Ukazatele

Ukazatel na hodnotu je *adresa* paměťového místa, které tuto hodnotu obsahuje.

V C hovoříme o *ukazatelích*, když jde o paměťové místo, které samo obsahuje adresu.

Deklarace

```
double* p;
```

čteme buď: `p` je proměnná typu `double*`,
tedy typu *ukazatel na proměnnou typu* `double`
hvězdička jako postfixový *typový* operátor

anebo: `p` je takové, že odkazovaná hodnota `*p`
je *proměnná typu* `double`
hvězdička jako prefixový operátor dereference

Dereferencování

```
double* p;
double x;
p = &x;
*p = 5;
```

`p = &x;` do `p` se vloží *adresa* proměnné `x`

`*p = 5;` do proměnné, na niž ukazuje `p`, se vloží číslo 5

Dereferencující operátor `*` zpřístupní hodnotu, na niž ukazatel odkazuje.

Vyskytne-li se v *r-výrazu* (na pravé straně přiřazení, v podmínce nebo v argumentu volané funkce) proměnná, dereferencuje se implicitně.

Pro explicitní dereferencování použijeme `*`.

Je-li `p` ukazatel na strukturu s položkou `t`,
lze zápis `(*p).t` zkrátit na `p->t`.

Operátor `&` zabraňuje dereferencování.

Lze ho číst jako *referencování*: `&x` je *adresa proměnné* `x`.

Ukazatele

```
int a, b;   // a, b jsou proměnné obsahující celá čísla
int *p;     // p je ukazatel na proměnné obsahující celá čísla
int **pp;   // pp je ukazatel na ukazatele na proměnné
a=3;        // uložení hodnoty 3 do proměnné a
p=&a;       // adresa proměnné do ukazatelové proměnné
b=*p+1;     // uložení hodnoty 4 do proměnné b
pp=&p;      // uložení ukazatele na adresu proměnné a do pp
**pp=2;     // uložení hodnoty 2 do proměnné a
*pp=&b;     // vložení adresy proměnné b do místa, něž ukazuje pp
(**pp)++;   // zvýšení dvojnásobně referencované hodnoty
```

Jaké jsou obsahy proměnných `a`, `b`, `p`, `pp`, když po předchozích příkazech provedeme ještě příkazy

```
p = &a;
**pp += *&b;
```

Nulový ukazatel

Zvláštní hodnotou je *nulový ukazatel* `NULL`, který má typ `void*` a je kompatibilní s ukazateli všech typů.

`NULL` je ukazatel, který nemá odkazovat na žádnou hodnotu. Jeho dereferencování způsobí chybu výpočtu, tj. výraz `*NULL` má nedefinovanou hodnotu.

`NULL` se používá u datových struktur, které jsou implementovány pomocí spojování prvků ukazateli.
Například poslední prvek dynamicky spojovaného seznamu, listy stromu apod.
Funkce, které vracejí ukazatel na prvek struktury nebo na oblast paměti, ve zvláštních případech nebo při chybových stavech vracejí `NULL`.

Předávání parametrů funkcím

Při volání funkcí v C se argumenty předávají *hodnotou*: argumenty se vyhodnocují před provedením příkazů v těle funkce.

V C se navíc *dereferencují* proměnné:
Je-li `n` proměná deklarovaná `int n=5;`, volání `abs(n)`, tak se funkci `abs` nepředá *proměnná* `n`, ale její *hodnota* 5.

Má-li funkce měnit hodnotu proměnné v argumentu, předáme jí její *adresu* pomocí operátoru `&`. Mluvíme pak o předávání parametrů *referencí*.

Pole a řetězce se implicitně nedereferencují. U nich tedy operátor `&` nepoužijeme.

