

C++ - Úvod

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Požadavky, zápočet, zkouška

2 Použitý styl v přednáškách

3 Rozdíly mezi C a C++

- Reference
- Ostatní...
- C++11, C++14, C++17

Požadavky, zápočet, zkouška

Přednáší, zkouší, cvičí & studenty děsí

- Ing. Roman Diviš
- roman.divis@upce.cz
- konzultační hodiny - vizte upce.cz

Alternativní výukové materiály pro C++

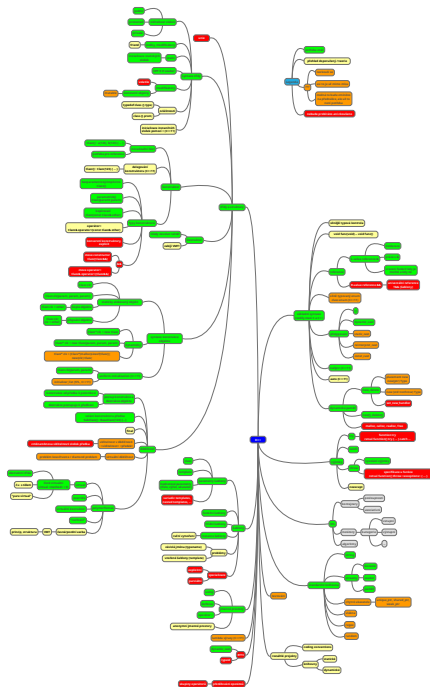
- <https://learnxinyminutes.com/docs/c++/>
- <https://en.cppreference.com/w/>
- <http://www.cplusplus.com/doc/tutorial/>
- <https://msdn.microsoft.com/en-us/library/3bstk3k5.aspx>

Zápočet

- docházka
- samostatně vypracovaná semestrální práce

Zkouška

- teoretický test (10 minut)
- program dle zadání (cca 150 minut)



Použitý styl v přednáškách

Základní informace

Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Příklad

Nullam mattis efficitur aliquam.

Chyba / příklad s chybou / nekorektní použití

Sed aliquam iaculis massa, vel tincidunt lacus tincidunt eget.

Poznámka (teorie, vhodné k zapamatování)

Proin porta urna ut ipsum ornare, a ultricies elit dictum.

Deprecated (staré, už nepoužívané)

Pellentesque habitant morbi tristique senectus et netus et malesuada fames.

Bonus (nepotřebujete ke zkoušce, ale souvisí s tématem)

Sed imperdiet pharetra est, sed ullamcorper neque.

Ukázka

Ukazatele a dynamická paměť

Ukázky korektní a nekorektní práce s ukazateli a dynamicky alokovanou pamětí:

✓ Good

```
int* pointer = nullptr;  
pointer = new int;  
*pointer = 123;
```

✗ Bad

```
int* pointer = 0xdeadbeef;  
*pointer = 123;
```

⚠ deprecated

```
int* pointer = (int*)malloc(sizeof(int));  
*pointer = 123;
```

Definice kódu

```
struct|class nazevDatovehoTypu [final] [dědičnost] {  
    [složky – atributy, metody, vnořené typy]...  
} [objekty];
```

```
[dědičnost]:  
    : [viditelnost] [virtual] předek1, ...
```

Definuje:

- Začínáme klíčovým slovem **struct** nebo **class**
- dále uvedeme název datového typu (Pes, Kocka, Student, ...)
- dále může být (ale nemusí) klíčové slovo **final**
- dále může být definováno dědění z předků
- Uvnitř třídy je možné definovat větší množství složek (...)

Slidy označené fialovými pruhy obsahují téma, které není vyžadováno u zkoušky a zápočtu.

