5. Strukturované datové typy, ukazatele

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Strukturované datové typy

  Pole

  Struct

  Union

- Část 2 – Ukazatele

  Ukazatele

  Předávání parametrů funkcím

- Část 3 – Zadání 4. domácího úkolu

Část I

Strukturované datové typy

Definice typu – operátor `typedef`

- Operátor `typedef` umožňuje definovat nový datový typ
- Slouží k pojmenování typů, např. ukazatele, struktury a uniony
- Například typ pro ukazatele na `double` a nové jméno pro `int`:

```
typedef double* double_p;
typedef int integer;
// totozne s double *x, *y;
double_p x, y;
// totozne s int i, j;
integer i, j;
```

- Nově zavedené typy lze používat v celém programu
- Výhodou je zpřehlednění zápisu u složitějších typů – ukazatele na funkce nebo struktury

I. Strukturované datové typy

Pole

Struct

Union

Motivace

- Výpočet průměrné teploty z hodnot naměřených ve všedních dnech

```c
#include <stdio.h>

int main(void)
{
  int t1, t2, t3, t4, t5, prumer;
  printf("zadejte teploty\n");
  scanf("%d", &t1);
  scanf("%d", &t2);
  scanf("%d", &t3);
  scanf("%d", &t4);
  scanf("%d", &t5);
  prumer = (t1+t2+t3+t4+t5+t5)/5;
  printf("%d\n", prumer);
  return 0;
}
```

- řešení je těžkopádné
- bylo by ale ještě horší, kdyby vstupními daty byly teploty za měsíc nebo za celý rok
- vhodnější je využít stukturovaný datový typ – **pole**

Pole

- Datová struktura pro uložení více hodnot stejného typu

  Typicky reprezentace posloupností

- Hodnoty uloženy v souvislém bloku paměti

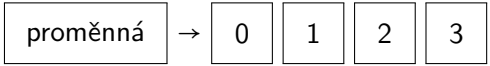
  Velikost bloku je určena počtem prvků a velikostí datového typu

- Proměnná typu pole reprezentuje adresu, kde blok začíná

  Definicí proměnné dochází k alokaci popřebné paměti

- Prvky pole mají stejnou velikost a známou relativní adresu

  Relativní adresa – posun vůči adrese prvního prvku



Deklarace pole

- Deklarací proměnné dojde k alokaci potřebné paměti

K inicializaci ale dojde jen za určitých okolností

`typ proměnná []`

- `[]` slouží také pro přístup k prvku pole (adresaci)

`proměnná [index]`

Příklad

```
1  int array[10];
3  printf("Velikost pole je %lu\n", sizeof(array));
4  printf("%i. prvek pole je %i\n", 4, array[4]);
```

```
Velikost pole je 40
4. prvek pole je -5789
```

Vlastnosti pole

- Pole je posloupnost prvků stejného typu

  Libovolný datový typ včetně strukturovaných

- K prvkům pole se přistupuje pořadovým číslem (indexem) prvku

  Index je celé nezáporné číslo

- Index prvního prvku je vždy roven **0**

  Index udává relativní vzdálenost prvku od adresy prvního prvku.

- C nekontroluje za běhu programu, zda je index platný!

  Je tedy možné adresovat paměť, která nebyla alokována

- Velikost pole (v bajtech) je dána počtem prvků pole **n** a **typem prvku**, tj. n * sizeof(typ)
- Velikost pole statické délky nelze měnit

  Ve starších standardech (< *C*99) je třeba, aby byla velikost pole známa v době překladu

- Pole může být jednorozměrné nebo vícerozměrné

Příklad – deklarace a přístup k prvkům 1/3

- Deklarace jednorozměrného a dvourozměrného pole

```
1  char simple_array[10];
2  int two_dimensional_array[2][2];
```

- Přístup k prvkům pole

```
1  m[1][2] = 2*1;
```

- Deklarace pole a tisk hodnot prvků

```
1  int array[5];
3  int velikost = sizeof(array)/sizeof(int);
5  for (int i = 0; i < velikost; ++i) {
6    printf("array[%i] = %i\n", i, array[i]);
7  }
```

