Jak zjednodušit ukazatele? Schovat do struktur!

1 K čemu je typ struktura?

Datový typ struktura slouží k tomu, abychom mohli do jediné proměnné ukrýt, či chcete-li, zabalit více rozmanitých proměnných.

K čemu je to dobré? Proč by to chtěl někdo dělat dobrovolně? Napadají mě minimálně dva důvody a jsou to vlastně velice podobné důvody, jako pro ukládání souborů do složek nebo jejich balení do zip archívů.

Prvním důvodem je přirozená potřeba programátorů mít data uspořádána tak, aby věci, které logicky patří k sobě, byly tak říkajíc v jednom balíku.[1] Je to základ programovací techniky, které se říká tvorba *abstraktních datových typů*. Všechny objektově orientované jazyky jsou založeny právě na tomto principu.

Druhý důvod je snazší práce s takovým balíkem dat. Když chcete poslat hromadu souborů jako přílohu e-mailem, je jednodušší je zabalit do archivu a mít jako přílohu jediný soubor. Podobně při programování je často snazší proměnné zabalit do struktury a pak si je s funkcí předávat v jednom kuse. Pokud je to v souladu s výše zmíněným prvním důvodem, lze takto elegantně snižovat počet parametrů funkcí.[2] Je rozdíl, jestli se musím při volání funkce starat o devět parametrů nebo třeba jenom o tři.

V jazyce C je pro použití struktur ještě třetí důvod – jsou syntakticky pohodlné a hodně věcí jde pomocí nich zjednodušit. Nevěříte? Ukážeme si to.

2 Práce s typem struktura

Datový typ struktura je navržen tak, aby si jej programátor mohl vytvářet sám podle aktuální potřeby. Nejde tedy o zabudovaný typ, jako je třeba `int` nebo `float`, u kterého je předem jasné, jak budou vypadat proměnné. Typ struktura si musí programátor nejprve *deklarovat*, ideálně globálně, tj. na začátku programu nebo modulu.

Deklaraci typu struktura si ukážeme na tomto příkladu:

```
struct osoba {
  char jmeno[21];
  int id;
};
```

Pozor! Toto je pouhá deklarace typu. Tímto se ještě žádná proměnná nevyrobila. Takto jenom říkáme překladači, že *máme v plánu* vyrábět proměnné podle takového receptu!

Složky mohou být libovolných typů a předepisují, z jakých částí se proměnná typu struktura bude skládat.

Proměnná typu struktura se obecně vyrábí (někde uvnitř funkce) takto:

```
struct osoba zamestnanec;
```

Název typu jsou skutečně dvě slova! To nevypadá moc příjemně, že? Naštěstí jde každý typ v jazyce C přejmenovat pomocí specifikátoru `typedef`. Struktury se běžně při deklaraci typu rovnou přejmenují na něco hezčího:

```
typedef struct osoba {
  char jmeno[21];
  int id;
} Tosoba;
```

Alternativně to jde i takto:

```
typedef struct osoba Tosoba;

// a teď ještě deklarovat složky
struct osoba {
  char jmeno[21];
  int id;
};
```

To je náš ideální tvar, protože teď jde proměnné vyrábět, předávat do funkcí přes parametry a vracet z funkcí jako funkční hodnotu podle úplně stejných pravidel, jako

1 V krabici, složce, šuplíku, … zvolte si analogii podle chuti. Prostě když jakýkoli nepořádek zavřete do krabice, najednou vypadá uklizeně.

2 Nepřehánějte to. Když začnete vytvářet nesourodé struktury proměnných, které nemají žádný jiný vzájemný vztah, než že je chcete předat stejné funkci, spíše si práci přiděláte.

u proměnných *jednoduchých datových typů*! Syntakticky nenajdete rozdíl, dokud nezačnete pracovat s jednotlivými složkami a to je velice jednoduché.

2.1 Proměnná

Deklarace proměnné a práce se složkami vypadá takto:

```
Tosoba zamestnanec;
zamestnanec.id = 12345;
strcpy(zamestanec.jmeno, "Aaron");

// Kopie proměnných jde dělat
// přiřazením! A to i když má složku
// typu pole.
Tosoba klon = zamestnanec;
```

Všimněte si, že deklarace proměnné typu struktura nevypadá nijak odlišně od deklarace proměnné například typu `int`.

Proměnné typu struktura jde kopírovat přiřazením,[3] i když obsahují složku typu pole – to u samotných polí nejde.[4]

Složky proměnné typu struktura jsou přístupné pomocí tečkového operátoru a jména příslušné složky. Složka proměnné je taky proměnná.