Rozdíly mezi C a C++

Základní rozdíly

- znakové literály jsou typu **char**
- proměnné je možné deklarovat „kdekoliv“
- silnější typová kontrola
 - ▶ **void** func(); – v C++ funkce bez parametrů
 - ▶ **void*** prom – nelze přiřadit bez konverze typu
- ukazatel nikam není NULL, ale **nullptr**^{C++11} (rvalue, nelze přiřadit do neukazatele)
- funkce je nutné deklarovat před jejich zavoláním
- nový logický typ **bool** (**true** nebo **false**)
- při použití datového typu struktury se uvádí pouze název struktury (nikoliv **struct** NazevStruktury jako v C)

Dynamická alokace paměti



- ▶ funkce `malloc`, `free`, `calloc` a `realloc` jsou nahrazeny
 - ▶ neumí pracovat s objekty

- nové operátory `new`, `new[]`, `delete` a `delete[]`
 - ▶ umí pracovat s objekty
 - ▶ realokaci je nutné řešit ručně (`new`, `memcpy()`, `delete`)

Reference

L-value reference

- L-value reference představují ukazatele na „pojmenované“ proměnné
 - ▶ ne na dočasné objekty
- kompilátor referencování a dereferencování řeší za nás (odpadá starost s operátory * a &)
- po vytvoření reference nejde změnit kam ukazuje
- veškerá manipulace je pak automaticky přeposlána na referencovaný objekt



```
int alfa = 100;
int& refAlfa = alfa; // není potřeba &

refAlfa++; // alfa = 101
cout << (&alfa == &refAlfa); // == true
refAlfa = 0; // alfa = 0;
```

L-value reference. . .

- lze je použít jako
 - ▶ lokální proměnné
 - ▶ atributy tříd
 - ▶ parametry metod
 - ▶ návratové hodnoty

✓

```
// předávání hodnotou  
// = kopie objektu  
void nakrmKocku(Kocka k) {  
    k.nakrmena(RYBA);  
    // nakrmili jsme kopii micky, micka ↔  
    // zatím chce hladu  
}
```

```
Kocka micka;  
nakrmKocku(micka);
```

✓

```
// předávání odkazem  
// = stejný objekt  
void nakrmKocku(Kocka& k) {  
    k.nakrmena(RYBA);  
    // nakrmili jsme micku  
}
```

```
Kocka micka;  
nakrmKocku(micka);
```

L-value reference. . .

- reference je možné vytvořit na libovolný datový typ (i ukazatele)
 - ▶ ale nelze vytvořit ukazatel na referenci



```
int i = 10;
int* ui = &i;

int*& refui = ui; // ok
//int&* urefi; // ne — ukazatel na referenci

// následující dva řádky jsou totožné
*ui = 100;
*refui = 100;
// i == *ui == *refui == 100
```

L-value reference. . .

- konstatní reference
 - ▶ nelze přiřadit jinou hodnotu (pomocí operátoru =)
 - ▶ s objekty lze omezeně manipulovat
 - ▶ **const** type& ref;
- referenční funkce
 - ▶ funkce, která vrací referenci
 - ▶ za return musí být l-hodnota



```
Kocka& mnoukej(Kocka& k) {  
    cout << "mnau ";  
    k.zamnoukano();  
    return k;  
}  
  
Kocka micka;  
mnoukej(mnoukej(mnoukej(micka)));  
// micka 3x zamňoukala
```

R-value reference^{C++11}

- ▶ představuje optimalizaci využití dočasných objektů
- ▶ zapisuje se jako: `datovyTyp&& prom`
- ▶ přidává move sémantiku (konstruktor, operátor =)
 - ▶ `std::move()`
 - ▶ `std::forward()`

Ostatní...

Standardní vstupy a výstupy (obrazovka, soubor, paměť)

- pro vstup a výstup vytvořeny objekty – proudy
 - ▶ hlavičkový soubor `iostream`
 - ▶ výstup na obrazovku pomocí
`std::cout << "data" << 123 << prom << ...;`
 - ▶ vstup z klávesnice pomocí
`cin >> prom1 >> prom2 >> prom3 >> ...;`



```
#include <iostream>
```

```
void main() {  
    std::cout << "Zadej cislo: ";  
    int cislo;  
    std::cin >> cislo;  
    std::cout << "Zadal jsi " << cislo << std::endl;  
}
```

Prostory jmen (namespace)

- řeší konflikty jmen
- podobné balíčkům z Javy (ale vše uvnitř je veřejné)
- přístup k prvkům pomocí :: nebo zpřístupnění pomocí operátoru **using**



```
namespace MojeKnihovna {  
    void knihovniFunkce() { ... }  
}  
  
void main() {  
    // knihovniFunkce(); // NE -> neexistuje  
    MojeKnihovna::knihovniFunkce();  
}
```


Prostory jmen (namespace)...

