4. Konstruktory a destruktory, RAII

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Část I

Konstruktory a destruktory

I. Konstruktory a destruktory

L-value R-value

Konstruktory a destruktory

Přesouvání (move sémantika)

L-value

- *locator value*, někdy také *left-hand side value*
- *z pohledu kódu* všechny identifikátory, které označují konkrétní místo v paměti
- fyzicky i jiné objekty, které mají nějaké místo v paměti, jen ho nejsme schopni identifikovat názvem
- do `l-value` jsme schopni přiřadit hodnotu
- výjimka jsou const proměnné, ale i ty jsou `l-value`

```
int a = 5;      // a je l-value
std::string b; // b je l-value
MyClass c;      // c je l-value
```

R-value

- *right-hand side value*
- dočasné a přechodné objekty
- (syntakticky) nemají adresovatelnou pamět – nemají identifikátor
- nelze do nich přiřadit

```cpp
int a = 5;              // a je l-value, 5 je r-value
std::string b = "hi";   // b je l-value, "hi" je r-value
float c = get_c();      // get_c() je/vraci r-value
5 = c;                  // ouch !
```

L-value reference

- reference na `l-value`
- nekonstantní nelze navázat na `r-value`

```
int a = 5;
int & ra = a; // ok
int & rb = 5; // ouch !
```

- konstantní `l-value` reference – také reference na `l-value`
- pri vázání na `r-value` ji díky `const` kompilátor převede na `l-value`

```
const int& rb = 5; // ok
void my_func(const int& a) {}
// ...
my_func(a); // ok
my_func(10); // ok
```

R-value reference

- reference na `r-value`
- nelze vázat na `l-value`
- platnost má jako běžná reference, jen nemusí být `const` pro `r-value`
- uvození `&&`
- dovoluje fungování move sémantiky (přesouvání – viz další část přednášky)

```
int && rb = 5; // ok
void my_func(int && a) {
  //...
}
my_func(a)    // ouch !
my_func(10); // ok
```

R-value reference

- dovoluje rozlišit dočasné a *trvalé* instance

```
void my_func(int && a) {
  // pro dočasné instance
}

void my_func(const int& a) {
  // pro trvalé instance
}

my_func(a);  // ok
my_func(10); // ok
```

L-value a R-value kontext

- lze rozlišit kontext volání metody objektu
- jiné chování při volání metody nad `l-value`, než nad `r-value`
- uvození `&` pro `l-value` a `&&` pro `r-value` za signaturou metody

```
class Test {
public:
  std::string getContext() & { return "l-value\n"; }
  std::string getContext() && { return "r-value\n"; }
};
//...
Test a;
std::cout << a.getContext();      // l-value
std::cout << Test().getContext(); // r-value
```

I. Konstruktory a destruktory

L-value R-value

Konstruktory a destruktory

Přesouvání (move sémantika)

Konstruktor

Už víme, že

- konstruktor je speciální metoda volaná při inicializaci objektu
- konstruktor má tzv. inicializační sekci
- jméno konstruktoru = jméno třídy

Přetěžování konstruktorů

- jako přetěžování funkcí/metod
- (od C++11) konstruktory můžou v inicializační sekci volat jiné konstruktory (tzv. delegující konstruktory)

Pořadí volání konstruktorů

- konstruktory atributů se volají před konstruktorem objektu
- volání konstruktorů při dědičnosti (později, v přednášce o OOP)

Kopírování

Hodnotová sémantika

- Inicializace/přiřazení je kopírování.

Implicitní kopírování

- Všechny atributy jsou zkopírovány (inicializace/přiřazení)
  - to je většinou to, co chceme
  - co když chceme jiné chování? (proč chceme jiné chování?)

Kopírovací konstruktor

- Popisuje, jak se objekt kopíruje při inicializaci
- Pokud ho nedefinujeme, kompilátor pro nás připraví implicitní
- Syntax: `Object(const Object& object)`
  - proč referenci? Protože teprve definujeme, jak kopírovat.
  - proč konstantní? Protože není slušné při kopírování měnit originál.

