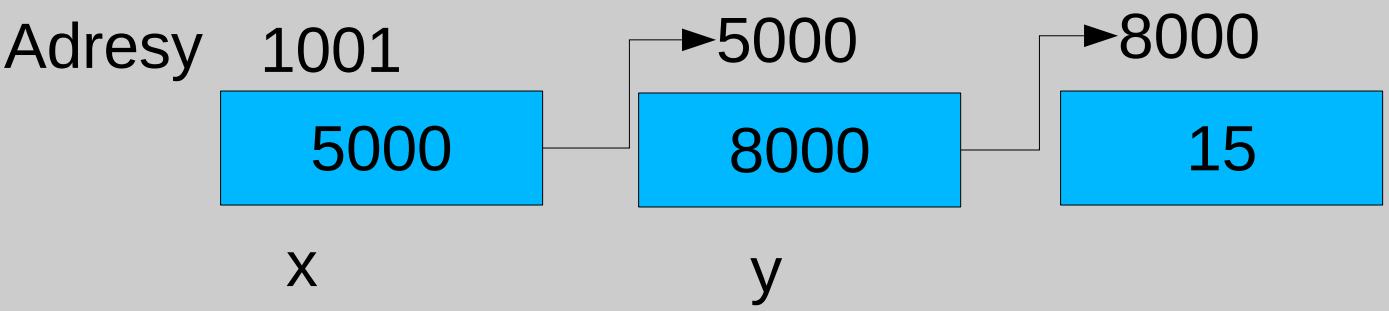
```
float traceM(float **tab,int k)
{
        int i;
        float sum;

        for(i=0,sum=0;i<k;i++)
                sum+=tab[i][i];

        return sum;
}

 float traceM(float *tab,int k)
 {
         int i;
         float sum;

         for(i=0,sum=0;i<k*k;i+=k+1)
                  sum+=*(tab+i);

         return sum;
 }
```

Sortowanie tablicy

```
void sortuj(int *tab,int s)
{
	int test,i;

	do
	{
		test=0;
		for(i=0;i<s-1;i++)
		{
			if(tab[i]>tab[i+1])
			{
				przestaw(tab+i,tab+i+1);
				test=1;
			}
		}
		s--;
	}
	while(test);
}
```

Wskaźnik na wskaźnik na ...

```
int **x;
```




```
int *y;

    *x=&y;


int ****x;
```

Wskaźniki na typ void

Deklarowanie wskaźników na typ void:

*void *p;*

to powoduje, że p staje się wskaźnikiem uniwersalnym, któremu można przypisać wskaźniki na dowolny typ!

Na wskaźnikach typu void nie można wykonywać żadnych operacji arytmetycznych, ale można je porównywać między sobą i zerem.

Nie można wykonywać dereferencji bez rzutowania:

```
int x;
float f;
void *p = &x;    // p wskazuje na x
*(int*)p = 2;
p = &r;          // p wskazuje na r
*(float*)p = 1.1;
```

```
void przestaw(void *a,void *b)
{
  float tmp;

  tmp=*(float *)a;
  *(float *)a=*(float *)b;
  *(float *)b=tmp;
}
```

```
void przestaw(void  **a,void  **b)
{
  void  *tmp;

  tmp=*a;
  *a=*b;
  *b=tmp;
}
```

Dynamiczna alokacja pamięci

Funkcja alokująca pamięć:

*void * malloc(size_t size);*

Funkcja nie czyści alokowanego obszaru. Można też użyć calloc. Konieczne jest załadowanie <stdlib.h>

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

int main()
{
	int *tmp;
	int x;
	printf(”Podaj rozmiar tablicy : ”);
	scanf(”%d”,&x);

	tmp= malloc(x*sizeof(int));
	if(!tmp)
		printf(”Pamiec nie zostala zaalokowana!”);

	free(tmp);

	return 0;
}
```

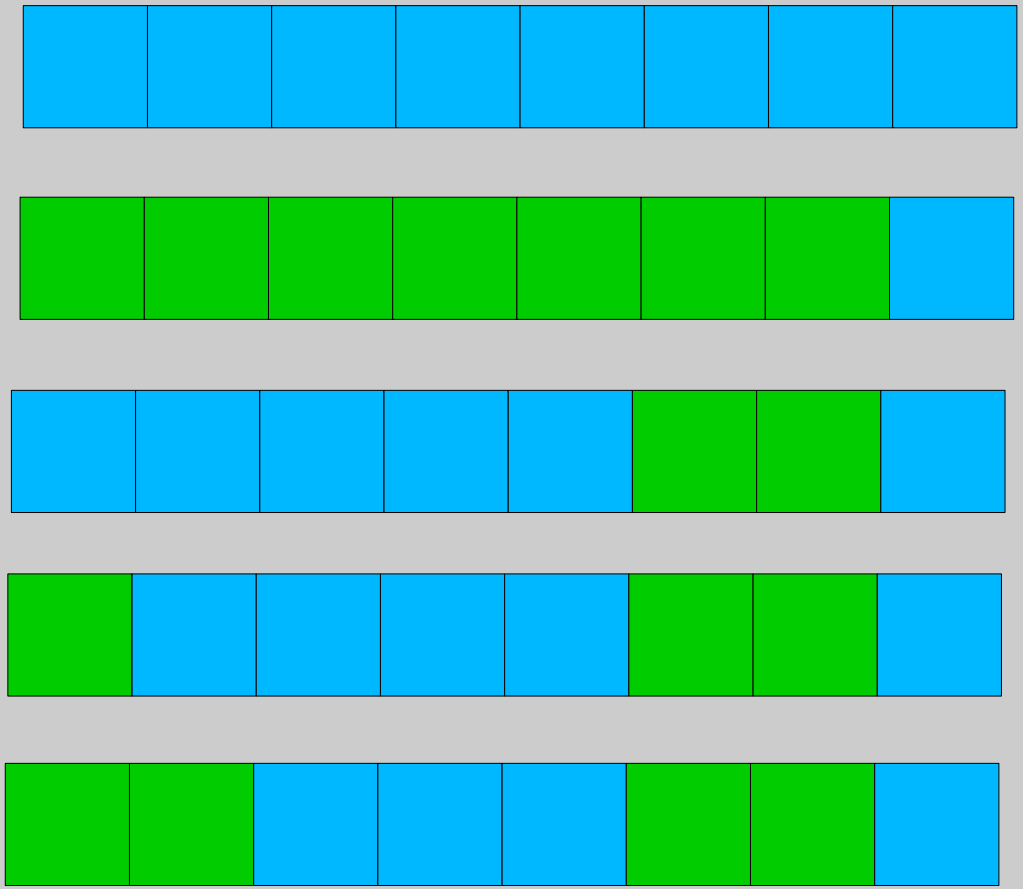
Inne funkcje alokujące pamięć

*void * cmalloc(size_t nmemb, size_t size);*

*void *realloc(void *ptr, size_t size);*

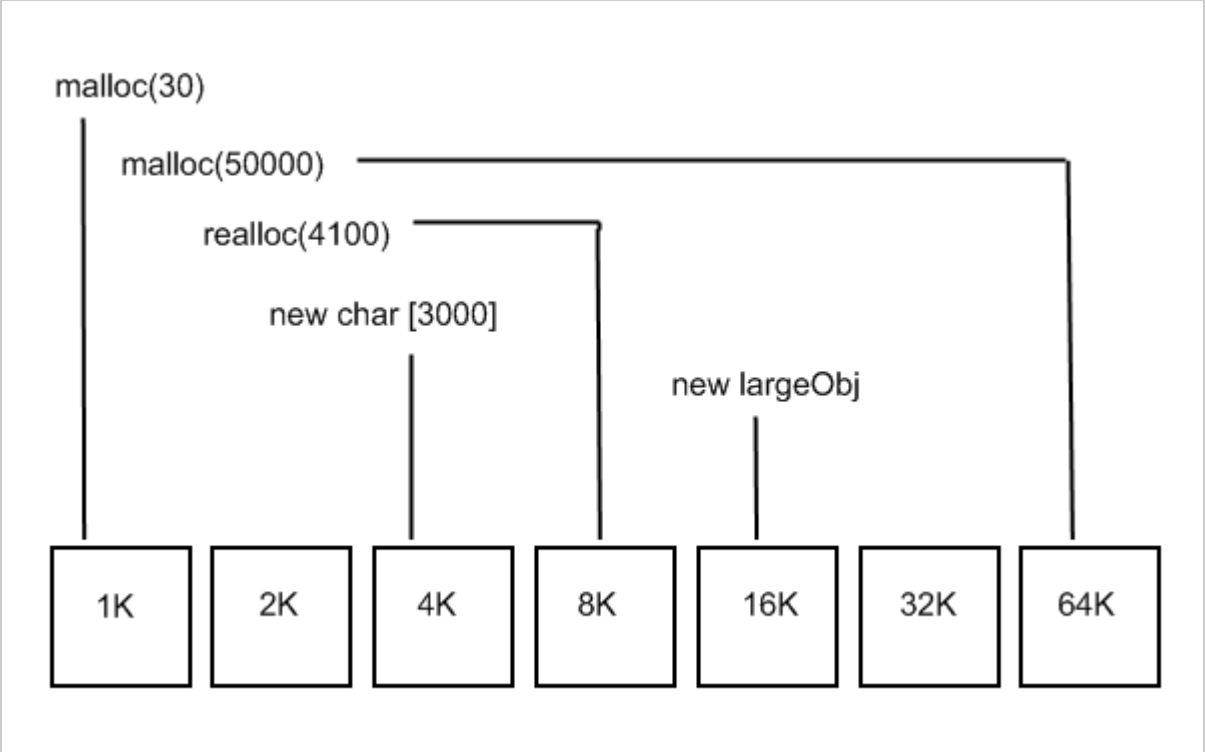
```
int main()
{
        int **tmp;
        int k[]={3,6,4,9};

        tmp=malloc(4*sizeof(int *));
        if(!tmp)
        {
                printf(”Pamiec nie przydzielona”);
                return(-1);
        }
        for(i=0;i<4;i++)
        {
                tmp[i]= malloc(k[i]*sizeof(int));
                if(!tmp[i])
                {
                        printf(”Pamiec nie przydzielona”);
                        return(-1);
                }
        }
        /* ..........................................................*/

        for(i=0;i<4;i++)
                free(tmp[i]);

        free(tmp);
}
```

Fragmentacja pamięci



Strategia alokatorów pamięci

Każdy alokator pamięci ma własną strategię w jaki sposób dokonywać alokacji. Każda strategia jest dla pewnych zadań optymalna a dla innych nie. Najczęściej występującą strategią jest alokacja w blokach, i każdy blok jest 2 razy większy od poprzedniego.



Alokacja, która mieści się w danym bloku, ale nie mieści się w bloku mniejszym jest wykonana w tym bloku. Jeżeli nie ma bloku, który spełnia wymogi to taki blok zostaje utworzony. W tej strategii problemem jest sytuacja, w której alokowany obszar jest znacząco mniejszy od bloku (duża ilość pamięci jest zmarnowana).

Duże znaczenie na fragmentację pamięci ma również czas życia obiektów Obiekty o krótkim czasie życia mają minimalny wpływ na fragmentację!

Jeżeli w programie występuje dużo obiektów o różnych wielkościach defregmentacja będzie większy niż w przypadku dużej ilości obiektów o Tych samych rozmiarach.

Jak rozpoznać problemy z defragemnatcją

- Program zaczyna działać znacząco wolniej
- W niektórych przypadkach nie udaje się zaalokować pamięci

Funkcja main i jej argumenty

```
#include <stdio.h>

main(int argc, char *argv[])
{
        int i;
        for(i=1;i<argc;i++)
                printf("%s%s",argv[i],(i<argc-1) ? " " : "");
        printf("\n");

        return 0;
}

#include <stdio.h>
int main(int argc, char *argv[])
{
        while(--argc>0)
                printf("%s%s",*++argv,(argc>1) ? " " : "");
        printf("\n");
        return 0;
}
```

Wskaźnik na funkcje

Funkcja nie jest zmienną, mimo to istnieje możliwość zdefiniowania wskaźnika na funkcję.

typ_zwracany_przez_funkcje (*nazwa_funkcji)(argumenty funkcji)

Nie wolno wykonywać na wskaźnikach na funkcje operacji arytmetycznych!

funkcja qsort i jej definicja w stdlib.h

void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const void *));

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int cmpstringp(const void *p1, const void *p2)
{
    return strcmp(* (char **) p1, * (char **) p2);
}
main(int argc, char *argv[])
{
    int j;
    qsort(&argv[1], argc - 1, sizeof(char *), cmpstringp);
    exit(1);
}
```

```
void use_int(void *);
void use_float(void *);
void pozdrowienia(void (*)(void *), void *);

int main(void)
 {
  char ans;
  int i_age = 22;
  float f_age = 22.0;
  void *p;
  printf("Użyj int (i) lub float (f)? ");
  scanf("%c", &ans);
  if (ans == 'i') {
    p = &i_age;
    pozdrowienia(use_int, p);
  }
  else
  {
    p = &f_age;
    pozdrowienia(use_float, p);
  }
  return 0;
}
```

```
void pozdrowienia(void (*fp)(void *), void *q)
{
  fp(q);
}
void use_int(void *r)
{
  int a;
  a = * (int *) r;
  printf("Liczba całkowita, masz %d lat.\n", a);
}
void use_float(void *s)
{
  float *b;
  b = (float *) s;
  printf("Liczba zmiennoprzecinkowa, masz %f lat.\n", *b);
}
```

Złożone deklaracja wskaźników

char **argv;/*wskaźnik na wskaźnik na char */

int (*day)[13];/* wskaźnik na tablicę 13 elementową typu int */

int *day[13];*/* 13 elementowa tablica wskaźników na typ int*/

void * comp(); */* funkcja comp zwracająca wskaźnik na void*/
void (*comp)(); /* wskaźnik na funkcje zwracającą void */

char (*(*x())[])();/* x: funkcja zwracająca wskaźnik na tablice wskaźników do funkcji zwracającej char*/

char (*(*x[3])())[5]; /*x: tablica [3] wskaźników na funkcje zwracającą wskaźnik na tablic 5 elementową typu char.