- **using** se nepoužívá v hlavičkových souborech!



```
namespace MojeKnihovna {  
    void knihovniFunkce() { ... }  
}  
  
// ...  
  
using namespace MojeKnihovna;  
  
void main() {  
    knihovniFunkce();  
}
```

Výjimky

- ošetřování chyb pomocí výjimek
 - ▶ blok **try**
 - ▶ blok **catch**
 - ▶ příkaz **throw**
 - ▶ modifikátor **noexcept**^{C++11}



```
try {  
    pripravVlakna();  
    provedVypocet();  
    uklidVlakna();  
} catch (ThreadException& threadException) {  
    cerr << "Vlakna spadla" << endl;  
} catch (CalculationException& calculationException) {  
    cerr << "Vypocet spadl" << endl;  
}
```

Šablony (generické datové typy a funkce)

- genericita na stereoidech
- **template**<**typename** T> ...
- šablony
 - ▶ funkcí
 - ▶ metod
 - ▶ datových typů
 - ▶ šablon
- specializace šablon
 - ▶ parciální
 - ▶ explicitní



```
template<typename T, int Size>
struct Array {
    T& get(int index) { return _array[index]; }
    T _array[Size];
}
```

RTTI – run time type information

- obdoba reflexe z Javy
- podstatně jednodušší
 - ▶ v podstatě umí jen identifikovat typ a vypsát jeho jméno (tvar není standardizován)
- definuje dva operátory
 - ▶ **typeid** – vrací strukturu **type_info** s informacemi o typu
 - ▶ **dynamic_cast** – chytré přetypování v rámci hierarchie dědičnosti
- využívá přítomnost VMT v objektech (vyžaduje virtuální metodu, viz později polymorfismus)

Nové operátory pro přetypování

- **dynamic_cast**

- ▶ využítá RTTI (Run Time Type Information)
- ▶ užitečný, bezpečný, přetypování z předka na potomka (i naopak)

- ▶ **static_cast**

- ▶ běžné přetypování

- ▶ **const_cast**

- ▶ pro odstranění **const** nebo **volatile** modifikátorů

- ▶ **reinterpret_cast**

- ▶ nestandardní přetypování



```
Object* obj = new Kocka("Micka");  
Kocka* kocka = dynamic_cast<Kocka*>(obj);  
if (kocka != nullptr)  
    kocka->mnoukej();
```

C++11, C++14, C++17

- ▶ definuje, že proměnná nebo funkce obsahuje/vrací konstatní výraz a může být využit na místě, kde se očekává konstatní výraz (definice statického pole, ...)



```
constexpr int pocetBajtuVObrazu(int sirka, int vyska, int bpp) {  
    return sirka * vyska * (bpp / 8);  
}
```

```
unsigned char obrazoveBity[pocetBajduVObrazu(800, 600, 32)];
```

auto^{C++11}, decltype^{C++11}

- ▶ **auto**^{C++11} lze použít jako zástupný symbol pro libovolný datový typ
 - ▶ konkrétní datový typ dohledá kompilátor v době kompilace
 - ▶ stále se jedná o silně typovaný jazyk, nelze pak přiřadit jiný typ do takové proměnné
- ▶ **decltype**^{C++11} slouží jako zástupný symbol datového typu definovaného podle výsledku výrazu



```
std::map<KeyObject<Person>, Person> map;
```

```
auto iterator = map.rbegin();
```

```
// auto = std::map<KeyObject<Person>, Person>::reverse_iterator
```

```
auto v = 1; // v = int
```

```
decltype(v) w = v; // w = int
```


lambda výrazy^{C++11}

- ▶ lambda výrazy (anonymní funkce) představují možnost zapsat funkci prakticky kamkoliv do kódu



```
auto function = [] (int p1, int p2) { return p1 + p2; }  
  
function(10, 20); // = 30
```

for-each^{C++11}

- ▶ **for** (definiceProměnné: kontejner){ ... }
- ▶ umožňuje jednoduše procházet kontejnery/kolekce
- ▶ funguje nad statickým polem a objekty, které definují metody `begin()`, `end()`



```
std::vector<int> container;  
for (int value : container) {  
    ...  
}
```

for-each^{C++11}

×

```
// for-each podle C++/CLI použitelné v MSVC  
// není ve standardu C++ → nepoužívat!  
for each (int value in container) {  
    ...  
}
```

