Ukazatelé (pointery)

Ukazatel představuje adresu v paměti. Je vždy svázán s nějakým datovým typem.
int i; int *p_i;
Tyto dva příkazy můžeme napsat najednou: int *p_i, i;
Pozor!! Napíši-li int * p_i, p_j; je pointer jen p_i, p_j je proměnná typu integer.
int i, *p_i=&i; /* definice a současně inicializace */ Nebo: p_i=&i;
& adresní, referenční operátor
* derefenční operátor
Operátor & je možno použít jen na proměnné, ne na konstanty
Příklady:

```
int i, *p_i;
i=3;
*p_i=4;
i=*p_i;
*p_i=i;
p_i=&i; *p_i=4;
```

Operátor * má vyšší prioritu než operátor +, takže příkaz i=*p_i1+1; je totéž jako i=(*p_i1)+1;
Operátor ++ má stejnou prioritu jako *, ale je použit jako postfix, takže i=*p_i1++; má význam i=*p_i1; p_i1++; Toto se používá při práci s řetězci:

```
#include <stdio.h>
#include <conio.h>
#define MAX 20
char *p_c, *p_d, *p_t;
int main()
{
   char c[20]="Ahoj              ";
   char d[20];
   p_c=c; p_d=d;
   /* tisk retezce c */
   for  (p_t=p_c; p_t<p_c+MAX; p_t++)
   printf("%c",*p_t);
   /* kopirovani retezce */
   for(p_t=p_c; p_t<p_c+MAX; )
      *p_d++=*p_t++; /* totez jako *p_d=*p_t; p_d++; p_t++; */
   /* ted ukazuje p_d na první bajt za nově zkopírovaným blokem */
   p_d=p_d-MAX; /* p_d ukazuje na zacatek zkopirovaneho retezce */
   /* tisk zkopirovaneho retezce */
   for  (p_t=p_d; p_t<p_d+MAX; p_t++)
   printf("%c",*p_t);    getch();
   return 0;
}
```

Odečítání pointerů má význam pouze, ukazují-li na stejné pole dat. Sčítat pointery nelze!
Nulový pointer NULL je možno přiřadit pointerům všech typů. Označuje, že pointer neukazuje momentálně na nic, tzn. nemá přidělenu paměť.

Konverze pointerů

```
char *p_c; int *p_i;
p_c=p_i; /* nevhodné!! */
p_c=(char *)p_i; /* lepší */

void vymen(int * p_x, int * p_y)
{
   int pom;
   pom=*p_x; *p_x=*p_y; *p_y=pom;
}
…..
int i=5,j=3;
vymen(&i,&j);
```

Pointery a funkce

Máme definovánu funkci

```
double sectidbl(double f,double g) {return(f+g);};
double (*p_fd)(); /* p_fd je pointer na funkci vracející double */
```

Prázdné závorky jsou nutné, jinak by šlo o pointer na typ double. Závorky kolem jména proměnné jsou také nutné, jinak by to byla deklarace funkce.

```
p_fd=sectidbl; /* přiřadí pointeru p_fd adresu funkce sectidbl() */.
```

Definice s využitím operátoru typedef

typedef float * P_FLOAT; vytvoří nový typ jako pointer na float a pojmenuje jej P_FLOAT.

Pointerová aritmetika

Platné operace s pointery jsou

- součet pointeru a celého čísla n – odkazujeme na n-tý prvek za prvkem, na který právě ukazuje pointer. K pointeru se nepřičítá celé číslo, ale násobek tohoto čísla a velikosti typu, na který pointer ukazuje.
- rozdíl pointeru a celého čísla – odkazuje n n-tý prvek před prvkem, na který právě ukazuje pointer
- porovnání dvou pointerů stejného typu
- rozdíl dvou pointerů stejného typu má smysl, ukazují-li pointery na stejné pole dat

