Uzupełnienie dot. przekazywania argumentów

```
#include <stdio.h>
#include <stdlib.h>
struct nowa { int f; char line[20000]; int k; } reprezentant;
     int main()
{   void funkcja7( struct nowa x);
    reprezentant.k=17;       funkcja7(reprezentant);
   printf(" reprezentant.k= %d\n",reprezentant.k);    exit(0);
} /*  koniec  funkcji  main  */
          void funkcja7( struct nowa x)
         {  printf("\n funkcja7  x.k=%d\n",x.k);
            x.k=-18;
            printf("\n funkcja7  x.k=%d\n",x.k);
         }/*   koniec funkcji  funkcja7  */
```

Uzupełnienie dot. przekazywania argumentów

Program wypisze:

funkcja7 x.k=17

funkcja7 x.k=-18

reprezentant.k= 17

Wniosek: w programie wywołującym nie ma zmiany argumentu

Uzupełnienie dot. przekazywania argumentów

```
#include <stdio.h>
#include <stdlib.h>
int nowa[10000];
   int main()
{    void funkcja6( int a[10000] );     nowa[10]=27;
     funkcja6( nowa );                  printf("a[10]=%6d\n",nowa[10]);
     exit(0);
} /*  koniec  funkcji  main  */
        void funkcja6( int a[10000] )
       {  printf("\n funkcja6  a[10]=%d\n",a[10]);
          a[10]=-18;
         printf("\n funkcja6  a[10]=%d\n",a[10]);
       }/*   koniec funkcji  funkcja6  */
```

Uzupełnienie dot. przekazywania argumentów

Program wypisze:

funkcja6 a[10]=27

funkcja6 a[10]=-18

a[10]= -18

Wniosek: w programie wywołującym jest zmiana argumentu!

Przykład – wskaźnik do struktury

```
#include <stdio.h>
#include <stdlib.h>
        struct alfa { char buf[16*16*16];   /* 4096 */      int k;
                        };
typedef struct alfa koral;        koral array[10];
koral * wskaznik;
          int   main()
{         wskaznik = array;
           printf("%p\n",wskaznik);
          wskaznik++;
           printf("%p\n",wskaznik);
           exit(0);
}/* koniec  main  */
```

Przykład – wskaźnik do struktury

Program wypisze:

0x8049640

0x804a644

Wniosek: wskaźnik (jego zawartość) zmodyfikowała się o ilość bajtów potrzebnych na jeden element typu, na który może wskazywać

Uwaga: niech program sam wylicza, jaka ilość bajtów jest potrzebna !

adres zmiennej

Do pobrania adresu zmiennej używa się jednoargumentowego operatora & (uwaga & może mieć także znaczenie dwuargumentowego operatora bitowego iloczynu logicznego)

Jednoargumentowy operator * jest "używany do wskazywania", tzn. jego argument jest adresem zmiennej.

Wskaźniki
(wskaźnik do funkcji)

Nazwa funkcji jest jednocześnie adresem jej początku – czyli adresem miejsca w pamięci, gdzie zaczyna się kod odpowiadający instrukcjom tej funkcji.

(czyli jest podobnie jak w przypadku tablic!)

Wskaźnik powinien na coś wskazywać (choćby na NULL)

```
float  * f4 = NULL;


/*  trzeba uważać by obiekt wskazywany nie zniknął !!
*/
```

malloc (dynamiczna alokacja pamięci) - malloc3.c

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

struct st
 { int pole1;
   int pole2;
   float pole3;
   double buf[100000];
 } z1, *p1, *p2;

  main()
{
  p1= &z1;
  p1->pole1=11.;
  p1->pole2=13.;
  p1->pole3=15.;

  printf("\n");
  printf("  pole1 %5d   pole2 %5d   pole3 %11.3f\n",z1.pole1,z1.pole2,z1.pole3);

  printf(" sizeof...%d \n\n",sizeof(z1) );