Odczytywanie złożonych wskaźników

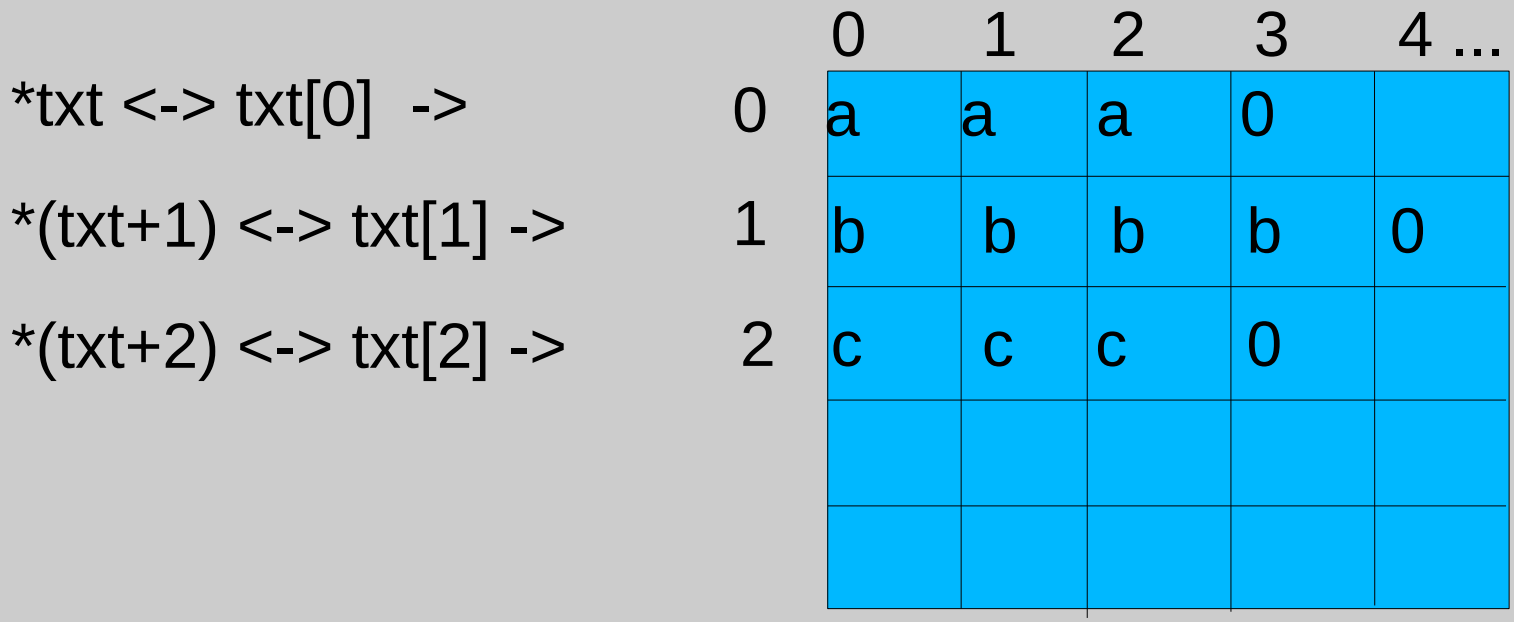
- Rozpocznij odczytywanie od nawiasów najbardziej zgłębionych, idź na prawo a następnie na lewo. Kiedy natrafisz na nawiasy odwróć kolejność marszu.

- int * (* (*fp1) (int)) [10];
  1) Zacznij od nazwy zmiennej fp1
  2) Nie ma nic na prawo, więc idź na lewo * mamy wskaźnik
  3) Wyjdź z nawiasów, na prawo jest (int) to znaczy że mamy wskaźnik do funkcji, która jako argument ma int
  4) Idź na lewo, mamy *, więc funkcja zwraca wskaźnik
  5) Wychodzimy z nawiasów i na prawo mamy [10], a więc jest to 10 elementowa tablica
  6) na lewo jest *, czyli 10 elementowa tablica wskaźników
  7) dalej na lewo jest int

- int *(*(*arr[5])())()
  1) Zacznij od zmiennej arr
  2) Idź na prawo i mamy tablicę 5 elementową
  3) Idź na lewo wskaźnik *
  4) Wychodzimy z nawiasów, idziemy na prawo i mamy () czyli funkcję
  5) Idź na lewo, mamy *, czyli funkcja zwraca wskaźnik
  6) Wychodzimy z nawiasów i idziemy na prawo, znajdujemy () czyli mamy funkcję
  7) Idziemy na lewo i mamy wskaźnik co oznacza że funkcja zwraca wskaźnik
  8) Dalej na lewo jest int, czyli funkcja zwraca wskaźnik typu int

```
char txt[3][80];

*txt <-> txt[0] <-> &txt[0][0]
*(txt+1)  <-> txt[1] <-> &txt[1][0]
(*txt)[3] <-> txt[0][3]
```

Wskaźniki na pierwsze elementy wierszy:



Uwaga nie można txt przypisać wskaźnikowi na wskaźnik typu char!!!!!

Można natomiast tak:

```
char *p=*txt;
```

lub:

```
**txt='x';
**(txt+1)='z';
*(*(txt+2)+1)='?'
```

Rezultat ”xaa”, ”zbbb”,”c?ccc”

Przekazywanie tablic „wielowymiarowych” do funkcji

```
f(int daytab[2][13]);

f(int daytab[][13]);

f(int (*daytab)[13]);

f(int **daytab);
```

```
int  tab[2][ 4 ] {{1,2,3,4},{3,4,5,6}};
int  (*w) [ 4 ] ;
int*  wsk ;

for( w = tab ; w < tab + 2 ; w++ )
        for( wsk = *w ; wsk < *w + 4 ; wsk++ )

           {
               printf( ” %d ”, *wsk );

           }
```

Struktury

Struktura to zestaw jednej lub kilku zmiennych, tego samego lub różnych typów, reprezentowanych przez jedną nazwę.

Definicja:

```
struct nazwa_struktury
{
        definicja pól struktury;
};
```

```
struct punkt
{
        int x;
        int y;
};
```

```
struct punkt3d
{
        int x;
        int y;
        int z;
};
```

```
struct trojkat
{
        struct punkt p1;
        struct punkt p2;
        struct punkt p3;
};
```

Są to deklaracje typów danych, a nie deklaracje zmiennej. Nie wydzielona zostaje pamięć!

Deklaracja zmiennych typu struktura i inicjalizacja

```
struct punkt x;
struct trojkat p;
struct { ...} x,y; /* wprowadzenie struktury bez podania jej nazwy*/

struct punkt x={10,20};
```

Dostęp do pól struktury:

```
        nazwa_struktury.pole;

x.x=10;
p.p1.x=20;
```

- Dozwolone operacja na strukturach:
  - kopiowanie lub przypisywanie (całości)
  - pobranie adresu
  - odwoływanie się do pól struktury
-
- Przekazywanie struktury do funkcji odbywa się przez:
  - przekazywanie pól, każde oddzielnie
  - przekazanie całej struktury
  - przekazanie wskaźnika do struktury

Przykłady

```
struct punkt fPunkt(int x, int y)
{
        struct punkt tmp;

        tmp.x=x;
        tmp.y=y;

        return tmp;
}

struct punkt pTrans(struct punkt x, struct punkt vec)
{
        x.x+=vec.x;
        x.y+=vec.y;

        return x;
}
```

```
struct trojkat pTransTroj(struct trojkat x, struct punkt vec)
{
        x.p1.x+=vec.x;
        x.p2.x+=vec.x;
        x.p3.x+=vec.x;

        x.p1.y+=vec.y;
        x.p2.y+=vec.y;
        x.p3.y+=vec.y;

        return x;
}
```

```
struct trojkat pTransTroj(const struct trojkat *x, const struct punkt
*vec)
{
        struct trokat tmp;
        tmp.p1.x=x->p1.x+vec->x;
        tmp.p2.x=x->p2.x+vec->x;
        tmp.p3.x=x->p3.x+vec->x;

        tmp.p1.x=x->p1.y+vec->y;
        tmp.p1.x=x->p2.y+vec->y;
        tmp.p1.x=x->p3.y+vec->y;

        return tmp;
}
```

Tablice struktur

```
struct punkt3d p[]={{1,2,3},{3,1,1},{4,3,2},{3,3,3}};
struct punkt3d p[4];

int main()
{
        struct punkt3d *p;

        p=(struct punkt3d *) malloc(sizeof(struct punkt3d )*3);

        ..........................

        free(p);
}
```

```
#include <stdio.h>
#define BAZA_R 6
struct baza
{
  char *nazwisko;
  char *adres;
}b[BAZA_R]={{"AAAA","ul sakjkj"},{"BBBB","ul ooooo"},{"CCCC","ul iiiii"},{"DDDD","ul
ssssss"},{"EEEE","ul wwww"},{"FFFF","ul qwqww"}};

struct baza * szukaj(char *naz)
{
  int i;
  for(i=0;i<BAZA_R;i++)
  {
        if(!strcmp(b[i].nazwisko,naz))
            return &b[i];
  }
  return 0;
}
int main()
{
          struct baza *tmp;

          tmp=szukaj(”CCCC”);
          if(tmp)
                    printf(”Adres poszukiwanego: %s”,tmp->adres);
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct baza
{
  char *nazwisko;
  char *adres;
};
int main()
{
        struct baza *b;
        int i;
        char buff[100];

        b=(struct baza *) malloc(sizeof(struct baza) * 5);
        for(i=0;i<5;i++)
        {
                printf("Podaj nazwisko : ");
                scanf("%s",buff);
                b[i].nazwisko=(char *)malloc(sizeof(char)*(strlen(buff)+1));
                strcpy(b[i].nazwisko,buff);

                printf("Podaj adres : ");
                scanf("%s",buff);
                b[i].adres=(char *)malloc(sizeof(char)*(strlen(buff)+1));
                strcpy(b[i].adres,buff);

        }
```

```
        for(i=0;i<5;i++)
        {
                free(b[i].nazwisko);
                free(b[i].adres);
        }
        free(b);

}
```

Pola bitowe struktury

Pola bitowe w strukturach stosowane są do oszczędzania pamięci i pozwalają stworzyć zmienną, która zajmuje mniej niż jeden bajt (np 1 bit).

Deklaracja pola bitowego:

typ [id_pola]:wielkość pola;

typ: może być char, unsigned char, int, unsigned int
wielkość pola: obszar zajmowany przez pole, w bitach.

Pola zajmują tyle bitów ile wyspecyfikowano jako wielkość bitów. Wartości w nich zapisane są traktowane jako liczby całkowite.

```
struct pola_bitowe
{
        unsigned int pole1:1;
        unsigned int pole2:3;
};
```

```
#include <stdio.h>

struct pojazd
{
        unsigned int drzwi: 3;
        unsigned int kola:4;
        unsigned int abs:1;
        unsigned int pasazer:3;
}s;

int main()
{
                s.drzwi=0;
                s.kola=2;
                s.pasazer=1;

                s.pasazer+=20; /*Co sie teraz stanie?*/
}
```

Ułożenie struktur w pamięci jest zależne od implementacji (systemu, architektury), przez co w różny sposób może zachodzić tzw. wyrównywanie na granicy słowa oraz w różny sposób mogą być układane pola bitowe. Dlatego niedopuszczalne jest **pobieranie adresów pól bitowych** bo nie wiadomo jak będą się one rozkładać wewnątrz struktury.

Typedef

Typedef – pozwala utworzyć nową nazwę typu danych (nie tworzy nowego typu danych!!!)

typedef typ nazwa_typu;

np.

typedef int Cal;

Cal staje się synonimem int, może więc być używany w deklaracji tak samo jak int.

typedef struct trojkat troj;

……

*troj pTransTroj(troj *x, struct punkt *vec);*

Typdef ułatwia tworzenie kodu, który łatwo przenosi się na inne maszyny. W takich przypadkach typedef używa się do danych, które są zależne od architektury maszyny, przy przenoszeniu wystarczy tylko zmienić typedef.

Czy można to samo zrobić przy użyciu #define ?

