Strukturovaná data

Karel Richta a kol.

katedra počítačů FEL ČVUT v Praze

Přednášky byly připraveny s pomocí materiálů, které vyrobili Ladislav Vágner, Pavel Strnad, Martin Hořeňovský, Aleš Hrabalík

Programování v C++, A7B36PJC
09/2015, Lekce 3

https://cw.fel.cvut.cz/wiki/courses/a7b36pjc/start

Reprezentace složitějších dat

- Dosud jsme pracovali se základními datovými typy.
- Způsob uložení triviálních informací není v C a C++ určen striktně, jen jsou stanovena určitá pravidla a závisí na počítači a operačním systému.
- Nyní se budeme zabývat strukturováním dat.

Odvozené (strukturované) typy:

- pole
- struktura (záznam)
- třída
- union (sjednocení)

Odvozené (strukturované) typy

- **Pole**
  - jednorozměrná pole prvků libovolného typu (kromě funkce)
  - indexováno vždy od 0
- **Struktura** (záznam)
  - obsahuje pojmenované položky různých typu uložené za sebou, implicitně přístupné
- **Třída**
  - obsahuje pojmenované položky různých typu uložené za sebou, implicitně nepřístupné zvenku
- **Union** (sjednocení)
  - pojmenované položky různých typů uložené "přes" sebe

Funkce a ukazatelé

- **Funkce**

  specifikuje funkci typem návratové hodnoty a typy parametrů

- **Ukazatel**

  adresa datového objektu (proměnné nebo konstanty) nebo funkce specifikovaného typu

- **Klasifikace typů**:
  - aritmetické typy: celočíselné + racionální
  - skalární typy: aritmetické + ukazatelé

Popis typu struktura

- Syntaxe:

  struct *značka* { *seznam popisů položek* }

- Popisy položek mají podobný tvar, jako deklarace
  - nesmí obsahovat paměťovou třídu
  - položka nesmí být typu funkce nebo stejná struktura
  - položka muže být ukazatelem na funkci nebo stejnou strukturu

- Příklad:

```
struct osoba {
  char jmeno[20];
  char prijmeni[25];
  int rok_narozeni;
};  /* deklarace struktury */
struct osoba Jan;
/* deklarace proměnné typu struct osoba */
```

Popis typu struktura (pokračování)

- Značka (jméno, tag) struktury není v C identifikátorem typu

```
osoba Petr;   /* v C je to chyba */
```

- Identifikátor typu lze pro strukturu zavést deklarací typu

```
typedef struct Complex {
  float Real, Imag;
} Complex;
Complex a, b;                    /* O.K. */
```

Další příklady struktur

- struktura pro binární strom nebo dvousměrně zřetězený seznam:

```
struct tnode {
        char      slovo[20];
        int       pocet;
        struct tnode *levy, *pravy;
};
```

- anonymní struktura:

```
struct { int prvni, druha; } p1[9], *p2;
```

- vzájemně odkazované struktury:

```
struct S1;
struct S2 { int Obsah; struct S1 *Dalsi; };
struct S1 { long Obsah; struct S2 *Dalsi; };
```

Příklad: Tabulka zaměstnanců

```
#define MAXZAM 10

typedef struct osoba {
   unsigned int ID;
   char jmeno[25];
   char prijmeni[36];
   float plat;
} ZAM;              /* definice typu ZAM */

ZAM tab[MAXZAM];  /* tabulka zaměstnanců */
```

Popis typu třída

- Syntaxe:

  class *značka* { *seznam popisů položek* }

- Popisy položek mají podobný tvar, jako deklarace
  - nesmí obsahovat paměťovou třídu
  - položka nesmí být typu funkce nebo stejná třída
  - položka může být ukazatelem na funkci nebo stejnou třídu
  - položka může být popisem metody
- Příklad:

```
class Osoba {
  char jmeno[20];
  char prijmeni[25];
  int rok_narozeni;
};  /* deklarace třídy */
Osoba Jan;
/* deklarace proměnné typu class Osoba */
```