  while(1)
  {
  p2=malloc(sizeof(z1));
  if( p2==NULL)
          { printf("\n .....zabraklo pamieci..."); break;
          }
  (*p2).pole1=21;
  (*p2).pole2=22;
  (*p2).pole3=23;
/*  free(p2);*/
  }
```

malloc

void * **malloc** *(size_t size)*

funkcja ta zwraca wskaźnik do nowo zaalokowanego bloku pamięci o długości w bajtach *size ;*

w przypadku niepowodzenia funkcja zwraca wskaźnik zerowy NULL

uwaga – blok pamięci pozostaje niezainicjalizowany (tzn. jego zawartość jest niezdefiniowana)

free

void **free** *(void *ptr)*

funkcja ta zwalnia (dealokuje) blok pamięci który został przydzielony przez **malloc**, wskazywany przez *ptr*

realloc

void * **realloc** *(void *ptr, size_t newsize)*

Funkcja ta zmienia wielkość bloku którego adres jest *ptr* – nową wielkością jest *newsize*. Uwaga – ponieważ przestrzeń za końcem bloku może być zajęta, **realloc** może dokonać kopiowania bloku pamięci do nowego adresu, tam gdzie jest więcej wolnej pamięci. **realloc** zwraca adres bloku; jak widać nie musi to być adres *ptr*. Jeśli się nie uda zmienić wielkości bloku pamięci, to wtedy **realloc** zwraca wskaźnik zerowy NULL.

Jeśli ptr będzie wskaźnikiem o wartości NULL, to **realloc** zadziała tak jak **malloc(newsize)** .

Wskaźnik do struktury

```
struct nowa { float f1;
              char line[20000];
              int k;
            }   reprezentant;
struct nowa * pf;
pf = & reprezentant;
      (*pf).f1=33.4 ;   /*  nawiasy  są konieczne  */
      (*pf).k  = 17;
        pf -> k   =34;    /* operator wskaźnika struktury   */
```

Wskaźnik do struktury-2

```
struct faa *ptr;

ptr  =  (struct faa *)  malloc (sizeof (struct faa));

if (ptr == 0) abort ();


memset (ptr, 0, sizeof (struct faa));

  /*  memset służy do inicjalizowania pamięci  */
```

Wskaźnik do struktury-3

```
#include <stdio.h>

#include <stdlib.h>

 int main()

{

struct nowa { int f;

            char line[20000];

            int k;

          }   reprezentant;

struct nowa * pf;

    .....
```

Wskaźnik do struktury-4

```
  .....
pf = malloc(sizeof(struct nowa) );


memset(pf,1, sizeof(struct nowa) );
  /*      memset(pf,0, sizeof(struct nowa) );    */
printf("\n %d  %d  %d\n",pf->f, pf->line[10], pf->k);
} /*  koniec  funkcji  main  */
```

wynik programu 16843009 1 16843009

calloc

void * **calloc** *(size_t count, size_t eltsize)*

funkcja zwraca blok pamięci dostatecznie długi aby mógł pomieścić *count* elementów, każdy o długości *eltsize* ;

zawartość pamięci jest zerowana ! (tzn. inicjalizowana zerami)

calloc

void * **calloc** *(size_t count, size_t eltsize)*

```
void * calloc (size_t count, size_t eltsize)
{ size_t size = count * eltsize;
  void *value ;
   value = malloc (size);
  if (value != 0) memset (value, 0, size);
  return value;
} /*   koniec funkcji  calloc    */
```

memset

void * **memset** *(void *block, int c, size_t size)*

funkcja kopiuje wartość *c* (jako unsigned char) do *size* pierwszych bajtów, począwszy od adresu wskazywanego przez wskaźnik *block*

(czyli do bloku pamięci wskazywanego przez wskaźnik *block*)

PRZYKŁAD:

```
struct nowa * pf;
void   *  wsk;
pf   =   malloc  ( sizeof  (struct nowa)  );
wsk = memset(pf,1, sizeof(struct nowa) );
printf("\n\n pf, wsk %p %p\n",pf,wsk);
```

memset

void * **memset** *(void *block, int c, size_t size)*

W przypadku powodzenia **memset**() zwraca wskaźnik równy wskaźnikowi *block*

memset

void * **memset** *(void *block, int c, size_t size)*

memset najczęściej używa się w stosunku do pamięci dynamicznie przydzielonej przez malloc (czy calloc), ale można jej użyć także do np. ”wyzerowania” większej struktury danych (przykład poniżej)

memset

void * **memset** *(void *block, int c, size_t size)*

```
/* przyklad inicjalizowania pamieci
   przygotowanej dla macierzy structur */
#include <stdio.h>
#include <stdlib.h>
struct nowa { int f;
              char line[992];
              int k;            /* alfa[] zdefiniowana statycznie*/
            } alfa[100];    /* kontynuacja na następnej stronie */
```

memset

void * **memset** *(void *block, int c, size_t size)*

```
int main()
{  alfa[10].k= -223;
   printf("%d\n",alfa[10].k);
   memset(alfa,0,sizeof(alfa));
   printf("%d     %d\n",alfa[10].k, sizeof(alfa));
   exit(0);
}/* koniec  funkcji  main*/
          /*  wydruk              -223
                                  0   100000                    */
```

memcpy

Function: void * **memcpy** *(void *to, const void *from, size_t size)*

The memcpy function copies *size* bytes from the object beginning at *from* into the object beginning at *to*. The behavior of this function is undefined if the two arrays *to* and *from* overlap; use memmove instead if overlapping is possible.

The value returned by memcpy is the value of *to*.

Here is an example of how you might use memcpy to copy the contents of an array ”of struct foo”:

