6. Strukturované datové typy

B0B99PRPA – Procedurální programování

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přehled témat

- Část 1 – Strukturované datové typy
  - Pole
  - Vícerozměrná pole
  - Struct
  - Union
- Část 2 – Textové řetězce
  - Textové řetězce
  - Funkce main

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`:

```
1  typedef double* double_p;
2  typedef int integer;
3  // totozne s double *x, *y;
4  double_p x, y;
5  // totozne s int i, j;
6  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

Vícerozměrná 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

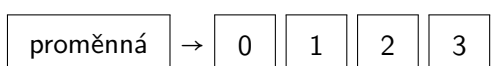
  Velikost bloku je určena počtem prvků a velikostí datového typu

- 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;
}
```

lec06/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;
}
```

lec06/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  }
```

Pole a ukazatele

- Ukazatel ukazuje na vyhrazenou část paměti proměnné

```
int *p;      // ukazatel (adresa)
sizeof(p);   // velikost promenne
```

- Pole je označení souvislého bloku paměti

```
int a[10];   // souvisly blok pameti
sizeof(a);   // 10*sizeof(int)
```

- Obě proměnné odkazují na paměť, kompilátor s nimi však pracuje rozdílně
  - Proměnná typu pole je symbolické jméno pro místo v paměti, kde jsou uloženy hodnoty prvků pole

    Kompilátor nahrazuje jméno přímo paměťovým místem
  - Ukazatel obsahuje adresu, na které je příslušná hodnota

    Nepřímé adresování
- Při předávání pole jako parametru funkce je předáváno pole jako ukazatel.

Pole a ukazatele

- Proměnná typu tříprvkové pole int

  `a[3] = {1,2,3};`

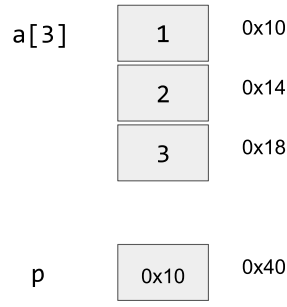
  Odkazuje na adresu prvního prvku pole

- Proměnná ukazatel na integer

  `int *p = a;`

  Obsahuje adresu prvního prvku pole

- `a[0]` reprezentuje hodnotu na adrese `0x10`.



- Hodnota `p` je adresa `0x10`, kde je uložena hodnota 1. prvku pole.
- Přiřazení `p = a` je možné.
- Kompilátor zajistí přiřazení adresy prvního prvku do ukazatele.
- Přístup k 2. prvku lze použít jak `a[1]` tak `p[1]`.
- Oběma přístupy se dostaneme na příslušné prvky pole, způsob je však odlišný

Ukazatele využívají tzv. pointerovou aritmetiku.

Ukazatelová artitmetika

- S ukazateli lze provádět aritmetické operace + a −, tj. přičítat nebo odčítat celé číslo
  - `ukazatel = ukazatel` stejného typu + (nebo −) a celé číslo (`int`)
  - Nebo lze používat zkrácený zápis např. ukazatel `+= 1` a unární operátory např. `ukazatel++`
- Aritmetické operace jsou užitečné pokud ukazatel odkazuje na více položek daného typu (souvislý blok paměti)
  - Např. pole položek příslušného typu
  - Dynamicky alokovaný souvislý blok paměti
- Přičtením hodnoty celého čísla k ukazateli posouváme hodnotu ukazatele na další prvek, např.

```
int a[10];
int *p = a;
int i = *(p+2); //odkazuje na hodnotu a[2]
```