Motivační příklad

- Mějme třídu `Vektor`, která popisuje vektor celých čísel, data uložena v dynamické paměti
- Implementace obsahuje přetížený operátor pro přístup k datům

```
1  class Vektor {
2    int * data, velikost;
3  public:
4    Vektor(int v):velikost(v) {
5      data = new int[velikost];
6      for (int i = 0; i < velikost; i++) data[i] = 0;
7    }
8    ~Vektor() { delete [] data; }
9    int & operator[] (int idx);
10 };
```

lec06/03-vektor.cpp

Motivační příklad


```
1   int & Vektor::operator[] (int idx) {
2     return data[idx];
3   }
4
5   int main()
6   {
7     Vektor a(5);
8     a[1] = 10;
10    for (int i = 0; i < 5; i++)
11      std::cout << a[i] << " ";
13    std::cout << std::endl;
15    return 0;
16  }
```

lec06/03-vektor.cpp

Motivační příklad 3/3

```cpp
int main()
{
  Vektor a(5), b(5);

  a[4] = 3;
  b = a;      // problém č. 1
  b[3] = 5;   // problém č. 2

  for (int i = 0; i < 5; i++) std::cout << a[i] << " ";
  std::cout << std::endl;
  for (int i = 0; i < 5; i++) std::cout << b[i] << " ";
  std::cout << std::endl;
  return 0;
}
```

lec06/03-vektor.cpp

Mělká kopie

- Objekt byl okopírován použitím operátoru `=`
- Třída `Vektor` nemá operátor `=` přetížený, takže si jej kompilátor vymyslí:
  - objektové členské proměnné kopíruje jejich operátorem `=`
  - primitivní datové typy (včetně ukazatelů a ukazatelů na objekty) kopíruje po bytech
- Binární kopie ukazatele na dynamicky alokovaná data referencuje stejnou paměť (tedy stejná data)
- To se nazývá mělká kopie (shallow copy).
- Mělká kopie není problémem, pokud jsme si toho vědomi.
- Destruktor ale problémem je (stejný dynamicky alokovaný blok je uvolněn dvakrát nebo i vícekrát).

Hluboká kopie

- Sémantika operátoru = (deep copy) je jasná:
  - chceme okopírovat zdrojová data (vpravo) na cílová (vlevo),
  - původní data na levé straně budou zřejmě zničena, objekty musí zůstat nezávislé.
- Kopírování adresy je nedostatečné, je třeba kopírovat obsah:
  - musí být připraven nový dynamicky alokovaný blok,
  - obsah pole musí být zkopírován, dříve alokované pole musí být zrušeno.

```
1  Vektor & Vektor::operator = (const Vektor & src) {
2    if (&src == this) return *this;
3    delete [] data;
4    velikost = src.velikost;
5    data = new int [velikost];
6    for (int i = 0; i < velikost; i++)
7        data[i] = src.data[i];
8    return *this;
9  }
```

Kopírující konstruktor

- Deklarace

```
Vektor b = a;
```

deklaruje nový objekt b a inicializuje jej obsahem objektu a.

- Deklarace je identická jako

```
Vektor b(a);
```

- V obou předchozích případech inicializace se volá kopírující konstruktor :

```
Vector(const Vector & src);
```

- Implicitní kopírující konstruktor:
  - objektové členské proměnné jsou kopírovány jejich kopírujícími konstruktory,
  - primitivní datové typy (včetně ukazatelů a ukazatelů na objekty) jsou kopírovány po bitech.

Toto je v podstatě mělká kopie objektu.

Kopírující konstruktor 2/3

```
Vektor::Vektor(const Vektor & src)
{
  velikost = src.velikost;
  data = new int [velikost];
  for (int i = 0; i < velikost; i++)
    data[i] = src.data[i];
}
```

- Konstruktor je podobný operátoru =, ale
  - chybí podmínka,
  - chybí delete.

Proč?