Výměna hodnot ve dvou proměnných

Napište funkci `void swap(...)` se dvěma parametry, která vymění hodnoty ve dvou proměnných typu `float` pomocí třetí, lokální proměnné.

```
#include <stdio.h>

void swap    /*  DOPLŇTE definici funkce  */

int main (void) {
  float a = 2.0, b = 3.0;
  printf("Before:  a = %5f,  b = %5f\n", a, b);
  swap /* DOPLŇTE volání funkce */ ;
  printf("After:   a = %5f,  b = %5f\n", a, b);
}
```

https://github.com/li-ska/izp-cv/tree/master/06

Dělení se zbytkem

Napište funkci

```
bool divMod (int a, int b, int* quotient, int* remainder)
```

která jedním voláním:

- Vrátí `false`, když `b` $= 0$.
- Když `b` $\neq 0$, funkce vrátí `true` a také spočítá
  - celočíselný podíl `quotient` $= \left\lfloor \frac{\texttt{a}}{\texttt{b}} \right\rfloor$
  - zbytek po dělení `remainder` $=$ `a` $-$ `b` $\cdot$ `quotient`

Podíl a zbytek funkce vloží do svého třetího a čtvrtého argumentu, který se předává *odkazem*. Pro výpočet použijte standardní operace `/` a `%`.

Dělení se zbytkem

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

bool divMod (int a, int b, int* quotient, int* remainder) {

      /*  DOPLŇTE  */
}

int main(void) {
  int a[4] = {0, 23, 28, 35};
  int b[4] = {0, 1, 3, 7};
  int q; int r;
  for (int i=0; i<4; i++)
    for (int j=0; j<4; j++) {
      if (divMod(a[i], b[j], &q, &r))
          printf("%3d / %2d = %3d    %3d %% %2d = %3d\n",
              a[i], b[j], q, a[i], b[j], r);
      else printf("%3d / %2d =  _|_   %3d %% %2d =  _|_\n",
              a[i], b[j], a[i], b[j]);
    }
  return 0;
}
```

Dělení se zbytkem

Dává funkce správné výsledky, když dělenec a dělitel:

- jsou oba kladná čísla?
- jsou oba záporná čísla?
- mají různá znaménka?

`https://en.wikipedia.org/wiki/Modulo#Variants_of_the_definition`.

Množinové operace

Definujeme typ konečných množin:

```
typedef char Elem;      // typ prvků množiny

typedef struct {
    int card;           // počet prvků množiny
    Elem elems[100];    // pole prvků
} Set;
```

Pak například prázdnou množinu lze deklarovat jako proměnnou

```
Set emptyset;
emptyset.card = 0;
```

nebo lépe jako konstantu

```
const Set emptyset = {0};
```

Množinové operace

Definujte podobně funkce

- příslušnosti prvku do množiny
  `bool member_set(Elem el, Set *s)`
- přidání prvku do množiny – funkce změní svůj druhý argument
  a jako svůj výsledek vrátí počet prvků množiny
  `int insert_set(Elem el, Set *s)`
- průniku dvou množin – funkce změní svůj třetí argument
  a jako svůj výsledek vrátí počet prvků průniku
  `int intersect_set(Set *a, Set *b, Set *c)`
- sjednocení dvou množin – funkce změní svůj třetí argument
  a jako svůj výsledek vrátí počet prvků sjednocení
  `int union_set(Set *a, Set *b, Set *c)`

Všechny množinové operace udržují invariant pro typ `Set`:

`card` $\geqslant 0$ a v poli `elems` se neopakují prvky.

Množinové operace

Invariant je podmínka, které mohou funkce využívat, ale kterou musí také udržovat – musí zajistit, aby po návratu z funkce byly splněny invarianty všech dat.

Invariant typu `Set`:

- $0 \leqslant$ `card` < 100
- mezi prvky `elems[0]`, ..., `elems[card-1]` se žádná hodnota neopakuje.

Jak bychom mohli invariant typu `Set` zesílit, abychom mohli funkci `member_set` implementovat efektivněji?

Jakou časovou složitost má funkce `member_set` (při pův. invariantu)?

Jaké časové složitosti funkce `member_set` lze dosáhnout, jestliže invariant typu `Set` vhodně zesílíme?