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, Martin Mazanec

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

Popis typu struktura

- Syntaxe:

  `struct` *značka* { *seznam popisů položek* };

Středník je nutný!

- 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 {
        std::string jmeno;
        std::string prijmeni;
        int rok_narozeni;
}; /* deklarace struktury */
Osoba Jan;
/* deklarace proměnné typu Osoba */
```

Popis typu struktura (pokračování)

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

```
Osoba Petr; /* chyba v C */
struct Osoba Petr; /* správně v C i C++ */
```

- Identifikátor typu lze pro strukturu zavést deklarací typu

```
typedef struct {
        float Real, Imag;
} Complex;

Complex a, b; /* OK */
```

Další příklady struktur

- struktura pro binární strom nebo dvousměrně zřetězený seznam:

```
struct TNode {
  std::string slovo;
  int pocet;
  TNode *levy, *pravy;
};
```

- anonymní struktura:

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

- vzájemně odkazované struktury:

```
struct S1; /* neúplná deklarace, lze použít jen pro
vytvoření ukazatelů a referencí */
struct S2 { int Obsah; struct S1 *Dalsi; };
struct S1 { long Obsah; struct S2 *Dalsi; };
```

Příklad: Tabulka zaměstnanců

```
const int MAXZAM = 30;

struct Osoba {
        unsigned int ID;
        std::string jmeno;
        std::string prijmeni;
        float plat;
}; /* definice struktury Osoba */

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

Příklad: Tabulka zaměstnanců

```
const int MAXZAM = 30;

typedef struct {
        unsigned int ID;
        std::string jmeno;
        std::string prijmeni;
        float plat;
} Osoba; /* definice typu Osoba */

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

Selekce

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

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

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

```
struct { int a; char b; } x, *px = &x;
x.a = 1;
x.b = 'a';
px->a = 4;
px->b = 'd';
(*px).b = 'd'; /* totéž */
```

Popis typu třída

- Syntaxe:

  `class` ***značka*** **{** ***seznam popisů položek*** **};**

Středník je nutný!

- 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 {
      std::string jmeno;
      std::string prijmeni;
      int rok_narozeni;
};/* deklarace třídy */
Osoba Jan;
/* deklarace proměnné typu class Osoba */
```

Příklad: Tabulka zaměstnanců

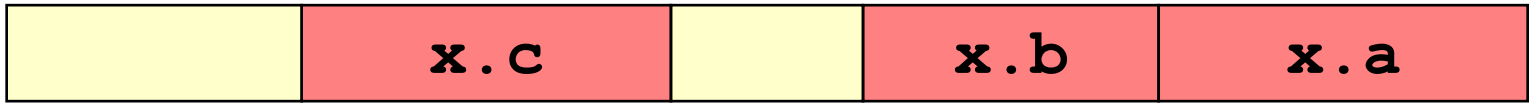
```
const int MAXZAM = 30;

class Osoba {
        unsigned int ID;
        std::string jmeno;
        std::string prijmeni;
        float plat;
}; /* definice třídy Osoba */

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

Bitová pole

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



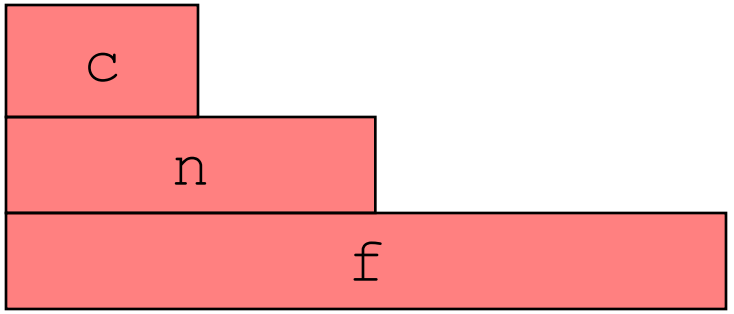
Příklad: bitová pole

```
struct TCP_Header {
    unsigned Header_Length : 4;
    unsigned : 6; // nevyužito
    unsigned URG : 1;
    unsigned ACK : 1;
    . . .
    unsigned FIN : 1;
    unsigned Window_Size : 16;
};
```

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.

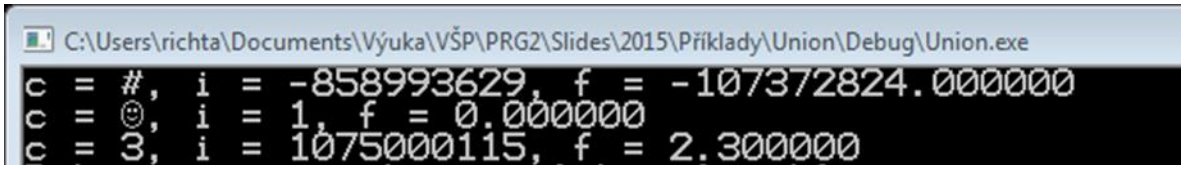
```
union U z = 'a';
```

Příklad (jen v C, v C++ to nejde)