2.2 Struktura jako jednoduchý parametr funkce

Při předávání struktury přes parametry platí stejné pravidla jako pro jednoduché typy. Při předávání hodnotou funkce obdrží kopii parametru.

```
// Parametr předávaný hodnotou – tedy
// jako kopie argumentu
void osobaTisk(Tosoba o);

// někde dále v kódu…
// Argument při volání
osobaTisk(zamestnanec);
```

2.3 Parametr předávaný odkazem

Při předávání odkazem, musíme parametr deklarovat jako ukazatel a při volání předat adresu původní proměnné.

```
// Parametr předávaný odkazem
void osobaZmen(Tosoba *o, char*jmeno)
{
  // složky jsou teď přístupné přes ->
  strcpy(o->jmeno, jmeno);
}

// někde dále v kódu…
// Argument při volání předávám
// odkazem, tedy musím předat adresu.
osobaZmen(&zamestnanec, "Baron");
```

Tady je vidět první důkaz, že práce s ukazateli je pohodlnější se strukturami. Ve funkci místo dereferenčního operátoru * (a tečky) teď používáme šipkový operátor.

Při volání funkce předáváme argument zase úplně stejně jako kdyby to byla jednoduchá proměnná, třeba typu `int`.

2.4 Struktura jako funkční hodnota

Protože proměnné typu struktura jde kopírovat přiřazením, může být tento typ použit i jako návratový typ funkce. To opět u polí nejde.

```
Tosoba osobaVyrob(char*jmeno, int id)
{
  Tosoba o;
  o.id = id;
  strcpy(o.jmeno, jmeno);
  return o;
}

// někde dále v kódu…
Tosoba z = osobaVyrob("Cecil", 9752);
```

To je celkem šikovné, ne? Bez struktury by taková funkce vůbec nešla vyrobit. Když bychom chtěli vrátit dvě či více proměnných zároveň, muselo by se to dělat pomocí dvou nebo více parametrů předávaných odkazem. V takové funkci by se to pak hemžilo hvězdičkami, až by z toho oči přecházely.

3 Tohle platí pro modernější variantu jazyka – C99. Při překladu je potřeba ji zapnout přepínačem překladače -std=C99. U starších verzí jazyka C to nešlo.

4 Na druhou stranu uvnitř struktury jde mít jen pole pevné velikosti. Pole s velikostí specifikovanou pomocí proměnné sem vložit nejde. To ale nevadí, protože sem můžeme vložit ukazatel a pole alokovat dynamicky.

3 Struktura jako finta pro zjednodušení ukazatelů

Podle mých zkušeností ukazatele nejvíce matou, když se začne kombinovat dynamická alokace a parametry předávané odkazem. Pak se v kódu mohou začít vyskytovat „vícehvězdičkoví generálové“, tj. ukazatele na ukazatele, často vznikající v důsledku ukazatelů předávaných odkazem.

Podívejte se například na tento příklad:

```
bool rozsirPole(int** pole, int n) {
  int* np = realloc(*pole,
                    n*sizeof(int);
  if (np != NULL) {
    *pole = np; return true;
  }
  return false;
}
// někde dále v kódu…
int *p = malloc(5*sizeof(int));
// a ještě dále…
rozsirPole(&p, 25);
```

Netočí se vám z těch hvězdiček a ampersandů hlava? Tak si zkuste představit, že to dynamické pole bude vícerozměrné. Pak tam počet hvězdiček, ale i parametrů ještě naroste. V některých situacích dokonce reálně hrozí, že by bylo potřeba používat dereferenci a indexování zároveň!

Ukážeme si, že pomocí struktur se jde hromadě hvězdiček úplně vyhnout.

3.1 Struktura obsahující dynamické pole

Složka struktury může být také typu pole, ale musí to být pole s pevným rozměrem. To se hodí pro krátká pole, např. krátké textové řetězce (jméno, příjmení, …). Potom je výhodou, že není potřeba ta pole alokovat. Když jsou uvnitř struktury, jde je spolu s celou strukturou pohodlně kopírovat přiřazovacím operátorem (a tedy i vracet z funkcí pomocí return).

U dlouhých polí se z výhody stává přítěž. Kopírování trvá dlouho, neuvážené kopírování proměnných vede k plýtvání pamětí. Pro dlouhá pole se více hodí dynamická alokace.

Dynamicky alokované pole se musí deklarovat jako ukazatel a alokovat pomocí funkce `malloc`. To už jsme řešili v minulém cvičení.

