5. Reference, lambda funkce, přesouvání, RAII

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Část I

Další konstrukce C++

I. Další konstrukce C++

Výjimky

- V programování se výjimkou rozumí nějaká neočekávaná událost, která může vést i ke zhroucení samotného programu.
- Výjimky v C++
  - kód, ve kterém může nastat výjimka, uzavřeme do bloku `try`
  - pokud bloku `try` dojde k nějaké nečekané situaci
    - řízení programu se předá do bloku `catch`
    - bloků `catch` může být více
  - pokud k žádné výjimce nedojde, blok `catch` se přeskočí

dělení nulou, zápis za konec pole, čtení neexistujícího souboru, ...

```
1  try {
2    // nějaký kód, který může způsobit výjimku
3  }
4  catch (typ vyjimky) {
5    // kód, který se provede při vyvolání výjimky
6  }
```

Výjimky – příklad

```
1  #include <iostream>
2  #include <exception>
4  int main() {
5    double a, b;
7    std::cin >> a >> b;
9    try {
10     if (b == 0) throw "Deleni nulou.\n";
11     std::cout << a / b << std::endl;
12   }
13   catch (const char* e) {
14     std::cout << "Vyjimka - " << e;
15   }
17   return 0;
18 }
```

Ošetření více výjimek

- V programu může dojít k více výjimkám
- Je možné napsat více bloků `catch` pro různé výjimnky.
  - Výjimky jsou rozlišeny svým datovým typem
- Existuje blok `catch(...)`, který dokáže zachytit všechny výjimky.

```
1  try {
2    NejakaNebezpecnaFunkce();
3  }
4  catch (int) {
5    // zachycení výjimky datového typu int
6  }
7  catch (...) {
8    // zachycení všech neošetřených výjimek
9  }
```

Standardní výjimky v C++

C++ definuje třídu standardních výjimek, včetně interface a datových typů.

- `<exceptions>`
  - `std::exception` – bázová třída
  - `virtual what()` – vrací řetězec s popisem výjimky
  - `terminate()` – ukončuje program, volá `abort()`
- `<stdexcept>`
  - `std::out_of_range` – přístupu mimo rozsah
  - `std::overflow_error` – přetečení
- `<new>`
  - `std::bad_alloc` (memory allocation fails)
  - `std::nothrow`

Definice vlastního chování systému výjimek

```
1  void myunexpected () {
2    cerr << "unexpected called\n";
3    throw 0;      // throws int - ošetřeno výjimkou
4  }
6  void myfunction () {
7    throw 'x';    // throws char
8  }
10 int main (void) {
11   set_unexpected (myunexpected);
12   try { myfunction(); }
13   catch (int) { cerr << "caught int\n"; }
14   catch (...) { cerr << "caught other exception type\n"; }
15   return 0;
16 }
```

Co když není výjimka zachycena?

```
1  void myFunction()
2  {
3    std::cerr << "Nezachycena vyjimka.\n";
4    std::cerr << "Program bude ukoncen.\n";
5    exit(1);
6  }
8  int main()
9  {
10   std::set_terminate(myFunction);
11   throw 1;
12   return 0;
13 }
```

Výjimka při alokaci dynamické paměti

```
try {
  while (true) { // throwing overload
    new int[100000000ul];
  }
} catch (const std::bad_alloc& e) {
  std::cout << e.what() << '\n';
}
while (true) { // non-throwing overload
  int* p = new (std::nothrow) int[100000000ul];
  if (p == nullptr) {
    std::cout << "Allocation returned nullptr\n";
    break;
  }
}
```

I. Další konstrukce C++

Výjimky

Reference

Přetížené operátory

Lambda funkce

Reference

- slabší, ale bezpečnější forma ukazatele
- nemůže být NULL (C++11: `nullptr`)
- nemůže se přesměrovat jinam
- musí být vždy inicializovaná
- funguje jako alias
- vytváří se automaticky přiřazením do referenčního typu nebo voláním funkce

```
1  int a = 1;
2  int b = 2;
3  int& ref = a; // ref je reference na a
4  // cokoli provedeme s ref, jako bychom provedli s a
5  ref = b;
6  ref += 7; /* jaka je ted hodnota a, b? */
```

