5 Typ ukazatel, strukturované datové typy

Kreslíková, 26. 9. 2023

5.1 Úvod

V předchozích kapitolách jsme si ukázali přehled jednoduchých datových typů, které slouží především pro práci s číselnými hodnotami. V této kapitole se podíváme na datové typy, které nám umožní řešit abstraktnější a složitější problémy mnohem elegantněji, než by to bylo možné pomocí našich dosavadních znalostí.

V prvé řadě se podíváme na datový typ ukazatel. Ačkoli jde principiálně stále o jednoduchý číselný datový typ, jeho význam sahá mnohem dále. Použití tohoto typu umožňuje pracovat efektivněji s pamětí a využívat jí větší množství než bez něj. Hraje také klíčovou úlohu při tvorbě dynamických datových struktur, o kterých budeme mluvit v dalších kapitolách.

Tento datový typ je poměrně blízko hardwaru počítače a úzce souvisí s jeho fungováním. O tom si povíme v části věnované alokaci paměti.

Dále se v této kapitole seznámíme s datovým typem pole, který má v jazyce C k typu ukazatel velmi blízký vztah.

Nakonec se podíváme na to, jaký vliv má typ ukazatel na funkce v jazyce C, protože i když v jiných programovacích jazycích tento vliv nemusí být na první pohled patrný, princip fungování podprogramů je všude stejný.

Některé vyšší programovací jazyky se typ ukazatel snaží před programátory skrývat (např. Java, Python) právě kvůli jeho principiální blízkosti hardwaru počítače. Skutečnost je ale taková, že vnitřně je tyto jazyky používají téměř u všech proměnných. Znalost ukazatelů prostřednictvím jazyka C vám bude velmi užitečná i při programování v těchto jazycích.

5.2 Datový typ ukazatel

Ukazatel je proměnná, která obsahuje paměťovou adresu. Jeho hodnota říká, kde je uložen nějaký objekt. Součástí deklarace ukazatele je i informace o typu dat, která jsou na získané adrese očekávána. Typ ukazatele může být odvozen z libovolného typu datového objektu (**int, float, char, …**), funkčního typu a z neúplného typu [HePa13, str. 144] .

Obecný formát definice proměnné typu ukazatel:

*typ *jméno_proměnné;*

Příklad: Definice proměnné typu ukazatel.

```
int *integer_ptr;    // integer_ptr je ukazatel na int
```

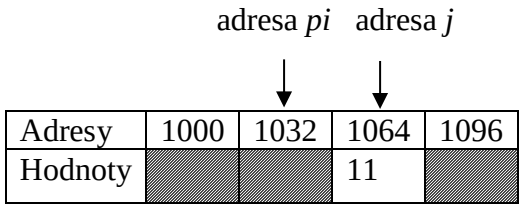
Hodnota *integer_ptr* představuje adresu paměti a **integer_ptr* je místo v paměti o rozměru datového typu *int (sizeof(int))*, na které lze uložit celočíselnou hodnotu, a odkud lze tuto hodnotu získat. Často používáme pojem ***dereference ukazatele***.

Jazyk C má dva speciální ukazatelové operátory: * (dereferenční – vrací hodnotu uloženou na adrese operandu) a & (referenční – vrací adresu proměnné operandu).
Když je hodnota proměnné získána přes ukazatel, říkáme, že je získána ***odkazem*** (dereferencí, nepřímo).
Následující hypotetický příklad pracuje s adresou proměnné *j* , kam zapíše hodnotu 123:

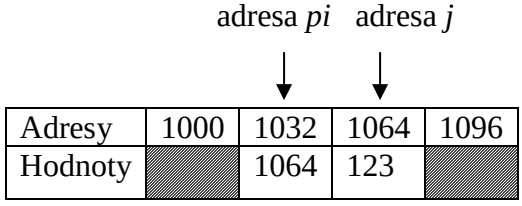
Příklad: Reference a dereference ukazatele.

```
int *pi;
int j = 11;

pi = &j;   // získání adresy proměnné j
*pi = 123; // dereference ukazatele pi – zapsání/získání
           // odkazované hodnoty
```

Situace po inicializaci *j*. Obsah ostatních okolních adres je nedefinovaný:



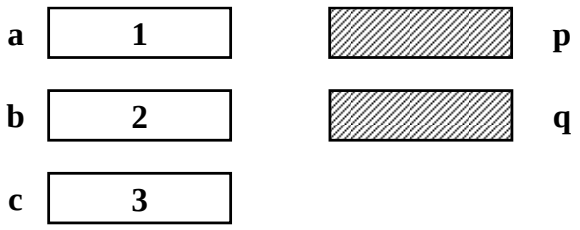
Situace po přiřazení pomocí ukazatele (*pi = &j*)



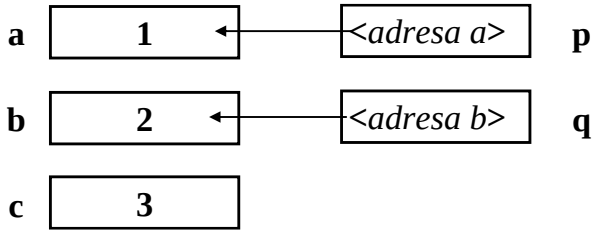
Po definici proměnné *j* typu **int**, a ukazatele *pi* na typ **int** v prvním řádku následuje získání adresy proměnné *j*. Na tuto adresu se odkazuje ukazatel *pi*. Jeho dereferencí na levé straně příkazu přiřazení uložíme příslušnou hodnotu na paměťové místo, na které ukazuje.

Příklad: dva ukazatele mohou ukazovat na stejné místo.

