5. 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++

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 }
```

lec05/01-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 jako agument funkce

- Jak napsat vlastní funkci, která přijímá lambda funkci jako parametr?
- Generická funkce `accumulate_if` bude rozšířením `accumulate` – do výsledku zahrne pouze ty hodnoty, které splní zadanou podmínku:

```
1  template <class Iter, class T, class Predicate>
2  T accumulate_if (Iter st, Iter en, T init, Predicate op) {
3    for ( ; st != en; ++st)
4      if (op (*st)) init = init + *st;
5    return init;
6  }
```

- Parametrem pro testování prvků může být funkce, funktor nebo lambda funkce.
  - Kompilátor si typ Predicate odvodí podle typu předaného skutečného parametru.
  - Datový typ lambda funkce, která nezachycuje proměnné, je kompatibilní s ukazatelem na funkci. Pro lambda funkce lze vždy použít generický typ `std::function` (`<functional>`).

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

```
1  auto get_count = [n = 1]() mutable {
2    return n ++;
3  };
5  std::cout << get_count () << std::endl ; // 1
6  std::cout << get_count () << std::endl ; // 2
7  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;
4  // chyba
5  // lambda se pokousi ovlivnovat promennou zachycenou hodnotou
6  generate_n (back_inserter (seq), 100, [s]() {return s++;});
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
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

I. Další konstrukce C++

Lambda funkce

L-value R-value

Přesouvání (move sémantika)

RAII

Specialitky

L-value

- *locator value*, někdy také *left-hand side value*
- *z poledu 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`

```
1 int a = 5; // a je l- value
2 std :: string b; // b je l- value
3 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

```
1  int a = 5;                // a je l-value, 5 je r-value
2  std :: string b = "hi"; // b je l-value, "hi" je r-value
3  float c = get_c ();       // get_c () je ( vraci ) r-value
4  5 = c; // ouch !
```

L-value reference

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

```
1  int a = 5;
2  int & ra = a; // ok
3  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`

```
1  const int& rb = 5; // ok
2  void my_func ( const int& a) {}
3  // ...
4  my_func (a); // ok
5  my_func (10); // ok
```

R-value reference 1/2

- 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)

```
1  int && rb = 5; // ok
2  void my_func (int && a) {
3    //...
4  }
5  my_func (a) // ouch !
6  my_func (10); // ok
```

R-value reference 1/2

- dovoluje rozlišit dočasné a *trvalé* instance

```
1  void my_func (int && a) {
2    // pro dočasné instance
3  }
5  void my_func (const int& a) {
6    // pro trvalé instance
7  }
9  my_func (a);  // ok
10 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

```
1  class Test {
2  public:
3    std::string getContext () & {return "l-value\n";}
5    std::string getContext () && {return "r-value\n";}
6  };
7  //...
8  Test a;
10 std::cout << a.getContext ();       // l-value
11 std::cout << Test().getContext (); // r-value
```

I. Další konstrukce C++

Lambda funkce

L-value R-value

Přesouvání (move sémantika)

RAII

Specialitky

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

```
class string {
  int len, max; char * ptr;
  string (const string & src) { // copy constructor
    len = src.len;
    max = src.max;
    ptr = new char [max];
    for (int i = 0; i <= len; i++) ptr[i] = src.ptr[i];
  }
  string (string && src) { // move constructor
    len = src.len;
    max = src.max;
    ptr = src.ptr;
    src.ptr = NULL; /* !!! */
  }
};
```

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"); // using move op =
```

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) { // no & or &&
  // this is always != &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, `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`

I. Další konstrukce C++

Lambda funkce

L-value R-value

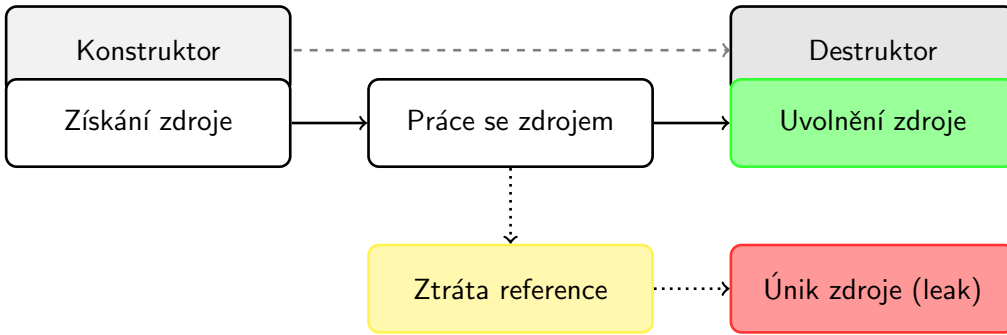
Přesouvání (move sémantika)