✓

```
// neplést for-each cyklus s for_each algoritmem!  
for_each(  
    container.begin(),  
    container.end(),  
    [](int value) { ... });
```

for-each^{C++11}

- lze kombinovat s **auto**



```
std::vector<Game::Map::Object<GUID, WorldType>> container;  
  
for (auto& object : container) {  
    ...  
}
```

tuple (N-tice)^{C++11}

- ▶ slouží pro předávání N hodnot bez nutnosti tvořit třídu
- ▶ využívá šablon s proměnným počtem parametrů



```
std::tuple<double, char, std::string> get_student(int id)
{
    if (id == 0)
        return std::make_tuple(3.8, 'A', "Lisa Simpson");
    if (id == 1)
        return std::make_tuple(2.9, 'C', "Milhouse Van Houten");
    if (id == 2)
        return std::make_tuple(1.7, 'D', "Ralph Wiggum");

    throw std::invalid_argument("id");
}
```

tuple (N-tice)^{C++11}



```
int main()
{
    auto student0 = get_student(0);
    std::cout << "ID: 0, "
                << "GPA: " << std::get<0>(student0) << ", "
                << "grade: " << std::get<1>(student0) << ", "
                << "name: " << std::get<2>(student0) << '\n';

    double gpa1;
    char grade1;
    std::string name1;
    std::tie(gpa1, grade1, name1) = get_student(1);
    std::cout << "ID: 1, "
                << "GPA: " << gpa1 << ", "
                << "grade: " << grade1 << ", "
                << "name: " << name1 << '\n';
}
```

Další novinky z C++11, 14

- ▶ silně typované výčty – **enum class** `NazevEnumu { ... }`
 - ▶ jednotlivé výčtové konstanty nejsou dostupné z globálního prostoru, ale z `NazevEnumu::konstanta`
- ▶ nové řetězcové literály – `u8"Řetězec v UTF-8"`,
`u"UTF-16"`, `U"UTF-32"`
- ▶ RAW řetězcové literály – `R"(retezec " s ' divnoznaky)"`
- ▶ uživatelské řetězcové literály
- ▶ **static_assert**
- ▶ atributy – `[[atribut]]`
- ▶ ...

Structured bindings^{C++17}

- ▶ umožňuje jednoduše rozdělit složitý typ na elementární proměnné
- ▶ funguje na tuple interface, statické pole nebo pokud typ má pouze veřejné nestatické složky



```
// C++14
```

```
double gpa1;
```

```
char grade1;
```

```
std::string name1;
```

```
std::tie(gpa1, grade1, name1) = get_student(1);
```

```
...
```

```
// C++17
```

```
auto& [gpa2, grade2, name2] = get_student(2);
```

```
...
```


Structured bindings^{C++17}

- lze kombinovat i s for-each



```
std::map<System::GUID, Game::Map::Object<GUID, WorldType>> ←  
    container;
```

```
for (const auto& [key, value] : container) {  
    ...  
}
```

Další novinky z C++17

- ▶ **constexpr if**
- ▶ init-statement pro if/switch
- ▶ automatické odvození typů pro šablony objektových typů
- ▶ fold expressions (... v šablonách s proměnným počtem parametrů)
- ▶ ...