Unia

Unia to specjalny rodzaj struktury która może zawierać (w różnych chwilach) obiekty o różnych typach i rozmiarach. Pozwala to manipulować różnymi rodzajami danych w jednym obszarze pamięci.
np.

```
union uu
{
        int i;
        float f;
        char *s;
}u;
```

Unie to właściwie struktury, w których wszystkie pola nachodzą na siebie. Rozmiar uni to rozmiar maksymalnego elementu uni (rozmiar struktury to suma rozmiarów wszystkich elementów).

http://irc.essex.ac.uk/www.iota-six.co.uk/c/h5_unions.asp

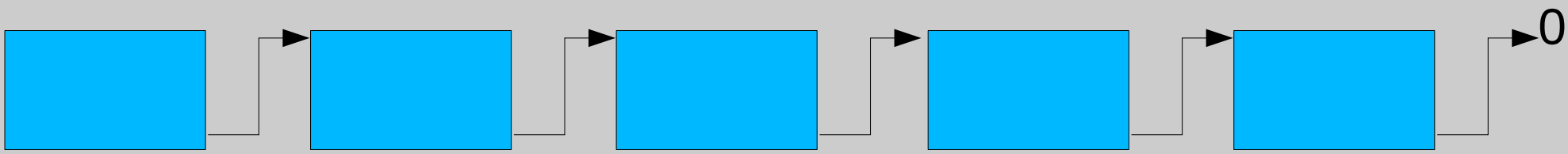
```
#include <stdio.h>
enum which_member
{
  INT,
  FLOAT
};
union int_or_float
{
  int int_member;
  float float_member;
};
int main()
{
        union int_or_float my_union1;
        enum which_member my_union_status1;

        my_union1.int_member=5;
        union_status1=INT;

        my_union1.float_member=5.0;
        union_status1=FLOAT;
```

```
        switch (my_union_status1)
        {
                case INT:
                        my_union1.int_member += 5;
                        break;
                case FLOAT:
                        my_union1.float_member += 23.222333;
                        break;
        }
}
```

Dynamiczne struktury danych

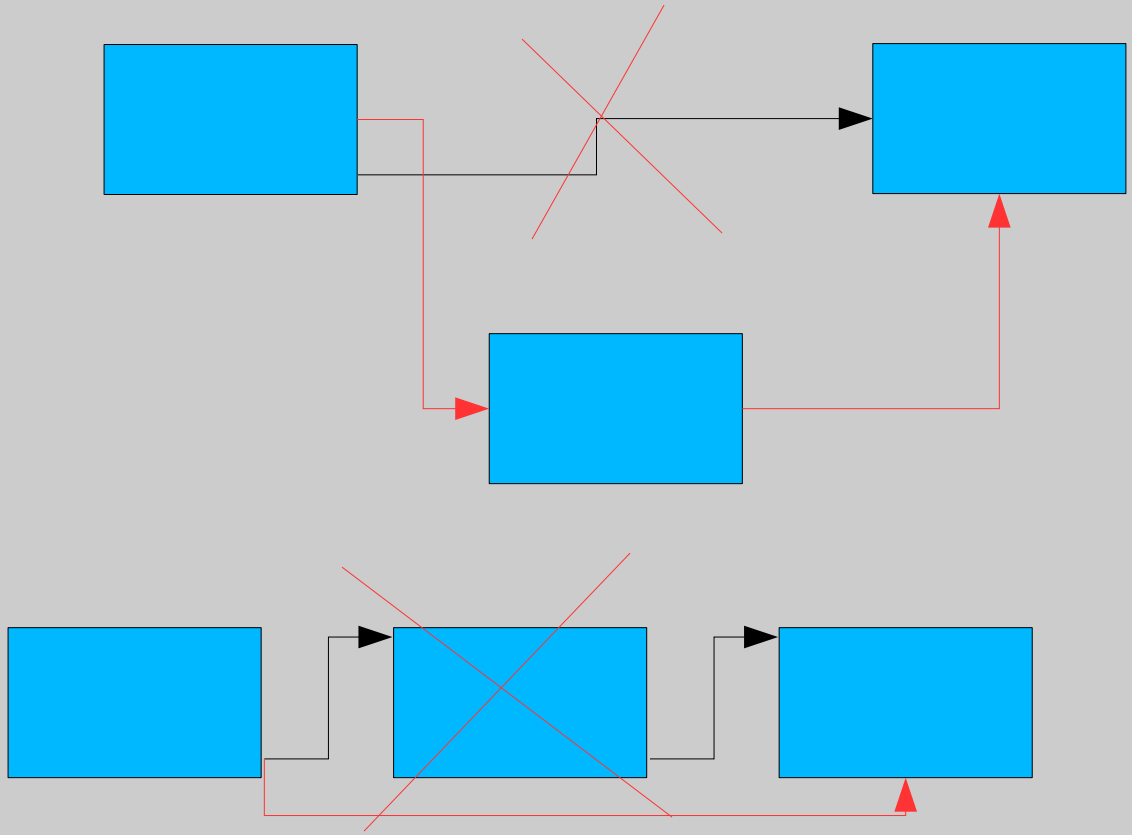
- listy:
  - jednokierunkowa
  - dwukierunkowa
  - cykliczna
  - z wartownikiem
- drzewa:
  - AVL
  - czerwono-czarne
  - BST
- kolejki:
  - LIFO (Last In First Out) - stos
  - FIFO (First In First Out)

Listy

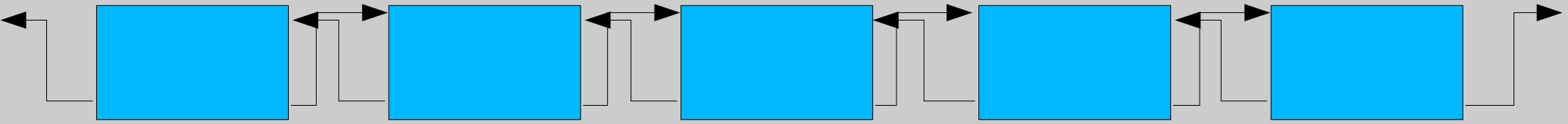


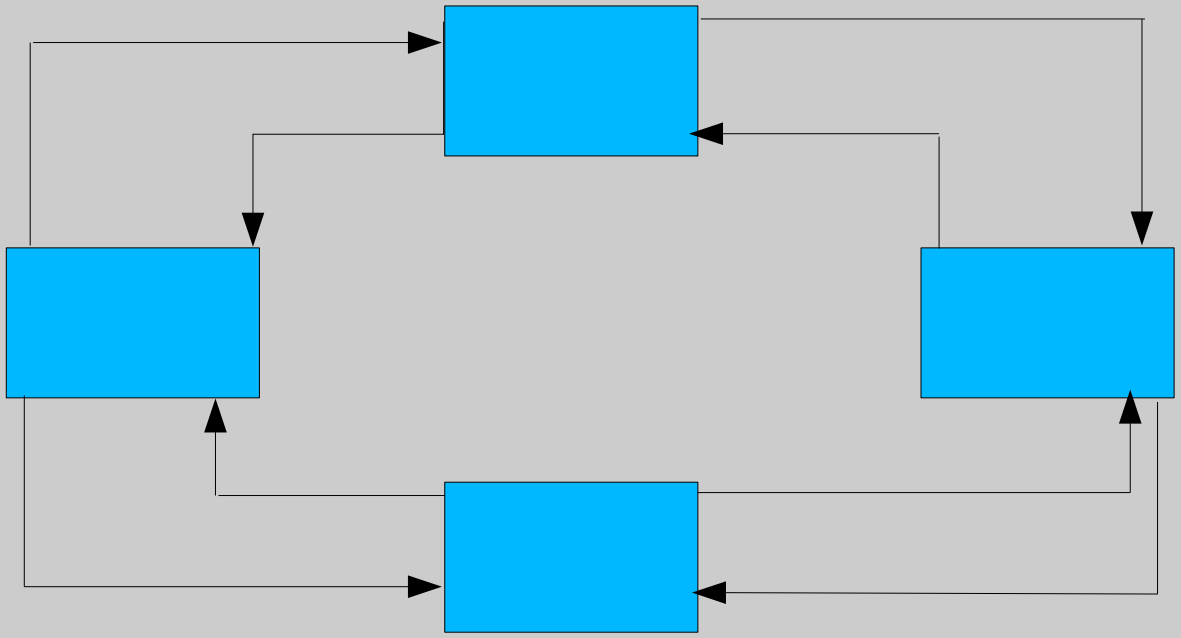

```
struct wezel
{
        char *nazwisko;
        char *adres;
        ..............
        struct wezel *next;
}
```

Listy: dodawanie i usuwanie



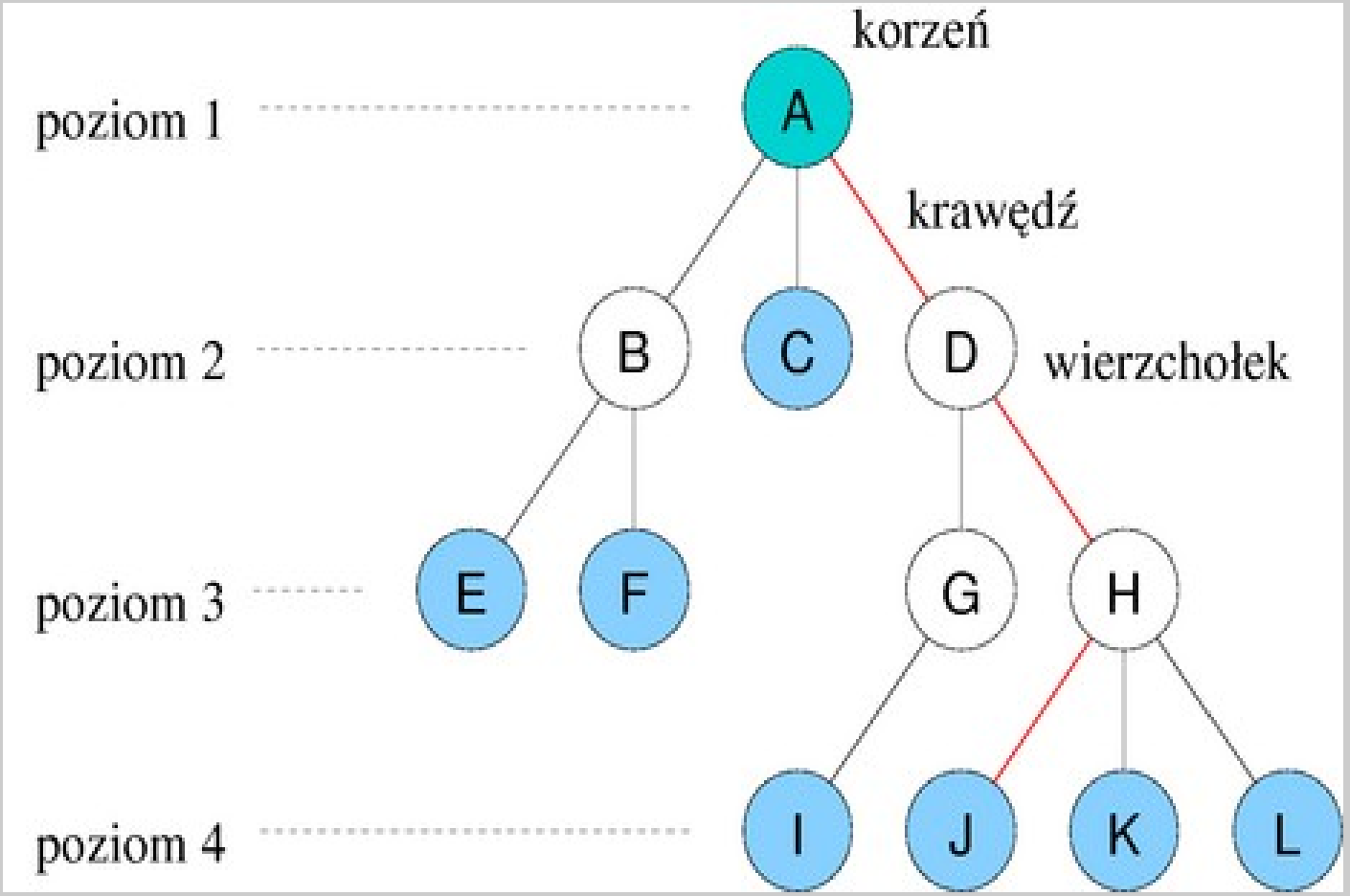
0 0





```
struct wezel
{
        char *nazwisko;
        char *adres;
        .................
        struct wezel *next;
        struct wezel *prev;
}
```

Drzewo




```
struct tree {
        struct tree *left;
        struct tree *right;
        char *dane;
};
```

Przykład zastosowania stosu

(2+3)/7 zapis infiksowy

/7+2 3 notacja polska (zapis przedrostkowy)

Odwrotna Notacja Polska to sposób zapisu wyrażeń arytmetycznych za pomocą notacji postfiksowej.

2 3 + 7 /

((2+3)/7+(5-2)*3)*4

2 3 + 7 / 5 2 – 3 * + 4 *

Algorytm wyliczania wartości przedstawionej w postaci ONP

- Wyzeruj stos.
- Dla wszystkich symboli z wyrażenia ONP wykonuj:
  - jeśli i-ty symbol jest liczbą, to odłóż go na stos,
  - jeśli i-ty symbol jest operatorem to:
    - zdejmij ze stosu jeden element (ozn. a),
    - zdejmij ze stosu kolejny element (ozn. b),
    - odłóż na stos wartość b operator a.
  - jeśli i-ty symbol jest funkcją to:
  - zdejmij ze stosu oczekiwaną ilość parametrów funkcji(ozn. a1...an)
  - odłóż na stos wynik funkcji dla parametrów a1...an
- Zdejmij ze stosu wynik.