```
struct foo *old, *new; int arraysize; ...
memcpy (new, old, arraysize * sizeof (struct foo));
```

mempcpy

Funkcja: void * **mempcpy** (*void * to, void * from, size_t size*)

mempcpy jest bardzo podobna do funkcji memcpy. Kopiuje *size* bajtów z miejsca wskazywanego przez *from* do miejsca wskazywanego przez *to*. Jednakże w przypadku sukcesu zwraca wskaźnik do bajtu następującego po ostatnim zapisanym bajcie, to znaczy zwraca

((void *) ((char *) *to* + *size*))

memmove

Function: void * **memmove** *(void *to, const void *from, size_t size)*

memmove copies the *size* bytes at *from* into the *size* bytes at *to*, even if those two blocks of space overlap. In the case of overlap, memmove is careful to copy the original values of the bytes in the block at *from*, including those bytes which also belong to the block at *to*.

przykład dynamicznej alokacji macierzy dwuwymiarowej

program malloc9b.c (na osobnym zbiorze)

Jak czytać skomplikowane deklaracje?
(”Symfonia C++” J.Gręboszz, 8.17.1)

.

```
void  zz  ( int (*aa) (float), float t );

/*  co to jest za obiekt ?  */
```

Jak czytać skomplikowane deklaracje? ("Symfonia C++" J.Grębosz, 8.17.1)

1. Zaczynamy czytanie od nazwy "**tego co jest deklarowane**" .

2. Od tej nazwy posuwamy się w prawo. To dlatego, że tam mogą stać najmocniejsze (jeśli chodzi o priorytet) operatory: operator wywołania funkcji **(....)** bądź operator indeksowania tablicy **[]** .

3. Jeśli w prawo już nic nie ma, lub natkniemy się na zamykający nawias – wówczas zaczynamy czytanie w lewo. Kontynuujemy tak długo, dopóki wszystkiego nie przeczytamy, lub dopóki nie natkniemy się na zamykający nawias.

4. Jeśli napotkamy taki nawias, to wychodzimy z czytaniem na zewnątrz nawiasu. Znowu zaczynamy czytać w prawo, czyli wracamy do punktu 2.

5. I tak dalej, dopóki nie przeczytamy wszystkiego w tej deklaracji.

Jak czytać skomplikowane deklaracje? (”Symfonia C++” J.Gręboszz, 8.17.1)

Jak czytamy? Słownik!

* jest wskaźnikiem mogącym pokazywać na...

(typ1,typ2,...) jest funkcją wywoływaną z argumentami typ1, typ2.....

[n] jest n-elementową tablicą.....

Jak czytać skomplikowane deklaracje?
(”Symfonia C++” J.Gręboszz, 8.17.1)

```
float  *  fp;

char  *  tan (float p, int s);

char   (* tas ) (float p, int s);

char * (* tas) (float p, int s);
```

Jak czytać skomplikowane deklaracje?
(”Symfonia C++” J.Grębosz, 8.17.1)

```
int   ( *  (*fw)  (int a, char * b)  )   [2] ;
```

Jak czytać skomplikowane deklaracje?

```
struct kos { float as;
             int k[10000];
           }  kos1;
struct kos kos2;
struct kos *fp1;
struct kos (*fp2);
struct kos (*fp3)(int a, int b);

struct kos  * (*fp4[6])(  struct kos *a,  struct kos (*b)(void) );
```

Jak czytać skomplikowane deklaracje?

```
float  *  fw [6] (int a);
/*  tego powyżej kompilator zbudować nie potrafi.... */
/*  ale właściwie nie jest to potrzebne    */

/* natomiast kompilator potrafi zbudować poniższy obiekt*/
char  (* fp) [20];  /* czy może być przydatny ?   */
```

argumenty wiersza poleceń

funkcja main ma dwa argumenty

```
int main(int argc, char * argv [ ] )
```

(ich nazwy to argc, argv, tak są zwyczajowo nazywane, choć nie są to nazwy obowiązujące)

argumenty wiersza poleceń

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
 int   main  (int argc,  char *argv[] )
   /*    albo  int main( int argc,   char  **argv)    */
{ int n;
  for(n=0;n<argc; ++n)
  {  printf("\n            n=%d  param= %s",n, argv[n]);
     printf("\n to samo... n=%d  param= %s",n, *(argv+n) );
     printf("\n");
  }     exit(0);
 }  /*    koniec     funkcji     main     */
```