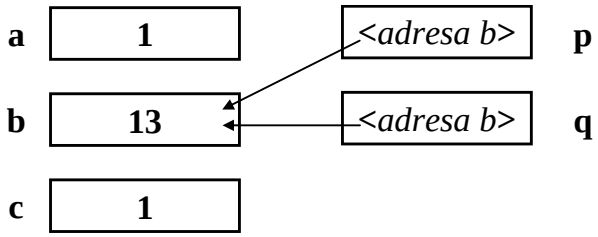
```
int a = 1;
int b = 2;
int c = 3;
int *p;
int *q;
```




```
p = &a; // reference a
q = &b; // reference b
```




```
c = *p;   // dereference
p = q;    // přiřazení ukazatele
*p = 13;  // přepíše proměnnou b hodnotou 13
```



Definice ukazatele s inicializací

Příklad: Definice ukazatele s inicializací.

```
int i = 7;
int *pi = &i;
```

Nulový ukazatel

V <*stdio.h*> je definována konstanta *NULL*, jejíž význam je nulový (prázdný) ukazatel. Tuto hodnotu můžeme přiřadit každému ukazateli, bez ohledu na datový typ, na nějž ukazuje. Konstanta *NULL* se používá k označení, že ukazatel neukazuje nikam. Pokud má ukazatel hodnotu *NULL,* je zřejmé, že neukazuje nikam, což lze snadno otestovat. Používání konstanty *NULL* je tedy doporučeno vždy, když ukazatel neukazuje na nějaký objekt v paměti.

Příklad: Definice ukazatele a jeho inicializace na *NULL*.

```
int *pi = NULL;
```

```
// přiřazení s otestováním, zda ukazatel někam ukazuje
if (pi != NULL)
  *pi = 20;
```

Inicializace ukazatele

Obvyklou chybou začátečníků je použití ukazatele bez jeho předchozí inicializace (nebo alokace paměťového místa). Neinicializovaný ukazatel může ukazovat na kritickou oblast paměti a jeho použití může vést (v operačních systémech jako je MS-DOS) i k havárii systému. Takto tedy ne:

Příklad: Neinicializovaný ukazatel.

```
float *px, *py;

*px = 12.5;  // zápis do nedefinovaného místa paměti
*py = 54.1;
```

Příklad: Nekompatibilita typů.

```
int q;
double *fp;

fp = &q;
// v tomto případě překladač vypisuje varování

*fp = 100.23; //  přepíše paměť sousedící s q
```

Překladač jazyka C používá základní (bázový) typ ukazatele pro určení, jak velký je objekt, na který ukazatel ukazuje. Tak překladač například pozná, kolik bajtů se má kopírovat při nepřímém přiřazení a kolik bajtů se má porovnávat, když provádí nepřímé porovnání. Proto je velmi důležité používat vždy správný základní typ. S výjimkou zvláštních případů není vhodné používat ukazatel jednoho typu pro odkaz na objekt jiného typu.

Konstantní ukazatel, ukazatel na konstantu

V jazyce C lze vytvořit ***konstantní ukazatel***. Takový ukazatel pak nelze modifikovat. Lze ale modifikovat hodnotu paměti, na kterou tento ukazatel odkazuje. Naopak, když vytvoříme ***ukazatel na konstantu***, bude možné měnit samotný ukazatel (adresa pak může ukazovat na jinou konstantu), ale nebude možné modifikovat odkazovanou hodnotu.

Příklad: Konstantní ukazatel, ukazatel na konstantu.

```
int i;
int *pi;    // pi je neinicializovaný ukazatel na typ int
int * const CP = &i; // konstantní ukazatel na int
const int CI = 7;    // celočíselná konstanta

// neinicializovaný ukazatel na celočíselnou konstantu
const int *pci;

// konstantní ukazatel na konstantu
const int * const CPC = &CI;
```

Obecný ukazatel

Ukazatele, o kterých jsme dosud pojednávali, byly spojené s nějakým konkrétním typem. Tato skutečnost je přínosem, neboť umožňuje typovou kontrolu. Jsou však okamžiky, kdy potřebujeme ukazovat ukazatelem do paměti a nemáme na mysli konkrétní datový typ. Norma pro takovou situaci zavádí prázdný fiktivní typ **void** a ukazatel na něj **void ***.

Datový typ **void *** lze přetypovat na jakýkoli typový ukazatel. Pro praktické použití (např. pro zápis) je potřeba obecný ukazatel vždy přetypovat na ukazatel konkrétního typu. Ukazatel typu **void *** lze přiřadit jakémukoliv ukazateli.

Příklad: Datový typ **void** a přetypování ukazatele.

```
int inum = 10;
float fnum = 3.14;
void * pgeneric = &inum;

*(int *)pgeneric = 20; //přetypování je nutné, inum je nyní 20
pgeneric = &fnum;
*(float *)pgeneric = 2.72; //nastaví fnum na 2.72
```

Konverze ukazatelů
Pokud je to potřeba, lze datový typ ukazatel přetypovat na jiný typ ukazatele pomocí operátoru přetypování. Toho se používá zejména, pokud pracujeme s obecným ukazatelem (viz předchozí příklad). Stejně jako u přetypování základních datových typů jde ovšem o potenciálně nebezpečnou konstrukci. Programátor bere na sebe veškerá rizika, která z toho vyplývají (např. při chybném přetypování).

Adresová aritmetika
Pro datový typ ukazatel jsou v jazyce C definovány aditivní operátory *(+, -, ++, --)* pro sčítání a odečítání ukazatelů a celých čísel a relační operátory pro ukazatele na stejný typ *(<, <=, >, >=, ==, !=)*. Aditivní operátory nemodifikují hodnotu adresy o celočíselný operand, ale o příslušný násobek velikosti bázového typu (+ dále se to komplikuje zarovnáváním paměti).

V tomto předmětu nebudeme tuto schopnost jazyka C využívat. Více se o ní dozvíte v předmětu Jazyk C nebo v doporučené literatuře.

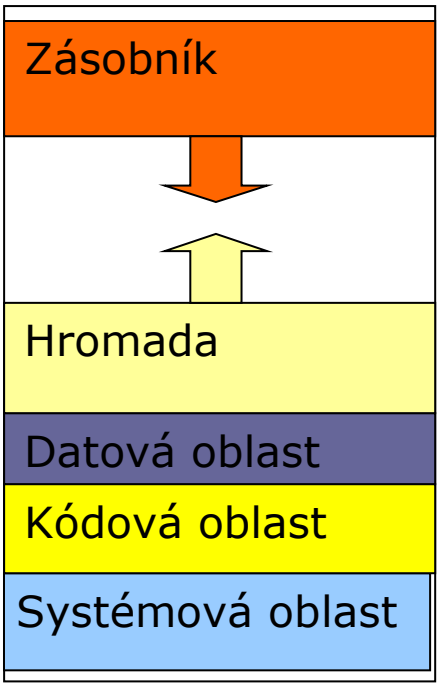
Vícenásobný nepřímý odkaz
Této možnosti se využívá hlavně u funkcí pro předávání ukazatelů odkazem a při vytváření složitějších dynamických datových struktur. Později uvidíme, jak se vícenásobný ukazatel používá pro vytváření dynamických vícerozměrných polí.