```c
#include <stdio.h>

typedef union {
        char  c;
        int   i;
        float f;
} ZN_INT_FLT;

int main() {
        ZN_INT_FLT a;
        a.c = '#';
        printf("c = %c, i = %d, f = %f \n", a.c, a.i, a.f);
        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);
        return 0;
}
```



Výčtový typ

- Syntaxe:

```
enum značka { seznam literálů };
```

- Příklad:

```
enum Color { Red, Green, Blue };
```

- Literály jsou synonyma celočíselných hodnot

```
/* Red = 0, Green = 1, Blue = 2 */
```

  - Takto zavedená jména jsou globální konstanty – nelze tedy v různých výčtech používat stejná jména.

Výčtový typ (pokračování)

- Literálu lze explicitně přiřadit hodnotu:

```
enum Masky { Nula, Jedna, Dva, Ctyri = 4, Osm = 8 };
/* Dva = 2, Ctyri = 4, Osm = 8 */

enum Sign { Minus = -1, Zero, Plus };
/* Minus = -1, Zero = 0, Plus = 1 */

enum SelfRef { E1, E2, E3 = 5, E4, E5 = E4 + 10, E6 };
/* E4 = 6, E5 = 16, E6 = 17 */
```

- Výčtový typ je kompatibilní s `int` i co do délky vnitřní reprezentace:

```
enum Sign s; /* totéž co int s; */
```

Výčtový typ jako třída (C++)

- V C++ je v zájmu bezpečnosti možno zavést výčtový typ jako třídu:

```
enum class Colors { Red, Blue, Green };
```

- Hodnoty typu **Colors** nejsou kompatibliní s typem **int**, ani implicitně konvertované na **int** – jsou to prostě hodnoty typu **Colors**.
- Každá hodnota výčtové třídy se musí uvádět spolu s kvalifikátorem:

```
Colors mycolor;

mycolor = Colors::blue;
if (mycolor == Colors::green) mycolor = Colors::red;
```

- Hodnoty výčtových tříd mohou být reprezentovány různými základními typy, dle rozsahu výčtu. Např.:

```
enum class EyeColor : char { blue, green, brown };
```

- Pak **EyeColor** je typ se stejnou velikostí jako **char** (1 byte) (ale nikoliv kompatibilní s **char**).

Pole

- Homogenní složený objekt, prvky uloženy za sebou a odkazovány indexy.
- Indexy se počítají od 0 do N-1, kde N je rozměr pole (počet prvků).
- Samotný identifikátor pole se převádí na typ ukazatele na první prvek (má význam adresy prvního prvku) např.:

```
int A[10];
```

- Pole A má deset prvků typu `int`:

```
A[0], … , A[9]
```

- Samotné A je typu `int*` a má hodnotu &A[0].

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!

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 – chybějící prvky se vždy vynulují */
char Msg[] = "text";
char Msg[5] = { 't', 'e', 'x', 't', '\0' };  /* totéž */
char Mag[4] = "text";/* bez závěrečné 0 */
```

Příklad: Inicializace tabulky

```
const int MAXZAM = 30;

struct Osoba {
        unsigned int ID;
        std::string jmeno;
        std::string prijmeni;
        double plat;
}; /* definice třídy Osoba */

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

int pocet = 3;/* počet zaměstnanců */
```

Řetězce v C++

- Patří mezi datové typy z prostoru std.
- Knihovna má hlavičkový soubor **<string>**

```
#include <string>
using std::string;
string s = "Ahoj";

// nebo

#include <string>
using namespace std;
string s = "Ahoj";

// nebo

#include <string>
std::string s = "Ahoj";
```

Příklad práce s řetězci

```
#include <iostream>
#include <string>
using namespace std;

int main() {
	char question1[] = "What is your name? ";
	string question2 = "Where do you live? ";
	char answer1[80];
	string answer2;
	cout << question1;
	cin >> answer1;
	cout << question2;
	cin >> answer2;
	cout << "Hello, " << answer1;
	cout << " from " << answer2 << "!\n";
	return 0;
}
```

```
What is your name? Homer
Where do you live? Greece
Hello, Homer from Greece!
```

Příklad práce s řetězci

```
// operace zřetězení
#include <iostream>
#include <string>

int main()
{
   std::string str("ahoj!");
   str = str + str;
   str = str + str;
   std::cout << "str contains: " << str << '\n';
   return 0;
}
```

```
Výstup:
str contains: ahoj!ahoj!ahoj!ahoj!
```

Příklad práce s řetězci

```
// string::copy – kopírování řetězce
#include <iostream>
#include <string>

int main()
{
   char buffer[20];
   std::string str("Test string...");
   std::size_t length = str.copy(buffer, 6, 5);
   buffer[length] = '\0';
   std::cout << "buffer contains: " << buffer << '\n';
   return 0;
}
```

Kolekce

- Součástí standardní knihovny C++.
- Typickým zástupcem je **`vector`**.
- Obecně slouží pro uložení určitého množství objektů, liší se podle způsobu ukládání a vybírání objektů:

  `bitset, deque, list, map, multimap, multiset, queue, priority_queue, set, stack`

- Kromě třídy vektor nemají přístup k prvkům přes závorky (indexy).