Reference vs. ukazatele

- Ukazatel může být neinicializovaný

```
1 int* ptr;
2 int& ref; /* chyba pri kompilaci */
```

- Ukazatel může být `nullptr`

```
1 int* ptr = nullptr;
2 int& ref = nullptr; /* chyba pri kompilaci */
```

- Ukazatel se může přesměrovat

```
1 int a = 1;
2 int b = 2;
3 int* ptr = &a;
4 ptr = &b;
5 int& ref = a;
6 ref = b; /* v cem je rozdil? */
```

Reference

Používat reference nebo ukazatele?

- pokud možno dávejte přednost referencím
- ukazatele používejte jen tam, kde dává smysl nullptr a přesměrování
- pro dynamickou alokaci jsou vhodnější chytré ukazatele (o těch později)

Na co si dát pozor?

- reference neřeší problém životnosti
- nedržet si referenci na něco, co už přestalo existovat
- nevracet z funkcí reference na lokální proměnné

Klíčové slovo `const` 1/2

- **Připomenutí:** const a ukazatele

```
1  int a = 1, b = 2;
2  int* ptrX = &a;
3  const int* ptrY = &a;
4  int* const ptrZ = &a;
```

- Které řádky obsahují chybu?

```
1  ptrX = &b; /* A */
2  *ptrX = 7; /* B */
3  ptrY = &b; /* C */
4  *ptrY = 7; /* D */
5  ptrZ = &b; /* E */
6  *ptrZ = 7; /* F */
```

Klíčové slovo `const` 2/2

- Použití s referencemi

```
1  int a = 1;
2  const int& cref = a;
3  std::cout << cref << '\n'; // čtení je OK
4  cref = b; /* zapis/modifikace je chyba pri kompilaci */
```

Význam konstantních referencí

- deklarujeme záměr neměnit proměnnou (read-only)
- zároveň ale nevytváříme kopii
  - vhodné např. pro předávání větších objektů do funkcí
  - nedává smysl pro primitivní typy a malé objekty
- viděli jsme minule (string, vector)

Konstantní metody 1/2

```
1  class Person {
2    int age;
3  public:
4    void setAge(int a) { age = a; }
5    int getAge() { return age; }
6  };
8  void f(const Person& person) {
9    std::cout << person.getAge() << '\n';
10 }
12 int main() {
13   Person john;
14   john.setAge(20);
15   f(john);
16 }
```

Konstantní metody 2/2

- pokud metoda nehodlá měnit vnitřní stav objektu, musíme ji takto označit
- klíčové slovo const za hlavičkou metody

```
1  class Person {
2    // ...
3    int getAge() const { return age; }
4  };
```

- pokus o změnu stavu objektu v const metodě ohlásí překladač jako chybu

Konstatní atributy

- atribut se dá nastavit pouze v inicializační sekci konstruktoru
- potom už se nedá vůbec změnit

```
1  class Person {
2    std::string name;
3    int age;
4    const std::string genome;
5    // ...
6  };
```

- neměnné objekty
- neměnné části objektů (větší bezpečnost)

Kde používat `const`?

- všude, kde dává smysl (tj. skoro všude)
- pište `const` všude a jen tehdy, pokud zjistíte, že danou proměnnou (atribut, referenci) budete chtít měnit, `const` smažte
- pište `const` ke všem metodám a jen tehdy, pokud zjistíte, že chcete, aby daná metoda měnila stav objektu, `const` smažte

Proč?

- překladač vás hlídá → větší bezpečnost
- jasně deklarujete svůj záměr → větší čitelnost kódu
- překladač může využít pro optimalizace

Kde se spíš nepoužívá?