RAII

Specialitky

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 - inicializátory
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

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

- přímý přístup ke spravovanému ukazateli

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

- vzdání se vlastnictví

```
1 // tohle raději nedělejte!
2 Object* rawPtr = ptr.release();
3 // ptr už není vlastníkem ukazatele, který je nyní uložen
4 // 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

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

- návratová hodnota není problém

```
1 std::shared_ptr<Rect> VytvorCtverec (int a) {
2   auto rect = std::make_shared<Rect> (a, a);
3   // ...
4   return rect ;
5 }
```

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

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

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

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

- Dá se zjistit, zda originál stále existuje

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

I. Další konstrukce C++

Lambda funkce

L-value R-value

Přesouvání (move sémantika)

RAII

Specialitky

Small String Optimization

- optimalizace `std::string` pro malé řetězce
- normálně by byla potřeba dynamická alokace
- `std::string` od C++17 dovoluje uchovávat krátké řetězce přímo v sobě
  - 15 znaků pro MSVS
  - 23 znaků pro GCC/clang

```
1 // bez dynamicke alokace
2 std::string a{"Hello"};
4 // s dynamickou alokaci
5 std::string a{"Stay at home unless you want to get intubated by
      psychiatrist"};
```

String Views

- třída `std::string_view` poskytuje *pohled* na jinou řetězcovou pamět (`<string_view>`)
- sama neuchovává řetězec
- lze napr. poznat rozdíl pri volání substr
  - `std::string` vrací *kopii* podřetězce
  - `std::string_view` vrací pouze pohled na podřetězec (v podstatě dva ukazatele)
- hodí se napr. pri parsování vstupu

```
1  // 2 instance std::string, dvakrat rozkopirovany retezec
2  std::string s{"nejaky velmi dlouhy retezec v dynamicke pameti"};
3  auto s2 = s. substr (7);
5  // 2 instance std::string_view, ale jen jedna instance retezce
6  std::string_view sv(s);
7  auto sv2 = sv. substr (7);
```

Datový typ `std::optional`

- třída `std::optional` obaluje jiný typ, který může, ale nemusí být poskytnut
- šablonový typ, konstruktor s instancí daného typu nebo implicitní
- `std::nullopt` lze použít pro vytvoření prázdného

```
#include <optional>
//...
std::optional<int> getValue (size_t idx) {
  if (myMap.find(idx) != myMap.end())
    return myMap[idx];
  return std::nullopt;
}
```

- ověření, zda má hodnotu: metoda `has_value()`, přetížený operátor `bool()`
- vyzvednutí hodnoty: metoda `value()`, přetížený operátor dereference

Datový typ `std::variant`

- obaluje jednu hodnotu různých typů – typově bezpečný `union` z C
- alokace vždy na zásobníku
- šablonový typ, parametry šablony jsou všechny typy, které může potenciálně ukládat
- vždy ukládá právě jeden z nich (přiřazení, konstruktor)
- vyzvednutí funkcí `std::get<T>`
- `get()` špatného typu vyvolá výjimku `std::bad_variant_access`

```
1  #include <variant>
2  // --
3  std::variant<int, double, long long> var;
5  var = 1;
6  var = 15.4;
7  var = 99999LL;
```

Datový typ `std::any`

- obaluje jednu hodnotu libovolného typu – typově bezpečný `void*` z C
- není šablonový typ, alokace vždy na haldě
- vyzvednutí pretypováním `std::any_cast<T>`
- přetypování na špatný typ vyvolá výjimku `std::bad_any_cast`
- typ musí být kopírovatelný

```
1  #include <any>
3  std :: any v;
4  v = 42;
5  v = " retezec ";
7  try {
8    int ival = std::any_cast<int> (v);
9  }
10 catch (std::bad_any_cast & bac) { }
```

Uživatelsky definované literály

- možnost definovat si vlastní literály
- např. převody jednotek
- *tváří* se jako `operator""`
- uživatelské literály by měly začínat podtržítkem

Literály bez podtržítka jsou vyhrazené pro standard jazyka

```
1  // napr.  vse na metry
2  constexpr long double operator"" _cm (long double cm) {
3    return cm / 100.0;
4  }
6  long double A4_vyska = 29.7_cm; // 0.297
```