Abychom si to zjednodušili a zvýšili přehlednost, zabalíme si teď pole, respektive ukazatel na něj, do struktury. Zároveň s ním tam přibalíme i skutečnou délku alokovaného pole, protože, jak už jistě víte, pole si svou skutečnou délku nepamatuje.

```
typedef struct {
  float *prvek;
  int delka;
} Tpole;
```

Takové pole se dá používat velice pohodlně. Jde je například vracet jako funkční hodnotu.

```
// pole jako funkční hodnota
Tpole novePole(int delka) {
  Tpole p;
  p.prvek=malloc(delka*sizeof(float));
  if (p.prvek == NULL)
    p.delka = 0;
  p.delka = delka;
  return p;
}
```

Také jej jde předávat hodnotou. V tom případě jde měnit prvky pole, ale už ne samotný ukazatel nebo délku.

```
// pole předávané hodnotou
void vynasob(Tpole p, float k) {
  for (int i = 0; i < p.delka; ++i) {
    p.prvek[i] *= k;
  }
}
```

Při předávání odkazem jde měnit i složky struktury samotné.

```
// pole předávané odkazem
bool rozsirPole2(Tpole *p, int n) {
  float *np = realloc(p->prvek,
                      n*sizeof(float));
  if (np != NULL) {
    p->prvek = np; p->delka = n;
    return true;
  }
  return false;
}
```

U vícerozměrných polí je tento postup ještě účinnější.

Úkol č. 0 – Pod a nadprůměrný soubor

Uprav své řešení úlohy z minulého cvičení tak, aby využívala zde popsaný datový typ Tpole.

4 Struktura s odkazem na „sebe samu“

Struktura může obsahovat položku typu ukazatel na libovolný typ, tedy i na svůj vlastní. Tento princip využívá překvapivě hodně abstraktních dynamických datových typů, s nimiž budeme pracovat v dalších cvičeních – například seznam, fronta, zásobník, strom či graf.

Společnou vlastností těchto datových struktur je, že na rozdíl od pole, jde u nich pohodlně dynamicky (tj. za života programu) přidávat nebo rušit prvky a to dokonce nejenom na konec, ale i na začátek nebo dokonce doprostřed. Podrobněji je budeme probírat v dalších cvičeních.

Teď se zaměříme na jednoduchý seznam. Prvkem seznamu bude struktura nesoucí hodnotu a ukazatel na další prvek téhož typu.

```
typedef struct _prvek Tprvek;

struct _prvek {
  float hodnota;
  Tprvek * dalsi;
};
```

Princip seznamu tvořeného takovými prvky demonstruje tento obrázek.

Máme zde ukazatel na první prvek. Pak každý další prvek obsahuje ukazatel na svého následníka. Poslední prvek má svůj ukazatel nastavený na NULL.

Tento seznam se jmenuje *jednosměrně vázaný seznam*. Na podobném principu jsou založeny i další typy seznamu. Například obousměrně vázaný seznam má ukazatel na další i předchozí prvek, kruhový seznam nemá konec, protože poslední prvek ukazuje znovu na první prvek. S takovými strukturami se dají realizovat různé zajímavé algoritmy. S některými se ještě seznámíme.

Úkol č. 1 – nový prvek

Vytvoř funkci `novyPrvek`, která dynamicky alokuje jeden prvek seznamu a vloží do něj zadanou hodnotu – desetinné číslo. Funkce bude vracet ukazatel na nově vyrobený prvek nebo NULL, když se to nepovede.

Ukazatel na další prvek nezapomeň inicializovat hodnotou NULL.

Připrav si papír a tužku a kresli si dílčí kroky všech následujících algoritmů. Načrtni si seznam podobně, jako je to na obrázku na této straně. U následujících algoritmů velmi záleží na pořadí jednotlivých kroků!

Úkol č. 2 – vložení na začátek seznamu

Vytvoř funkci `vlozNaZacatek`, která dostane jako parametry číselnou hodnotu, kterou chci vložit na začátek seznamu a ukazatel na první prvek seznamu (může být i NULL, když chci vkládat do prázdného seznamu).

Funkce vyrobí nový prvek a *za něj* připojí původní seznam. Vrátí ukazatel na tento nový prvek a tedy na upravený seznam.

Úkol č. 3 – výpis seznamu

Vytvoř funkci `vypisPrvky`, která dostane ukazatel na první prvek seznamu, projde všechny prvky seznamu a vypíše na výstup hodnoty všech prvků seznamu. Zastaví se až na prvku, jehož ukazatel na další prvek obsahuje hodnotu NULL.

Úkol č. 4 – zrušení seznamu

Vytvoř funkci `zrusSeznam`, která dostane ukazatel na první prvek seznamu a postupně všechny prvky seznamu uvolní pomocí funkce `free`.

Úloha – otoč pořadí

Pomocí podprogramů z předchozích úkolů vytvoř program, který

- zpracuje vstup (může být i soubor) s předem neznámým počtem prvků (desetinná čísla),
- vypíše načtené prvky v opačném pořadí, než v jakém byly načteny.

Bonus: Uživatel zadá číslo N a program vypíše jen posledních N zadaných hodnot. Použitý seznam nesmí mít nikdy více než N prvků.