- C++17 není prozatím široce podporováno (zejména v MSVC)
- Většina novinek bude dostupná nejdříve od MSVC 2017!
- Viz <https://blogs.msdn.microsoft.com/vcblog/2017/05/10/c17-features-in-vs-2017-3/>

C++ - Objektové typy v C++

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Objektové typy

- Základní syntaxe objektových typů
- Datové složky a operace
 - Atributy (datové složky)
 - Metody
 - Speciální metody

2 Vytváření objektů

- Staticky alokované objekty
 - Volání funkcí – předávání parametrů, návratová hodnota
- Dynamicky alokované objekty
- Konstatní a nekonstantní objekty

Objektové typy

Objektové typy

- **class** – plnohodnotný objektový typ
- **struct** – plnohodnotný objektový typ
- **union** – omezený objektový typ (neumí dědičnost, polymorfizmus)

Základní vlastnosti:

- vícenásobná dědičnost
 - ▶ neexistují rozhraní
- polymorfizmus
- H x CPP soubor
 - ▶ H – deklarace (celý typ, atributy, prototypy metod)
 - ▶ CPP – definice (pouze těla metod, statické atributy)

struct x class

- **struct** i **class** jsou zaměnitelné
- **struct** může dědit z **class** a naopak
- liší se pouze ve výchozí úrovni viditelnosti složek (**public** x **private**)

Základní syntaxe objektových typů

Definice struct a class

```
struct|class nazevDatovehoTypu [final] [dědičnost] {  
    [složky – atributy, metody, vnořené typy]  
} [objekty];
```



```
struct Pes {  
    ...  
};
```



```
class Pes {  
    ...  
};
```

dopředná deklarace

- pokud potřebujeme pracovat s typem, ale ne s jeho vnitřkem
- **struct|class** nazevDatovehoTypu;

```
struct|class nazevDatovehoTypu [final] [dědičnost] {  
    [složky – atributy, metody, vnořené typy]  
} [objekty];
```

- [objekty] – umožňuje vytvořit staticky alokované objekty od dané třídy v globálním prostoru



```
struct Pes {  
    ...  
} punta, rafan;  
  
void main() {  
    punta.stekej();  
    rafan.stekej();  
}
```

```
struct | class nazevDatovehoTypu [final] [dědičnost] {  
    [složky – atributy, metody, vnořené typy]  
} [objekty];
```

- [**final**]^{C++11} – zakazuje dědit z této třídy
- [dědičnost] – specifikace předků
 - ▶ : [viditelnost] [**virtual**] předek1, ...



```
struct Pes : public virtual Zvire, public IIdentifikovatelný, ←  
    IKopirovatelný {  
    ...  
};
```

```
struct|class nazevDatovehoTypu [final] [dědičnost] {  
    [složky – atributy, metody, vnořené typy]  
} [objekty];
```

- [složky] – dále dle typu atribut, metoda, vnořený typ



```
struct Pes {  
  
    void stekej() { ... }  
  
    string jmeno;  
  
};
```

Datové složky a operace

Viditelnost / přístupová práva

- 3 úrovně:
 - ▶ **public** – přístupné všude
 - ▶ **protected** – přístupné z potomků
 - ▶ **private** – přístupné pouze z vnitřku třídy
- výchozí hodnota:
 - ▶ **struct** – **public**
 - ▶ **class** – **private**
- změna viditelnosti:
 - ▶ uvozuje se blok (**viditelnost**: ...)
 - ▶ bloky se mohou opakovat, být zpřeházené

friend

- ▶ přátelé třídy (**friend**) mají přístup i k **protected** a **private** složkám

Viditelnost / přístupová práva



```
struct Pes {  
    // implicitní public: (platí pro struct)  
    // implicitní private: (platí pro class)  
  
    string jmeno; // je public  
  
private:  
    int id; // je private  
    int vek; // je private  
  
protected:  
    int pocetChlupu; // je protected  
  
public:  
    int pocetNohou; // je public  
};
```

Atributy (datové složky)

Atributy

```
[static] datovýTyp názevAtributu [inicializátor];
```

- **static** – statický atribut (váže se na třídu, ne na objekt)
- [inicializátor]^{C++11} – umožňuje nastavit výchozí hodnotu (nelze u statických atributů)



```
int pocetChlupu = 12345;  
static int pocetPsu;
```

- Dále existují modifikátory: **mutable**, **volatile**

Statické atributy

- statické atributy potřebují vyhradit paměť a inicializovat – v CPP souboru

PES.H

```
struct Pes {  
    static int pocetPsu;  
};
```

PES.CPP

```
int Pes::pocetPsu = 0;
```

Statické atributy (použití)