Příklad: Ukazatel může ukazovat i na jiný ukazatel.

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

5.3 Alokace paměti

Všechny proměnné jsou uloženy v paměti, kde zabírají prostor úměrný velikosti svého datového typu *(sizeof(typ)* + zarovnávání). Využitelné paměti počítače také říkáme ***paměťový prostor***. Podle typu dat rozdělujeme paměťový prostor na kódovou oblast, oblast statických dat (globální proměnné a konstanty), hromadu (*heap* – dynamické proměnné) a zásobník (*stack* – lokální proměnné a parametry funkcí).



Obrázek 1: Typické rozdělení paměťového prostoru programu.

Operaci, která vyhradí v paměti místo pro proměnnou pak říkáme ***alokace paměti***. Podle způsobu a místa rozdělujeme alokaci na statickou a dynamickou.

Statická alokace
Při této alokaci je nutné znát už v době překladu velikost dat, která budou takto alokována. Data jsou pak alokována v datové oblasti programu. Skutečná alokace paměti se provádí během zavádění programu do paměti.

Všechny definice globálních proměnných, které jsme doposud uskutečnili, alokovaly paměť tímto způsobem.

Dynamická alokace
Tento způsob alokace umožňuje teoreticky využít veškerou dostupnou paměť počítače. Při této alokaci se využívají prostory paměti zvané ***zásobník*** a ***hromada***.

Dynamická alokace na zásobníku
Do prostoru zásobníku se ukládají veškeré lokální proměnné a parametry funkcí. Při každém zavolání funkce se automaticky vyhradí prostor pro lokální proměnné a parametry, takže programátor se o to nemusí vůbec starat. Tato paměť se automaticky uvolní v okamžiku ukončení funkce.

Všechny lokální proměnné a parametry, které jsme doposud používali, byly alokovány na zásobníku.

Dynamická alokace na hromadě
Paměť se přiděluje z oblasti, které se říká hromada (*heap*). Do této paměti neexistuje možnost přímého přístupu pomocí proměnné, proto musíme použít ukazatele. Tuto paměť jsme prozatím nepoužívali.

Všechny předchozí způsoby alokace paměti má programátor minimální možnost ovlivnit. Alokace i uvolnění paměti se dělo automaticky. Naopak tento způsob alokace je výhradně v režii programátora, který musí zajistit jak správnou velikost alokovaných dat, tak jejich správné uvolnění z paměti.

Alokace na hromadě v jazyce C
Jazyk C neposkytuje sám o sobě žádné syntaktické prostředky pro alokaci paměti na hromadě. Namísto toho je ve standardní knihovně jazyka C (rozhraní <*stdlib.h*>) funkce *malloc*, která má přidělování této paměti na starosti.

```
void *malloc(size_t size);
```

Tato funkce má jediný parametr – *size* (kompatibilní s **int**), kterým se specifikuje počet bajtů, které se mají alokovat na hromadě. Funkce vrací ukazatel na takto alokovaná data nebo *NULL,* pokud došlo k chybě, například při nedostatku volné paměti. Funkce *malloc* vrací obecný ukazatel, který je kompatibilní se všemi typovými ukazateli, takže jej lze bez problémů přiřadit.

Protože datové typy v jazyce C nemají normou určenou velikost, v zájmu přenositelnosti se funkce *malloc* používá spolu s operátorem *sizeof*.

Příklad: Na všech platformách alokuje správně velkou paměť.
```
int *pi = malloc(sizeof(int));
```

Pokud při alokaci dojde k chybě, funkce *malloc* vrací hodnotu *NULL*. Tato hodnota může signalizovat například nedostatek paměti a slušný program by měl tento stav detekovat, aby mohl buďto nějakou paměť uvolnit, nebo aby se pokusil program ukončit (a zavřít otevřené soubory a uvolnit jiné alokované zdroje).

Příklad: Alokace paměti s testováním úspěšnosti.
```
if ((pi = malloc(sizeof(int))) == NULL)
{
  zpracujChybu(ERR_MALLOC);
}
```

Poznámka:
U jednodušších projektů se na toto někdy rezignuje, protože dokonalé uvolnění všech zdrojů v tomto případě vyžaduje v jazyce C poměrně velké úsilí a nárůst složitosti kódu. Vy toto ale ošetřujte vždy! Problémy, které laxní přístup způsobí, jsou vždy horší než práce navíc s ošetřením. V jazycích s výjimkami se tento stav ošetřuje mnohem jednodušeji.

Nikdy nesmíme ztratit ukazatel na takto alokovanou paměť, protože bychom ji nemohli uvolnit a zbytečně by zabírala místo v paměti. Takto ztracenou paměť by nemohl použít ani náš ani jiný program. U dlouho běžících programů mohou takovéto paměťové úniky (memory leaks) vést až k vyčerpání veškeré dostupné paměti a havárii systému.

Příklad: Ztráta ukazatele na alokovanou paměť.
```
int *pi = malloc(sizeof(int));
...
pi = pj; // ztráta původního ukazatele
```

Uvolnění paměti
Protože v počítači není k dispozici nekonečně velká paměť, je nutné s ní rozumně hospodařit. V okamžiku, kdy program dynamicky alokovaná data nepotřebuje, měl by paměť uvolnit pro další použití. K tomu slouží funkce *free*.

```
void free(void *ptr);
```

Této funkci není potřeba předávat údaj o velikosti alokovaných dat, protože ten je obsažen už v těchto datech (typicky v prostoru před skutečným ukazatelem, který vrátí funkce *malloc*).

Příklad: Alokace a uvolnění paměti.

```
int *pi = malloc(sizeof(int));
...
free(pi);
```

Platí pravidlo: ***Ke každému volání funkce malloc patří jedno volání funkce free.***

Rozumný OS po úspěšném skončení programu uvolní i celou hromadu.