- nepište je k návratovým hodnotám předávaným hodnotou
- nebývá zvykem je psát k parametrům funkcí předávaným hodnotou

Problém s kopírováním 1/2

- V čem je problém s následujícím kódem (pokud vůbec)?

```
1  class Person {
2    std::string name;
3  public:
4    Person(std::string name) : name(name) {}
5    void rename(std::string newName) {
6      name = newName;
7    }
8  };
```

Problém s kopírováním 2/2

- Klasické řešení (zcela dostatečné)

```
// ...
void rename(const std::string& newName) {
  name = newName;
}
```

- Moderní řešení (od C++11)

```
// ...
void rename(std::string newName) {
  name = std::move(newName);
}
```

- Výhoda: pokud je vstupem dočasný objekt, neděje se žádná kopie
- Ještě se k tomu dostaneme později

I. Další konstrukce C++

Výjimky

Reference

Přetížené operátory

Lambda funkce

Přetížené operátory

- Pro přetěžování operátorů jsou jistá omezení
  - přetíženy mohou být jen existující operátory,

    Není možné zavést nový operátor, např. operátor $.
  - aritu, asociativitu a prioritu operátorů nelze změnit,

    Např. operátor += musí být binární.
  - nelze přetěžovat operátory vestavěných typů.

    Nelze změnit způsob sčítání čísel typu integer.
- Přetížení je třeba si dobře rozmyslet
  - přetížené operátory mají význam v matematice (např. naše komplexní čísla, velká čísla, vektory, ...) a řetězce (konkatenace – spojování),
  - kolekce (jako vektory, množiny, tabulky, ...) používají přetížené operátory pro přístup k uloženým hodnotám,
  - jiné třídy mohou vyžadovat přetížené operátory `<<`, `>>` a `=`,
  - jste-li na pochybách, dejte přednost metodě před přetíženými operátory.

Přehled operátorů

- Operátory, které lze přetížit

`+`	`-`	`*`	`/`	`%`	`^`	`&`	`|`
`~`	`!`	`=`	`<`	`>`	`+=`	`-=`	`*=`
`/=`	`%=`	`^=`	`&=`	`|=`	`<<`	`>>`	`>>=`
`<<=`	`==`	`!=`	`<=`	`>=`	`&&`	`||`	`++`
`--`	`->*`	`,`	`->`	`[]`	`()`	`new`	`delete`
`new []`	`delete[]`						

- Operátory, které nelze přetížit

`.`	`.*`	`::`	`?:`	`sizeof`

Binární operátory

`+, -, *, ..., +=, -=, ...`

- zápis operace: `x @ y`
- signatura: `T1 @ T2 -> T3`
- metoda v T1: `T3 operator @ (T2)`
- alt. volání: `x.operator @ (y)`
- funkce: `T3 operator @ (T1, T2)`
- alt. volání: `operator @ (x, y)`

Přetížení operátoru `@` automaticky nepřetíží operátor `@=`. Je-li to požadováno, operátor musí být přetížen separátně.

Příklad přetížení binárního operátoru

```
1  struct cplx {
2    float im, re;
4    cplx operator+(cplx & a) {
5      cplx tmp;
6      tmp.im = this->im + a.im;
7      tmp.re = this->re + a.re;
8      return tmp;
9    }
10 };
12 cplx operator-(cplx & a, cplx & b) {
13   cplx tmp;
14   tmp.re = a.re - b.re;
15   tmp.im = a.im - b.im;
16   return tmp;
17 }
```

Unární prefixové operátory

`+, -, *, &, ~, !, ++, --`

- zápis operace: `@ x`
- signatura: `@ T1 -> T2`
- metoda v T1: `T2 operator @ ()`
- alt. volání: `x.operator @ ()`
- funkce: `T2 operator @ (T1)`
- alt. volání: `operator @ (x)`