Pole a ukazatele – příklad

```c
#include<stdio.h>

int main(void)
{
    int *p;         // ukazatel na integer
    int (*ptr)[5]; // ukazatel na pole peti prvku int
    int arr[5];

    p = arr;        // ukazatel na prvni prvek pole
    ptr = &arr;     // ukazatel na cele pole

    printf("sizeof(p) = %lu\n", sizeof(p));
    printf("sizeof(ptr) = %lu\n", sizeof(ptr));
    printf("p = %p, ptr = %p\n", p++, ptr++);
    printf("p = %p, ptr = %p\n", p, ptr);

    return 0;
}
```

lec06/array-pointer.c

Funkce s polem jako argumentem

```
3  void fce(int A[]) {
4    int B[] = { 2, 4, 6 };
5    printf("[A] = %lu, [B] = %lu\n", sizeof(B), sizeof(B));
6    for (int i = 0; i < 3; ++i) {
7      printf("A[%i]=%i B[%i]=%i\n", i, A[i], i, B[i]);
8    }
9  }

13 int array[] = { 1, 2, 3 };
14 fce(array);
```

lec06/array-argument.c

- Po překladu (`gcc -std=c99`) na **amd64**
  - `sizeof(A)` vrátí velikost 8 bajtů (64-bitová adresa)
  - `sizeof(B)` vrátí velikost 12 bajtů (3×4 bajty – `int`)
- Pole se funkcím předává jako ukazatel na první prvek

I. Strukturované datové typy

Pole

Vícerozměrná pole

Struct

Union

Vícerozměrná pole

- Pole lze definovat jako vícerozměrná, např. 2D matice

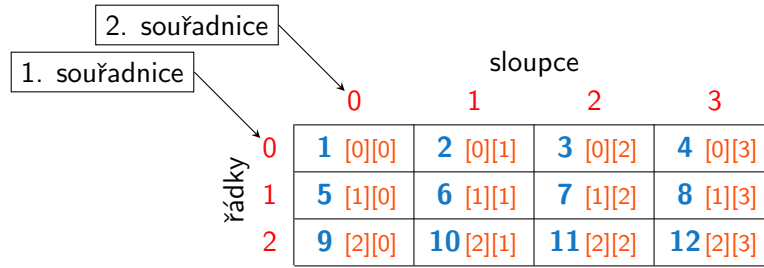
```
4   int m[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
5   printf("[m]: %lu == %lu\n", sizeof(m), 3*3*sizeof(int));
7   for (int r = 0; r < 3; ++r) {
8     for (int c = 0; c < 3; ++c) {
9       printf("%3i", m[r][c]);
10    }
11    printf("\n");
12  }
```

lec06/array-two-dimensions.c

```
[m]: 36 == 36
  1  2  3
  4  5  6
  7  8  9
```

Vícerozměrná pole

```
int a[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```



Vícerozměrná pole

```
int a[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

- Každý řádek lze považovat za 1D pole
- `a` je pole tří prvků, každý prvek je pak 1D pole čtyř prvků

`a` – první prvek pole `a`

`a+1` – druhý prvek pole `a`

`a+2` – třetí prvek pole `a`

`*a` – první prvek pole `a[0]`

`*(a+1)` – první prvek pole `a[1]`

`*(a+2)` – první prvek pole `a[2]`

Vnitřní reprezentace vícerozměrných polí

- Vícerozměrné pole je vždy souvislý blok paměti

```
1 int *pm = (int *)m; // ukazatel na souvislou oblast
2 printf("m[0][0]=%i m[1][0]=%i\n", m[0][0], m[1][0]);
3 printf("pm[0]=%i pm[3]=%i\n", pm[0], pm[3]);
```

- Dvourozměrné pole lze také definovat jako ukazatel na ukazatele (pole ukazatelů) na hodnoty konkrétního typu, např.
  - `int **a;` – ukazatel na ukazatele
  - V obecném prípade však takový ukazatel nemusí odkazovat na souvislou oblast, kde jsou alokovány jednotlivé prvky.
  - Při přístupu jako do jednorozměrného pole `int *b = (int *)a;` tedy nelze garantovat přístup do druhého řádku jako v přechozím příkladě.

Vícerozměrná pole jako parametr funkce

- Parametr funkce je ukazatel na pole, např. typu `int`

```
1  int (*p)[3] = m; // pointer to array of int
2  printf("Size of p: %lu\n", sizeof(p));
3  printf("Size of *p: %lu\n", sizeof(*p));
```

- Funkci nelze deklarovat s argumentem typu `[][]` např.