Kopírující konstruktor

- Kopírující konstruktor a operátor = jsou použity pro vytvoření hluboké kopie objektu.

Cílový objekt

- operátor = má na levé straně plně funkční objekt
  - zdroje alokované pro cílový objekt musí být před kopírováním uvolněny
- kopírující konstruktor nemá žádný objekt k modifikaci
  - cíl má již rezervovanou (neinicializovanou) paměť, která musí být inicializována

- Kopírující konstruktor a operátor = jsou těsně svázány s destruktorem.
- Je-li důvod pro implementaci destruktoru, je také důvod pro implementaci kopírujícího konstruktoru a operátoru = (pokud kopírování není zakázáno).

Zákaz kopírování

- Kopírování lze zakázat pomocí klíčového slova delete

```
class Objekt
{
public:
  Objekt ();
  // --
  // zakázání kopie konstruktorem
  Objekt(const  Objekt & obj) = delete;
  // zakázání kopie přiřazením
  Objekt& operator= (const  Objekt &) = delete;
};
```

Destruktory

Konec života objektů

- lokální objekty: na konci bloku
- atributy objektů: zároveň1 s koncem života objektu, kterému patří
- globální objekty: na konci programu
- dynamicky alokované objekty: do C++11 (v zásadě pouze) explicitně

Pořadí volání destruktorů

- opačné pořadí než volání konstruktorů2
- destruktory atributů se volají po destruktoru hlavního objektu

Všimněte si, že

- volání destruktorů je **deterministické**, víme přesně, kdy nastane konec života objektu
- tato vlastnost umožňuje princip RAII

Pravidla

Rule of Zero

- pokud možno, nepište kopírovací konstruktor/přiřazení ani destruktor
- vhodné pro třídy, které přímo nespravují žádný zdroj (resource)

Rule of Three

- jakmile třída spravuje nějaký zdroj, pak je typicky třeba explicitně definovat všechny tři (nebo alespoň některé zakázat):
  - kopírovací konstruktor
  - kopírovací přiřazovací operátor
  - destruktor

Rule of Five (od C++11)

- přidává se ještě přesouvací (move) konstruktor/přiřazení

I. Konstruktory a destruktory

L-value R-value

Konstruktory a destruktory

Přesouvání (move sémantika)

Motivace

- Uvažujme `std::vector` obsahující složité objekty (např. řetězce). Pokud se vyčerpá kapacita, musí:
  - alokovat nové větší pole pro objekty,
  - zkopírovat objekty ze starého pole do nového pole jejich kopírujícími konstruktory,
  - uvolnit původní pole (zavolá destruktury všech původních objektů).
  - operace bude velmi režijně náročná.
- Není kopírování řetězců hloupé?
  - řetězec typicky obsahuje informaci o kapacitě, využité délce a ukazatel na pole znaků (dynamicky alokované),
  - rozšíření pole řetězců provede hlubokou kopii, tedy kopíruje celé řetězce včetně polí znaků,
  - následně jsou staré řetězce likvidovány (včetně polí znaků),
- Nešlo by pole znaků recyklovat,
- Nestačila by zde varianta mělké kopie?

Přesouvací konstruktor

- Samotná mělká kopie ale nestačí
- Zdrojový a cílový objekt sdílí data
  - to až tak nevadí, zdroj stejně brzy zanikne,
  - ale destruktor zničí sdílená data.
- Mělkou kopii nelze použít vždy
  - někdy potřebujeme objekty skutečně kopírovat (kopírující konstruktor, deep copy),
  - pro přesouvání C++11 zavádí nový konstruktor – **přesouvací konstruktor**.
- Přesouvací konstruktor dostává parametrem zdrojový objekt:
  - použije složky zdrojového objektu pro inicializaci (zde např. ukazatel na řetězcová data),
  - zdrojový objekt *vytěží*,
  - zajistí, aby šlo zdrojový objekt bezpečně uvolnit destruktorem (zde např. ukazatele nastaví na `NULL`).