5.4 Pole

Každá proměnná, kterou jsme dosud v našich programech používali, nabývala v každém okamžiku pouze jedné hodnoty určitého typu. V této kapitole začínáme studovat tzv. složené (agregované) datové typy. Proměnné složených typů označují zpravidla skupinu určitých hodnot. Do této třídy patří pole, struktury a uniony. Pole je kolekce proměnných stejného typu, které mohou být označovány společným jménem. K prvkům pole přistupujeme prostřednictvím identifikátoru pole a indexu. Pole v jazyce C obsazuje spojitou oblast operační paměti tak, že první prvek pole je umístěn na nejnižší přidělené adrese a poslední na nejvyšší přidělené adrese [HePa13, str. 173].

Jednotlivá proměnná v poli se nazývá ***prvek pole***. Pole umožňují pohodlně zpracovávat skupinu dat stejného typu.

Pro deklaraci jednorozměrného pole se používá obecný formát:

*typ_pole jméno_pole**[**velikost**]***

typ_pole určuje typ prvků pole (bázový typ).
jméno_pole představuje identifikátor pole.
velikost je kladný počet prvků pole.

Příklad: Deklarace pole o deseti prvcích typu **int**.

```
int mojePole[10];
```

indexy prvků pole *mojePole*

0 1 2 3 4 5 6 7 8 9

Prvek pole se získá ***indexováním*** pole pomocí čísla prvku. V jazyce C začínají všechna pole nulovým indexem. To znamená, že pokud chceme pracovat s prvním prvkem pole, zadáme jako index 0.

Obecný formát přístupu k prvku pole

*jméno_pole**[**výraz**]***

```
mojePole[0]  = 100;
//  odkazuje na první prvek pole mojePole

mojePole[1]  = 5  // odkazuje na druhý prvek pole mojePole
```

Jazyk C ukládá jednorozměrná pole do souvislé oblasti paměti. První prvek pole má nejnižší adresu.

Příklad: Naplní prvky pole hodnotou indexu.

```
int pole[5];

for(int i=0; i<5; i++)
  pole[i] = i;
```

Výsledek:

pole[0]	pole[1]	pole[2]	pole[3]	pole[4]
0	1	2	3	4

Hodnotu prvku pole lze použít všude tam, kde lze použít obyčejnou proměnnou nebo konstantu odpovídajícího typu. Prvek pole je L-hodnota.

Příklad: Naplní pole *sqrs* druhými mocninami čísel od 1 do 10 a pak je vypíše.

```
#include <stdio.h>
#define N 10

int main(void)
{
  int sqrs[N];

  for(int i=1; i<=N; i++)
    sqrs[i-1] = i*i;

  for(int i=0; i<N; i++)
    printf("%d ", sqrs[i]);

  return 0;
}
```

Meze polí

Jazyk C nekontroluje žádné meze indexů pole. Je-li např. deklarováno pole s pěti prvky, překladač přesto umožní přístup k (neexistujícímu) desátému prvku příkazem *a[9] = 5;* . Pokus o práci s neexistujícími prvky má ovšem obvykle katastrofální následky a způsobuje často zhroucení programu. Záleží jen na programátorovi, aby zajistil, že hranice pole nebudou nikdy překročeny. (Podstata bezpečnostní chyby *buffer-overflow*.)

Příklad: Načtení celého čísla za hranicí pole.

```
int count[5];
scanf("%d", &count[9]);  // nelze, překladač to nehlídá
```

Přiřazení, kopie pole
V jazyce C nelze přiřadit jedno celé pole jinému poli. Jazyk C neposkytuje operátor pro přiřazení polí.

Příklad: Nelze přiřadit jedno celé pole jinému poli.
```
char a1[10], a2[10];
.
.   // naplnění prvků pole a1 hodnotami
.
a2 = a1; // nelze
```

Chceme-li zkopírovat hodnoty všech prvků jednoho pole do jiného pole, pak to musíme udělat samostatným kopírováním jednotlivých prvků (nebo kopií paměti – *memcpy*).

Příklad: Naplní pole *a1* čísly *1* až *10* a pak je zkopíruje do pole *a2* a zobrazí.
```
#include <stdio.h>
#define POCET 10
int main(void)
{
  int a1[POCET], a2[POCET];
  for(int i=1; i<=POCET; i++)
    a1[i-1] = i;
  for(int i=0; i< POCET; i++)
    a2[i] = a1[i];
  for(int i=0; i< POCET; i++)
    printf("%d ", a2[i]);

  return 0;
}
```

Použití řetězců
Jednorozměrné pole se často používá pro ukládání řetězců. Jazyk C nemá zabudovaný datový typ řetězec. Místo toho je řetězec definován jako ***pole znaků, ukončené nulovým znakem ('\0').***

Skutečnost, že řetězec musí být ukončen nulovým znakem, znamená, že musíme nadefinovat pole, do kterého se vejde řetězec o jeden bajt delší, než je požadovaná délka řetězce, aby zbylo místo pro nulový znak. Řetězcová konstanta je ukončena nulovým znakem automaticky překladačem.

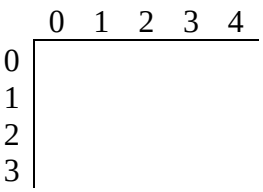
Příklad: Čte řetězec zadaný z klávesnice. Pak vypíše obsah řetězce po znacích
```
#include <stdio.h>
#define N 80
int main(void)
{
  char str[N];
  printf("Zadejte řetězec (méně než %d znaků): \n", N-1);
  fgets(str, N, stdin); // nikdy ne gets, je to nebezpečné!
  int i = 0;
  while(str[i] != '\0')
  {
    printf("%c", str[i]);
    i++;
  }

  return 0;
}
```

5.5 Vícerozměrná pole

Kromě jednorozměrného pole můžeme vytvářet pole se dvěma nebo více rozměry. Pro přidání dalšího rozměru stačí zadat jeho velikost v hranatých závorkách. ***Dvourozměrné*** pole je v podstatě pole jednorozměrných polí.

Příklad: Deklarace dvourozměrného pole.