MAIN.CPP

```
Pes pes = PsiBouda::vytvorPsa();  
Pes::pocetPsu++;  
  
cout << "Pocet psu: " << Pes::pocetPsu;
```

Metody

Metody

deklaraceMetody:

```
[static] [virtual] datovýTypNávratovéHodnoty názevMetody([↔  
parametry]) [const] [override] [final] [noexcept];
```

- **static** – statická metoda (váže se na třídu, ne na objekt)
- **virtual** – virtuální metoda (polymorfizmus)
- **const** – konstantní metoda (nemění stav objektu)
- **override**^{C++11} – označuje přetíženou metodu z předka (polymorfizmus)
- **final**^{C++11} – zakazuje další přepisování metody v potomcích
- **noexcept**^{C++11} – označuje metodu, která nevyvolává výjimky

Metody...

- ▶ modifikátor **inline** – optimalizace, nevolá funkci, vkládá kód přímo na místo volání; kompilátor dosadí automaticky, pokud je metoda definována uvnitř třídy
- ▶ modifikátor **volatile** – nestálé metody
- ▶ deklaraceMetody = **delete**; – C++11, smazání metody
- ▶ deklaraceMetody = **default**; – C++11, vynucení implementace metody kompilátorem

Deklarace a definice metody uvnitř třídy



```
struct Pes {  
    // deklarace — úplný funkční prototyp, bez těla  
    void stekej();  
  
    // definice — deklarace + tělo  
    void kousej(Osoba& osoba) {  
        osoba.zran(KOUSNUTI_DO_KOTNIKU);  
    }  
};
```

Deklarace a definice metody vně třídy

- definice metody vně třídy musí specifikovat, že jde o metodu dané třídy
 - ▶ používá se operátor ::
 - ▶ uvádějí se pouze modifikátory **const**, **override**, **noexcept**



```
struct Pes {  
    // deklarace  
    void kousej(Osoba& osoba);  
};  
  
// definice  
void Pes::kousej(Osoba& osoba) {  
    osoba.zran(KOUSNUTI_DO_KOTNIKU);  
}
```


Metody a this

- uvnitř instančních metod je dostupný ukazatel **this** (odpovídá Třída*)
 - ▶ není potřeba, implicitně je možné se odkazovat na složky třídy



```
struct Pes {  
    void obnovZdravi() {  
        // this je Pes*, následující výrazy jsou totožné  
        _zdravi = 100;  
        this->_zdravi = 100;  
        (*this)._zdravi = 100;  
    }  
  
private:  
    int _zdravi;  
};
```

Speciální metody

Speciální metody

Existuje několik speciálních metod souvisejících s vytvářením a rušením objektů:

- konstruktory (bez parametrů, s parametry, kopírovací, konverzní, move)
- destruktory
- operátor =

Základní verzi těchto metod nám vytvoří automaticky kompilátor, pokud některou z metod nadefinujeme sami, kompilátor pak nemusí nic vytvářet (chybějící bezparametrický konstruktor).

Vznik a zánik objektu

struct Pes dědí z (**struct** Zvire dědí z (**struct** Objekt))

Proces vzniku a zániku objektu Pes

- konstruktor Objekt
- konstruktor Zvire
- konstruktor Pes
 - ▶ objekt žije, lze volat metody...
- destruktory Pes
- destruktory Zvire
- destruktory Objekt

Konstruktor

```
názevTřídy([parametry]) [inicializačníČást];
```

```
inicializačníČást:  
: atribut1(hodnota1), ...
```

- nemá návratovou hodnotu (ani **void**)!
- inicializační část – pro inicializaci datových složek
 - ▶ je potřeba pro inicializaci referenčních atributů

- ▶ modifikátor **explicit** – zakazuje implicitní konverze

Bezparametrický konstruktor



```
struct Pes {  
  
    Pes() {  
        cout << "konstruktor Pes()" << endl;  
    }  
  
};
```

Parametrický konstruktor



```
struct Pes {  
  
    Pes(int zdravi) {  
        cout << "konstruktor Pes(int)" << endl;  
        _zdravi = zdravi;  
    }  
  
private:  
    int _zdravi;  
};
```