Kopírující vs. přesouvací konstruktor

```
1   class string {
2     int len, max; char * ptr;
4     string(const string & src) { // copy constructor
5       len = src.len;
6       max = src.max;
7       ptr = new char [max];
8       for (int i = 0; i <= len; i++) ptr[i] = src.ptr[i];
9     }
11    string (string && src) { // move constructor
12      len = src.len;
13      max = src.max;
14      ptr = src.ptr;
15      src.ptr = NULL; /* !!! */
16    }
17  };
```

Přesouvací konstruktor

- Přesouvací konstruktor má parametr v podobě `&&`:
  - nový C++ typ – reference na pravou stranu,
  - obdoba C++ typu reference,
  - umožní volanému takto předaný objekt *vytěžit*
- Jak se liší reference:

`&`	volající musí předat zapisovatelný objekt s paměťovou reprezentací. Nelze předávat konstantní objekt nebo dočasný objekt. Volaný může objekt číst i zapisovat.
`const &`	volající musí předat objekt s paměťovou reprezentací (i konstantní) nebo dočasný objekt. Volaný může objekt pouze číst.
`&&`	volající musí předat objekt k *vytěžení* – buď dočasný objekt nebo objekt označený k *vytěžení* pomocí `std::move`. Volaný může objekt číst i zapisovat. Objekt musí být zrušitelný destruktorem.
`const &&`	nemá využití

Přesouvací operátor =

- Obdoba operátoru =, použije se pro optimalizaci kopírování tam, kde objekt na pravé straně zaniká.

```
string & operator = (string && src) {
if (this == &src) return *this;
  delete [] m_Ptr;
  ptr = src.ptr;
  len = src.len;
  max = src.max;
  src.ptr = NULL;
  return *this;
}
// ...
string s;
s = string ("hello"); // presouvaci operator =
```

Přesouvací operátor =

- Přesouvací operátor = často doprovází přesouvací konstruktor (a naopak).
- Přesouvací a kopírující operátor = lze spojit do jednoho technikou **copy & swap**.
- Musíme ale vytvořit odpovídající přesouvací a kopírující konstruktor.

```
string (const string & src) { ... }
string (string && src) { ... }
string & operator = (string src) { // ani & nebo &&

  // != &src
  swap (ptr, src.ptr);
  swap (max, src.max);
  swap (len, src.len);
  return *this;
}
```

Přesouvací operátor `=` – copy & swap

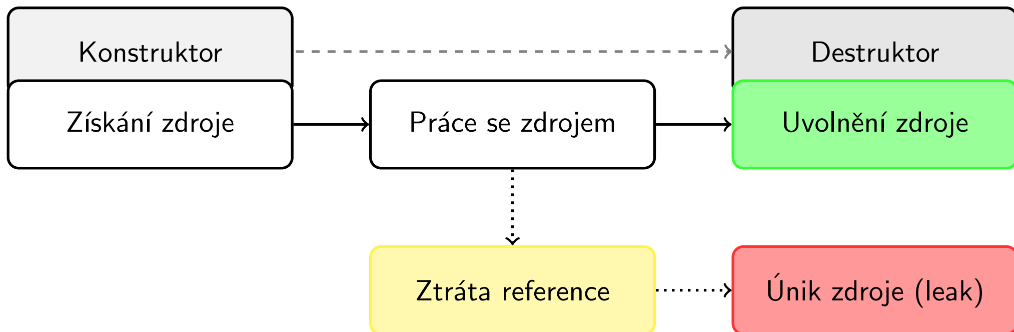
- Parametrem operátoru `=` je nová instance `src`,
- Pro její inicializaci se použije buď přesouvací nebo kopírující konstruktor,
- V těle operátoru `=` se vymění obsah
- Ukazatel `this` obsahuje přesunutý/zkopírovaný obsah pravé strany
- Parametr `src` obsahuje původní obsah instance `this`
- Po návratu z operátoru `=` se zavolá destruktor `src`, které uvolní původní nepotřebný obsah `this`,
- Díky kopírování do nové instance `src` máme jistotu, že `this != &src`