Příklad – motivační příklad pomocí pole 2/3

```c
#include <stdio.h>

int main()
{
  int teploty[7], prumer = 0;

  for (int i = 0; i < 7; i++)
  {
    scanf("%d", &teploty[i]);
    prumer += teploty[i];
  }
  printf("%d\n", prumer);
  return 0;
}
```

lec05/teploty-pole.c

Příklad – statistika výskytu znaků v souboru 3/3

```c
#include <stdio.h>

int main(void)
{
  int h[26] = {0};
  char a;

  while(scanf("%c", &a) != EOF) {
    if('A' <= a && a <= 'Z') h[a-'A']++;
  }

  for (int i = 0; i < 26; i++) {
    printf("%c:% 4i\n", i+'A', h[i]);
  }

  return 0;
}
```

lec05/hist.c

Inicializace pole

- Inicializace pole hodnotami

```
1  double d[] = {0.1, 0.4, 0.5};
```

Pole bude mít velikost 3 prvků

- Inicializace pole textovým literálem

```
1  char str[] = "hallo";
```

Pole bude mít velikost 6 prvků

- Inicializace částečným výčtem

```
1  int a[] = {[4] = 10};
```

Pole bude mít velikost 5 prvků, všechny kromě 4. prvku budou inicializovány na 0

- Inicializace všech prvků pole na 0

```
1  int a[5] = {}; // možné, ale nelze v pedantic módu
2  int a[5] = {0};
```

Pole variabilní délky

- Standard C99 umožňuje určit velikost pole za běhu programu

  Ve starších standardech bylo třeba znát velikost v době překladu

- Nejedná se o možnost změny velikosti pole za běhu programu
- Pole variabilní délky nelze v definici inicializovat

Příklad

```
1  printf("zadej velikost pole: ");
2  scanf("%d", &n);
4  int pole[n];
6  for (int i = 0; i < n; i++) {
7    pole[i] = i * i;
8  }
```

I. Strukturované datové typy

Pole

Struct

Union

Struktura `struct`

- Struktura je konečná množina prvků (proměnných), které nemusí být stejného typu
- Skladba struktury je definovaná uživatelem jako nový typ sestavený z již definovaných typů (včetně jiných struktur)
- K prvkům struktury přistupujeme **tečkovou notací**
- Pro struktury stejného typu je definována operace přiřazení `=`

```
struct1 = struct2;
```

Na rozdíl od pole, kde je přiřazení nutné realizovat po prvcích.

- Struktury (jako celek) nelze porovnávat relacním operátorem `==`
- Struktura může být návratovou hodnotou funkce

Struktura struct – příklad

- Bez zavedení nového typu (typedef) je nutné pred identifikátor jména struktury uvádět klíčové slovo struct

```
1  struct record {
2    int number;
3    double value;
4  };
5  // NELZE! - typ record neni znam
6  record r;
7  // vyzadovano kl.  slovo struct
8  struct record r;
```

Název struktury

```
1  typedef struct {
2    int n;
3    double v;
4  } item;
6  item i;
```

Název datového typu

- Zavedením nového typu typedef můžeme používat typ struktury již bez uvádění klíčového slova struct

Definice jména a typu struktury

- Uvedením `struct record` zavádíme nové jméno struktury `record`

```
1  struct record {
2    int number;
3    double value;
4  };
```

Definice idenfikátoru `record` ve jmenném prostoru struktur (tag namespace)

- Definicí typu `typedef` zavádíme nové jméno typu record

```
1  typedef struct record record;
```

Definujeme globální idenfikátor record jako jméno typu struct record

- Obojí lze kombinovat v jediné definici jména a typu struktury

```
1  typedef struct record {
2    int number;
3    double value;
4  } record;
```

Inicializace struktury

- Struktury:

```
struct record {
  int number;
  double value;
};
```

```
typedef struct {
  int n;
  double v;
} item;
```

- Proměnné typu struktura můžeme inicializovat prvek po prvku

```
1  struct record r;
2  r.value = 21.4;
3  r.number = 7;
```

- Podobně jako pole lze inicializovat přímo pri definici

```
1  item i = { 1, 2.3 }; // Pozor na pořadí položek!
```

- Inicializovat lze i konkrétní položky (ostatní jsou nulovány)

```
1  struct record r2 = { .value = 10.4};
```

