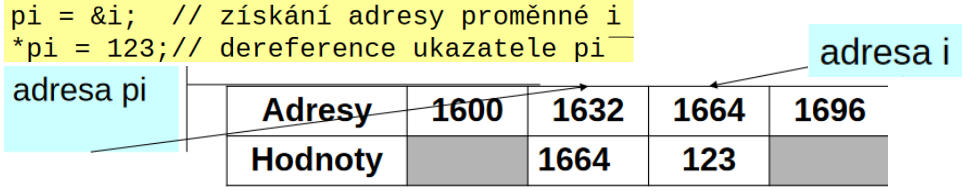
Ukazatel

- **Datový typ ukazatel**
  - **Obecný formát definice proměnné typu ukazatel:**
    - Bázový_typ *jméno_proměnné;
  - Do proměnné se ukládá adresa paměti
  - **Bázový typ**
    - Určuje velikost odkazované paměti
    - Libovolný datový typ, funkce, void
    - Počet bajtů při kopírování paměti a při nepřímém porovnávání
    - Nikdy vzájemně nepřiřazovat ukazatele různých typů (až na výjimky)!
  - **Operátory**
    - **& referenční**
      - Vrací adresu proměnné operandu.
      - ```
        int x = 10;
        int *px; // (int *) je datový typ
        px = &x; // reference; px,&x - adresa
        ```
    - *** dereferenční**
      - Vrací hodnotu uloženou na adrese operandu (hodnota získaná odkazem)
      - *px – místo v paměti o rozměru sizeof(int)
      - ```
        int y = *px; // dereference = odkaz
        ```
  - **Příklad: Zápis hodnoty 123 do proměnné i přes ukazatel.**
    - ```
      int i = 11;
      int *pi;
      ```
    - Situace po inicializaci i, obsah okolních paměťových míst není definovaný:
Adresy	1600	1632	1664	1696
Hodnoty			11	
    - ```
      pi = &i;  // získání adresy proměnné i
      *pi = 123;// dereference ukazatele pi
      ```
    - 

  - **Nulový ukazatel**
    - **Konstanta NULL ze <stdio.h>**
      - Lze přiřadit každému ukazateli
      - Pro test, že ukazatel nikam neukazuje
    - **Zásady bezpečného programování**
      - Ukazatel musí obsahovat bezpečnou adresu
      - Nepoužívané ukazatele nastavit na NULL

        - Testovat ukazatele, zda nejsou NULL
    - **Použití ukazatele bez inicializace**
        - Častá chyba začátečníků
        - Zápis do nealokované paměti havárie
- **Konstantní ukazatel**
    - Nelze měnit, odkazovanou paměť ano
- **Ukazatel na konstantu**
    - Lze měnit, odkazovanou paměť ne

```
int i;
int *pi;             // ukazatel
int * const CP = &i; // konstantní ukazatel
const int CI = 7;    // celočíselná konstanta
// neinicializovaný ukazatel na konstantu
const int *pci;
// konstantní ukazatel na konstantu
const int * const CPC = &CI;
```

- **Typový ukazatel**
    - Umožňuje typovou kontrolu
- **Obecný ukazatel (void *)**
    - Neumožňuje typovou kontrolu
    - Ale lze jej přetypovat na jakýkoli typový ukazatel, kompatibilní se všemi ukazateli
    - Nelze použít dereferenční operátor, pro praktické použití nutno přetypovat
    - Pro situace, kdy je potřeba měnit typ ukazatele za běhu programu podle kontextu řešené úlohy (jde o speciální případy!)

```
int inum = 10;
float fnum = 3.14;
void *pgeneric = &inum;
//přetypování je nutné, inum je nyní 20
*(int *)pgeneric = 20;

pgeneric = &fnum;
//nastaví fnum na 2.72
*(float *)pgeneric = 2.72;
```

- **Konverze ukazatelů**
    - Pomocí operátoru přetypování
    - Potenciálně nebezpečná operace – programátor na sebe bere veškerou zodpovědnost za rizika!

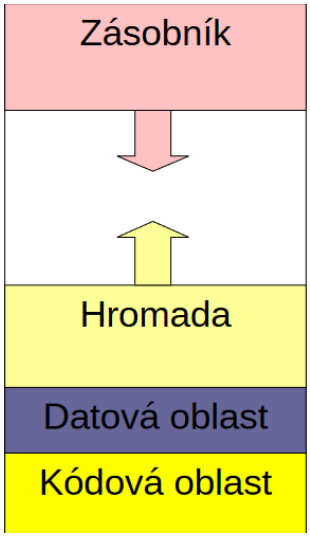
- **Ukazatel odkazující na jiný ukazatel**
    - Funkce – předávání ukazatelů odkazem
    - Tvorba složitějších dynamických datových struktur (seznam, zásobník, strom, …)
    - Např. Při vytváření vícerozměrných polí

```
int i = 12;
int *pi = &i;
int **ppi = &pi; //ukazatel na ukazatel na int
int j = **ppi;   //dvojitá dereference
```