Část II

RAII

RAII – Resource Acquistion Is Initialization

- někdy též zváno scope-based resource management
- správa zdroje spjatá s životním cyklem objektu
    - inicializace objektu: získání zdroje (acquire)
    - destruktor: uvolnění zdroje (release)
- ideálně: jeden objekt spravuje jeden zdroj



RAII – Příklady zdrojů

- paměť na haldě
- string, vector, všechny kontejnery
- práce se soubory v C++
- chytré ukazatele (C++11)
- zamykání, mutexy (C++11)

V C++ vše funguje na stejném principu

Různé jazyky často implementují automatickou správu paměti, ale ne automatickou správu zdrojů (v posledních letech se to trochu zlepšuje). C++ má automatickou správu zdrojů už skoro od svého počátku.

Chytré ukazatele – smart pointers

- Cílem je poskytnout prostředek pro bezpečnou správu paměti a programátorské chyby
  - neuvolnění paměti
  - vícenásobné uvolnění paměti
  - ukazatele do neplatné paměti
  - záměna `delete` a `delete []`

Problémem je i nejasné vlastnictví `raw pointerů` a odpovědnost za jejich uvolňování

- Chytré ukazatele umožňují automatickou dealokaci v místě zániku vlastníka

Garbage collector

- Přístup k hodnotám
  - pomocí dereference, vrací ukazatel na datový typ parametry šablony
  - metodou `get`
- Několik variant chytrých ukazatelů:

  `std::unique_ptr`, `std::shared_ptr`, `std::weak_ptr`
- Inicializátory:

  `std::make_unique`, `std::make_shared`

Chytré ukazatele – `std::unique_ptr`

- Nejjednodušší, ale pokrývá většinu našich možných potřeb
- Má nulovou režii za běhu (na rozdíl od dalších chytrých ukazatelů)
- Použitelný pro jednotlivé objekty i pole – podporuje operátory `*`, `->` a `[]`

Základní princip

- Objekt, který uvnitř drží ukazatel (koncept vlastnictví)
- Na konci života objektu se zavolá delete na vlastněný ukazatel
  - tedy zavolá destruktor a dealokuje
  - lokální `std::unique_ptr`: automaticky, na konci bloku
  - `std::unique_ptr` jako atribut: automaticky po destruktoru hlavního objektu
  - explicitně: zavoláním metody `reset()`
- Vlastnictví se nedá sdílet (proto unique)
  - ale může se explicitně předat

Chytré ukazatele – std::unique_ptr

- Alokace
- pokud neinicializujeme, je automaticky nullptr

```
// v C++11
std::unique_ptr<Object> ptr(new Object(params));
// od C++14 - inicizalizátory
auto ptr = std::make_unique<Object>(params);
```

- Přístup k objektu – stejně jak u klasických ukazatelů (*, ->)
- Pozor, nedefinované chování, pokud by byl ptr == nullptr

```
ptr->method();
function(*ptr);
```

Chytré ukazatele – std::unique_ptr

- test, zda std::unique_ptr obsahuje nějaký ukazatel

```
if (ptr) { ... }
```

- přímý přístup ke spravovanému ukazateli

```
Object* rawPtr = ptr.get();
// ptr je stále vlastníkem ukazatele
```

- vzdání se vlastnictví

```
// tohle raději nedělejte!
Object* rawPtr = ptr.release();
// ptr už není vlastníkem ukazatele, který je nyní uložen
// v rawPtr, a je třeba jej uvolnit ručně!
```

Chytré ukazatele – std::unique_ptr

- předávání vlastnictví jinému std::unique_ptr
- std::unique_ptr se nedá kopírovat!

```
std::unique_ptr<Object> newPtr = ptr; // chyba!
// normální operátor= vytváří kopii, ukazatel by nebyl unikátní
```