Struktura struct – přiřazení

- Struktury:

```
struct record {          typedef struct {
  int number;              int n;
  double value;            double v;
};                       } item;
```

```
struct record rec1 = { 10, 7.12 };
struct record rec2 = { 5, 13.1 };
item i;
print_record(rec1); /* number(10), value(7.12) */
print_record(rec2); /* number(5), value(13.10) */
rec1 = rec2;
i = rec1; /* NELZE! */
print_record(rec1); /* number(5), value(13.10) */
```

Stuktura struct – změna hodnoty

lec05/struct-compound-literal.c

```c
struct point {
  int x, y;
};

int main(void)
{
  struct point b = { .y = 3, .x = 2 };
  // b = 5, 6; NELZE!
  // b = .x = 5, .y = 6; NELZE!
  b.x = 5;
  b.y = 6;  // C89
  // C99:  compound literal - anonymni struktura
  b = (struct point){5, 6};
  b = (struct point){ .x = 5, .y = 6 };

  return 0;
}
```

Struktura struct – kopie paměti

- Jsou-li dve struktury stejně veliké, můžeme přímo kopírovat obsah příslušné paměťové oblasti

```
1  struct record r = {7, 21.4};
2  item i = {1, 2.3};
3  print_record(r); /* number(7), value(21.400000) */
4  print_item(i);   /* n(1), v(2.300000) */
5  if (sizeof(i) == sizeof(r)) {
6    printf("i and r are of the same size\n");
7    memcpy(&i, &r, sizeof(i));
8    print_item(i); /* n(7), v(21.400000) */
9  }
```

- V tomto případě je interpretace hodnot v obou strukturách identická, obecně tomu však být nemusí

Struktura struct – velikost

- Vnitřní reprezentace struktury nutně nemusí odpovídat součtu velikostí jednotlivých prvků
- Při překladu zpravidla dochází k zarovnání prvků na velikost slova příslušné architektury
  - rychlejší přístup, větší paměťové nároky
  - např. 8 bytů v případě 64 bitové architektury
- Překladači (clang a gcc) lze explicitně předepsat kompaktní paměťovou reprezentaci

```
1  struct record {int number; double value;};
2  struct record_p {int n; double v;} __attribute__((packed));

4  typedef struct {int n; double v;} item;
5  typedef struct __attribute__((packed)) {int n; double v;} item_p;

7  printf("i: %lu d: %lu\n", sizeof(int), sizeof(double));
8  printf("record: %lu\n", sizeof(struct record));
9  printf("item: %lu\n", sizeof(item));

11 printf("record_p: %lu\n", sizeof(struct record_p));
12 printf("item_p: %lu\n", sizeof(item_p));
```

Struktura struct – bitové pole

- Struktura umožňuje specifikovat velikost členských proměnných mimo násobky bytů
- Časté použití v mikroprocesorové technice

```
1  typedef struct {
2    unsigned char MODE:2;   // 00 - d.in, 01 - d.out, 10 - a.in
3    unsigned char PUPDR:2;  // 00 - nic; 01 - pull-up; 10 - pull-down
4    unsigned char STATUS:2;
5    unsigned char IN:1;
6    unsigned char OUT:1;
7  } GPIO;
9  GPIO portA;
10 portA.MODE = 2; // port A je v modu analog in
11 printf("port A mode: %d\n", portA.MODE);
```

- Co se stane, když se pokusíme zapsat do členské proměnné číslo s větším rozsahem?

I. Strukturované datové typy

Pole

Struct

Union

Proměnné se sdílenou pamětí – sjednocení `union`

- Množina prvků (proměnných), které nemusí být stejného typu
- Prvky unionů sdílejí společně stejná paměťová místa – překrývají se
- Velikost unionu je dána velikostí největšího z jeho prvků
- Skladba unionu je definována uživatelem jako nový typ sestavený z již definovaných typů
- K prvkům unionu se přistupuje tečkovou notací
- Pokud nedefinujeme nový typ, je nutné k identifikátoru proměnné unionu uvádět klíčové slovo `union`

```
1  union Nums {
2    char c;
3    int i;
4  };
6  Nums nums; /* NELZE! typ Nums neni znam! */
7  union Nums nums;
```

Příklad


```c
union Nums {
  char c;
  int i;
  double d;
};

printf("Velikost char %lu\n", sizeof(char));
printf("Velikost int %lu\n", sizeof(int));
printf("Velikost double %lu\n", sizeof(double));
printf("Velikost Nums %lu\n", sizeof(union Nums));
```

lec05/union.c

```
Velikost char: 1
Velikost int: 4
Velikost double: 8
Velikost Nums: 8
```

