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

  - Zavedením typů operátorem `typedef` umožňuje používání nových jmen typů v celém programu
  - Výhoda zavedení nových typů je především 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

‘

```
1  #include <stdio.h>
3  int main(void)
4  {
5    int t1, t2, t3, t4, t5, prumer;
6    printf("zadejte teploty\n");
7    scanf("%d", &t1);
8    scanf("%d", &t2);
9    scanf("%d", &t3);
10   scanf("%d", &t4);
11   scanf("%d", &t5);
12   prumer = (t1+t2+t3+t4+t5+t5)/5;
13   printf("%d\n", prumer);
14   return 0;
15 }
```

- 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

lec05/teploty.c

Pole

- Datová struktura pro uložení více hodnot stejného typu

  Typicky reprezentace posloupností

- Hodnoty uloženy v souvislém bloku paměti

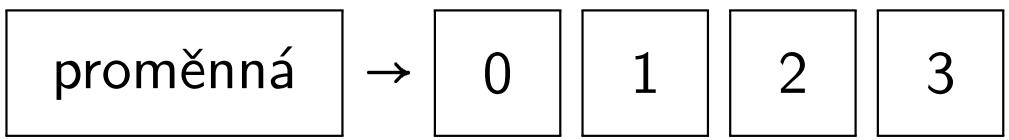
  Velikost bloku je určena počtem prvků a velikostí datového typu

- 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



Definice pole

- Definicí 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

```
int array[10];
printf("Velikost pole je %lu\n", sizeof(array));
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

Pole – příklad 1/3

- Definice jednorozměrného a dvourozměrného pole

```
/* jednorozmerne pole prvku typu char */
char simple_array[10];
/* dvourozmerne pole prvku typu int */
int two_dimensional_array[2][2];
```

- Přístup k prvkům pole

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

- Definice pole a tisk hodnot prvků

```
int array[5];

for (int i = 0; i < 5; ++i) {
  printf("array[%i] = %i\n", i, array[i]);
}
```

Pole – příklad

- Motivační příklad pomocí pole

```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/temploty-pole.c

Pole – příklad 3/3

- Statistika výskytu znaků v souboru

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5    int h[26] = {0};
6    char a;
7
8    while(scanf("%c", &a) != EOF) {
9      if('A' <= a && a <= 'Z') h[a-'A']++;
10   }
11
12   for (int i = 0; i < 26; i++) {
13     printf("%c:% 4i\n", i+'A', h[i]);
14   }
15
16   return 0;
17 }
```

lec05/hist.c

Inicializace pole

- Inicializace pole hodnotami

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

Pole bude mít velikost 3 prvků

- Inicializace pole textovým literálem

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

Pole bude mít velikost 6 prvků

- Inicializace částečným výčtem

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

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

- Inicializace všech prvků pole na 0

```
int a[5] = {};
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

```
printf("zadej velikost pole: ");
scanf("%d", &n);

int pole[n];
for (int i = 0; i < n; i++) {
  pole[i] = i * i;
}
```

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
- K prvkům struktury přistupujeme **tečkovou notací**
- Pro struktury stejného typu je definována operace přirazení

  ```
  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`

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

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

```
record r; /* NELZE! - typ record neni znam */
struct record r; /* vyzadovano kl. slovo struct */
item i; /* definice typu pomoci typedef */
```

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

Definice jména struktury a typu struktury

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

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

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

- Definicí typu `typedef` zavádíme nové jméno typu record

```
typedef struct record record;
```

Definujeme globální idenfikátor record jako jméno typu struct record

- Obojí lze kombinovat v jediné definici jména a typu struktury

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

Inicializace struktury `struct`

- Struktury:

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

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

- Proměnné typu struktura můžeme inicializovat prvek po prvku

```
struct record r;
r.value = 21.4;
r.number = 7;
```

- Podobně jako pole lze inicializovat přímo pri definici

```
item i = { 1, 2.3 };
```

Pozor na pořadí položek!

- nebo pouze konkrétní položky (ostatní jsou nulovány)

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

Struktura `struct` – přiřazení

- Struktury:

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

```
typedef struct {
  int n;
  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

```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;
}
```

lec05/struct-compound-literal.c

Struktura `struct` – kopie paměti

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

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

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

Struktura `struct` a velikost 1/2

- Vnitřní reprezentace struktury nutně nemusí odpovídat součtu velikostí jednotlivých prvků

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

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

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

```
i: 4 d: 8
record: 16
item: 16
```

- Při kompilaci zpravidla dochází k zarovnání prvků na velikost slova příslušné architektury – rychlejší přístup, větší paměťové nároky

Např. 8 bytů v případě 64 bitové architektury.

Struktura `struct` a velikost 2/2

- Můžeme explicitně předepsat kompaktní pamětovou reprezentaci
  - direktiva `__attribute__((packed))` pro překladače `clang` a `gcc`

```
struct record {
  int n;
  double v;
} __attribute__((packed));
// --
typedef struct __attribute__((packed)) {
  int n;
  double v;
} item;
printf("record: %lu\n", sizeof(struct record));
printf("item: %lu\n", sizeof(item));
```

```
record: 12
item: 12
```

Struktura `struct` – bitové pole

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

```
typedef struct {
  unsigned char MODE:2;
  unsigned char PUPDR:2;
  unsigned char STATUS:2;
  unsigned char IN:1;
  unsigned char OUT:1;
} GPIO;
```

I. Strukturované datové typy

Pole

Struct

Union

Proměnné se sdílenou pamětí – union

- Množina prvků (proměnných), které nemusí být stejného typu
- Prvky unionů sdílejí společně stejná paměťová místa

Prekrývají 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

```
union Nums {
  char c;
  int i;
};
Nums nums; /* NELZE! typ Nums neni znam! */
union Nums nums;
```

Příklad union 1/2

```
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 union 2/2

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

```
1  #include <stdio.h>
2
3  int main()
4  {
5    int a = 10;
6    int *p = &a;
7    printf("a = %i\n", a);
8    *p = 20;
9    printf("a = %i\n", a);
10
11   return 0;
12 }
```

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

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

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

Předávání parametrů funkcím – příklad

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

lec05/function-call.c

```
Pred volanim a:
Po volani a:
```

Pole jako parametr funkce

- Jako parametr funkce se předává ukazatel s adresou pole
- Datovým typem parametru může být pole nebo ukazatel

```
void fce1 (int * a) {
  printf ("a[0] = %d\n", a[0]);
}

void fce2 (int a[]) {
  printf ("a[0] = %d\n", a[0]);
}

int pole[] = {10, 20, 30};
fce1 (pole);
fce2 (pole);
```

lec05/function-array.c

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

Struktura jako parametr funkce

- Struktury lze předávat jako parametry funkcí hodnotou

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

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

- Nebo ukazatelem

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

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: Kreslení (ASCII art)

- **Motivace:** Zábavným a tvůrčím způsobem získat praktickou zkušenost s cykly a jejich parametrizací na základě uživatelského vstupu.
- **Cíl:** Použití cyklů a vnořených cyklů.
- **Zadání:** `https://cw.fel.cvut.cz/wiki/courses/b0b99prpa/hw/hw04`
  - Načtení parametrizace pro vykreslení obrázku domečku s využitím vybraných ASCII znaků
  - Ošetření vstupních hodnot
  - **Volitelné zadání** rozšiřuje obrázek domečku o plot
- **Termín odevzdání:** **2.11.2018, 23:59:59**

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