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



Příklad: Zaplní pole součiny indexů a pak zobrazí pole po řádcích.

```
#include <stdio.h>
#define RADKY 4
#define SLOUPCE 5

int main(void)
{
  int matice[RADKY][SLOUPCE];

  for(int r=0; r<RADKY ; r++)
    for(int s=0; s<SLOUPCE; s++)
      matice[r][s] = r*s;

  for(int r=0; r<RADKY; r++)
  {
    for(int s=0; s<SLOUPCE; s++)
    {
      printf("%d ", matice[r][s]);
    }
    printf("\n");
  }

  return 0;
}
```

Inicializace pole

Prvkům pole lze přiřadit počáteční hodnoty. Inicializace se provádí zadáním seznamu hodnot, které mají prvky pole obsahovat.

Obecný formát inicializace pole:

*typ jméno_pole**[**velikost**]**=**{**seznam-hodnot**};***

Seznam-hodnot je seznam konstant typově kompatibilních se základním typem pole, oddělených čárkami. Seznam se zpracovává zleva doprava, tj. první konstanta bude umístěna do první pozice pole, druhá do druhé atd.

Příklad: Celočíselné pole o pěti prvcích inicializováno mocninami čísel 1 až 5.
```
int pole[5] = {1, 4, 9, 16, 25};
```

Příklad: Znakové pole o třech prvcích inicializováno znaky *'A' 'B' 'C'*.
```
char a[3] = {'A', 'B', 'C'};
```

Má-li pole znaků obsahovat řetězec může být inicializováno pomocí řetězce uzavřeného v uvozovkách. Jelikož řetězec v jazyce C musí končit nulovým znakem, musíme zajistit, aby bylo deklarované pole dostatečně dlouhé, aby se do něj tento nulový znak vešel. Když je použita řetězcová konstanta, přidá překladač nulový znak na konec automaticky.

Příklad: Pole o pěti prvcích, do kterého je uložen řetězec.
```
char name[5] = "Adam";
```

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

Vícerozměrná pole se inicializují podobně jako jednorozměrná. Hodnoty prvků se uvádějí v pořadí podle řádků. Musíme si ale uvědomit, že jde o pole polí, takže musíme jednotlivé prvky správně uzávorkovat. Pokud bychom vnitřní pole neuzávorkovali, překladač by ohlásil varování (ale prvky by vložil ve správném pořadí).

Příklad: Inicializace vícerozměrného pole.
```
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}
};
```

Není nutné vyjmenovat všechny prvky pole.

Příklad: Nulové prvky v inicializaci není nutno uvádět.
```
#define N 5

int pole2[N] = {5};

int pole3[N] = {1, [3]=2}; // pole3[3] má hodnotu 2

// neuvedené prvky pole mají hodnotu 0
```

Pole s neurčenou velikostí
Pokud se inicializuje jednorozměrné pole, není nutné specifikovat jeho velikost. Pokud velikost není zadána, překladač spočítá počet inicializačních hodnot a použije tento počet jako

velikost pole. Pole, jejichž velikost není explicitně dána, se nazývají pole s ***neurčenou velikostí.***

Příklad: Pole o osmi prvcích inicializované hodnotami mocniny čísla *2*.
```
int pwr[] = {1, 2, 4, 8, 16, 32, 64, 128};
```

Příklad: Pole s neurčenou velikostí pro uložení textu výzvy.
```
char prompt[] = "Zadejte svoje jmeno: ";
```

U vícerozměrných polí je nutné určit všechny rozměry mimo nejlevějšího. (Přemýšlejte proč tomu tak je. V kontextu předchozích informací na to není těžké přijít.)

Příklad: Vícerozměrné pole s neurčenou velikostí.
```
int mesice[][3] =
{
  {1, 2, 3},
  {4, 5, 6},
  {7, 8, 9},
 {10, 11, 12}
};
```

Globální, lokální deklarace pole a inicializace
Pokud je pole deklarováno na ***globální*** úrovni, lze pro specifikaci velikosti použít výhradně konstanty zapsané pomocí ***literálů*** nebo definované pomocí klauzule *#define*.
U polí deklarovaných ***lokálně*** lze velikost specifikovat pomocí ***celočíselného výrazu*** (ISO C99). V tomto případě bude pole alokováno na zásobníku a stejně jako ostatní lokální proměnné zanikne v okamžiku ukončení funkce, ve které je deklarováno.

Pole, které je deklarováno s konstantním rozměrem (nezáleží, zda je globální, či lokální), může být inicializováno pomocí inicializátoru (ISO C99). Neinicializované pole bude mít nedefinované hodnoty prvků.

Příklad: Inicializované pole, velikost pole je určena pomocí *#define*
```
#define N 5
int pole1[N] = {3, 4, 1, 0, 2}; //prvky lze vyjmenovat
```

Pole s rozměrem specifikovaným proměnnou (lze pouze ve funkcích) nelze výše uvedeným způsobem inicializovat. Inicializace se v tomto případě provádí pomocí cyklu (nebo pomocí *memcpy*).

Příklad: Pole inicializované pomocí cyklu.
```
void funkce(int n)
{
  int pole[n]; //nelze použít inicializátor
  for (int i = 0; i < n; i++)
    pole[n] = 0;
...
}
```

Alokace polí (uložení vícerozměrných polí v paměti)
Dvourozměrné staticky alokované pole je uloženo v paměti po řádcích v jediném souvislém bloku paměti.

Příklad: Dvourozměrné staticky alokované pole.
```
int x[2][3]
// alokuje v paměti 2*3 prvků, např. 2 * 3 * sizeof (int) bajtů
```

Předpokládejme, že počáteční adresa pole je 100, pak je obsazení paměti:

100	112	124
řádka č. 0	*řádka č. 1*	*první volný bajt paměti*

detailněji:

100	104	108	112	116	120	124
x[0][0]	*x[0][1]*	*x[0][2]*	*x[1][0]*	*x[1][1]*	*x[1][2]*	*volno*

Alokace dynamického pole
Často potřebujeme pracovat s poli, u nichž neznáme velikost v době překladu. V takovém případě musíme použít dynamickou alokaci.

Dynamické pole alokované na zásobníku (ISO C99)
V některých situacích si vystačíme s polem, které bude deklarováno jako lokální proměnná. V tom případě můžeme použít stejnou syntaxi, jako v případě statické alokace. Místo konstanty při specifikaci rozměru můžeme použít jakýkoliv celočíselný výraz.