- std::unique_ptr třeba přesouvat (std::move)

```
std::unique_ptr<Object> newPtr = std::move(ptr);
// newPtr teď vlastní ukazatel, který předtím vlastnil ptr
// ptr teď nevlastní nic, je tedy ekvivalentní nullptr
```

- funguje nejen při inicializaci, ale i při přiřazení

```
auto ptrA = std::make_unique<Object>();
auto ptrB = std::make_unique<Object>();
ptrA = std::move(ptrB); // kdo co vlastní teď?
```

Chytré ukazatele – std::unique_ptr

- použití pro alokaci polí

```
// v C++11
std::unique_ptr<Object[]> ptr(new Object[size]);
// od C++14 { používejte raději takto
auto ptr = std::make_unique<Object[]>(size);
// alokuje paměť pro size objektů
// a všechny inicializuje bezparametrickým konstruktorem
```

- použití pro alokací polí primitivních typů

```
auto ptr = std::make_unique<int[]>(size);
// alokuje paměť pro size intů
// a všechny inicializuje na 0
```

- inicializuje tedy jako `new int[size]()`

Jak správně pracovat s ukazateli v moderním C++

- je třeba rozmyslet, kdo bude vlastníkem ukazatele
  - ten pak má `std::unique_ptr` ukazující na daný objekt
- ostatní (ne-vlastníci) smí mít klasický (raw) ukazatel na tentýž objekt (lépe referenci)
  - je třeba zaručit, aby vlastník ukazatele přežil všechny ne-vlastníky, kteří ukazovaný objekt používají

Jak předávat `std::unique_ptr` do funkce?

- hodnotou typu `std::unique_ptr`: volající pak musí použít `std::move` a tím se vzdává vlastnictví ukazatele
- referencí: volaná funkce může sebrat vlastnictví (nedoporučované)
- const referencí: v podstatě OK, ale volaná funkce může modifikovat odkazovaný objekt (je to jakoby Object* const)
- surový ukazatel: může být Object * nebo const Object *
- úplně nejlépe – referencí na Object: volající musí zajistit, že není `nullptr`

Chytré ukazatele – std::unique_ptr

- předávání pomocí std::move

```
auto rect = std::make_unique<Rect>(2 , 3);
// ...
auto rect2 = std::move(rect);
rect2->CalcSurface();
rect->CalcSurface(); // chyba !
```

- návratová hodnota funkce

```
std::unique_ptr<Rect> VytvorCtverec(int a) {
  auto rect = std::make_unique<Rect>(a, a);
  // ...
  return std::move(rect);
}
```

Chytré ukazatele – std::shared_ptr

- std::shared_ptr slouží pro společné vícenásobné sdílení paměti
- Společně s kopií std::shared_ptr se o ní vytváří záznam v interním čítači referencí
  - konstruktor čítač inkrementuje
  - destruktor čítač dekrementuje, poslední prvek zruší i odkazovaný objekt
- std::make_shared vytvoří shared pointer tak, že z dodaného objektu si vytvoří kopii

```cpp
std::shared_ptr<Rect> rect = std::make_shared<Rect>(2, 3);
auto rect2 = rect; // lze kopírovat
```

- návratová hodnota není problém

```cpp
std::shared_ptr<Rect> VytvorCtverec(int a) {
  auto rect = std::make_shared<Rect>(a, a);
  // ...
  return rect;
}
```

Chytré ukazatele – std::weak_ptr

- realizuje dočasné vlastnictví, odkazuje na objekt zprostředkovaně přes vlastníka
- umí sdílet ukazatele s std::shared_ptr, aniž by je vlastnil

```
std::weak_ptr<double> xx;
std::shared_ptr<double> bb(new double);
xx = bb;
```

- pro přístup/práci nutno zkonvertovat na std::shared_ptr pomocí metody

```
std::shared_ptr aa = xx.lock(); // zablokuje zruseni puvodniho xx
```

- Dá se zjistit, zda originál stále existuje

```
if (xx != nullptr) // ukazatel je v pořádku => bb ještě existuje
if (xx.expired())  // bb je zrušeno
```