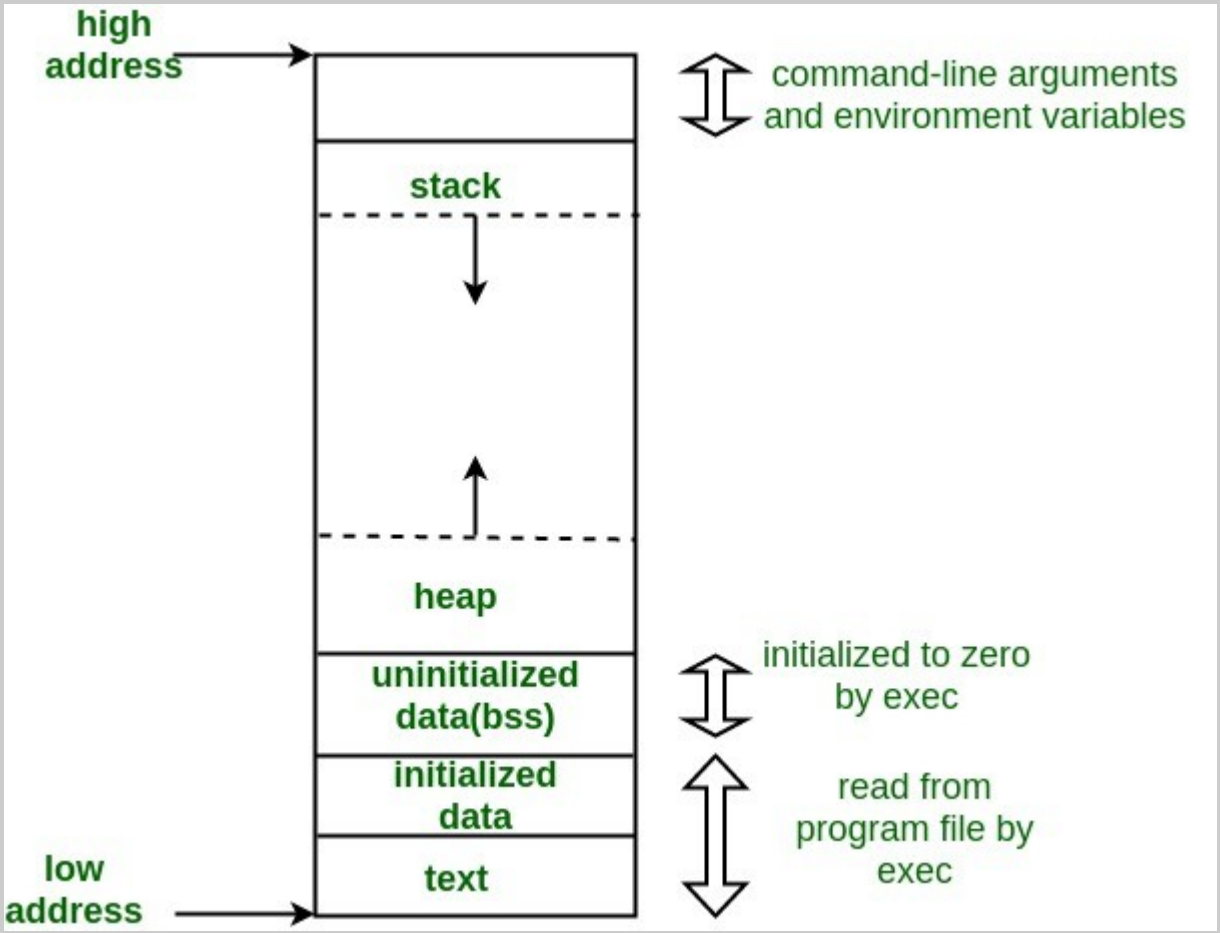
Zarządzanie pamięcią: sterta i stos

Stos to specjalny obszar w pamięci komputera, w którym przechowywane są zmienne tymczasowe tworzone przez funkcję. Za każdym razem gdy funkcja jest wywoływana wszystkie jej zmienne są umieszczane na stosie a jak się funkcja skończy zmienne ze stosu są zdejmowane, a zwolniony obszar jest udostępniany dla innych zmiennych odkładanych na stosie

Trzeba jednak pamiętać, że dla każdego systemu operacyjnego istnieje ograniczenie wielkości stosu (dla różnych systemów, różna jest wielkość stosu)

Sterta region pamięci komputera, który zarządzany jest przez programistę tzn. pamięć jest przydzielana i zwalniana przez programistę. W przeciwieństwie do stosu sterta nie ma ograniczeń na ilość pamięci (jedynym ograniczeniem jest ilość pamięci zainstalowana na urządzeniu). Ponadto zmienne umieszczone na stercie są dostępne dla wszystkich funkcji w programie.

Warstwy pamięci w języku C



- Warstwa tekstu (warstwa kodu)– miejsce w którym umieszczony jest kod programu, umieszczony jest poniżej stosu i sterty co uniemożliwia przypadkowe nadpisanie tego obszaru
- Segment danych – w tym obszarze znajdują się zmienne globalne i statyczne inicjalizowane przez programistę, ten obszar nie jest segmentem tylko do odczytu, gdyż zmienne mogą ulec zmianie w trakcie działania programu. Można jednak segment ten podzielić na 2 części obszar tylko do odczytu i odczytu zapisu. i tak np: globalnie zdefiniowana zmienna s[] = “hello world” - jest do odczytu i zapisu, natomiast char* string = “hello world”, napis umieszczony jest w obszarze tylko do odczytu natomiast wskaźnik do odczytu i zapisu
- Segment danych niezainicjowanych – zawiera wszystkie zmienne globalne i statyczne, które nie zostały zainicjowane w kodzie
- Stos – rośnie w odwrotnym kierunku niż sterta, w momencie gdy się nakryją następuje zgłoszenie braku pamięci, obszar stosu ma strukturę kolejki LIFO, dokładniejszy opis później
- Sterta – obszar zarządzany przy użyciu malloc, realloc, free

```
#include <stdio.h>

int main(void)
{
    return 0;
}
text       data       bss       dec       hex    filename
 960        248        12      1220       4c4    memory-layout

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
text       data       bss       dec       hex    filename
 960        248        16      1224       4c8    memory-layout
```

```
int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
text      data      bss      dec      hex    filename
960        252       12     1224      4c8    memory-layout

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}

text      data      bss      dec      hex    filename
960        256       8     1224      4c8    memory-layout
```

Przepełnienie bufora

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

int main(int argc, char **argv) {
  char haslo_poprawne = 0;
  char haslo[16];

  if (argc!=2) {
    fprintf(stderr, "uzycie: %s haslo", argv[0]);
    return -1;
  }

  strcpy(haslo, argv[1]); /
  if (!strcmp(haslo, "poprawne")) {
    haslo_poprawne = 1;
  }

  if (!haslo_poprawne) {
    fputs("Podales bledne haslo.\n", stderr);
    return -1;
  }
  puts("Witaj, wprowadziles poprawne haslo.");
  return 1;
}
```

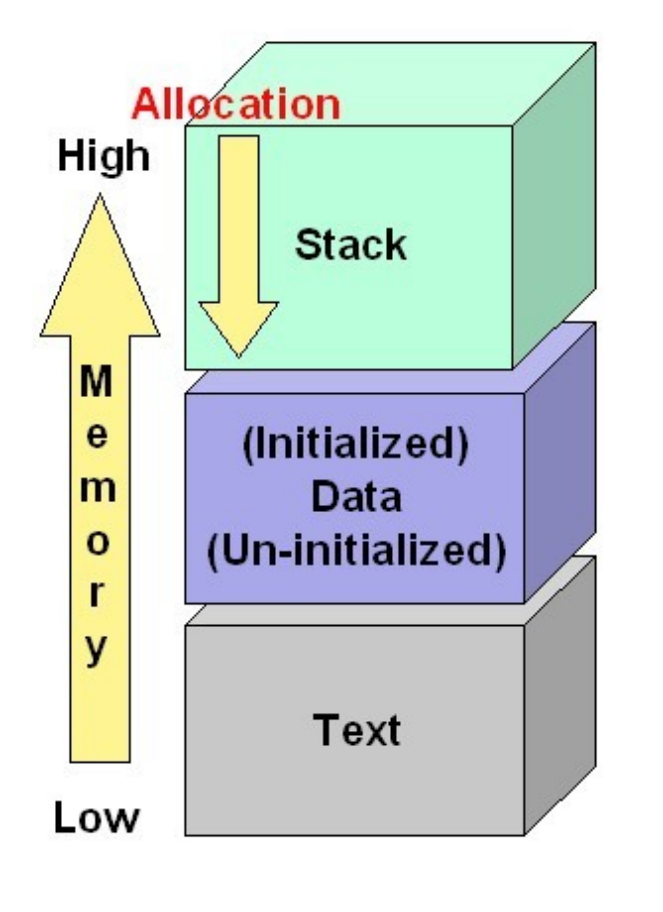
Stos

Pierwszym istotnym zagadnieniem jest mechanizm funkcjonowania tzw. stosu. Stos służy między innymi do przekazywania parametrów do funkcji i do tworzenia zmiennych lokalnych funkcji. Jest czymś w rodzaju podręcznego schowka dla programisty, mechanizmem łatwego dostępu do danych lokalnych wewnątrz poszczególnych funkcji. Inną, ale nie mniej ważną rolą stosu jest przekazywanie informacji towarzyszących wywołaniu poszczególnych funkcji. Dzięki temu stos pełni rolę bufora przechowującego wszystkie dane potrzebne do zainicjowania i wykonania funkcji. Stos jest alokowany przy wejściu do funkcji i zwalniany przed jej opuszczeniem. Ponadto nie ulega zmianom jako struktura - najwyżej zmieniają się dane w nim przechowywane.

Budowa stosu

Stos ma budowę kolejkową, określaną jako Last In, First Out (LIFO). Zgodnie z nazwą, element kolejki zapisany jako ostatni jest odczytywany jako pierwszy, a zapisany jako pierwszy odczytywany jako ostatni. Stosy sterowane są przez wewnętrzne procedury procesorów, przede wszystkim ESP i EBP.

Stosy mają charakter "odwrócony". W praktyce oznacza to, iż szczyt stosu stanowią komórki pamięci o najniższych adresach. Najniższe adresy są nadawane danym najnowszym - z punktu widzenia przepełnienia bufora jest to bardzo ważne. Ponieważ danym nowszym przydziela się niższe adresy, a zarazem są one umieszczane na szczycie stosu, to nadpisanie bufora od adresu najniższego do najwyższego umożliwia nadpisanie danych z dna stosu.


Allocation
High
Stack
Memory
(Initialized)
Data
(Un-initialized)
Text
Low

Każde wywołanie funkcji powoduje utworzenie nowej ramki stosu o następującej budowie (należy pamiętać, że lista uporządkowana jest w kolejności malejących adresów):

- parametry funkcji,
- adres powrotu funkcji,
- wskaźnik ramki,
- ramka procedur obsługi wyjątków,
- lokalnie zadeklarowane zmienne i bufory,
- rejestry zachowane przez funkcję wywołaną.

Jak wygląda stos dla poniższego przykładu?

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
}
void main() {
  function(1,2,3);
}
--------------
```

Dół pamięci góra pamięci

buffer2 buffer1 sfp ret a b c

<------ [ ][ ][ ][ ][ ][ ][ ]

Szczyt stosu dół stosu

```
void function(char *str) {
   char buffer[16];

   strcpy(buffer,str);
}

void main() {
  char large_string[256];
  int i;

  for( i = 0; i < 255; i++)
    large_string[i] = 'A';

  function(large_string);
}
```

```
  Dół pamięci                                                  góra pamięci

                     buffer            sfp   ret   *str
  <------           [                ][    ][    ][    ]
  Szczyt stosu                                                  dół stosu
```

Wynik: segmentation violation

Ponieważ litera A → 0x41 to adres powrotu zostaje ustawiony na 0x41414141 a to jest poza adresami przeznaczonymi dla procesu

```
void function(int a, int b, int c) {
   char buffer1[5];
   char buffer2[10];
   int *ret;
   ret = &a -1;
   (*ret) += 8;
}
void main() {
  int x;
  x = 0;
  function(1,2,3);
  x = 1;
  printf("%d\n",x);
}
```

Niebezpieczne funkcje

```
char *strcat(char *dest, const char *src);
char *strcpy(char *dest, const char *src);
int sprintf(char* buffer, const char* format ...);
char *gets(char *buffer);
```

Inne funkcje: fscanf(), scanf(), vsprintf(), realpath(), getopt(), getpass(), streadd(), strecpy() i strtrns().

Obrona przed atakiem „buffer overflow”

Pierwsze podejście polega na przechwytywaniu "w locie" wszystkich odwołań do funkcji uznawanych za niebezpieczne i zamianę ich na odwołania do funkcji bezpiecznych. System ten nie wymaga dostępu do kodu źródłowego programów, ani też nie ingeruje w budowę systemu operacyjnego, a przechwytywanie jest przezroczyste dla użytkownika. Projektem reprezentującym to podejście jest pakiet Libsafe dostępny dla użytkowników Linuksa.

Druga metoda codziennej obrony przed przepełnieniem bufora polega na stałym sprawdzaniu przez odpowiednie biblioteki, czy podczas wywołań systemowych nie jest wykonywany kod znajdujący się na stosie. Jeśli tak się dzieje, użytkownik jest o tym powiadamiany i może zawczasu podjąć kroki obronne. Przykładem programu realizującego tego typu ochronę jest SecureStack firmy SecureWave.

printf

int printf(char *format, arg1, arg2,...)
Funkcja konwertuje i formatuje i wypisuje przekazane argumenty na standardowe wyjście. Zwraca liczbę wypisanych znaków.

Inna deklaracja funkcji printf:

int printf(char *fmt, ...)

gdzie ... oznacza że liczba argumentów może się zmieniać.
Deklaracja ... może znajdować się tylko na końcu listy argumentów.
W <stdarg.h> są zdefiniowane makra, które umożliwiają pobieranie listy argumentów o nieokreślonej liczbie. *va_list* jest używany żeby zadeklarować zmienną, która odpowiada poszczególnym argumentom. *va_start* initializuje zadeklarowaną zmienną na pierwszy argument bez nazwy, *va_arg* zwraca jeden argument i przesuwa zmienną na następny. *va_end* czyści listę, musi być wywołana zanim skończy się funkcja.

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

int average( int first, ... )
{
        int count = 0, sum = 0, i = first;
        va_list marker;

        va_start( marker, first ); /* inicjalizacja*/
        while( i != -1 )
        {
                sum += i;
                count++;
                i = va_arg( marker, int);
        }
        va_end( marker );

return( sum ? (sum / count) : 0 );
}
int main( void )
{
        printf( "Average is: %d\n", average( 2, 3, 4, -1 ) );
}
```

Zmienna liczba argumentów.