- **Alokace paměti**
  - **Paměťové nároky proměnných**
    - **Sizeof(typ) + zarovnání**
  - **Paměťový prostor**
    - Zásobník (stack) – lokální proměnné + parametry funkcí
    - Hromada (heap) – dynamické proměnné
    - Dat. Oblast – globální proměnné + konstanty
    - Kódová oblast – kód programu
  - **Dynamická alokace**
    - Zásobník + hromada
    - Teoreticky lze využít celou paměť
    - Za běhu programu
    - **Dynamická (automatická) alokace na zásobníku**
      - Lokální proměnné, parametry funkcí
      - Automaticky při volání funkcí
      - Při skončení funkce se automaticky uvolní
    - **Dynamická alokace na hromadě**
      - Na hromadě nelze přímo vytvořit pojmenovanou proměnnou – nutno použít ukazatel
      - Programátor je zodpovědný za správnou alokaci i uvolnění (dealokaci) paměti
      - Jazyk C nemá syntaktické prostředky pro práci s hromadou.
      - Přidělování paměti provádí funkce malloc z rozhraní <stdlib.h>
      - ```
        void *malloc(size_t size);
        ```
      - Size – počet alokovaných **bajtů**
      - Vrací obecný ukazatel nebo NULL, pokud došlo k chybě (např. Není dostatek paměti)
      - Nutné používat spolu se sizeof! - na všech platformách alokuje správně velkou paměť
        - ```
          int *pi = malloc(sizeof(int));
          ```
      - Vždy je třeba otestovat návratovou hodnotu.
        - ```
          if ((pi = malloc(sizeof(int))) == NULL)
          { // zpracuj chybu
            return ERR_MALLOC;
          }
          ```
      - **Nikdy nesmíme ztratit ukazatel na alokovanou paměť!**
      - Ztráta ukazatele – memory leak – ztracenou paměť nelze získat zpět!

        Příklad: ztráta ukazatele na alokovanou paměť.

        - ```
          int *pi = malloc(sizeof(int));
          ...
          pi = pj; // ztráta původního ukazatele
          ```



- S pamětí je nutno dobře hospodařit - nepoužívanou paměť uvolnit – free()
  - ```
    void free(void *ptr);
    ```
  - Platí pravidlo: **Ke každému volání funkce malloc() patří jedno volání funkce free().**
  - ```
    int *pi = malloc(sizeof(int));
    ...
    free(pi);
    ```

- **Datový typ pole**
  - **Složený (agregovaný) typ**
    - Označuje skupinu hodnot
    - Pole, struktura, union
  - **Pole**
    - Kolekce hodnot (prvků) stejného typu
    - V C – spojitá oblast paměti
  - **Prvek pole**
    - Přístup pomocí identifikátoru pole a indexu
  - **Pro deklaraci jednorozměrného pole se používá obecný formát:**
    - typ_pole jméno_pole[velikost];
  - Prvek pole lze získat **indexováním**.
  - V C všechna pole začínají indexem 0
    - První prvek: mojePole[0]
  - **Jazyk C nekontroluje meze polí!**
    - Musí ohlídat programátor
    - Indexace mimo meze pole – zhroucení programu
    - nebo poškození vnitřních dat – podstata bezp. chyby buffer-overflow!

      Příklad: načtení celého čísla za hranicí pole.

      ```
      int count[5];
      scanf("%d", &count[9]);
      // nelze!, překladač to nehlídá
      ```
  - V C nelze přiřazovat pole – neexistuje operátor přiřazení pro pole
    - Neplést s přiřazováním ukazatelů

      Příklad: nelze přiřadit najednou celé pole jinému poli.

      ```
      char prvni[10], druhe[10];
      .
      .   // naplnění prvků pole prvni hodnotami
      .
      druhe = prvni; // nelze
      ```

- **Použití řetězců**
  - Textový řetězec = pole typu char
  - V C – řetězec ukončen znakem '\0'
  - Pole musí být minimálně o znak delší než řetězec.
  - Řetězcové konstanty jsou ukončeny nulou automaticky
  -

0	1	2	3	4
'A'	'd'	'a'	'm'	'\0'

Příklad: čte řetězec zadaný z klávesnice. Pak vypíše obsah řetězce po znacích.

```
char str[N];
printf("Zadejte řetězec znaků:\n");
fgets(str, N, stdin);// nikdy ne gets – hrozí přetečení !!!
                                                // [HePa13, str. 206]
int i = 0;
while(str[i] != '\0')
{
  printf("%c", str[i]);
  i++;
}
```

- **Vícerozměrná pole**
  - **Dvourozměrné pole**
    - **Pole jednorozměrných polí**

Příklad: deklarace dvourozměrného pole

	0	1	2	3	4
0					
1					
2					
3					

```
#define RADKY 4
#define SLOUPCE 5
.
.
float matice[RADKY][SLOUPCE];
```

- **Inicializace pole**
  - Obecný formát inicializace polí pomocí inicializátoru:
    - **typ jméno_pole[velikost]={seznam-hodnot};**
  - seznam hodnot – typově kompatibilní konstanty oddělené čárkami

Příklad: pětiprvkové celočíselné pole inicializováno mocninami čísel 1 až 5.

```
int pole[5] = {1, 4, 9, 16, 25};
```

  - Textové řetězce lze inicializovat jednodušeji – překladač doplní znak '\0'

Příklad: pětiprvkové pole, do kterého je uložen řetězec.

```
char name1[5] = "Adam";
char name2[]  = "Adam";
```

0	1	2	3	4
'A'	'd'	'a'	'm'	'\0'

- **Vícerozměrná pole** – závorkovat
  - jinak překladač vypíše varování

```
int mesice[4][3] =
      {{1,2,3},{4,5,6},{7,8,9},{10,11,12}};

int mesice[4][3] =
{
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
  {10, 11, 12}
};
```