Příklad 2/2

```c
union Nums n;
printf("c: %3d i: %10d d: %lf\n", n.c, n.i, n.d);
n.c = 'a';
printf("c: %3d i: %10d d: %lf\n", n.c, n.i, n.d);
n.i = 5;
printf("c: %3d i: %10d d: %lf\n", n.c, n.i, n.d);
n.d = 3.14;
printf("c: %3d i: %10d d: %lf\n", n.c, n.i, n.d);
```

lec05/union.c

```
c: 112 i: 1818984816 d: 0.000000
c:  97 i: 1818984801 d: 0.000000
c:   5 i:          5 d: 0.000000
c:  31 i: 1374389535 d: 3.140000
```

Inicializace union

- Proměnnou typu union můžeme inicializovat při definici

```
union {
  char c;
  int i;
  double d;
} numbers = { 'a' };
```

Pouze první položka (proměnná) může být inicializována.

- V C99 můžeme inicializovat konkrétní položku (proměnnou)

```
union {
  char c;
  int i;
  double d;
} numbers = { .d = 10.3 };
```

Část II

Ukazatele

II. Ukazatele

Ukazatele

Předávání parametrů funkcím

Adresování

- Každá proměnná v paměti má tři hlavní charakteristiky
  - Jméno (identifikátor)
  - Obsah (hodnota)
  - Adresa – jednoznačná identifikace obsazeného místa v paměti

Vyjímkou jsou proměnné v paměťové třídě register.

- **Přímé adresování** – přístup k obsahu na adrese prostřednictvím jména proměnné
- **Nepřímé adresování** – přístup prostřednictvím jiné proměnné, která obsahuje adresu – tzv. reference
  - nepřímé adresování umožňuje funkcím přistupovat k paměti alokované mimo tělo funkce
  - nepřímé adresování umožňuje funkcím mít vic než jednu návratovou hodnotu (viz `scanf`)

Ukazatel (pointer)

- Ukazatel (pointer) je proměnná, jejíž hodnota je adresa v paměti
- Pointer může být inicializován adresou jiné proměnné

  Odkazuje na oblast paměti, kde je uložena hodnota proměnné.

- Ukazatel má typ proměnné, na kterou může ukazovat

  Důležité pro ukazatelovou aritmetiku

- Ukazatel může odkazovat na
  - proměnné všech typů – primitivní i strukturované
  - funkce
  - jiné ukazatele
- Ukazatel může být též bez typu (void)
  - Velikost proměnné nelze z vlastnosti ukazatele určit
  - Pak může obsahovat adresu libovolné proměnné
- Prázdná adresa ukazatele je definovaná hodnotou konstanty `NULL`

Definice ukazatele

- Deklarace ukazatele

```
// ukazatel na promennou typu char
char *p1;
// ukazatel na promennou typu int
int *p2;
```

- Inicializace ukazatele

```
char a;
// ukazatel p1 inicializovan adresou promenne a
p1 = &a;
int b;
// ukazatel p2 inicializovan adresou promenne b
p2 = &b;
```

- Referenční operátor `&` – vrací adresu paměti, kde je uložena hodnota proměnné, před kterou je uveden

Dereferenční operátor

- Vrací l-hodnotu (l-value) odpovídající hodnotě na adrese ukazatele
- Umožňuje číst a zapisovat hodnotu na adrese dané obsahem ukazatele

Příklad lec05/pointer.c

```c
#include <stdio.h>

int main()
{
  int a = 10;
  int *p = &a;
  printf("a = %i\n", a);
  *p = 20;
  printf("a = %i\n", a);

  return 0;
}
```

```
a = 10
a = 20
```

Ukazatele, proměnné a jejich hodnoty

- Proměnné jsou názvy adres, kde jsou uloženy hodnoty příslušného typu
- Kompilátor pracuje přímo s adresami

V případě kompilace se zpravidla jedná o adresy relativní.