Příklad: Alokace pole na zásobníku.
```
void praceNaPoli(int delka)
{
  int pole[delka][delka+2];
  ...
}
```
V tomto případě bude pole alokováno na zásobníku a v okamžiku ukončení funkce automaticky zanikne.

Poznámka: Z praktického hlediska nelze tuto konstrukci doporučit, protože moderní operační systémy ve snaze minimalizovat škody, které může způsobit chybný program (například s chybnou rekurzivní funkcí), podstatně omezují velikost paměti využívané jako zásobník. Zatímco pomocí hromady lze využít veškerou dostupnou paměť, pomocí zásobníku toho dosáhnout nelze. Dalším problémem je, že neexistují prostředky pro zjištění, zda je pro zamýšlenou alokaci na zásobníku dostatek místa. Na hromadě tuto informaci získat lze, a tak se program s případným nedostatkem paměti může vyrovnat ještě dříve než způsobí systémovou chybu.

Přesto tuto konstrukci uvádíme, protože je užitečné ji znát. Hodí se například pro rychlé prototypování, kdy potřebujeme vyzkoušet funkčnost nějakého algoritmu a nechceme se zdržovat ošetřováním výjimečných stavů. Při použití v reálné aplikaci je potřeba být velmi obezřetný a raději alokovat dynamické pole na hromadě.

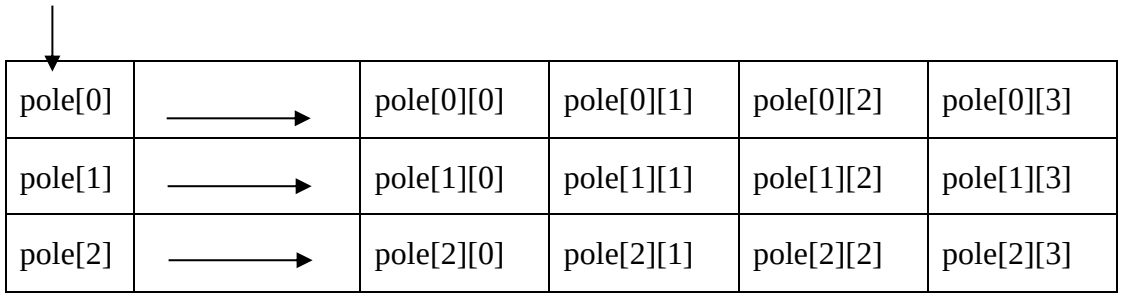
Dynamické pole alokované na hromadě
Pokud potřebujeme, aby dynamicky alokované pole přetrvalo v paměti i po ukončení funkce, musíme použít dynamickou alokaci na hromadě. V tomto případě ovšem nemůžeme proměnnou deklarovat jako datový typ pole. Zde využijeme příbuznosti mezi poli a ukazateli a k alokovaným datům budeme přistupovat přes ukazatel. Jak už víme, s ukazateli můžeme pracovat pomocí operátoru indexování jako s polem.

Příklad: Alokace pole na hromadě.
```
int *pole = malloc(delka * sizeof(int));
```

Takto získané pole bude mít nedefinovanou hodnotu svých prvků a bude nutné jej inicializovat pomocí cyklu nebo pomocí funkce *memset*.

Dynamická, vícerozměrná pole
Pokud chceme dynamicky vytvořit na hromadě vícerozměrné pole, musíme to udělat pomocí ukazatelů. Budeme uvažovat dvourozměrné pole. Nejprve si uvědomíme, že dvourozměrné pole je vlastně jednorozměrné pole řádků (opět jednorozměrných polí). Když to převedeme na ukazatele, tak zjistíme, že musíme vyrobit pole ukazatelů na jednorozměrná pole. Pole ukazatelů vyrobíme lehce pomocí funkce *malloc*. Jednotlivé řádky ovšem musíme alokovat pomocí cyklu.

int **pole



Příklad: Alokace dvourozměrného pole.
```
// nejprve pole ukazatelů na int
int **pole = malloc(RADKU*sizeof(int *));
if (pole == NULL) error(E_NOMEM);
// potom jednotlivé řádky
for (int i = 0; i < RADKU; i++)
{
  pole[i] = malloc(SLOUPCU*sizeof(int));
  if (pole[i] == NULL) error(E_NOMEM);

}
// nyní lze pole používat běžným způsobem
pole[1][2] = 10;
```

Příklad: Takto to nelze, protože překladač nemá informaci o rozměrech pole.
```
int **pole = malloc(RADKU*SLOUPCU*sizeof(int));
pole[1][2] = 10; // chyba za běhu!!
```

Při uvolnění pole z paměti je potřeba postupovat opačně – nejprve zavolat *free* na všechny řádky a až poté uvolnit pole ukazatelů na **int**.

5.6 Použití ukazatelů pro práci s poli

Samotný identifikátor pole má význam konstantního ukazatele na první prvek. Jelikož je jméno pole bez indexu ukazatelem na začátek pole, můžeme pracovat s prvky pole pomocí ukazatelové aritmetiky. Poznamenejme však, že použití operátoru indexace je v tomto případě programátorsky čistší konstrukce.

Příklad: Zobrazení prvních třech prvků pole pomocí ukazatelové aritmetiky.

```
int a[10] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
int *p = a; // přiřadí do p adresu začátku pole a

// toto zobrazí první, druhý a třetí prvek
printf("%d %d %d\n", *p, *(p+1), *(p+2));
// zobrazení stejných prvků pomocí pole a
printf("%d %d %d\n", a[0], a[1], a[2]);
```

Pro použití ukazatelové aritmetiky pro přístup k prvkům vícerozměrného pole se musí vypočítat pozice prvku v poli. Pro výběr prvku pomocí indexů pole to překladač dělá automaticky.

Abychom získali požadovaný prvek, musíme vynásobit číslo řádku požadovaného prvku počtem sloupců pole a pak přičíst číslo sloupce požadovaného prvku.

Příklad: Přístup k prvku dvourozměrného pole pomocí ukazatele.

```
float hotovost[4][3];
float *p = *hotovost;
```

Pomocí **(p + (2*3) + 1)* získáme totéž co použitím *hotovost[2][1]*.