  `int fce(int a[][]);`
  - kompilátor nemůže správně spočítat index, pro přístup na `a[i][j]` se používá adresová aritmetika jinak

    Pro int m[row][col] totiž m[i][j] odpovídá hodnote na adrese *(m + col * i + j)

- Je však možné funkci deklarovat například jako
  - `int g(int a[]);` což odpovídá deklaraci `int g(int *a);`
  - `int fce(int a[][13]);` – je znám počet sloupců
  - nebo `int fce(int a[3][3]);`

I. Strukturované datové typy

Pole

Vícerozměrná 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;
5
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 {
  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

lec06/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

Vícerozměrná 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 1/2

```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));
```

lec06/union.c

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

Příklad


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

lec06/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

```
1  union {
2    char c;
3    int i;
4    double d;
5  } numbers = { 'a' };
```

Pouze první položka (proměnná) může být inicializována.

- V C99 můžeme inicializovat konkrétní položku (proměnnou)

```
1  union {
2    char c;
3    int i;
4    double d;
5  } numbers = { .d = 10.3 };
```

Část II

Textové řetězce

II. Textové řetězce

Řetězcové literály

- Formát – posloupnost znaků a řídicích znaků uzavřená v uvozovkách

  `"Sud kulatý, rys tu pije!\n"`

  - Řetězcové konstanty oddělené oddělovači (white spaces) se sloučí

    `"Tu je kára," " ten to ryje!\n"`

    se sloučí do

    `"Tu je kára, ten to ryje!\n"`
- Typ – pole typu char zakončené znakem `'\0'`
  - Pole pro uložení řetězce musí být o jeden prvek větší než délka samotného řetězcového literálu
  - Pokud ukončovací znak chybí, překladač nepozná konec řetězce
  - Při inicializaci literálem se automaticky vytvoří správně velké pole

```
char text[] = "PRPA";
```

Textový řetězec – pole znaků

- Textový řetězec můžeme inicializovat jako pole znaků `char[]`

```c
char str[] = "123";
char s[] = {'5', '6', '7'};

printf("sizeof(str): %lu\n", sizeof(str));
printf("sizeof(s)  : %lu\n", sizeof(s));
printf("str: '%s'\n", str);
printf("  s: '%s'\n", s);
```

lec06/string-array.c

```
sizeof(str): 4
sizeof(s)  : 3
str: 123
  s: 567890
```

Pokud není řetězec ukončen, výpis pokračuje, dokud není nalezen znak '\n'

Textový řetězec – ukazatel

- Na textový řetězec lze odkazovat ukazatelem na znak `char*`

```
1  char a[] = "ahoj";
2  char *b = "ahoj";
3  char *c = "ahoj";
5  printf("sizeof(a): %lu\n", sizeof(a));
6  printf("sizeof(b): %lu\n", sizeof(b));
8  printf ("adresa a: %p\n", a);
9  printf ("adresa b: %p\n", b);
10 printf ("adresa c: %p\n", c);
```

lec06/string-pointer.c

```
sizeof(a): 5
sizeof(b): 8
 adresa a: 0x7ffe708d1b60
 adresa b: 0x4006d4
 adresa c: 0x4006d4
```

Inicializace textového řetězce – pole a ukazatel

- Inicializace řetězce reprezentovaného polem
  - Alokace správně velkého pole
  - Nakopírování literálu na dané místo v paměti
  - Lze získat informaci o velikosti řetězce
- Inicializace řetězce reprezentovaného ukazatelem
  - Kompilátor umístí literál do zvláštního segmentu paměti
  - Ukazatel je inicializován adresou
  - Pokud v tomto segmentu již existuje stejný literál, ukazatel je inicializován jeho adresou
  - Tento segment paměti je pouze pro čtení, proto nelze měnit libovolně znaky řetězce

    Více o segmentech paměti na příští přednášce
  - Nelze přímo získat informaci o velikosti řetězce

    Operátor sizeof vrací velikost ukazatele, 8 bytů pro 64b OS

Textový řetězec – přiřazení

- Textový řetězec reprezentovaný polem `char[]`

```
char str[5];
str = "prpa"; // nelze zkompilovat
```

