

KIV/TSI - Seminář C++

06. Copy a move sémantika, šablony

Martin Úbl

KIV ZČU

2020/2021

- zjištění pozice ve streamu
 - `tellg` (input), `tellp` (output)
 - vrací instanci `streampos`
- nastavení pozice ve streamu (`seek`)
 - `seekg` (input), `seekp` (output)
- např. zjištění velikosti souboru v bajtech:

```
std::ifstream input("soubor.txt");
```

```
input.seekg(0, input.end);  
int fileSize = input.tellg();  
input.seekg(0, input.beg);
```

- standardní proudy
 - proudové obaly nad implicitně otevřenými soubory
 - `std::cin` - standardní vstup, `stdin`
 - `std::cout` - standardní výstup, `stdout`
 - `std::cerr` - chybový výstup, `stderr`
- instance proudů lze předávat obecnou referencí
- lze tedy mít funkci schopnou zapisovat do/číst z libovolného streamu
 - parametr `std::ostream&`
 - předáním `std::cout` zapíše do konzole
 - předáním instance `std::ofstream` zapíše do souboru
 - parametr `std::istream&`
 - předáním `std::cin` přečte z konzole
 - předáním instance `std::ifstream` přečte ze souboru

- *l-value*
- „locator value“, někdy „left-hand side value“
- všechny identifikátory, které označují konkrétní místo v paměti
- „z pohledu kódu“; fyzicky i jiné objekty 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 paměť - nemají identifikátor
- nelze do nich přiřadit

```
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
- klasická reference, s jakou jsme pracovali
- 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
- při vázání na *r-value* 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

- *r-value* reference
- reference na *r-value*
- nelze vázat na *l-value*
- `std::move`
- platnost má jako běžná reference, jen nemusí být `const` pro *r-value*
- uvození `&&`
- dovoluje fungování move sémantiky

```
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 dočasne  
}  
void my_func(const int& a) {  
    // pro trvale  
}  
  
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";  
    }  
    std::string GetContext() && {  
        return "r-value";  
    }  
};
```

```
a.GetContext();           // l-value  
Test().GetContext();     // r-value
```

„C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.“

- Bjarne Stroustrup

„There are only two kinds of languages: the ones people complain about and the ones nobody uses.“

- Bjarne Stroustrup

- životnost objektu by měla být ohraničena i sémanticky
- copy sémantika
 - „klonuje“ zdroj
 - při konstruování nebo přiřazením
- move sémantika
 - „přesouvá“ zdroj
 - při konstruování nebo přiřazením
- příklady:
 - `std::unique_ptr` - pouze move
 - `std::thread` - pouze move
 - `std::ostream` a `std::istream` - pouze move
 - STL kontejnery - move i copy
- spíše než sada exaktních pravidel jde o sémantické odlišení kontextu
- lze lépe vyjádřit, kdo jaký objekt vlastní

- copy konstruktor a copy assignment
- objekt lze kopírovat při vytváření jiného
- nebo přiřazením do již existujícího objektu
- nemodifikuje zdrojový objekt

```
// copy konstruktor
```

```
MyClass(const MyClass& other)  
    : mValue(other.mValue) {  
}
```

```
// copy assignment operator
```

```
MyClass& operator=(const MyClass& other) {  
    mValue = other.mValue;  
}
```

Copy a move sémantika

- move konstruktor a move assignment
- objekt lze přesunout při vytváření jiného
- nebo přiřazením do již existujícího objektu
- modifikuje zdrojový objekt („vyprazdňuje“ ho)

```
// move konstruktor
```

```
MyClass(MyClass&& other)  
    : mValue(other.mValue) {  
    other.mValue = 0;  
}
```

```
// move assignment operator
```

```
MyClass& operator=(MyClass&& other) {  
    mValue = other.mValue;  
    other.mValue = 0;  
}
```

Copy a move sémantika

- copy konstrukce

```
MyClass a(other);
```

- move konstrukce

```
MyClass b(std::move(other));
```

- copy assignment

```
a = other;
```

- move assignment

```
b = std::move(other);
```

- pokud máme v úmyslu vyjádřit vlastnictví, existují pravidla
- *Rule of zero*
 - nedefinujeme sami move/copy konstruktory a operátory, ani destruktory
 - využíváme existující třídy, co to dělají za nás
 - např. `std::unique_ptr`
- *Rule of five*
 - definujeme vše:
 - copy konstruktor
 - copy assignment operátor
 - move konstruktor
 - move assignment operátor
 - destruktory

„If you think it's simple, then you have misunderstood the problem.“
- Bjarne Stroustrup

- šablony
- prvek generického programování
- dovoluje zobecnit implementaci pro více parametrů
- generování kódu v čase kompilace
- parametrizace:
 - typem
 - hodnotou
 - proměnným seznamem (tzv. variadické šablony)
- klíčové slovo `template`, `typename` a špičaté závorky
- šablonové funkce, metody, třídy, `constexpr` výrazy, ...