Vector (C++)

- Patří mezi datové typy z prostoru std – **`std::vector`** .
- Datový typ reprezentující kolekci hodnot stejného typu, které jsou uloženy sekvenčně za sebou, ale lze je vybírat přímým přístupem.
- Nachází se v hlavičkovém souboru **`<vector>`**

```
#include <vector>
using std::vector;
```

- Prvky vektorů mohou být objekty různých typů, využívá se šablon, které budeme probírat podrobně později.
- Příklad:

```
vector<int> vectorInt;
```

- Inicializace:

```
vector<int> v2(v1); // inicializuje v2 obsahem v1
vector<int> v3(6, 4); /* v3 bude obsahovat 6 prvků s hodnotou 4 */
```

Definice vektoru a operace s ním

- Příklad:

```
vector<int> v1;
vector<string> v2(v1); // Chyba
vector<int> v3(n, 20);
vector<string> v4(10); // Co tam bude?
```

- Vektory jsou dynamické, není třeba definovat jejich velikost při definici.
- Operace s vektorem:

```
v.empty()    /* test na prázdnost, vrací bool */
v.size()     /* velikost vektoru - vrací datový typ vector::size_type
*/
v.push_back()       /* vložení prvku na konec vektoru */
v[n]  /* výběr n-tého prvku */
v1 = v2      /* přiřazení vektorů – do položek v1 se kopírují položky
v2 */
v1 == v2     /* porovnání vektorů na rovnost */
!= , <, <= , >, >= /* relační operace s vektory */
```

- Pozor!

```
vector<int> ivec; // Prazdny vektor
for (vector<int>::size_type ix = 0; ix != 10; ++ix)
ivec[ix] = ix; // Chyba
```

- Přístup přes závorku nepřidá nové prvky!

Příklad: Inicializace tabulky pomocí vektoru

```
#include <vector>

const int MAXZAM = 30; /* není třeba */

struct Osoba {
        unsigned int ID;
        std::string jmeno;
        std::string prijmeni;
        double plat;
}; /* definice třídy Osoba */

std::vector<Osoba> vec = {
        { 1, "Karel", "Novák", 10000.0 },
        { 2, "Jana", "Nováková", 8000.0 },
        { 3, "Dáša", "Poláková", 14000.0 },
}; /* tabulka zaměstnanců */

int pocet = 3; /* není třeba, používej vec.size() */
```

Iterátory

- Slouží jako obecný přístup k jednotlivým prvkům uložených v kolekcích.
- Nejsou konkurencí přístupu přes závorky.
- Datový typ iterátoru je dán typem kolekce:

```
vector<typ>::iterator iter;
```

- Získání iterátoru:

```
vector<int> vec(10);
vector<int>::iterator i1 = vec.begin();
vector<int>::iterator i2 = vec.end();
```

- Dereference iterátoru:

```
cout << *i1;        // cout << vec[0];
cout << *(i1 + 1); // cout << vec[1];
cout << *(i1 + 9); // cout << vec[9];
cout << *i2;        // chyba, vec.end() je za posledním prvkem vec
cout << *(i2 - 1); // cout << vec[9];
*i1 = 100;          // vec[0] = 100;
```

Operace s iterátory

Základní operace s iterátory jsou:

```
!= , ++, *
```

Procházení kolekce pomocí iterátoru:

```
for (vector<int>::iterator i1 = v.begin(); i1 != v.end(); i1++)
{
   *i1 = 5; // ok
   cout << *i1 << endl;
}
```

Nebo lépe:

```
for (auto i1 = v.begin(); i1 != v.end(); i1++)
{
   *i1 = 5; // ok
   cout << *i1 << endl;
}
```

Příklad práce s iterátorem vektoru

```
#include <vector>
#include <iostream>
using namespace std;

int main() {
   vector<int> ivec; // prázdný vektor
   ivec.push_back(5);
   ivec.push_back(2);
   ivec.push_back(11);
   ivec.push_back(68);

   for (auto i1 = ivec.begin(); i1 != ivec.end(); i1++) {
      cout << *i1 << endl;
   }

   return 0;
}
```

Další operace s iterátory

Operace s iterátory

```
== , !=
i1++, ++i1, i1--, --i1
i2 + n, i2 - n
i1 - i2 // oba iteratory patri stejne instanci
```

Procházení kolekce pomocí iterátoru:

```
for (vector<int>::iterator i1 = v.begin();
        i1 != v.end(); i1++)
{
   *i1 = 5; // ok
   cout << *i1 << endl;
}
```

Procházení vektoru - const_iterator

Slouží čistě pro prohlížení:

```
for (vector<int>::const_iterator i1 = v.cbegin();
   i1 != v.cend(); i1++)
{
   *i1 = 5; // chyba
   cout << *i1 << endl;
}
```

The End