```
#include <stdarg.h>

void minprintf(char *fmt, ...)
{
        va_list ap;
        char *p,*sval;
        int ival;
        double dval;

        va_start(ap,fmt);
        for(p=fmt;*p;p++)
        {
                if(*p!='%')
                {
                        putchar(*p);
                        continue;
                }
```

```
        switch(*++p)
        {
            case 'd':
                ival=va_arg(ap,int);
                printf("%d",ival);
                break;
            case 'f':
                dval=va_arg(ap,double);
                printf("%f",dval);
                break;
            case 's':
                for(sval=va_arg(ap,char *);*sval;sval++)
                    putchar(*sval);
                break;
            default:
                putchar(*p);
                break;
        }
    }
    va_end(ap);
}
```

scanf

int scanf(char *format, ...)

Funkcja odczytuje znaki ze standardowego wejścia interpretując je stosowanie do podanego formatu, rezultat zapamiętuje w odpowiednich argumentach. Wszystkie argumenty (po format) muszą być wskaźnikami.

int sscanf(char *string, char *format, ...)

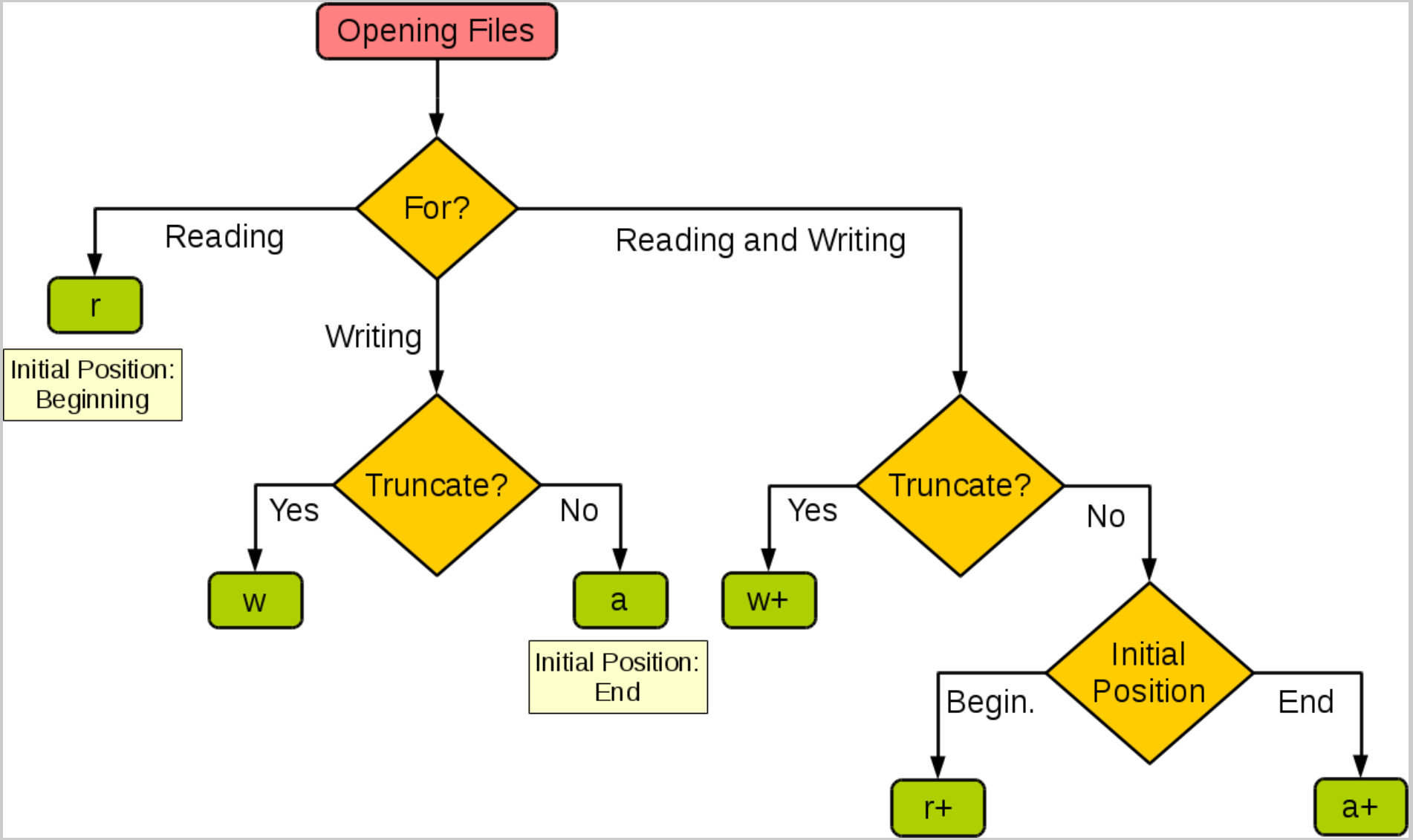
Pliki

Przed odczytem pliku lub zapisem do niego plik musi zostać otwarty.
Otwarcie:

```
FILE *fopen(char *name, char *mode);
```

możliwe tryby otwarcia: ”r”,”r+”,”w”,”w+”,”a”,”a+” (”t”,”b”).
Podstawowe instrukcje do czytania i zapisywanie do pliku:
int getc(FILE *) - zwraca znak z pliku
int putc(int ,FILE *) - przesyła znak do pliku
int fscanf(FILE *, char *, ...);
int fprintf(FILE *, char *, ...);
char *fgets(char *s, int size, FILE *stream);
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
Po wykonaniu operacji na pliku plik musi zostać zamknięty.

int fclose(FILE *)


Opening Files
For?
Reading
Writing
Reading and Writing
r
Initial Position:
Beginning
Truncate?
Yes
No
w
a
Initial Position:
End
Truncate?
Yes
No
w+
Initial
Position
Begin.
End
r+
a+

```
#include <stdio.h>
#include <stdlib.h>
int input(char *s,int length);
int main()
{
    char *buffer;
    size_t bufsize = 32;
    size_t characters;

    buffer = (char *)malloc(bufsize * sizeof(char));
    if( buffer == NULL)
    {
        printf("Unable to allocate buffer");
        exit(1);
    }

    printf("Type something: ");
    characters = getline(&buffer,&bufsize,stdin);
    printf("%zu characters were read.\n",characters);
    printf("You typed: '%s'\n",buffer);
    return(0);
}
```

Strumienie

- Każdy program w momencie uruchomienia "otrzymuje" od razu trzy otwarte strumienie:
-
- stdin (wejście) = odczytywanie danych wpisywanych przez użytkownika
- stdout (wyjście) = wyprowadzania informacji dla użytkownika
- stderr (wyjście błędów) = powiadamiania o błędach

- int fputc (int c, FILE *stream)
- int putc (int c, FILE *stream)
- int putchar (int c)
- int fputs (const char *s, FILE *stream)
- int puts (const char *s)
- int putw (int w, FILE *stream)
-

```
#include <stdio.h>

void filecopy(FILE *if,FILE *of)
{
        int c;

        while((c=getc(if))!=EOF)
                putc(c,of);
}
main(int argc, char *argv[])
{
        FILE *fp;

        if(argc==1)
                filecopy(stdin,stdout);
        else
                while(--argc>0)
                        if((fp=fopen(*++argv,"r"))==NULL)
                        {
                                printf("Nie mozna otworzyc pliku %s\n",*argv);
                                return 1;
                        }
                        else
                        {
                                filecopy(fp,stdout);
                                flocse(fp);
                        }
        return 0;
}
```

Błędy io

Do sygnalizacji błędów powinno używać się strumienia stderr. Znaki wysyłane do strumienia stderr zazwyczaj (?) ukazują się na ekranie nawet jeśli standardowe wyjście jest przekierowane.

Użycie wcześniej opisanych instrukcji z stderr :

*fprintf(stderr,”%s”,*argv);*

*int ferror(FILE *fp)- zwraca wartość niezerową jeżeli wystąpił błąd w strumieniu*

*char * strerror(int)*

*int feof(FILE *fp) – zwraca wartość niezerową jeżeli osiągnięty został koniec pliku*

*void clearerr(FILE *stream) czyści flagę końca pliku i błędu dla danego strumienia*

Pozycjonowanie strumienia

*int fseek(FILE *stream, long offset, int origin);*

możliwe wartości dla origin:
SEEK_SET Początek pliku
SEEK_CUR Obecna pozycja
SEEK_END Koniec pliku

long ftell(FILE *stream);
void rewind(FILE *stream);
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, fpos_t *pos);

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

void main( void )
{
   FILE *f;
   char line[81];
   int  result;
   f = fopen( "plik.out", "w+" );
   if(!f)
   {
      fprintf(stderr, "Plik plik.out nie zostal otwarty\n" );
     exit(-1);
   }
  fprintf( f, "Ustawiam czytanie na to miejsce w pliku: To jest plik 'plik.out'.\n" );
  result = fseek( f, 41, SEEK_SET);
  if( result )
          fprintf( stderr,"Fseek failed\n" );
  else
  {
          printf( "Odczyt pliku od miejsca wskaznika: .\n" );
          fgets( line, 80, f );
          printf( "%s", line );
  }
  fclose(f);

}
```

```
#include <stdio.h>
int main( void )
{
   FILE *f;
   int data1=1, data2=-37;

   f=fopen( "plik", "w+" );

   if(f)
   {
      fprintf( f, "%d %d", data1, data2 );
      printf( "Zapisane wartosci to: %d and %d\n", data1, data2 );
      rewind( f );
      fscanf(f, "%d %d", &data1, &data2 );
      printf( "Odczytane wartosc: %d and %d\n", data1, data2 );
      fclose(f);
   }
}
```

```
#include <stdio.h>
int main( void )
{
   FILE *f;
   char list[30];
   int  i, numread, numwritten;

   if((f= fopen("plik.out", "w+t" )) != 0 )
   {
     for ( i = 0; i < 25; i++ )
        list[i] = (char)('z' - i);
     numwritten = fwrite( list, sizeof( char ), 25, f);
     printf( "Zapisano %d znakow\n", numwritten );
     fclose(f);
   }
   else
     fprintf( stderr,"Nie moge otworzyc pliku\n" );
   if( (f=fopen("plik.out", "r+t" )) != 0 )
   {
     numread = fread( list, sizeof( char ), 25, f);
     printf( "Liczba odczytanych znakow = %d\n", numread );
     printf( "Zawartosc bufora = %.25s\n", list );
     fclose(f);
   }
   else
     fprintf(stderr,"Nie moge otworzyc pliku\n" );
}
```

```
#include <stdio.h>
int main( void )
{
   FILE   *f;
   fpos_t pos;
   char   buffer[20];
   f=fopen("plik", "rb" ) ;
   if(!f)
   {
      fprintf(stderr,"Nie moge otworzyc pliku\n" );
      return -1;
   }
   fread( buffer, sizeof( char ), 8, f);
   if( fgetpos(f, &pos ) != 0 ) {
      fprintf(stderr,"Blad w fgetpos" );
      return -1;
   }
   fread( buffer, sizeof( char ), 13, f);
   printf( "po fgetpos: %.13s\n", buffer );

   if( fsetpos(f, &pos ) != 0 ) {
      fprintf(stderr, "Blad w fsetpos" );
      return -1;
   }
   fread( buffer, sizeof( char ), 13, f);
   printf( "po fsetpos: %.13s\n", buffer );
   fclose(f);
}
```

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

struct fullname {
  char firstName[40];
  char lastName[10];
} info;

int main(void){
  FILE *fp;

  if((fp=fopen("test", "rb")) == NULL) {
    printf("Cannot open file.\n");
    exit(1);
  }

  int client_num = 10;

  /* find the proper structure */
  fseek(fp, client_num*sizeof(struct fullname), SEEK_SET);


  fread(&info, sizeof(struct fullname), 1, fp);

  fclose(fp);
}
```

Procesy w systemie Unix

- Każdy program uruchamiany w systemie Unix-owym nazywany jest procesem. Proces charakteryzowany jest przez numer PID (process identifier), który jest unikalny.
- W języku C (pod systemem UNIX) można utworzyć nowy proces przy użyciu funkcji fork()
- Funkcja fork() duplikuje proces, w którym jest uruchomiona. Nowo utworzony proces jest procesem potomnym (dzieckiem), proces w którym użyto fork() to rodzic.
- Proces potomny dostaje unikalny numer PID
- Wszystkie sygnały dla procesu potomnego są zerowane
- fork(), jeżeli zakończył się poprawnie zwraca 2 wartości w procesie potomnym zwraca 0, w procesie nadrzędnym PID nowo utworzonego procesu

```
#include <stdlib.h>
#include <unistd.h>
 #include <stdio.h>
 int main(int argc, char **argv)
{
            int pid;
            switch (pid = fork())
           {
                        case 0:
                                    printf("ID potomka: pid=%d\n", getpid());
                                   break;
                        default:
                                   printf("ID nadrzędnego procesu: pid=%d, child pid=%d\n", getpid(), pid);
                                    break;
                        case -1:
                                    printf(„Ups”);;
                                   exit(1);
           }
            exit(0);
}
```

```
#include  <stdio.h>
#include  <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

```
int main(void)
{
    pid_t child_pid, wpid;
    int status = 0;
    int i;
    int a[3] = {1, 2, 1};

    printf("rodzic pid = %d\n", getpid());
    for (i = 0; i < 3; i++)
    {
        printf("i = %d\n", i);
        if ((child_pid = fork()) == 0)
        {
            printf("proces potomny (pid = %d)\n", getpid());
            if (a[i] < 2)
            {
                printf("Tak\n");
                exit(1);
            }
            else
            {
                printf("Nie\n");
                exit(0);
            }
            /*NOTREACHED*/
        }
    }

    while ((wpid = wait(&status)) > 0)
    {
        printf("Exit status potomka %d byl nastepujacy %d (%s)\n", (int)wpid, status,
               (status > 0) ? "Tak" : "Nie");
    }
    return 0;
}
```

Demony

- Demon to proces, który uruchamiany jest w tle i nie jest pod kontrolą użytkownika (nie jest połączony z terminalem)
- Tworzony jest zwykle przy użyciu funkcji fork(), a następnie zakończeniu procesu nadrzędnego i zamknięciu dostępu do terminala kontrolnego.
- Procesy, które są odłączone od procesu nadrzędnego nazywane są sierotami.
- Sieroty, które powstały przez przypadek (np. proces nadrzędny nie zakończył się poprawnie, padł) to zombi.