Příklad přetížení prefixového unárního operátoru

```
1  struct cplx
2  {
3    float im, re;
4    // komplexně sdružené číslo
5    cplx & operator~() {
6      this->im = -1 * this->im;
7      return *this;
8    }
9  };
10 // test nulovosti
11 bool operator!(cplx & a) {
12   return a.re == 0 && a.im == 0;
13 }
```

lec05/02-pretizeni.cpp

Unární postfixové operátory

`++, --`

- zápis operace: `x @`
- signatura: `T1 @ -> T2`
- metoda v T1: `T2 operator @ (int)`
- alt. volání: `x.operator @ (0)`
- funkce: `T2 operator @ (T1, int)`
- alt. volání: `operator @ (x, 0)`

Přiřazení

- zápis operace: `x = y`
- signatura: `T = T1 -> T`
- metoda v T1: `T & operator = (const T1 &)`
- alt. volání: `x.operator = (y)`
- funkce: -
- alt. volání: -
- Obvykle bývají `T1` a `T` stejné typy.
- Návratová hodnota může být buď typu `T &`, nebo `void`.

Indexování

- zápis operace: `x[y]`
- signatura: `T1[T2] -> T3`
- metoda v T1 (a): `T3 & operator [] (T2)`
- metoda v T1 (b): `const T3 & operator [] (T2) const`
- alt. volání: `x.operator [] (y)`
- funkce: -
- alt. volání: -
- První způsob přetížení je určen pro nekonstantní objekty a umožňuje použít výsledek jako l-value (tzn. na levé straně přiřazení).
- Druhý způsob přetížení je určen pro konstantní objekty, výsledek není modifikovatelný.

Dereference

- zápis operace: `x -> met`
- signatura: `T -> T1*`
- metoda v T: `T1* operator -> ()`
- alt. volání: `(x.operator -> ()) -> met`
- funkce: -
- alt. volání: -

Volání funkce

- zápis operace: `x (x1, x2, ..., xn)`
- signatura: `T (T1, T2, ..., Tn) -> Tx`
- metoda v T: `Tx operator () (T1, T2, ..., Tn)`
- alt. volání: `(x.operator () (x1, x2, ..., xn))`
- funkce: -
- alt. volání: -

I. Další konstrukce C++

Příklad – podmíněné kopírování mezi dvěma vektory

```
1  struct positive {
2    bool operator () (int i) {
3      return i>0;
4    }
5  }
6  // chceme kopírovat pouze kladná čísla
7  void copy_positive (std::vector<int> & a, std::vector<int> & b) {
8      ...
9      std::copy_if (a.begin(), a.end(), b.begin(), positive());
10 }
```

lec05/03-copy_if.cpp

- funkce `positive` je zbytečně vidět mimo `copy_positive`

V C++03 je možné definovat strukturu / třídu uvnitř funkce

- jméno funkce může kolidovat s jiným globálním indentifikátorem
- funkce je krátká, zápis její deklarace je významně delší

Lambda funkce

- Při používání STL funkcí se odkaz na jinou funkci nebo funktor používá poměrně často:
  - STL funkce `sort` potřebuje komparátor
  - STL funkce `copy_if` potřebuje funkci pro predikát
  - funkce pro vytváření vláken potřebují znát kód (funkci), kterou mají v novém vlákně spustit
- Uvedené funkce jsou typicky krátké a *jednorázové*, to platí zejména pro komparátory.
- Takovou funkci si lze deklarovat mimo místo použití, ale je to nepohodlné
- Lambda funkce jsou způsobem, kterým lze vytvořit takovou *jednorázovou* funkci:
  - funkce nemá jméno,
  - kód funkce existuje pouze v parametrech volání,
  - volaná funkce (např. sort) může lambda funkci využít jako jakoukoliv jinou funkci.

