Pointery a struktury

1 Struktury

O strukturách jsme si říkali, že na rozdíl od polí, kde všechny jeho prvky musí být stejného datového typu, mohou být struktury složeny z datovách prvků různých typů.

Pro připomenutí, struktury se definují následovně:

```
struct jmeno{
    polozka1
    ...
    polozkan
};
```

Vytvoření proměnných tohoto typu pak vypadá následovně.

```
struct jmeno promenna = {p1, ..., pn};
```

Proměnná `foo` v tomto případě je rovnou inicializovaná hodnotami p1, ... pn.

Přístup k jednotlivým položkám struktury se provádí pomocí tečkové notace.

```
promenna.polozka1
```

Abychom při definování proměnných nemuseli psát klíčové slovo struct, můžeme si strukturovaný typ pojmenovat s pomocí příkazu **typedef**.

```
typedef deklarace promenne;
```

Tedy:

```
typedef struct{
    polozka1
    ...
    polozkan
} JMENO;
```

Definice proměnných pak vypadá takto:

```
JMENO promenna = {p1, ..., pn};
```

Prvky struktury jsou v paměti ukládány za sebou (obdobně jako u polí), proto při deklaraci potřebujeme vědět, jak velká část paměti je potřeba pro jednotlivé proměnné. To je důvod, proč není možné, aby prvkem definované struktury byl prvek stejného typu, jako je právě definovaná struktura.

```
struct data{
    struct data d;  /* chyba */
    ...
}
```

Pokud ovšem potřebujeme, aby struktura obsahovala položku stejného typu, jako je právě definovaná struktura, tak tomuto požadavku můžeme vyhovět tak, že do struktury zahrneme pointer typu právě definované struktury.

2 Struktury a pointery

Pointer na strukturu se definuje stejným způsobem, jako pointer na jiný typ, tedy:

```
typedef struct{
    polozka1
    ...
    polozkan
} JMENO;

JMENO s, *p_s;
```

`s` je struktura typu `JMENO` a `p_s` je pointer na strukturu typu `JMENO`, která nemá samozřejmě zatím přidělenou paměť.

Paměť pro strukturu lze dynamicky přidělit pomocí funkce `malloc()`:

```
p_s= (JMENO*)malloc(sizeof(JMENO));
```

Pointer lze samozřejmě použít i pro odkaz na již existující strukturu, tedy např.:

```
p_s = &s;
```

Máme-li definici:

```
JMENO s, *p_s = &s;
```

Pak k prvkům struktury `s` můžeme přistupovat takto:

```
/* pomoci jmena struktury */
s.polozka1;

/* pomoci pointeru p_s komplikovane */
(*p_s).polozka1;

/* pomoci pointeru p_s jednoduseji */
p_s->polozka1;

/* tento zapis je chybny, protoze operator . ma prednost pred operatorem dereference *  */
*p_s.polozka1;
```

3 Příklad

Vše si ukážeme na příkladu `lineárního seznamu` obsahující celá čísla. Lineární seznam můžeme definovat následují strukturou:

```
typedef struct _node{
    int data;
    struct _node *next;
} node;
```

POZOR: Není možné použít tento zápis:

```
typedef struct {
    int data;
    node *next;
} node;
```

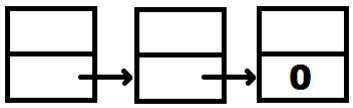
V okamžiku, kdy definujeme položku struktury next, není ještě znám typ struktury `node`.

Každému uzlu v seznamu bude odpovídat jedna struktura `node`m jejíž položka `data` bude obsahovat dané číslo.

Celý seznam si budeme v programu pamatovat jako pointer na první uzel seznamu. Položka `next` prvního uzlu bude ukazovat na druhý uzel atd.

Hodnota položky `next` posledního uzlu bude rovna 0.

Seznam si můžeme představit takto:



Pro ukázku si ukážeme, jak by mohla vypadat funkce pro přidání prvku na začátek seznamu.

```
node *add(node **list, int data)
{
    node *new = malloc(sizeof(node));
    new->data = data;
    new->next = *list;
    *list = new;
    return new;
}
```

4 Cvičení

1. Funkce pro práci se seznamem. Napište funkci, která:
   (a) vypíše všechny prvky seznamu.

(b) zjistí délku seznamu.

(c) přidá prvek na konec seznamu.

(d) smaže prvek na začátku seznamu.

(e) smaže prvek na konci seznamu.

(f) pro zadané i vypíše i-tý prvek seznamu.

(g) pro zadané i vypíše i-tý prvek seznamu od konce.

(h) pro zadané i smaže i-tý prvek seznamu.

(i) vytvoří kopii seznamu. Funkce musí fungovat tak, že pokud změníme kopii seznamu, originální seznam se nezmění.