Dynamické přidělování paměti

Funkce pro tento účel jsou v knihovně stdlib.h. Pro přidělení paměti je funkce malloc, jejíž jediný parametr je typu unsigned int (počet bytů). Funkce vrací pointer na void, který musíme přetypovat na pointer na vhodný typ. Není-li v paměti místo funkce vrací NULL. Pro uvolnění paměti se používá funkce free, jejímž parametrem je pointer, který ukazuje na začátek dříve přiděleného místa: free((void *)p_c); p_c=NULL;

Chceme-li alokovat paměť pro n prvků o velikosti size, použijeme calloc(n,size), která alokuje toto pole prvků stejně jako malloc(n*size) a navíc je vynuluje.

Jednorozměrná pole

V jazyku C začíná pole vždy prvkem s indexem 0. Je-li definice pole např. int a[10], bude první prvek a[0] a poslední a[9]. Jazyk C zásadně nekontroluje meze polí. Protože pole začíná od nuly, vypočte se adresa libovolného prvku ze vztahu

&a[i]=bázová adresa a + i *sizeof(typ)

Jméno pole, v tomto případě a, je považováno za adresu v paměti. Jazyk C neumožňuje pracovat s celým polem najednou. I pro vynulování pole je nutno použít cykl:
for(i=0;i<10;i++) a[i]=0;
Protože jméno pole představuje adresu, je možné předání odkazem, má-li pole být parametrem funkce. Samozřejmě to znamená, že položky pole, které byly ve funkci změněny, si ponechají novou hodnotu i po opuštění funkce.

Struktury
Zatímco pole je složený datový typ tvořený složkami stejného typu, je struktura složený datový typ tvořený složkami různých datových typů. Definice struktury může vypadat např. takto:

```
struct miry {
  int vyska;
  float vaha;
} pavel,honza;  /* proměnné typu struktury miry */
```

Definice proměnných můžeme od definice struktury oddělit:

```
struct miry {    /* definice struktury */
  int vyska;
  float vaha;
};
struct miry pavel,honza;  /* musíme opakovat klíčové slovo struct */
```

Nechceme-li stále znovu psát struct, použijeme typedef:

```
typedef struct miry {    /* definice struktury */
  int vyska;
  float vaha;
} MIRY;
MIRY pavel,honza;
```

Přístup k prvkům struktury je pomocí tečkové notice:
karel.vaha=85.5; pavel.vyska=178; honza.vyska=pavel.vyska;
Máme-li definován na strukturu pointer p_s
MIRY *p_s;
Můžeme se na prvky struktury odvolávat takto:
p_s->vyska

Následující příklad ilustruje dynamické využívání paměti. Vytváří tzv. lineární spojový seznam.

```
/* vytvoří lineární seznam s hodnotami od 1 do n */
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
struct record
{
        struct record *next;
        int prvek;
};
int n;
typedef struct record zaznam;
zaznam *p;
```

```
zaznam *q;
int create(int n);
void pruchod(zaznam * p);
void release(zaznam * p);
int create(int n) /* vytvori lin.seznam s hodnotami od 1 do n */
{
	/* p ukazuje na zacatek seznamu */
	p = NULL;
	while (n > 0)
	{
		q=(zaznam*) malloc(sizeof(zaznam));
		if (q!=NULL)
		{
			/* printf("%d",n); */
			q->prvek = n;
			q->next = p;
			p = q;
			n = n - 1;
		}
		else return 1;
	}
	return 0;
}
void pruchod(zaznam *p) /* vytiskne hodnoty uložené v lin.seznamu */
{
	while (p!=NULL)
	{
		printf("%d\n", p->prvek);
		p = p->next;
	}
	getch();
}
void release(zaznam *p) /* uvolní přidělenou paměť */
{
	while(p!=NULL)
	{
		q = p->next;
		free (p);
		p=q;
	}
}
void main()
{
	n = 9;
	p = NULL;
	if (create(n) != 1)
	pruchod(q);
	release(q);
}
```