Pozn.: Budeme se zabývat jednodušší podobou lambda funkcí tak, jak byly zavedeny v C++11. Referenci pro funkcionality zavedené v pozdějších standardech může najít zvídavý programátor např. zde: `https://en.cppreference.com/w/cpp/language/lambda`

Lambda funkce – obecná syntaxe

```
[capture-list] ( formal-parameters ) { lambda-body }
```

- `capture-list` je seznam zachycených (captured) proměnných – proměnných v nadařazené funkci, které jsou dostupné i v lambda funkci
- Zachycené proměnné si zachovávají hodnotu mezi vyvoláním lambda funkce:

`[]`	nezachycují se žádné proměnné
`[x,y]`	zachycují se proměnné zadaných jmen, obsah proměnných je zkopírován
`[&x, &y]`	zachycují se proměnné zadaných jmen, obsah proměnných je odkazován
`[&x,y]`	kombinace zachycení hodnotou a odkazem
`[&]`	všechny proměnné použité v lambda funkci jsou zachycené odkazem
`[=]`	všechny proměnné použité v lambda funkci jsou zachycené hodnotou
`[this]`	zachytne this ukazatel z nadřazené metody.

Pozn.: lambda funkce si může deklarovat své lokální proměnné, jejich hodnota se mezi vyvoláním lambda funkce nezachovává.

Lambda funkce – definice 1/2

- Prázdná

```
auto empty_fnc = [] () {};
```

- S návratovou hodnotou typu int (explicitně)

```
auto meaning_fnc = [] () -> int {
  return 42;
};
```

- S návratovou hodnotou typu int (explicitně)

```
std::function <int()> meaning_fnc = [] () {
  return 42;
};
```

Lambda funkce – definice 2/2

- Automatická dedukce návratové hodnoty

```
auto meaning_fnc = [] () {
  return 42;
};
```

- Se zachycením všech proměnných hodnotou

```
auto meaning_fnc = [=] () {
  return outer_var * another_var ;
};
```

- Se zachycením všech proměnných referencí

```
auto meaning_fnc = [&] () {
  outer_var = 15;
};
```

Lambda funkce – překlad

- Pokud lambda funkce nezachytává žádné proměnné, přeloží se jako obyčejná funkce,
- Pokud lambda funkce zachytává proměnné, vymyslí pro ni kompilátor třídu funktoru:
  - Deklaruje členské proměnné pro zachycené proměnné, konstruktor inicializuje členské proměnné zachycenými hodnotami,
  - kompilátor připraví přetížený `operator()` pro tělo lambda funkce,
  - takto vytvořený `operator()` je implicitně `const`, tedy lambda funkce nesmí měnit proměnné zachycené hodnotou (toto chování lze změnit deklaraci `mutable`),
  - Návratový typ je odvozen podle typu výrazu za `return`.
- Návratový typ lambda funkce lze explicitně zadat zápisem s šipkou (trailing return type).

Lambda funkce – příklad překladu

```
1  // lambda funkce
2  auto mod_fnc = [&a, b](int c) {
3    a = b * c;
4  };
5  // odpovídající třída vytvořená kompilátorem
6  class mod_fnc {
7    public :
8      mod_fnc (int &_a , int _b) : a(_a), b(_b) {}
9      void operator ()( int c) {
10       a = b * c;
11     }
12   private :
13     int &a;
14     const int b;
15 };
```

Lambda funkce – mutable

- Dovoluje modifikovat proměnnou zachycenou hodnotou
- Typicky v kombinaci s nějakou dočasnou lokální deklarací

```
auto get_count = [n = 1]() mutable {
  return n ++;
};
std::cout << get_count () << std::endl ; // 1
std::cout << get_count () << std::endl ; // 2
std::cout << get_count () << std::endl ; // 3
```

- **Pozn.:** proměnná `n` není vně nikde deklarována, její typ je dedukován na `int`

Lambda funkce – příklad

```
1  vector<int> seq;
2  int s = 123;
3
4  // chyba
5  // lambda se pokousi ovlivnovat promennou zachycenou hodnotou
6  generate_n (back_inserter (seq), 100, [s]() {return s++;});
7
8  // ok
9  // predchozi problem se da napravit pomoci mutable
10 // hodnota promenne s se se ale nezmeni
11 generate_n (back_inserter (seq), 100, [s]() mutable {return s++;});
12 // s = 123
13
14 // ok
15 // lambda ovlivnuje promennou zachycenou referenci
16 generate_n (back_inserter (seq), 100, [&s]() {return s++;});
17 // s = 223
```

lec05/lambda-generate.cpp