Příklad: Tabulka zaměstnanců

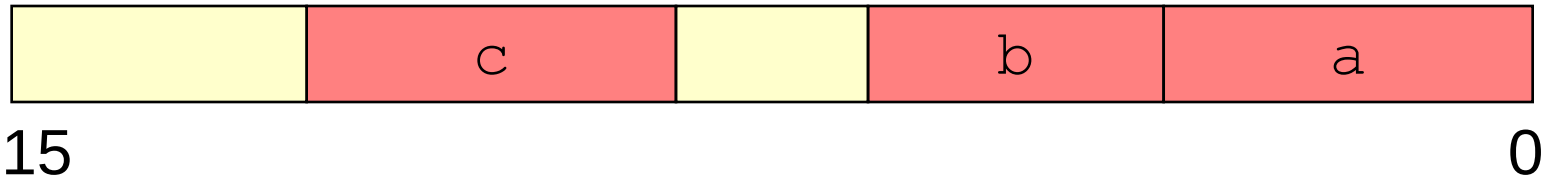
```
#define MAXZAM 10

class Osoba {
   unsigned int ID;
   char jmeno[25];
   char prijmeni[36];
   float plat;
};          /* definice třídy Osoba */

Osoba tab[MAXZAM]; /* tabulka zaměstnanců */
```

Bitová pole

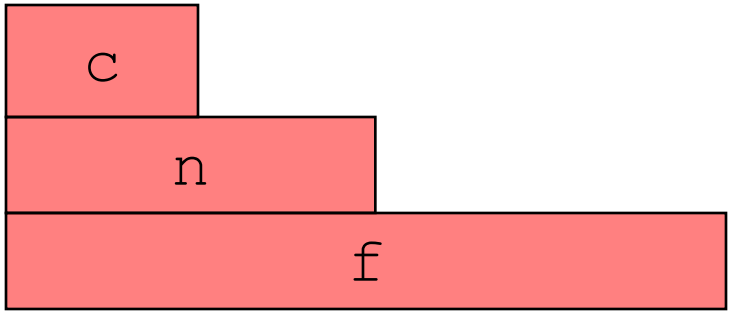
```
struct {
  unsigned a:4;
  signed b:3;
  int :2;
  int c:4;
}
```



Popis typu union

- Syntaxe stejná jako u struktury, místo **`struct`** je **`union`**
- Položky se nekladou za sebe, ale přes sebe
- Typ **`union`** se používá jako variantní záznam
- Příklad:

```
union U {
  char  c;
  int   n;
  float f;
} x;
```



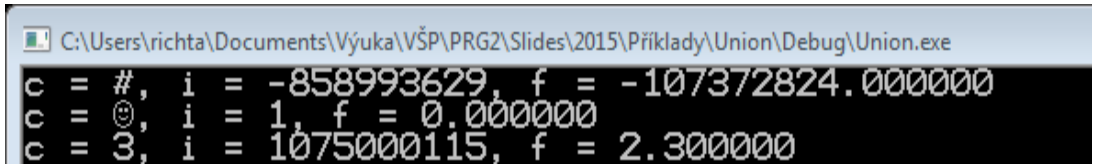
- Inicializuje se dle první položky.

```
union U z = 'a';
```

Příklad

```
#include <stdio.h>
typedef union {
  char  c;
  int   i;
  float f;
} ZN_INT_FLT;

int main() {
  ZN_INT_FLT a, *p_a = &a;
  a.c = '#';
  printf("c = %c, i = %d, f = %f \n", a.c, a.i, a.f);
  p_a->i = 1;     /* premaze znak '#' */
  printf("c = %c, i = %d, f = %f \n", a.c, a.i, a.f);
  a.f = 2.3;      /* premaze cislo 1 */
  printf("c = %c, i = %d, f = %f \n", a.c, a.i, a.f);
}
```



Deklarace funkce

- **Deklarací funkce** se deklaruje identifikátor funkce a případné typy parametrů, tělo funkce je dáno **definicí funkce**
- Parametry funkcí jsou nahrazovány **hodnotou**
- **Pořadí** vyhodnocení parametrů si určuje implementace C
- **Výsledek** funkce může být skalárního typu, struktura nebo sjednocení

Deklarace funkce (pokračování)

- Starý styl deklarace funkce v C (a jen v C) - bez specifikace parametrů - s **libovolným počtem parametrů**

```
long F();
int *G();
```

- Nebezpečné a nepoužívat, neboť při volání takto deklarované funkce se nekontroluje počet a typy parametrů!
- **Implicitní deklarace** - při volání nedeklarované funkce *F* se implicitně předpokládá deklarace:

```
extern int F();
```

- Nebezpečné, vyhýbat se!