Plik nagłówkowy signal.h

Zdefiniowane są 2 funkcje pozwalające obsłużyć różne sygnały.
Standardowy zestaw sygnałów:
SIGABRT – nietypowe zakończenie
SIGFPE – błąd w operacja arytmetycznych np. dzielenie przez 0
SIGILL – został wykryty ‘invalid object program’, często oznacza próbę wykonania nielegalnej instrukcji
SIGINT sygnał interakcyjny, często oznacza naciśniecie 'break'
SIGSEGV - niewłaściwy dostęp do pamięci, najczęściej spowodowany przy próbie zapisania wartości do obiektu wskazywanego przez niewłaściwy wskaźnik
SIGTERM – żądanie zakończenia programu
SIGALRM - sygnał wysyłany do procesu po upływie określonego czasu
SIGUSR1, SIGUSR2 – sygnały wysyłane do procesu żeby powiadomić o wystąpieniu zdarzenia zdefiniowanego przez użytkownika
Definicje funkcji:

```
void (*signal (int, void (*func)(int))) (int); //zwraca void  (*)(int)
int raise (int sig);
```

Do funkcji func może zostać przekazane SIG_DFL oznaczający standardową obsługę sygnału, lub SIG_IGN oznaczający ignorowanie sygnału.

Jeżeli powiodło się wywołanie funkcji *signal* zwracana jest wartości funkcji func, w przeciwnym razie SIG_ERR jest zwracany i ustawiony jest errno.

Przy użyciu funkcji raise program może wysyłać sygnał sig sam do siebie.

```
#include <signal.h>

void
abort(void)
{
        raise(SIGABRT);
}
```

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


FILE *temp_file;
void leave(int sig);

main() {
        (void) signal(SIGINT,leave);
        temp_file = fopen("tmp","w");
        for(;;) {
                /*
                 * Do things....
                 */
                printf("Ready...\n");
                (void)getchar();
        }
        /* can't get here ... */
        exit(EXIT_SUCCESS);
}

/*
 * on receipt of SIGINT, close tmp file
 * but beware - calling library functions from a
 * signal handler is not guaranteed to work in all
 * implementations.....
 * this is not a strictly conforming program
 */

void
leave(int sig) {
        fprintf(temp_file,"\nInterrupted..\n");
        fclose(temp_file);
        exit(sig);
}
```

Łańcuchy i operacja na łańcuchach.

```
#include <string.h>

char *strcat(char *dest, const char *src);
char *strncat(char *dest, const char *src, size_t n);
int strcmp(const char *s1, const char *s2);
int strncmp(const char *s1, const char *s2, size_t n);
char *strcpy(char *dest, const char *src);
char *strncpy(char *dest, const char *src, size_t n);
size_t strlen(const char *s);
char *strchr(const char *s, int c);
char *strrchr(const char *s, int c);
```

Przeszukiwanie łańcuchów

char *strrchr(const char *string, int c) – Znajduje ostatnie wystąpienie znaku c w zmiennej string

char *strstr(const char *s1, const char *s2) –Znajduje pierwsze wystąpienie łańcucha s2 w łańcuchu s1.

char *strpbrk(const char *s1, const char *s2) – Zwraca wskaźnik na pierwsze wystąpienie w łańcuchu s1 jakiegokolwiek znaku z łańcucha s2, jeśli nie ma żdanego znaku z s2 w s1 zwraca 0.

size_t strspn(const char *s1, const char *s2) – Zwraca liczbę znaków znajdujących się na początku łańcucha s1, które występują w s2.

size_t strcspn(const char *s1, const char *s2) -- Zwraca liczbę znaków znajdujących się na początku łańcucha s1, które nie występują w s2.

char *strtok(char *s1, const char *s2) – łamie łańcuch wskazywany przez wskaźnik s1 w sekwencje znaków które oddzielone są przez jeden lub więcej znaków z łańcucha wskazywanego przez wskaźnik s2

```
char *str1 = "Witaj";
char *ans;

ans = strchr(str1,'t');/* ans wskazuje str1+2*/
/*************************************************************/
char *str1 = "Witaj";
char *ans;

ans = strpbrk(str1,'pija');/*ans wskazuje na str+1*/
/*************************************************************/
char *str1 = "Witaj";
char *ans;

ans = strstr(str1,'aj');/*ans wskazuje na str1+3*/
/*************************************************************/
char *str1 = "Witaj co słychać";
char *t1;
for ( t1 = strtok(str1," ");t1 != NULL;t1 = strtok(NULL, " ") )
        printf("%s\n",t1);
```

Testowanie i konwersje

int isalnum(int c) – Prawda jeśli c jest znakiem alfanumerycznym, lub znakiem (‘'0''do '9’). UWAGA makroinstrukcja nie uwzględnia polskich znaków
int isalpha(int c) -- Prawda jeżeli c literą.
int isascii(int c) -- Prawda jeżeli c jest ASCII (0-127).
int iscntrl(int c) -- Prawda jeżeli c jest znakiem kontrolnym (0-31)
int isdigit(int c) -- Prawda jeżeli c znakiem '0' – '9'
int isgraph(int c) -- Prawda jeżeli c jest znakiem kodu drukarskiego.
int islower(int c) -- Prawda jeżeli c jest małą literą.
int isprint(int c) -- Prawda jeżeli c jest znakiem dającym się wydrukować
int ispunct (int c) -- Prawda jeżeli c punctuation character.
int isspace(int c) -- Prawda jeżeli c jest znakiem kodu separującego.
int isupper(int c) -- Prawda jeżeli c dużą literą
int isxdigit(int c) -- Prawda jeżeli c jest liczbą szesnastkową
int toascii(int c) – Zamień na znak ascii
int tolower(int c) – Zamień na małą literę
int toupper(int c) –Zamień na dużą literę

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

int main()
{
        int i;
        char *lan="To jest Lancuch";

        for(i=0;i<strlen(lan);i++)
                printf("%c",tolower(lan[i]));
        printf("\n");
}
```

```
char *str = "To jest napis";
while(*str)
        if isalpha(*str++)
```

Konwersja

łańcuch -> liczba

int atoi(const char *nptr);
long atol(const char *nptr);
double atof(const char *nptr);

liczba->łańcuch

char * itoa (int value, char * str, int base);/*funkcja nie jest ANSI-C*/

Konwersja liczby całkowitej określonej parametrem value na łańcuch w systemie określonym parametrem base zakres od 2 do 36.

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

int main ()
{
  int i;
  char buffer [33];
  printf ("Wprowadz liczbe: ");
  scanf ("%d",&i);
  itoa (i,buffer,10);
  printf ("dziesietna: %s\n",buffer);
  itoa (i,buffer,16);
  printf ("szesnastkowa: %s\n",buffer);
  itoa (i,buffer,2);
  printf ("dwojkowa: %s\n",buffer);
  return 0;
}
```

*void *memchr(const void *s, int c, size_t n);* - Funkcja przeszukuje n bajtów pamięci (s) w celu znalezienia znaku c.
*int memcmp(const void *s1, const void *s2, size_t n);* - Funkcja porównuje pierwszych n bajtów obszarów wskazywanych przez s1 i s2
*void *memcpy(void *dest, const void *src, size_t n);* - Funkcja kopiuje n bajtów z obszaru src do dest, obszary wskazywane przez dest i src nie powinny sie pokrywać
*void *memmove(void *dest, const void *src, size_t n);* Funkcja kopiuje n bajtów z obszaru src do dest, obszary wskazywane przez dest i src mogą się nakładać
void *memset(void *s,int c, size_t n) funkcja pozwala wypełnić n pierwszych bajtów obszaru pamięci wskazywanego przez source wartością c

```
char names[] = "Alan Bob Chris X Dave";
   if( memchr(names,'X',strlen(names)) == NULL
)
     printf( "X nie został znaleziony\n" );
   else
     printf( "X znaleziono\n" );
```

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

int main(void)
{
   char *buf1 = "aaa";
   char *buf2 = "bbb";
   char *buf3 = "ccc";

   int stat;

   stat = memcmp(buf2, buf1, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 1\n");
   else
      printf("buffer 2 is less than buffer 1\n");

   stat = memcmp(buf2, buf3, strlen(buf2));
   if (stat > 0)
      printf("buffer 2 is greater than buffer 3\n");
   else
      printf("buffer 2 is less than buffer 3\n");

   return 0;
}
```

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

int main(void)
{
   char src[] = "**********************************";
   char dest[] = "abcdefghijlkmnopqrstu-
vwxyz0123456709";
   char *ptr;

   printf("dst przed kopiowaniem: %s\n", dest);
   ptr = (char *) memcpy(dest, src, strlen(src));
   if (ptr)
      printf("dest po kopiowaniu %s\n", dest);
   else
      printf("memcpy failed\n");
   return 0;
}
```

Pliki nagłówkowe i standardowe typy

•Różne makra i typy danych używane są przez standardowe funkcje, są one zdefiniowane w różnych plikach nagłówkowych

- wszystkie nazwy definicji i makr występujących w plikach nagłówkowych są zarezerwowane. Nazwy te nie powinny być używane ani przedefiniowane
- wszystkie nazwy zaczynające się od podkreślenia są zarezerwowane

pliki nagłówkowe mogą być ładowane w dowolnej kolejności, i dowolną ilość razy, ale muszą być użyte zanim nastąpi wywołanie funkcji lub makra, które są wewnątrz danego pliku nagłówkowego zdefiniowane

Występuje 18 standardowych plików nagłówkowych:

<assert.h>	<locale.h>	<stdio.h>
<ctype.h>	<math.h>	<stdlib.h>
<errno.h>	<setjmp.h>	<string.h>
<float.h>	<signal.h>	<time.h>
<iso646.h>	<stdarg.h>	<wchar.h>
<limits.h>	<stddef.h>	<wctype.h>

Problemy z makrem

Makra mogą być uciążliwe podczas debuggowania programu. W standardowych plikach nagłówkowych mogą występować makra maskujące , które mają taką samą nazwę jak funkcja, można takie makra wyłączyć:

```
#include <ctype.h>
char *skip_space(char *p)
    {
    while (isspace(*p))       /*To moze byc makro*/
        ++p;
    return (p);
    }

#include <ctype.h>
#undef isspace                /*Usuwamy makro*/
int f(char *p) {
    while (isspace(*p))       /*Tu musi byc funkcja*/
        ++p;
```

Plik nagłówkowy stddef.h

- Plik te zawiera kilka definicji typów i makr, które są często używane w pozostałych plikach nagłówkowych.
- Typy i stałe zdefiniowane w stddef.h:
  - NULL
  - size_t
  - wchar_t
-

Plik nagłówkowy errno.h

- Zdefiniowane są następujące zmienne:
  - errno – wprowadzone do wykrywania błędów przy wywołaniu funkcji standardowej. W momencie uruchomienia programu errno przyjmuje wartość zero i od tego momentu nigdy nie jest automatycznie resetowana.Jeżeli wystąpi błąd funkcja ustawia errno na wartość niezerową (zazwyczaj -1). Oczywiście funkcja musi mieć udokumentowane użycie errno.

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

errno = 0;
if(some_library_function(arguments) < 0){
        /* error processing code... */
        /* may use value of errno directly */
```

  - makro EDOM
  - makro ERANGE

```c
#include <stdio.h>
#include <errno.h>
extern int errno ;
int main ()
{
  FILE * pFile;
  pFile = fopen ("unexist.ent","rb");
  if (pFile == NULL)
  {
    perror ("The following error occurred");
    printf( "Value of errno: %d\n", errno );
  }
  else
    fclose (pFile);
  return 0;
}
```

```c
#include <stdio.h>
#include <errno.h>
extern int errno ;
int main ()
{
  FILE * pFile;
  pFile = fopen ("unexist.ent","rb");
  if (pFile == NULL)
  {
    printf( "Error Value is : %s\n", strerror(errno) );
  }
  else
    fclose (pFile);
  return 0;
}
```

Plik nagłówkowy assert.h

Plik ten zawiera definicje makra assert bardzo przydatnego przy debug-gowaniu programu.

void assert(int wyrażenie)

Jeżeli *wyrażenie* daje wartość 0 (false) funkcja assert wypisze informa-cje o tym że wyrażenie jest nieprawidłowe oraz nazwę pliku źródłowego, w którym to nastąpiło numer linii i *wyrażenie.* Po tym następuje wywoła-nie funkcji *abort* i program zostaje zatrzymany.

```
assert(1 == 2);

/* Wynik */

Assertion failed: 1 == 2, file silly.c, line
15
```

Assert określa pewne warunki w określonych punktach programu które muszą być spełnione. Występują 3 typy warunków:

- Określają warunki na początku funkcji
- Określają warunki na końcu funkcji
- Warunki występujące w dowolnym miejscu programu

Nie spełnienie warunku znajdującego się w assert oznacza błąd Programu!

```
assert( size <= LIMIT );
```

```
int
magic( int size, char *format )
{
    int maximum;
    assert( size <= LIMIT );
    assert( format != NULL );
    ...
```

Jeżeli po licznych testach uznamy, że debuggowanie nie jest już potrzebne można wyłączyć *assert* definiując NDEBUG przed #include <assert.h>.
To powoduje jednak, że żadne wyrażenie użyte w assert nie będzie nie będzie wykonane. Poniższy przykład przedstawia „nieoczekiwane” działanie programu po wyłączeniu *assert*.

```
#include <assert.h>

void
func(void)
{
	int c;
	assert((c = getchar()) != EOF);
	putchar(c);
}
```

Kiedy należy używać funkcji assert?