  - Podle definice nelze aplikovat operátor přiřazení na datový typ pole
  - Je třeba, aby proměnná na levé straně byla modidikovatelná, čemuž v případě pole není – proměnná reprezentuje adresu již alokovaného pole
  - Přiřazení lze zajistit znak po znaku nebo pomocí funkce `strcpy()`

- Textový řetězec reprezentovaný ukazatelem `*char`

```
char *str;
str = "prpa"; // v pořádku
```

  - Literál je vytvořen v paměti a jeho adresou je inicializován dosud neinicializovaný ukazatel

Načtení textového řetězce funkcí `scanf()`

- Použitím `%s` může dojít k přepisu paměti

```
1  char str0[4] = "PRP";
2  char str1[5];
3  printf("String str0 = '%s'\n", str0);
4  printf("Enter 4 chars: ");
5  scanf("%s", str1);
6  printf("You entered string '%s'\n", str1);
7  printf("String str0 = '%s'\n", str0);
```

- Načtení maximálně 4 znaků zajistíme řídicím řetězcem `%4s`

```
1  char str0[4] = "PRP";
2  char str1[5];
3  scanf("%4s", str1);
4  printf("You entered string '%s'\n", str1);
5  printf("String str0 = '%s'\n", str0);
```

Načtení textového řetězce funkcí `scanf()`

- Načtení celého řádku (včetně bílých znaků)

```
1  char str[100];
2  scanf("%[^\n]", str);
```

Místo \n může být libovolný jiný znak

- Omezení vstupních znaků

```
1  scanf("%[0-9]", str); // podobně [a-z], [0-9a-z]
```

```
1234abc
1234
```

```
1  scanf("%[123]", str); // podobně [ABC]
```

```
112233126870
11223312
```

Zjištění délky textového řetězce

- Textový řetězec v C je pole (`char[]`) nebo ukazatel (`char*`) odkazující na část paměti, kde je uložena příslušná posloupnost znaků.
- Textový řetězec je zakončen znakem `'\0'`
- Délku textového řetězce lze zjistit sekvenčním procházením znak po znaku až k `'\0'`

```
1  int delka(char *str) {
2    int ret = 0;
3    while (str && (*str++) != '\0') {
4      ret += 1;
5    }
6    return ret;
7  }
8  ...
9  char *text = "Hello PRP!\n";
10 printf("%i %lu\n", delka(text), strlen(text));
```

Práce s textovými řetězci

- Základní operace jsou definovány v knihovne `<string.h>`
  - Funkce předpokládající dostatečný rozsah alokovaných polí

```
int strlen (char *s);
char* strcpy (char *dst, char *src);
int strcmp (const char *s1, const char *s2);
strcat ();
```

  - Funkce s explicitním limitem na maximální délku řetězců:

```
char* strncpy(char *dst, char *src, size_t len);
int strncmp (const char *s1, const char *s2, size_t len);
strncat ();
```

- Převod řetězce na číslo – `<stdlib.h>`

`atoi()`, `atof()` – převod celého a necelého čísla

```
long strtol(const char *nptr, char **endptr, int base);
double strtod(const char *nptr, char **restrict endptr);
```

Porovnávání textových řetězců

- Řetězce nelze porovnávat pomocí relačních operátorů – proměnné reprezentují adresy

```
// např.  str1 = 0x7f42, str2 = 0x654d
void doSomething (char *str1, char *str2) {
  if (str1 > str2) { // porovnává 0x7f42 > 0x654d!
```

- Porovnávání řetězců pomocí funkce `strcmp`

```
1  int compResult = strcmp (str1, str2);
2  if (compResult == 0) {
3    // equal
4  } else if (compResult < 0) {
5    // str1 comes before str2
6  } else {
7    // str1 comes after str2
8  }
```

Kopírování textových řetězců

- Řetězce nelze v C kopírovat pomocí operátoru `=`

```
1  char str1[] = "hello"; // e.g.  0x7f42
2  char *str2 = str1; // 0x7f42 stejný řetězec!
3  str2[0] = 'H';
4  printf("%s %s", str1, str2); // Hello Hello
5
```

- Kopírování řetězců pomocí funkce `strcpy`

```
1  char str1[] = "hello"; // e.g.  0x7f42
2  char str2[6];
3  strcpy (str2, str1);
4  str2[0] = 'H';
5  printf ("%s %s", str1, str2); // hello Hello
6
```

Argumentem strcpy je inicializovaný ukazatel. Tím může být pole, dynamicky alokované pole nebo ukazatel inicializovaný literálem.