	0	**1**	**2**
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11

Příklad: Můžeme indexovat ukazatel, jakoby to bylo pole.

```
char str[] = "Ukazatele jsou bezva\n";
char *p = str;
// cyklus trvá dokud se nenajde znak s kódem 0
int i=0;
while(p[i] != '\0')
  printf("%c", p[i++]);
```

Hodnotu ukazatele tvořeného jménem pole (v našem případě *str*) nelze změnit.

5.7 Vytvoření polí ukazatelů

Ukazatele lze uspořádat do pole stejně jako jiné typy dat.

Příklad: Pole celočíselných ukazatelů, které má 20 prvků.

```
int *pa[20];
```

Adresa celočíselné proměnné nazvané *mojePromenna* se přiřadí devátému prvku pole takto:

```
pa[8] = &mojePromenna;
```

Jelikož je *pa* pole ukazatelů, mohou prvky tohoto pole obsahovat jen adresy celočíselných proměnných. Pro přiřazení hodnoty celočíselné proměnné, na kterou ukazuje třetí prvek pole *pa* je potřeba použít dereferenční operátor:

```
*pa[2] = 100;
```

Příklad: Pole (s neurčenou velikostí) ukazatelů

```
enum errcodes {ERR_INOVR, ERR_OVR,
  ERR_PROFF, ERR_PAPER, ERR_DISKFULL, ERR_DISK};
void error(int err_num)
{
  const char *p[] =
  {
    "Vstup prekrocil pole",
    "Hodnota mimo rozsah",
    "Tiskarna neni zapnuta",
    "Chybi papir",
    "Plny disk",
    "Chyba pri zapisu na disk"
  };

  fprintf(stderr, "%s\n",p[err_num]);
}

...
  error(ERR_PAPER);
```

5.8 Vícenásobný nepřímý odkaz

Ukazatel může ukazovat i na jiný ukazatel.

Příklad: Deklarace ukazatele na ukazatel.

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

Této možnosti se využívá hlavně u funkcí pro předávání ukazatelů odkazem a při vytváření složitějších dynamických datových struktur. Později uvidíme, jak se vícenásobný ukazatel používá pro vytváření dynamických vícerozměrných polí.

5.9 Funkce a předávání argumentů

Funkce se probírají samostatně v kapitole 3., ale protože problematika předávání argumentů funkcím je natolik svázána s problematikou ukazatelů, že se jim budeme věnovat i zde.

5.9.1 Základní způsoby předávání argumentů v jazyce C

Rozlišujeme dva druhy předávání argumentů podprogramům. Jde o ***předávání hodnotou*** a o ***předávání odkazem.*** Jejich použití se liší v tom, co se děje s argumentem při zavolání podprogramu. Některé jazyky (Pascal) umožňují přímo při deklaraci parametru v podprogramu specifikovat pomocí klíčového slova, zda daný parametr bude předáván odkazem. Jazyk C žádné takové klíčové slovo nemá.

Předávání hodnotou

V tomto případě dojde při volání podprogramu ke kopírování hodnoty argumentu do parametru (hodnota se kopíruje na zásobník podprogramu). Parametr podprogramu v tomto případě funguje jako lokální proměnná a jakékoli změny hodnoty tohoto parametru se žádným způsobem neprojeví na hodnotě skutečného argumentu. Při tomto způsobu předávání lze předat podprogramu jako parametr i konstantu.

Předávání odkazem

Při předávání odkazem nedochází ke kopii hodnoty argumentu, ale k propojení formálního parametru podprogramu a skutečného argumentu (do parametru se předá odkaz (ukazatel) na proměnnou tvořící argument). Všechny změny, které podprogram provede s hodnotou parametru se pak projeví i v hodnotě argumentu. Při tomto způsobu předávání lze předat jako argument výhradně L-hodnotu.

Jazyk C nemá žádné klíčové slovo, které by umožňovalo deklarovat parametr předávaný odkazem. Namísto toho se v jazyce C využívají ukazatele. Ukazatele lze předávat funkcím jako argumenty předávané hodnotu. Pomocí dereferenčního operátoru (*) uplatněného na parametr je pak možné ve funkci modifikovat hodnotu argumentu.
Při volání funkce je nutné zajistit, aby byl předán ukazatel. U proměnných jednoduchých typů pak využijeme referenční operátor (&).

Příklad: Záměna dvou proměnných

```
#include <stdio.h>

/** Funkce pro záměnu hodnot dvou ukazatelů. */
void zamen(int *pa, int *pb)
{
  int pom = *pa;
  *pa = *pb;
  *pb = pom;
}

int main(void)
{
  int i = 7, j = 3;
  printf("i == %d, j == %d\n", i, j);

  // pozor,nesmí chybět &
  zamen(&i, &j);  // zavoláme funkci
  printf("i == %d, j == %d\n", i, j);

  return 0;
}
```

Příklad: Parametry předávané odkazem a hodnotou.

```
void proved(int x, int *y)
{
  x = *y;
  *y = 0;
  printf("x == %d, y == %d\n", x, *y); // zobrazí se 1  0

  return;
}
...
int a = 1, b = 2;
proved (b,&a);
printf("a == %d, b == %d\n", a, b);  // zobrazí se 0  2
```

Příklad: Načte zadaný počet čísel ze *stdin* a uloží je do pole.

```
#include <stdlib.h>
#include <stdio.h>
#define N 10
void nactiPole(int pole[], int delka)
{
  int i = 0;
  while(i < delka && scanf("%d", &pole[i]) == 1)
    i++;
  if(i != delka)
    chyba();
}
...
int *pole = malloc(N*sizeof(int));
nactiPole(pole, N);
...
free(pole);
```

5.9.2 Předávání polí

Datový typ pole je, jak už víme, příbuzný s datovým typem ukazatel. Při předávání polí do funkce se v jazyce C vždy předává ukazatel. Nikdy nedojde ke kopírování celého pole na zásobník podprogramu, i když to syntakticky vypadá, že pole předáváme hodnotou.

Příklad: Přičte hodnotu ke všem prvkům pole.

```
void prictiN(int pole[], int n, int delka)
{
  for(int i = 0; i < delka; i++)
  {
    pole[i] += n;
  }
}
...
prictiN(mojePole, 5, DELKA); // zde není &mojePole
```

Alternativní prototyp stejné funkce:

```
      void prictiN(int *pole, int n, int delka);
```

Poznámka:
Datový typ ukazatel není naprosto shodný s datovým typem pole, ačkoli s nimi lze pracovat téměř stejným způsobem. Rozdíl je v tom, že datový typ pole uchovává některá data navíc. Zvláště je to patrné u vícerozměrných polí. V tomto případě datový typ pole uchovává informaci o vyšších dimenzích pole, aby bylo možné správně vypočítat polohu indexovaného prvku.