W sytuacji gdy jestem pewien że dany warunek musi wystąpić (sytuacja w której nie występuje jest absolutnie wyjątkowa!)

```
int main(){
        SDL_Surface *screen;
        /** Initialize SDL */
        if(SDL_Init(SDL_INIT_VIDEO)!=0){
                fprintf(stderr,"Unable to initialize SDL: %s",SDL_GetError());
        }
        atexit(SDL_Quit);
        /** Sets video mode */
        screen=SDL_SetVideoMode(640,480,16,SDL_HWSURFACE);
        if(screen==NULL){
                fprintf(stderr,"Unable to set video mode: %s",SDL_GetError());
        }
        return (0);
}
```

```
int main(){
        SDL_Surface* screen;
        /** Initialize SDL */
        assert(SDL_Init(SDL_INIT_VIDEO)==0);
        atexit(SDL_Quit);
        /** Sets video mode */
        screen=SDL_SetVideoMode(640,480,16,SDL_HWSURFACE);
        assert(screen!=NULL);
        return (0);
}
```

Plik nagłówkowy setjmp.h

Występuj tu deklaracja jmp_buf oraz funkcji *setjmp* i *longjmp.* Jmp_buf jest używany do przekazywania informacji pomiędzy funkcją i makrem potrzebnej do wykonania skoku.
Deklaracja:

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);

setjmp to makro, które ustawia *jmp_buf* i zraca 0 przy pierwszym uruchomieniu. Przy kolejnych uruchomieniach zwraca wartość niezerową, która została użyta przy wywołaniu funkcji *longjmp.*
Argument val w funkcji *longjmp* powinien mieć wartość niezerową, jeżeli ustawiona zostanie wartość zero, wartość ta zostanie przestawiona na 1.
Jeżeli przed wystąpieniem *longjmp* nie było setjmp to program „padnie”.

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

void func(void);
jmp_buf place;

main(){
        int retval;

        /*
         * First call returns 0,
         * a later longjmp will return non-zero.
         */
        if(setjmp(place) != 0){
                printf("Returned using longjmp\n");
                exit(EXIT_SUCCESS);
        }

        /*
         * This call will never return - it
         * 'jumps' back above.
         */
        func();
        printf("What! func returned!\n");
}

void
func(void){
      /*
       * Return to main.
       * Looks like a second return from setjmp,
       * returning 4!
       */
      longjmp(place, 4);
      printf("What! longjmp returned!\n");
}
```

TRY CATCH W C

```
#include <stdio.h>
#include <setjmp.h>
#define TRY do{ jmp_buf ex_buf__; if( !setjmp(ex_buf__) ){
#define CATCH } else {
#define ETRY } }while(0)
#define THROW longjmp(ex_buf__, 1)
int
main(int argc, char** argv)
{
   TRY
   {
      printf("In Try Statement\n");
      THROW;
      printf("I do not appear\n");
   }
   CATCH
   {
      printf("Got Exception!\n");
   }
   ETRY;
   return 0;
}
```

```
#include <stdio.h>
#include <setjmp.h>
#define TRY do{ jmp_buf ex_buf__;
switch( setjmp(ex_buf__) ){ case 0:
#define CATCH(x) break; case x:
#define ETRY } }while(0)
#define THROW(x) longjmp(ex_buf__, x)
#define FOO_EXCEPTION (1)
#define BAR_EXCEPTION (2)
#define BAZ_EXCEPTION (3)
int
main(int argc, char** argv)
{
   TRY
   {
      printf("In Try Statement\n");
      THROW( BAR_EXCEPTION );
      printf("I do not appear\n");
   }
   CATCH( FOO_EXCEPTION )
   {
      printf("Got Foo!\n");
   }
   CATCH( BAR_EXCEPTION )
   {
      printf("Got Bar!\n");
   }
   CATCH( BAZ_EXCEPTION )
   {
      printf("Got Baz!\n");
   }
   ETRY;
   return 0;
}
```

Czas i liczby losowe

time.h

Występuje wiele funkcji (UNIX) pozwalających wyznacza czas, podstawowa to:

*time_t time(time_t *t);*

Zwraca czas od 1 stycznia 1970 w milisekundach.
Przydatna do mierzenia np. czasu trwania jakieś funkcji czy tez programu

```
#include <sys/types.h>
#include <time.h>
 main()
  {
        int i;
        time_t t1,t2;

        (void) time(&t1);
        for (i=1;i<=300;++i)
              printf(``%d %d %dn",i, i*i, i*i*i);
        (void) time(&t2);
        printf(Czas wykonania petli= %d ", (int) t2-t1);
}
```

Przydatna również do ustawiania posiewu generatora liczb pseudolosowych.

```
main()
{
        int i;
        time_t t1;

        time(&t1);
        srand(t1);

        for (i=0;i<5;++i)
                printf("%d ", rand());
}
```

void srand(unsigned int seed); - funkcja inicjująca posiew

int rand(void); - zwraca liczbę pseudolosową z zakresu 0 do RAND_MAX

Poszukiwanie błędów (debugowanie - odpluskwianie)

- Najczęściej występujące błędy:
  - źle zaalokowana/dealokowana pamięć
  - błędne założenia
  - niepoprawne warunki
-

Strategie debugowania

- Zastosuj odpowiednie narzędzia:
  - poszukujesz przyczyny wystąpienia „segmentation fault” użyj debugera
  - sprawdzasz pamięć, czy nie ma wycieków użyj valgrind (linux), purify (windows)

Zanim rozpoczniesz debugowanie przeanalizuj rozwiązanie, może jest ono zbyt złożone, spróbuj znaleźć prostsze rozwiązanie (nawet za cenę napisanie wszystkiego od początku)

Minimalizuj potencjalne problemy spowodowane „syndromem” copy-paste. Kopiowanie fragmentów kodu, szczególnie gdy nie zostały dobrze zdebugowane (a w przyszłości będą na pewno) prowadzi do frustracji spowodowanej debugowanie wiele razy tego samego. Dlatego należy używać funkcji !!

Testuj każdą funkcję, dzięki temu unikasz debugowania dużych problemów skupiając się na małych, a więc i prostszych. To oczywiście wymaga dyscypliny i wyczucia co mogło by pójść źle (doświadczenie).

Analizuj ostrzeżenia kompilatora.

Funkcja printf „kłamie”, operacje I/O są buforowane, więc to że jakiś printf nie wypisał na ekranie komunikatu (program wcześniej padł) nie znaczy, że program do tego miejsca nie dotarł. Jeżeli „lubisz” printf używaj fflush.

Standard C99

Nowe typy danych:

- bool

```
#include <stdbool.h>
int main()
{
    bool b = false;
    b = true;
}
```

- long long (long long int, signed long long signed long long int)
  zakres: [−9 223 372 036 854 775 807, +9 223 372 036 854 775 807]
- zespolone
  float _Complex, double _Complex, long double _Complex

Typy całkowite o ustalonych rozmiarach

int8_t int16_t int32_t int64_t	Typ całkowity ze znakiem, który zajmuje dokładnie: 8, 16, 32 i 64 bity
int_fast8_t int_fast16_t int_fast32_t int_fast64_t	Najszybszy typ całkowity ze znakiem, który zajmuje dokładnie: 8, 16, 32 i 64 bity
int_least8_t int_least16_t int_least32_t int_least64_t	Najmniejszy typ całkowity ze znakiem, który zajmuje dokładnie: 8, 16, 32 i 64 bity
intmax_t	Maksymalny rozmiar (w sensie bitów) dla typu całkowitego

Flexible array member

```
struct package_head {
  char name[20];
  size_t len;
  uint64_t data[];
};

Alokacja pamięci:

package_head *a = malloc(sizeof(package_head) + 10 * sizeof(uint64_t));
package_head *b = malloc(sizeof(*b) + 12 * sizeof(b->data[0]));
```

Kwalifikatory typu

- const
- volatile - znaczy ulotny. Oznacza to, że kompilator wyłączy dla takiej zmiennej optymalizacje typu zastąpienia przez stałą lub zawartość rejestru, za to wygeneruje kod, który będzie odwoływał się zawsze do komórek pamięci danego obiektu.
- restrict - może być zastosowane dla wskaźnika przekazywanego do funkcji. Oznacza ono że dostęp do wartości na który wskazuje wskaźnik, będzie odbywał się wyłącznie z użyciem tegoż wskaźnika.

Doprecyzowanie inicjalizacji

```
typedef struct
{
    int num;
    double dbl;
    char const* str;
} ttt;

int main(void)
{
    ttt bar = { .str = "str!", .dbl = 42 };
}
```

```
#include <stdio.h>

typedef struct {

int  a;

double b;

} s_t;

int main(void) {

s_t values[5] = {

 [0] = { .a =1, .b =13.2 },

 [1] = { 10, 10.0},

 [3] = { .a =4, .b =45.1 } };

printf("%d %f\n", values[1].a,
values[1].b);

}
```

Tablice o zmienne wielkości

```
void fun(int n)
 {
   int arr[n];
 }
int main()
{
 fun(6);
}
```

Funkcje inline

```
inline void funkcja()
{
//...
}
```

Podanie tego słowa powoduje, że wstawienie kodu funkcji w miejscu wywołania jest preferowane nad typowym mechanizmem wywoływania funkcji. Kompilator jednakże nie jest zobligowany do rozwinięcia tej funkcji w miejscu wywołania i może wygenerować kod wykonujący typowe wywołanie.

Funkcja snprintf

```
int snprintf(char *str, size_t size, const char *format, ...);
```

Końcowy przecinek

```
enum Channel {
    RED,
    GREEN,
    BLUE,
};
```

Tworzenie biblioteki

- Występują dwa rodzaje bibliotek: statyczne i dynamiczne
  - Biblioteki statyczne – dołączane są do programów w trakcie kompilacji
    Biblioteki statyczne tworzy się przy użyciu programu ar
    np. ar rcs libMylibrary.a file1.o file2.o
  - Biblioteki dynamiczne (so, dll) – mogą być ładowane w trakcie działania programu

Tworzenie biblioteki dynamicznej

- gcc -o file1.o -c -fpic file1.c
- gcc -shared -o mylibrary.so file1.o file2.o
- Program może ładować w trakcie pracy potrzebne mu biblioteki. Linker dostarcza w tym celu 4 funkcji:
  - dlopen ładowanie biblioteki,
  - dlsym zwrócenie wskaźnika do odpowiedniego symbolu (funkcji) w bibliotece,
  - dlerror obsługa błędów,
  - dlclose zamykanie biblioteki.

```
#include <dlfcn.h>      /* defines dlopen(), etc. */
void* lib_handle;       /* handle of the opened library */
lib_handle = dlopen("/full/path/to/library", RTLD_LAZY);
if (!lib_handle) {
    fprintf(stderr, "Error during dlopen(): %s\n", dlerror());
    exit(1);
}
```

```
/* first define a function pointer variable to hold the function's address */
struct local_file* (*readfile)(const char* file_path);
/* then define a pointer to a possible error string */
const char* error_msg;
/* finally, define a pointer to the returned file */
struct local_file* a_file;

/* now locate the 'readfile' function in the library */
readfile = dlsym(lib_handle, "readfile");

/* check that no error occured */
error_msg = dlerror();
if (error_msg) {
    fprintf(stderr, "Error locating 'readfile' - %s\n", error_msg);
    exit(1);
}
```

- /* finally, call the function, with a given file path */
- a_file = (*readfile)("hello.txt");

Pisanie dużych programów

W przypadku dużych programów powinno się podzielić program na moduły! Moduły składają się z funkcji, które są w jakiś sposób związane np. zestaw funkcji potrzebnych do rozwiązania jednego problemu w jednym pliku albo opisujących działania na jakimś obiekcie itp.
Każdy z modułów może być kompilowany niezależnie (przyspiesza to całkowitą kompilacje podczas pisanie programu).

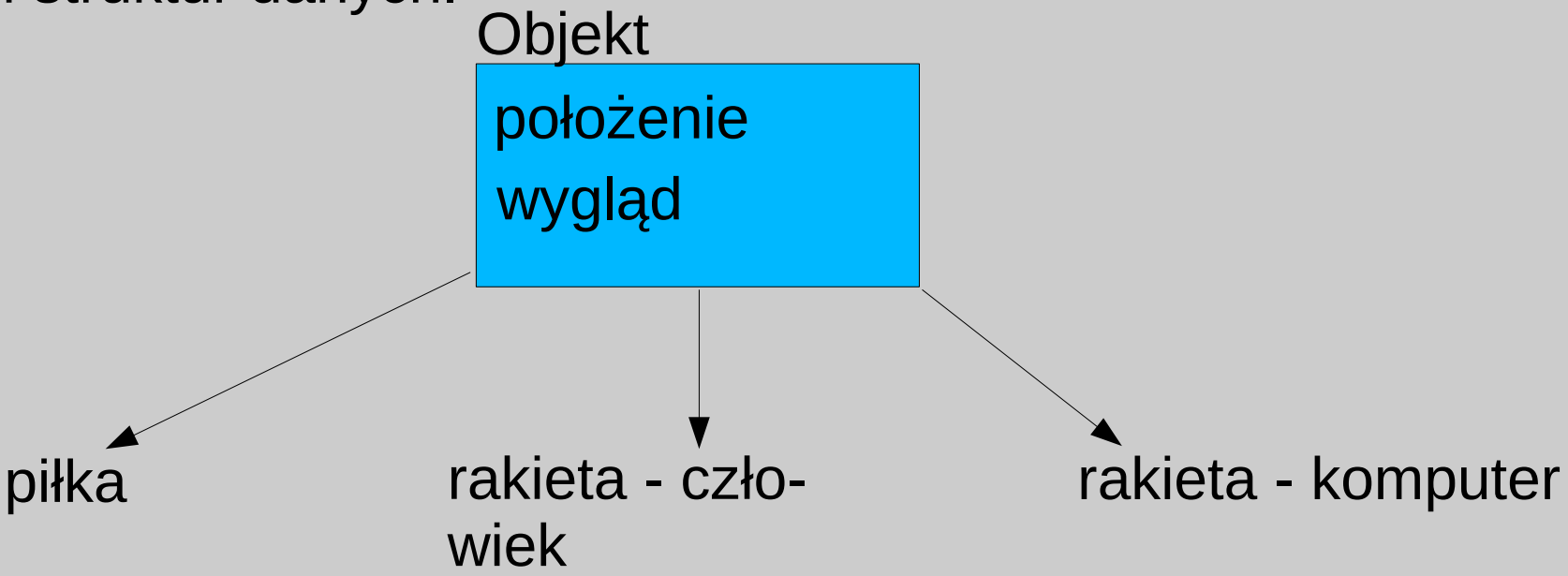
Dla poszczególnych modułów zmienne i definicje zazwyczaj definiuje się niezależnie, są jednak sytuacje w których dwa moduły korzystają z tego samego typu danych czy też definicji. Dlatego stosuje się pliki nagłówkowe w których występują definicje i deklaracje.

Dobre strony podziału na moduły

- Grupa programistów może pisać w tym samym czasie program (każdy pracuje nad innym modułem (funkcją).
- Można zastosować styl obiektowy. Każdy obiekt jest niezależny i z każdym związane są pewne operacje.
- Dobrze zaimplementowane obiekty lub funkcje mogą być użyte w innym programie.
- Gdy zostanie dokonana zmiana w jednym pliku tylko ten plik wymaga ponownej kompilacji.

Jak dokonać podziału programu na osobne pliki?

Projektowanie programu polega na podzieleniu problemu na małe pod-problemy. Często są z tym związane obiekty, które implementowane są w postaci struktur danych.



GNU make

GNU make jest to narzędzie pozwalające w sposób automatyczny kompilować (rekompilować) duży program.

Aby wykonać kompilacje programu najpierw trzeba stworzyć plik *makefile,* które zawiera informację dla programu make określające w jaki sposób kompilować program.
Prosty makefile składa się z reguł mających następującą postać:

cel .... : prerequisites (składniki) ...
komenda
.....

cel zazwyczaj oznacza nazwę programu (pliku), który zostanie wygenerowany
prerequisites plik, który musi być użyty aby stworzyć cel.
komenda komenda która zostanie wykonana (np. kompilacja lub linkowanie), w każdej linii komend pierwszy znak musi być tabulacją.

Przykład makefile-a

```
edit : main.o kbd.o command.o display.o \
        insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

Program make wywołany bez argumentów wczytuje plik Makefile i wykonuje pierwszą regułę zawartą w tym pliku. Dlatego często jako pierwszy definiuje się cel all, który zależy od wszystkich plików wynikowych jakie chcemy stworzyć w danym projekcie.