Deklarace funkce (pokračování)

- **ANSI styl** - funkční prototyp (lépe funkční profil) - deklarace obsahuje specifikace typů a případně i jmen parametrů (jména však nejsou významná):

```
int F(void);    /* funkce bez parametrů */
void G(long x, int y);   /* procedura */
char *H(char *, char *); /* bez jmen parametrů */
```

Deklarace funkce (pokračování)

- Funkce s **proměnným počtem parametrů** (nutno použít standardní knihovnu <stdarg.h>)

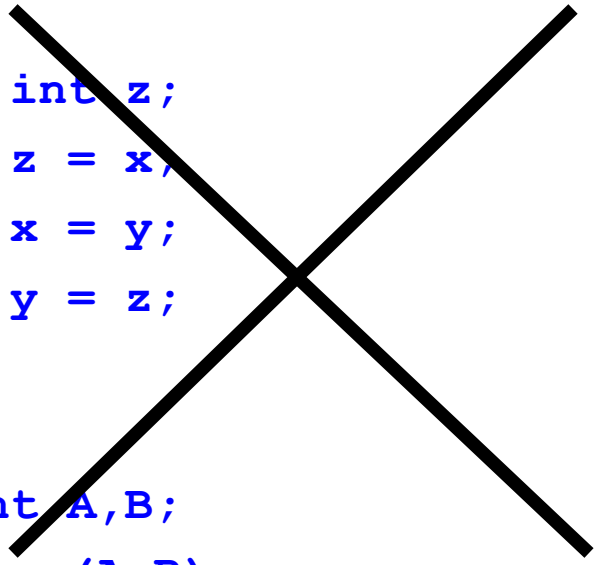
```
#include <stdarg.h>

int K(int, … ); /* proměnný počet parametrů */
int printf(char *format, … );
```

Příklad: funkce, která prohodí 2 buňky

```
void swap(int x, int y)
{
   int z;
   z = x;
   x = y;
   y = z;
}

int A,B;
swap(A,B);
```



Chybně

```
void swap(int *x, int *y)
{
   int z;
   z = *x;
   *x = *y;
   *y = z;
}

int A,B;
swap(&A,&B);
```

Správně

Cvičení: Pořadí vyhodnocení param.

```
#include <stdio.h>

void f(int x, int y, int z) {}

int p(int x)
{
  printf("Parametr: %d\n", x); return x;
}

int main()
{
  f(p(1),p(2),p(3));
  return 0;
}
```

Příklady na inicializaci polí

```
int P[3] = {1, 2, 3};
/* inicializací lze zadat počet prvků pole */
int Q[]  = {1, 2, 3};
int PP[4][3] = {{1, 3, 5}, {2, 4, 6}, {3, 5, 7}};
/* následující inicializace má stejný efekt */
int PP[4][3] = {1, 3, 5, 2, 4, 6, 3, 5, 7};
/* inicializace nemusí být úplná */
int QQ[][3] = { {1}, {2}, {3}, {4} };
/* pole má 4 řádky, inicializován je jen 1. sloupec
                    ostatní jsou vynulovány */
char Msg[] = "text";
char Msg[5] = {'t', 'e', 'x', 't', '\0'};  /* totéž */
char Mag[4] = "text"; /* bez závěrečné 0 */
```

Příklad: Inicializace tabulky

```
#define MAXZAM 10

typedef struct osoba {
   unsigned int ID;
   char jmeno[25];
   char prijmeni[36];
   float plat;
} ZAM;                  /* definice typu ZAM */

ZAM tab[MAXZAM] = {
   { 1, "Karel", "Novák", 10000.0 },
   { 2, "Jana", "Nováková", 8000.0 },
   { 3, "Dáša", "Poláková", 14000.0 },
};                      /* tabulka zaměstnanců */
int pocet = 3;          /* počet zaměstnanců */
```

Indexace

- Syntaxe: ***X* [*Y*]** , kde jeden výraz je typu ukazatel na *T* a druhý typu *int*, výsledek je typu *T*
- Jméno pole prvků typu *T* je konstantní ukazatel na *T*
- Příklady:

```
int a[10], i, *p;
a[1] = 5;
p = a;
*(p + 2) = 0;
p[2] = 0;
```

- Výraz *X* [*Y*] je ekvivalentní s *(*X* + (*Y*))
- Žádná kontrola indexů se neprovádí
- Vícerozměrná pole jsou pole polí, indexace v každém rozměru zvlášť

```
int mat[2][3];
mat[1][2] = 1;
```

- Pozor:

  mat[1,2] je dovoleno, neznamená však dvojitou indexaci!!