```
template<typename T>
void DoSomethingGeneric(const T& a) {
    ...
}
```

- již jsme je potkali
 - `std::array<int, 5>`
 - `std::vector<int>`
 - `std::function<void()>`
 - ...
- tyto třídy jsou obecné
- předem neví, jaký typ mají obalovat, vnitřně reprezentovat
- rozhodne se až v momentě tzv. specializace
 - explicitní nebo implicitní uvedení šablonových parametrů
 - vygeneruje se kopie kódu pro danou sadu parametrů

```
std::array<int, 5> arr;
```

- specializuje se pro `int` a velikost 5

- parametrizace typem
- klíčové slovo typename nebo class (ekvivalentní)

```
template<typename T>
void PrintSomething(const T& something) {
    std::cout << "Neco:␣" << something
               << std::endl;
}
```

- parametrizace hodnotou

```
template<size_t Dim>
class Vector {
private:
    std::array<double, Dim> mCoords;

    ...
};
```

- typicky pro nějaké analyticky definované třídy, které se mohou lišit parametry
- změnou hodnoty se v podstatě vytvoří nový typ

- parametrizace typem i hodnotou
- lze parametrizovat více typy i hodnotami a kombinovat

```
template<typename T, size_t Size>
std::array<T, Size> FillNatural() {

    std::array<T, Size> ret;
    std::generate(ret.begin(), ret.end(),
                  [n = 1]() mutable { return n++; });

    return ret;
}
```

- generické typy
- např. zásobník

```
template<typename T>
class Stack {
private:
    std::vector<T> mData;

public:
    void push(const T& value);
    T& top() const;
    void pop();

    bool empty() const;
};
```

- explicitní specializace funkce/metody
- např. jedna výjimka z generického chování

```
template<typename T>
void do_something(const T& value) {
    //
}
```

```
template<>
void do_something<double>(const double& value){
    //
}
```


- explicitní specializace metody
- např. jedna výjimka z jinak generického chování třídy

```
template<typename T>
class Test {
public:
    void Do_This();
    void Do_That();
};
```

```
template<>
void Test<int>::Do_This() {
    //
}
```

- každý typ, pro který se specializuje, musí podporovat vše, co se provádí v těle metody
 - např. inkrementace operátorem ++ - všechny typy musí mít definován operátor ++

```
template<typename T>
T& increment(T& val) {
    return val++;
}
```

```
int a = 5;
increment(a); // ok
```

```
std::string s{ "one_does_not_simply_increment"
               " a_string" };
increment(s); // ouch!
```

- podmíněná specializace
- `std::enable_if`
- dovoluje překladači udat podmínky, za kterých se má šablona specializovat
- podmínky, např.:
 - `std::is_integral`
 - `std::is_floating_point`
 - `std::is_signed`
 - `std::is_pod`
 - `std::is_class`
 - a další...

- lehce těžkopádné

```
template<typename T,  
        typename std::enable_if<  
            std::is_integral<T>::value  
            >::type* = nullptr>  
T& increment(T& val) {  
    val++;  
    return val;  
}
```

- dá se trochu zkrátit
 - místo `std::enable_if<...>::type` psát `std::enable_if_t<...>`
 - místo `std::is_integral<T>::value` psát `std::is_integral_v<T>`

- stále těžkopádné

```
template<typename T,  
        typename std::enable_if_t<  
            std::is_integral_v<T>  
            >* = nullptr>  
T& increment(T& val) {  
    val++;  
    return val;  
}
```

- constexpr šablony

```
template<typename T>  
constexpr T PI = T(3.141592653589793238L);
```

- C++20 - šablonové lambda funkce

- kuriozity
- např. faktoriál rekurzivní specializací šablon

```
template <int N>
struct Factorial {
    static const int value = N * Factorial<N - 1>;
};
```

```
template <>
struct Factorial<0> {
    static const int value = 1;
};
```

- pozn.: opět to neznamená, že to je dobrý nápad, jen demonstruje genericitu šablon; technicky vzato je systém šablon turingovsky úplný

- co dál?
- variadické šablony
- perfect forwarding
- např. `emplace_back`
 - in-place vložení prvku např. do vektoru
 - pak není třeba více kopií
 - nebo více `std::move`
- SFINAE