Zmienne w makefile-u

```
edit : main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
```

Powtarzanie fragmentów często prowadzi do błędu, żeby tego uniknąć stosuje się zmienne. Zmienne pozwalają zdefiniować raz łańcuch znaków i później używać go w innym miejscu.

```
objects = main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
```

Zmienne standardowe

- CC - nazwa kompilatora języka C
- CXX - nazwa kompilatora języka C++
- CFLAGS - opcje kompilatora języka C
- CXXLAGS - opcje kompilatora języka C
- LFLAGS - opcje dla linkera

Zmienne automatyczne

Symbol	Nazwa	Opis
$@	Cel	Obiekt, który znajduje się po lewej stronie dwukropka w regule.
$^	Wszystkie prerekwizyty	Obiekty, które znajdują się po prawej stronie dwukropka.
$<	Pierwszy prerekwizyt	Obiekt, który znajduje się bezpośrednio po prawej stronie dwukropka.
$?	Wszystkie prerekwizyty	nowsze niż cel (oddzielone spacjami)

```
hellomake: hellomake.c hellofunc.c
     gcc -o hellomake hellomake.c hellofunc.c -I.
```

```
CC=gcc
CFLAGS=-I.
hellomake: hellomake.o hellofunc.o
     $(CC) -o hellomake hellomake.o hellofunc.o -I.
```

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)
hellomake: hellomake.o hellofunc.o
        gcc -o hellomake hellomake.o hellofunc.o -I.
```

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o
%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)
hellomake: $(OBJ)
        gcc -o $@ $^ $(CFLAGS)
```

$@ ($^) - nazwa pliku znajdująca się po lewej (prawej) stronie :
$< - pierwszy element znajdujący się na liście zależności

Budowa makefile-a

- Makefile składa się z 5 części:
  - reguły jawne – określają kiedy i jak utworzyć jeden lub więcej obiektów, zawiera listę plików, od których zależy utworzenie *celu*
  - reguły wzorcowe - określają kiedy i jak utworzyć klasą obiektów bazując na ich nazwie
  - definicje zmiennych – definiuje łańcuch znaków, które mogą być użyte gdzieś w makefile-u
  - dyrektywy – są to komendy dla *make* aby wykonać jakieś specjalne działanie podczas odczytywania makefile-a
    - wczytanie innego makefile-a
    - podjęcie decyzji (na podstawie zmiennych) czy ominąć część makefile-a
    - definiowanie zmiennych za pomocą dyrektywy define
  - komentarze – linie zaczynające się od # oznacza komentarz

Dyrektywa PHONY

Aby uniknąć nieporozumień należy użyć dyrektywy .PHONY, która mówi, że dana reguła nie jest nazwą pliku.

np.

```
CFLAGS  := -O3 -Wall -c
LDFLAGS := -s -lm
.PHONY: all clean

all: hello.x
hello.o: hello.c hello.h
%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $^
%.x: %.o
    $(CC) $(LDFLAGS) -o $@ $^
clean:
    $(RM) *.o *.x
```

Instrukcja warunkowa

Instrukcja warunkowa składa się z 3 słów kluczowych:
if else endif

Ifeq ifneq ifdef

```
Ifeq warunek1
        Jeśli warunek1 prawdziwy
else warunek2
        Jeśli warunek2 prawdziwy
else
        Jeśli warunki nie prwadziwe
endif
```

```
ifeq ($(CC),gcc)
  libs=$(libs_for_gcc)
else
  libs=$(normal_libs)
endif

foo: $(objects)
	$(CC) -o foo $(objects) $(libs)
```

Funkcje w makefile-u

Funkcje umożliwiają wykonywanie operacji na łańcuchach znaków
Składnia:

```
$(funkcja argumenty)
```

Lub

```
${funkcja argumenty}
```

Argumenty funkcji są odseparowane od nazwy za pomocą spacji lub tabulacji, argumenty między sobą odseparowane są za pomocą przecinka.

Przykłady funkcji

$(wildcard ścieżka/*) – funkcja pozwalająca pobrać listę plików ze wskazanej ścieżki
$(foreach var,list,text)

Najpierw rozwijane jest var i list, później text

dir := a b c d

CFILES := $(foreach dir,$(SOURCES),$(notdir $(wilcard $(dir)/*.c)))

Spis funkcji można znaleźć w: https://www.gnu.org/software/make/manual/html_node/Functions.html

Wady makefile

- Brak śledzenia zależności plików
- Uzależnienie kompilacji od specyficznego systemu
- Podejmowanie decyzji o kompilacji tylko na podstawie czasu zmiany plików
- Inne znaczenie tabulacji i spacji.
- Mylące może być to, że instrukcje nie są wykonywane w kolejności napisanej w skrypcie

Cmake- (cross-platform make)

- Rozpoznaje jaki kompilator użyć do danego kodu źródłowego
- Rozpoznaje położenie bibliotek
- Potrzebny jest plik CmakeLists.txt, w którym definiuje się jak skompilować kod i gdzie zainstalować

Autoconf

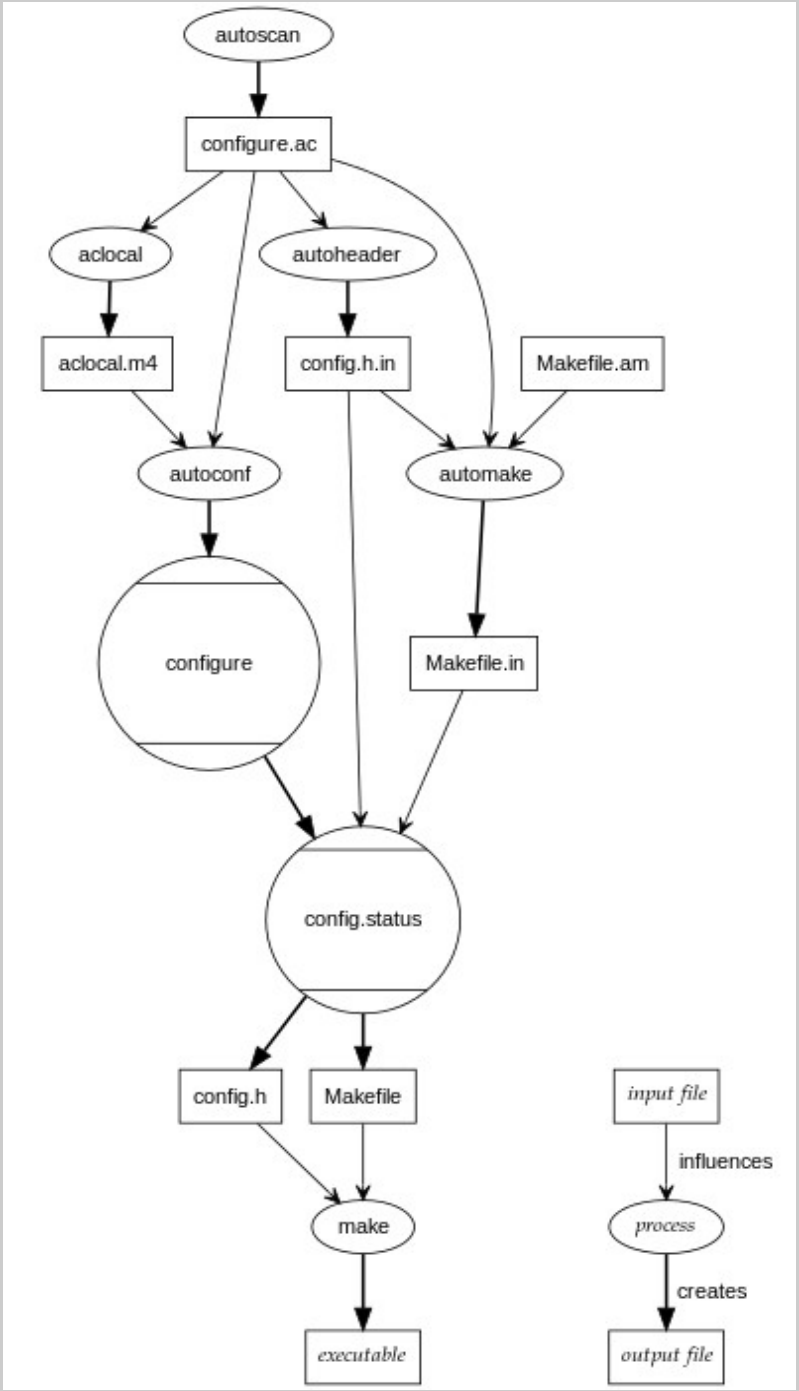
Autoconf – program produkujący skrypt shell-owy, który automatycznie konfiguruje źródła pakietu tak aby dostosować się do różnych systemów Posix-owych. Skrypty utworzone przez autoconf są niezależne od autoconf.

Plik konfiguracyjny generowany przez autoconf zazwyczaj nazywany jest *configure.*

Po uruchomieniu *configure* generuje następujące pliki:

- Jeden lub więcej Makefile-i
- Skrypt shell-owy config.status, który po uruchomieniu tworzy Makefil-e
- config.cache
- config.log
-

Do utworzenia pliku configure potrzebny jest specjalny „program”, który zapisuje się zazwyczaj w pliku configure.ac (lub configure.in)


autoscan
configure.ac
aclocal
autoheader
aclocal.m4
config.h.in
Makefile.am
autoconf
automake
configure
Makefile.in
config.status
config.h
Makefile
make
executable
input file
influences
process
creates
output file

Configure.ac

W pliku configure.ac znajdują się wywołania makr autoconf, które pozawalają sprawdzić czy dany system posiada narzędzia (cechy), które konieczne są do kompilacji danego pakietu.

Configure.ac może również zawierać kod bourne shell-a.

Przykład zastosowania makra w configure.ac

```
AC_CHECK_HEADER(stdio.h,[AC_DEFINE(HAVE_STDIO_H, 1,[Define to 1 if you have <stdio.h>.])],
   [AC_MSG_ERROR([Sorry, can't do anything for you])])

AC_CHECK_HEADER([stdio.h],[AC_DEFINE([HAVE_STDIO_H], 1,[Define to 1 if you have <stdio.h>.])],
 [AC_MSG_ERROR([Sorry, can't do anything for you])])
```

Standardowa struktura configure.ac

AC_INIT(package, version, bug-report-address)
information on the package
checks for programs
checks for libraries
checks for header files
checks for types
checks for structures
checks for compiler characteristics
checks for library functions
checks for system services
AC_CONFIG_FILES([file...])
AC_OUTPUT

Autoscan

Bardzo pomocny programem do wygenerowania pliku configure.ac jest autoscan. Autoscan analizuje pliku źródłowe pakietu znajdujące się we wskazanym katalogu i na tej podstawie generuje plik configure.scan, który może być wzorcem dla pliku configure.ac.

```c
/* hello.c: A standard "Hello, world!" program */

#include <stdio.h>

int main(int argc, char* argv[])
{
   printf("Hello, world!\n");

   return 0;
}
```

```
# Makefile: A standard Makefile for hello.c
all: hello
clean:
   rm -f hello *.o
```

Teraz można uruchomić autoscan
mv configure.scan configure.ac
Autoconf
mv Makefile Makefile.in
./configure

```c
/* hello.c: A program to show the time since the Epoch */

#include <stdio.h>
#include <sys/time.h>

int main(int argc, char* argv[])
{
   double sec;
   struct timeval tv;

   gettimeofday(&tv, NULL);
   sec = tv.tv_sec;
   sec += tv.tv_usec / 1000000.0;

   printf("%f\n", sec);

   return 0;
}
```

```c
/* hello.c: A program to show the time since the Epoch */

#include <stdio.h>
#include "config.h"

#ifdef HAVE_SYS_TIME_H
#include <sys/time.h>
#else
#include <time.h>
#endif

double get_sec_since_epoch()
{
   double sec;

   #ifdef HAVE_GETTIMEOFDAY
      struct timeval tv;

      gettimeofday(&tv, NULL);
      sec = tv.tv_sec;
      sec += tv.tv_usec / 1000000.0;
   #else
      sec = time(NULL);
   #endif

   return sec;
}

int main(int argc, char* argv[])
{
   printf("%f\n", get_sec_since_epoch());

   return 0;
}
```