Volání funkce

- Syntaxe:

  F (X1, X2, … Xn)

- kde *F* je ukazatel na funkci a *X1, … Xn* jsou výrazy tvořící skutečné parametry volání
- Identifikátor funkce je konstantním ukazatelem na funkci
- Při volání funkce s libovolným počtem parametrů (starý styl) se:
  - nekontroluje počet a typy parametrů
  - skutečné parametry se konvertují pomocí roztažení (promotion) a navíc typ `float` se konvertuje na `double`
- Při volání funkce deklarované ANSI stylem se:
  - kontroluje počet parametrů (není-li funkce deklarována s proměnným počtem parametrů)
  - kontroluje přípustnost skutečných parametrů a provádějí případné konverze (podobně jako při přiřazení)
- Pozor: závorky jsou třeba, i když se jedná o funkci bez parametrů!

Volání funkce (pokračování)

Příklady:

```
long Power(long, int);
long l; int a, b; float f;
l = Power(l, 10);
a = Power(a+3, b);
f = Power(f, 3);

void Swap(int *, int *);
int x, y;
Swap(&x, &y);

void CtiPole(int, int *);
int pole[10];
CtiPole(10, pole);
```

Selekce

- Zpřístupnění položky struktury, třídy nebo unionu
- Přímá selekce:

  X . položka kde *X* je výraz typu **`struct`** nebo **`union`**
- Nepřímá selekce (přes ukazatel):

  X -> položka kde *X* je výraz typu ukazatel na **`struct`** nebo **`union`**
- *X -> položka* je zkratkou za (**X*) *. položka*
- Příklad:

```
struct { int a; char b; } x, *px = &x;
x.a = 1;
px -> b = 'a';
(*px).b = 'a';          /* totéž */
```

Definice funkce

- **Starý styl (K&R C)**:

  hlavička

  deklarace formálních parametrů

  tělo = blok

  - hlavička obsahuje paměťovou třídu, typ návratové hodnoty, jméno funkce a seznam jmen formálních parametru
  - formální parametry typu `int` není třeba deklarovat

```
long Power(x,y)
long x,y;
{
  long result = 1;
  while (y-- >= 0) result *= x;
  return result;
}
Dummy() { }
```

  - volání funkcí definovaných starým způsobem se nekontroluje
  - proto nepoužívat

Definice funkce

- **ANSI styl:**

  hlavička

  tělo = blok

  - hlavička má tvar prototypu (profilu) funkce: paměťová třída, typ návratové hodnoty, jméno funkce a seznam deklarací parametrů
  - každý parametr je deklarován zvlášť

```
long Power(long x, int z)
{
  long result = 1;
  while (y-- >= 0) result *= x;
  return result;
}
void Dummy(void) { }
```

  - typ návratové hodnoty **`void`** = funkce nevrací hodnotu (procedura)
  - **`void`** v seznamu parametrů = funkce bez parametrů
  - volání funkcí definovaných ANSI stylem se kontroluje a provádějí se případné konverze skutečných parametrů

Definice funkce

- **Proměnný počet parametrů:**

  hlavička

  tělo = blok

  - hlavička: paměťová třída, typ návratové hodnoty, jméno funkce a seznam deklarací pevných parametrů zakončený …

```
#include <stdarg.h>
long Max(int n, …)
{
  long result = 0; va_list pt;
  va_start(pt,n);
  while (n-- >= 0) result += va_arg(pt,int);
  va_end(pt);
  return result;
}
long l = Max(4,2,45,3,17);
```

  - při volání funkcí s proměnným počtem parametrů se kontrolují pevné parametry, proměnné parametry se samozřejmě kontrolovat nedají

Parametry programu

- Funkce **`main`** může být bez parametrů:

  int main(void) { … }

- Funkce **`main`** může mít parametry:

  int main(int počet, char *slova[]) { … }

- 1.parametr udává počet slov v příkazovém řádku (slova jsou oddělena mezerou)
- 2.parametr je seznam slov

Příklad: při vyvolání programu copy.exe

>copy muj.txt tvuj.txt

dostane funkce **`main`** parametry:

počet == 3

slova[0] == "copy” slova[1] == "muj.txt"

slova[2] == "tvuj.txt” slova[3] == NULL

Parametry programu (pokr.)

Příklad: správné řešení programu copy.exe s parametry:

```
#include <iostream>

int main(int argc, char *argv[])
{
  if (argc < 2)
        std::cout <<
                "Chybne volani copy, spravne: copy vstup vystup\n";
  else
        std::cout <<
                "Spravne volani copy ve tvaru: copy " <<
                        argv[1] << " " << argv[2] << "\n";

  system("pause");
  return 0;
}
```

The End