Tip:
Díky ukazatelům není nutné vždy předávat pole jako celek. Zvláště při zpracovávání textových řetězců se často využívá toho, že funkci lze předat i ukazatel na prostřední prvek.

```
char* msg = "Hujer! Metelesku blesku!";
printf("%s\n", msg);   // Hujer! Metelesku blesku!
printf("%s\n", msg+7); // Metelesku blesku!
```

Tento příklad zároveň demonstruje principiální omezení jazyka C. Kvůli této vlastnosti jazyk C nemůže hlídat meze polí. Tradičně se údaje o délce dynamicky alokovaných polí ukládají do několika bajtů paměti před začátkem pole (přesně odsud vezme funkce *free* informace pro správné uvolnění bloku paměti). Pokud však jazyk dovoluje pracovat i s ukazatelem ukazujícím doprostřed pole jako s kterýmkoli jiným polem, je zřejmé, že například ve funkci se nikdy nelze spolehnout na to, že pole bude obsahovat informaci o svém rozměru. Dále si musíme uvědomit, že funkce musí stejným způsobem pracovat s polem předaným pomocí ukazatele, ať je pole alokováno na hromadě nebo na zásobníku, nebo je alokováno staticky.

Autoři jazyka C dali programátorům větší svobodu pro práci, ale zaplatili jsme za to ztrátou schopnosti kontrolovat meze polí. Kdybychom chtěli tento příklad naprogramovat v jazyce s kontrolou mezí polí (Pascal), pravděpodobně bychom se nevyhnuli nutnosti vytvářet kopii části textového řetězce. V porovnání s tím, co jsme udělali v jazyce C, je to značně neefektivní, ale bezpečné. Asi nejčastější chybou vyskytující se při programování v C je indexace za skutečnou hranicí pole, což mívá fatální následky.

5.10 Shrnutí

Pro pochopení témat probíraných v této kapitole je více než u jiných kapitol potřeba praktická zkušenost získaná samostatným řešením programovacích úloh. Nebojte se používat papír a tužku a zkoumat běh programu pomocí ladících nástrojů.

Po přečtení této kapitoly a procvičení probraných témat ve vlastních programech čtenář dokáže deklarovat a používat proměnné datového typu ukazatel a používat je pomocí referenčního a dereferenčního operátoru. Čtenář nyní také chápe, jaký je rozdíl mezi typem ukazatel a proměnnou typu ukazatel. Rovněž rozumí, co se myslí pod pojmy typový ukazatel a obecný ukazatel.

Čtenář již nyní rozumí pojmu dynamická alokace paměti a dokáže schematicky nakreslit strukturu programátorské abstrakce paměťového prostoru. U jednotlivých oblastí paměti dokáže popsat, k čemu a v jakých situacích se tyto oblasti používají.

Čtenář již dokáže v jazyce C správně dynamicky alokovat paměť na hromadě, je si vědom chyb, které je potřeba při této činnosti ošetřovat a také nutnosti tuto paměť uvolňovat v okamžiku, když už není potřeba.

Čtenář nyní také dokáže deklarovat, inicializovat a používat jedno i vícerozměrná pole a používat je ve funkcích. Chápe princip indexování a je si vědom toho, jaké meze indexů polí jsou v jazyce C dány. Je schopen pole vytvářet i dynamickou alokací na hromadě a to i pro pole s více rozměry. Čtenář již nyní chápe vztah mezi typem ukazatel a typem pole.

Čtenář v této kapitole dále získal znalosti o základních způsobech předávání argumentů funkcím v jazyce C. Dokáže vysvětlit rozdíl mezi předáváním argumentů hodnotou a odkazem. Dokáže deklarovat funkci s takovými parametry a dokáže s nimi správně zacházet jak uvnitř funkce, tak i vně při jejím volání. Je si také vědom zákonitostí při předávání polí jako parametrů funkce, které vyplývají z jejich blízkého vztahu s typem ukazatel.

5.11 Úlohy k procvičení

1. Deklarujte ukazatel na **double**.
2. Inicializujte celočíselné pole nazvané *items* hodnotami od 1 do 10.
3. Deklarujte pole s neurčenou velikostí. Pole bude obsahovat mocniny čísla 3 až do čísla 729.
4. Napište program, který předává funkci ukazatel na celočíselnou proměnnou. Uvnitř funkce přiřaďte proměnné hodnotu -5. Po návratu z funkce ukažte, že proměnná obsahuje skutečně hodnotu -5 tak, že ji vypíšete.
5. Schematicky znázorněte, jak bude v paměti vypadat dvourozměrné pole nad typem int, alokované staticky.
6. Schematicky znázorněte, jak bude v paměti vypadat dvourozměrné pole nad typem int, alokované dynamicky na hromadě.

5.12 Kontrolní otázky

1. Co je ukazatel?
2. Jaké jsou ukazatelové operátory a jaká je jejich funkce?
3. Vysvětlete, co je nulový ukazatel a k čemu slouží.
4. Charakterizujte statickou alokaci paměti.
5. Charakterizujte dynamickou alokaci paměti.
6. Co je pole?
7. Jak se provede kopie jednoho pole do jiného pole?
8. Jaké jsou možnosti předávání argumentů funkcím?
9. Jaká je výhoda použití ukazatelů místo indexování polí?
10. Jaký je vzájemný vztah polí a ukazatelů?
11. Proč lze při deklaraci vícerozměrného pole s inicializátorem nebo jako parametru funkce vynechat nejlevější rozměr pole?
12. Proč nelze při deklaraci pole jako globální proměnné použít pro specifikaci jeho rozměrů proměnné?
13. Lze v nějakém případě použít při deklaraci pole proměnné pro specifikaci jeho rozměrů. Pokud ano, které to jsou a proč je to v těchto případech možné?
14. Existuje nějaké principiální omezení jazyka C, které mu znemožňuje automaticky hlídat meze polí?