nowe operatory

&	. (kropka)
*	operator rzutowy
->	, (przecinek)
sizeof	

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

(*wskaźnik)

(inna nazwa to operator dereferencji)

sizeof

operator służy do uzyskania informacji o wielkości typu lub zmiennej (w bajtach)

```
struct alfa { char line[100];
              float ff[100];
            }  kos;
```

sizeof(struct alfa); sizeof(float);

sizeof(kos);

->

jest to wskaźnik do elementu struktury

wskaźnik -> element struktury

uwaga: nie jest sprawdzane czy wskaźnik wskazuje na strukturę !

pptr -> pole1;

operator rzutowy

```
float f, *ptr_f;

int n, *ptr_n;


n  = (float)  f;

f  = (int)     n;


ptr_f = (float *) ptr_n;   /* rzutowanie wskaźników */


ptr_n = (int *) n ; /* załadowanie konkretnego adresu */
```

sizeof i struktura

```
struct alfa {...lista...} reprezentant;

   sizeof (reprezentant);

   sizeof (  reprezentant. pole1 );  /* to odwołanie jest
                                        poprawne */
```

. /* kropka */

-kropka nie została wymieniona jako operator wśród 15 grup operatorów, ale ...jest operatorem dającym możliwość odwołania się do elementów struktury

-należy do operatorów o najwyższym priorytecie

krótka dygresja nt. struktury
struct

jedyne operacje jakie można wykonać na strukturze to znalezienie jej adresu (za pomocą operatora **&**) i odwołanie się do jej elementów (z użyciem operatora **.** lub kombinacji operatorów ***** i **.** albo ich odpowiednika **->**)

(w niektórych kompilatorach można przekazać strukturę jako argument funkcji - także w gcc z "Free Software Foundation")

&alfa; alfa.pole ; (*ptr_alfa).pole;

ptr -> pole ;

dygresja nt. struct

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

dygresja nt. struct

```
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  */
```

dygresja nt. struct - ciąg dalszy

/* inicjalizacja struktur */

```
struct klasa{ int n3;   float g4;  char  line[20]; };

......

typedef struct klasa typek;

typek  reprezentant1, reprezentant2={3,2.7,”Krakus”};
```

dygresja nt. struct - ciąg dalszy

do pól bitowych struktury nie można stosować operatora &

(natomiast jest poprawnym odwołanie przez operator & do elementu struktury który nie jest polem bitowym)

```
& (ptr->pole1);   /*  ptr jest wskaźnikiem do
                      struktury */
```

operator przecinkowy

```
/*   używany w celu grupowania instrukcji    */
k=7,f=13.4, funkcja(z);

k= (i=2,  j=12,  i*j);
    /*    wartość powyższego wyrażenia wynosi  24   */
```

operator przecinkowy

```
/*   używany w celu grupowania instrukcji    */

for(n1=1, n2=1;  n1<5  &&  n2<6;  ++n1,  ++n2)
    printf("\n  n1=%d   n2=%d\n", n1,n2);
```

operator spoza ortodoksyjnego C
&&

```
/*  adres  etykiety ;   operator &&  */
#include <stdio.h>
#include <stdlib.h>
     int  main()
{float n=1;
  char c;
  void *fp;
n:  ;
  ++n;
  printf("\n n=%f\n",n);    scanf("%c",&c);
  fp= && n;    /* to załadowanie fp można umieścić wcześniej... */
  goto *fp;
} /* koniec funkcji main*/
```

operator spoza ortodoksyjnego C
&&

Jedną z możliwości użycia adresów etykiet jest zainicjalizowanie tablicy o klasie pamięci static; tablica ta będzie przechowywać te adresy jako ”adresy skoków”.

```
static void *array[] = { &&etyk1, &&etyk2, &&etyk3 };

goto *array[i];

  /* oczywiście nie będzie sprawdzane czy  array[i] zostało
zainicjalizowane! */
```

1.wydaje się, że instrukcja switch jest wystarczająco bogatym w możliwości rozgałęźnikiem

2.skoki wykonywać można tylko w ramach danej funkcji

funkcje zagnieżdżone – tego nie ma w ANSI C

```
 int main()
{ float func(float w, float y);    float a,b,c;
   b=4; c=3;
   a=func(b,c);   printf("\n a=%f\n",a);    exit(0);
} /* koniec funkcji main*/
float func ( float  x1,  float  x2)
{ float ff(float f1)
      { return(f1*f1);
      } /* koniec funkcji ff,  lokalnej wewnatrz func  */
   return(ff(x1)+ff(x2));
} /*  koniec  funkcji    func            gcc –ansi  -pedantic  ”nazwa zbioru”
*/
```