- Ukazatel je proměnná, ve které je uložena adresa. Na této adrese se pak nachází hodnota nějakého typu (např. int).
- Ukazatele realizují tzv. nepřímé adresování
- Dereferenční operátor přistupuje na proměnnou adresovanou hodnotou ukazatele
- Operátor `&` vrací adresu, kde je uložena hodnota proměnné

Ukazatele a kódovací styl

- Symbol * lze zapisovat u identifikátoru proměnné nebo typu
- Preferujeme zápis u proměnné, abychom předešli omylům

```
char* a, b, c;     // ukazatel je pouze a
char *a, *b, *c;   // vsechny promenne jsou ukazatele
```

- Ukazatel na ukazatel: `char **a;`
- Typ ukazatel: `char*`, `char**`
- Neplatná adresa má symbolické jmého `NULL`
  - makro preprocesoru
  - v C99 lze i 0
  - proměnné v C nejsou automaticky inicializovány a ukazatele tak mohou odkazovat na neplatnou paměť

II. Ukazatele

Ukazatele

Předávání parametrů funkcím

Předávání parametrů funkcím

- V C jsou **parametry funkce předávány hodnotou**
- Parametry jsou lokální proměnné funkce (alokované v zásobníku), které jsou inicializované na hodnotu předávanou funkci

```
1  void fce(int a, char *b) {
2      /*  a - lokalni promena typu int (v zasobniku)
3          b - lokalni promena typu ukazatel na promenou typu
4              char (hodnota je adresa, take v zasobniku)  */
5  }
```

- Lokální změna hodnoty proměnné neovlivnuje hodnotu proměnné vně funkce
- Při předání ukazatele máme přístup na adresu původní proměnné, kterou můžeme měnit
- Ukazatelem tak realizujeme **volání odkazem**

Předávání parametrů funkcím – příklad

```c
void fce(int a, char* b) { // a - volání hodnotou, b - volání odkazem
  a += 1;
  (*b)++;
}
//--
int a = 10;
char b = 'A';
printf("%13s a=%d b=%c\n", "Pred volanim:", a, b);
fce(a, &b);
printf("%13s a=%d b=%c\n", "Po volani:", a, b);
```

lec05/function-call.c

```
Pred volanim: a=10 b=A
   Po volani: a=10 b=B
```

Pole jako parametr funkce

- Jako parametr funkce se předává ukazatel s adresou pole

Datovým typem parametru může být pole nebo ukazatel

```
1  void fce1 (int * a) {
2    printf ("a[0] = %d\n", a[0]);
3  }
5  void fce2 (int a[]) {
6    printf ("a[0] = %d\n", a[0]);
7  }
9  int pole[] = {10, 20, 30};
10 fce1 (pole);
11 fce2 (pole);
```

lec05/function-array.c

```
a[0] = 10
a[0] = 10
```

Struktura jako parametr funkce

- Struktury lze předávat jako parametry funkcí hodnotou

```
1  void print_record(struct record rec) {
2    printf("record: number(%d), value(%lf)\n",
3      rec.number, rec.value);
4  }
```

Vytváří se nová proměnná a původní obsah předávané struktury se kopíruje do zásobníku

- Nebo ukazatelem

```
1  void print_item(item *v) {
2    printf("item: n(%d), v(%lf)\n", (*v).n, (*v).v);
3    printf("item: n(%d), v(%lf)\n", v->n, v->v);
4  }
```

Kopíruje se pouze hodnota ukazatele (adresa) a pracujeme tak s původní strukturou

- Ukazatel na strukturu: `(*v).item` lze zapsat jako `v->item`

Část III

Zadání domácího úkolu

Zadání 4. domácího úkolu (HW04)

Téma: RLE kodér

- **Motivace:** Naprogramování složitějšího algoritmu
- **Cíl:** Použití cyklů a podmínek
- **Zadání:** `https://cw.fel.cvut.cz/wiki/courses/b0b99prpa/hw/hw04`
  - Načtení vstupních dat v podobně ASCII znaků
  - Kódování sekvencí stejných symbolů
  - Výstupem je bytový proud
  - Využití standardního chybového výstupu
- **Termín odevzdání:** **05.11.2021, 22:00PT**

Shrnutí přednášky

Diskutovaná témata

- Pole
- Struktury
- Uniony
- Ukazatele
- Předávání parametrů funkcím

- Příště: pole a ukazatele, vícerozměrná pole, textové řetězce

Diskutovaná témata

- Pole
- Struktury
- Uniony
- Ukazatele
- Předávání parametrů funkcím

- Příště: pole a ukazatele, vícerozměrná pole, textové řetězce