Spojování textových řetězců

- Řetězce nelze v C spojovat pomocí operátoru `+`

```
1  char str1[] = "hello ";
2  char str2[] = "prpa!";
3  char *str3 = str1 + str2; // nelze ani zkompilovat!
4
```

- Spojování řetězců pomocí funkce `strcat`

```
1  // výsledek se vloží do prvního argumentu strcat
2  // musí být tedy dostatečně dlouhý
3  char str1[13] = "hello ";
4  char str2[] = "prpa!";
5  // funkce přesune konec str1 na konec výsledku
6  strcat (str1, str2);
7  printf ("%s", str1);
8
```

Části textových řetězců

- Pro získání části textového řetězce lze využít ukazatel

```
1  char chars[] = "racecar";
2  char *str1 = chars;
3  char *str2 = chars + 4;
4  printf("%s\n", str1); // racecar
5  printf("%s\n", str2); // car
6
```

- Pozor na to, že ukazatele ukazují na stejná místa v paměti

```
1  str2[0] = 'f';
2  printf("%s\n", str1); // racefar
3  printf("%s\n", str2); // far
4
```

Pole textových řetězců

- Pole textových řetězců bude nejméně dvourozměrné

```
1  char *stringArray[5]; // pole ukazatelů char *s
3  char *stringArray[] = {
4    "my string 1",
5    "my string 2",
6    "my string 3"
7  };
9  printf ("%s\n", stringArray[1]);
```

II. Textové řetězce

Funkce `main` a její tvary

- Základní tvar funkce main

```
1 int main(int argc, char *argv[]) { ... }
```

- Alternativně pak také

```
1 int main(int argc, char **argv) { ... }
```

- Argumenty funkce nejsou nutné

```
1 int main(void) { ... }
```

- Rozšířená funkce o nastavení proměnných prostředí (Unix a MS Windows)

```
1 int main(int argc, char **argv, char **envp) { ... }
```

Přístup k proměnným prostředí funkcí **getenv()** z knihovny **stdlib.h**.

- Rozšířená funkce o specifické parametry Mac OS X

```
1 int main(int argc, char **argv, char **envp, char **apple);
```

Argumenty funkce `main`

- Základní tvar funkce `main`

```
1 | int main(int argc, char *argv[]) { ... }
```

- `argc` – obsahuje počet argumentů programu

  Včetne jména spouštěného programu.

  - Argumenty jsou textové řetězce oddělené mezerou (bílým znakem).
- `argv` – pole ukazatelů na hodnoty typu `char`
  - Pole `argv` má velikost (počet prvku) daný hodnotou `argc`
  - Každý prvek pole `argv[i]` obsahuje adresu, kde je uložen textový řetězec argumentů (tj. typ `char*`)
  - Textový řetězec (argument) je posloupnost znaku (typ `char`) zakončený znakem `'\0'`
  - Alokace paměti pro uložení argumentů (textových retězců) je provedena při spuštení programu

Argumenty programu

- Programu můžeme při spuštění předat argumenty příkazové rádky

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Pocet argumentu %i\n", argc);
  for (int i = 0; i < argc; ++i) {
    printf("argv[%i] = %s\n", i, argv[i]);
  }
  return argc > 0 ? 0 : 1;
}
```

- Voláním `return` ve funkci `main()` vracíme z programu návratovou hodnotu, se kterou můžeme dále pracovat

```
$ ./a.out >/dev/null; echo $?
$ 1
$ ./a.out first >/dev/null; echo $?
$ 0
```