Wskaźniki

- Typ wskaźnika
- Wskaźnik uniwersalny (void *)
- Rzutowanie wskaźnika na wskaźnik (operatory rzutowe dla wskaźników)

```
int * wsk1;

float * wsk2;

wsk1 = (int *)  wsk2;  /*rzutowanie*/
```

Wskaźniki

wskaźnik jako parametr wywołania funkcji

wskaźnik jako argument zwracany przez funkcję

(bardzo) bliskie pokrewieństwo tablica ←→ wskaźnik do niej

(wskaźniki a macierze)

```
float as[10];     /*   każdy element ma długość 4 bajtów */
float *  p3;


*(as) = 7.3 ;          /*   as[0]=7.3;     */
*(as+5)=-4.5;          /*   as[5]=-4.5;    */
```

/ dodawanie liczby całkowitej do wskaźnika skaluje się automatycznie */*

(wskaźniki a macierze)

```
#define SIZE 5

void funkcja3( int as[SIZE]);

     int  main ( )

{int n;

  static int array[SIZE]={0,1,2,3,4};

  for(n=0;n<5;++n) printf("  %d", *(array+n) );

  funkcja3(array);

  printf("\n\n po powrocie z funkcji  funkcja3\n");

  for(n=0;n<5;++n) printf("  %d", *(array+n) );

  printf("\n");   exit(0);

} /* koniec funkcji main*/
```

(wskaźniki a macierze)

```
void funkcja3( int as[SIZE])
{   as[3]=-3;
}/*  koniec funkcji   funkcja3  */
```

0 1 2 3 4

po powrocie z funkcji funkcja3

0 1 2 -3 4

(wskaźniki a macierze)

```
#define SIZE 5
void funkcja4( int as[SIZE]);
  main()
{int n;
  static int array[SIZE]={0,1,2,3,4};
  for(n=0;n<5;++n) printf("  %d", *(array+n) );
  funkcja4(array);
  printf("\n\n po powrocie z funkcji  funkcja4\n");
  for(n=0;n<5;++n) printf("  %d", *(array+n) );
  printf("\n");
} /* koniec funkcji main*/
```

```
void funkcja4( int * as)
{int n;
  printf("\n\n początek funkcji  funkcja4\n");
  for(n=0;n<5;++n) printf("  %d", *(as++) );
  as+=-2;    /*  as=as-2;  */
  (*as)=-3;
}/*  koniec funkcji   funkcja4  */
```

(wskaźniki a macierze)

0 1 2 3 4

początek funkcji funkcja4

0 1 2 3 4

po powrocie z funkcji funkcja4

0 1 2 -3 4

(wskaźniki)

```
    main()
{  void funkcja5 ( float** );    float n,*fp;    fp =  & n;
 printf("\n poczatek main fp=%p   &fp=%p\n",  fp,  &fp);
 funkcja5( &fp );
 printf("\n koniec   main fp=%p   &fp=%p\n",  fp,  &fp);  exit(0);
} /* koniec funkcji main*/
    void funkcja5(float ** wsk)
{ printf("\n poczatek funkcja5 *wsk=%p wsk=%p &wsk=%p\n",
                  *wsk,  wsk,  &wsk);
  *wsk=NULL;
  printf("\n koniec funkcja5 *wsk=%p wsk=%p &wsk=%p\n",
  *wsk, wsk, &wsk); }            /*  koniec funkcji   funkcja5  */
```

wynik programu :

poczatek main fp=0xbffffa64 &fp=0xbffffa60

poczatek funkcja5 *wsk=0xbffffa64 wsk=0xbffffa60 &wsk=0xbffffa5c

koniec funkcja5 *wsk=(nil) wsk=0xbffffa60 &wsk=0xbffffa5c

koniec main fp=(nil) &fp=0xbffffa60

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!)

przykład :

Wskaźniki
(wskaźnik do funkcji)

```
#include <stdio.h>
#include <stdlib.h>
   int main()
{   float func(float w, float y);
    float a, b=6, c=5;
    a=(*func) (b,c);     printf("\n a=%f\n",a);
    exit(0);
} /* koniec funkcji main*/
float func(float x1, float x2)
{  return  (  x1 * x2 );
}    /*   koniec  funkcji  func   */
```