Parametrický konstruktor s inicializační částí



```
struct Pes {  
  
    Pes(int zdravi) : _zdravi(zdravi) {  
        cout << "konstruktor Pes(int)" << endl;  
    }  
  
private:  
    int _zdravi;  
};
```


Kopírovací konstruktor

- kompilátor ho umí vytvořit automaticky
 - ▶ vytváří mělkou kopii (`memcpy(kopie, original, sizeof(Typ))`)



```
struct Pes {  
  
    Pes(const Pes& pes) {  
        cout << "konstruktor Pes(const Pes&)" << endl;  
        _zdravi = pes._zdravi;  
    }  
  
private:  
    int _zdravi;  
};
```

Delegování konstruktorů^{C++11}

- ▶ delegování umožňuje zavolat jiný konstruktor před vlastním vykonáním konstruktoru (od C++11)
- ▶ inicializační část pak musí obsahovat pouze volání delegovaného konstruktoru!

```
struct Pes {  
    Pes(std::string jmeno) : _jmeno(pes,_jmeno) {  
        cout << "konstruktor Pes(std::string)" << endl;  
    }  
  
    Pes(std::string jmeno, int vek) : Pes(jmeno) {  
        cout << "konstruktor Pes(std::string, int)" << endl;  
        _vek = vek;  
    }  
  
private:  
    std::string _jmeno;  
    int _vek;  
};
```

operátor =

- ▶ **operator**= přepíše obsah objektu jiným objektem
 - ▶ kompilátor vytváří automaticky
 - ▶ funguje na stejném principu jako kopírovací konstruktor

```
struct Pes {  
    Pes& operator=(const Pes& pes) {  
        if (this == &pes)  
            return *this;  
  
        _jmeno = pes._jmeno;  
        return *this;  
    }  
  
    private:  
        std::string _jmeno;  
};
```

Move konstruktor^{C++11}

- ▶ C++11 zavedlo novou „move“ sémantiku, funguje jako optimalizace, umožňuje převzít data z dočasných objektů
 - ▶ R-value reference (Typ&&)
 - ▶ move konstruktor (Typ(Typ&&))
 - ▶ move assignment operátor (**operator**=(Typ&&))

```
struct Pes {  
  
    Pes(Pes&& pes) : _jmeno(std::move(pes._jmeno)) {  
        cout << "konstruktor Pes(Pes&&)" << endl;  
    }  
  
    private:  
        std::string _jmeno;  
};
```

Destruktor

```
[virtual] ~názevTřídy();
```

- nemá návratovou hodnotu (ani **void**)!
- **virtual** – je potřeba při využívání polymorfizmu a dědičnosti!



```
struct Pes {  
    ~Pes() {  
        cout << "destruktor Pes()" << endl;  
    }  
};
```

Vytváření objektů

Vytváření objektů

- Staticky alokované objekty („statické objekty“)
 - ▶ na stacku (lokální proměnné, parametry funkcí)
 - ▶ globální prostor
 - ▶ statické atributy tříd
- Dynamicky alokované objekty
 - ▶ new, delete

Staticky alokované objekty

Staticky alokované objekty

- vytváří a ruší je kompilátor
 - ▶ destruktorky volá kompilátor!
- životnost
 - ▶ do konce bloku (lokální proměnné, parametry)
 - ▶ do konce programu (globální proměnné, statické atributy tříd)

Vytváření statických objektů



```
void main() {  
    // bezparametrický konstruktor  
    Pes rafan;  
    // parametrický konstruktor  
    Pes kousak("Kousak", 200);  
  
    // kopírovací konstruktor  
    Pes kopieKousaka(kousak);  
    Pes jinaKopieKousaka = kousak;  
    // operátor=  
    jinaKopieKousaka = kousak;  
}
```

Vytváření statických objektů...

×

```
// NELZE → jedná se o deklaraci funkce!
```

```
// funkce vracející objekt Pes, bez parametrů
```

```
Pes rafan();
```

```
// lze, ale nevhodné → může vést k vytvoření dočasného objektu a ↵  
kopírování
```

```
Pes kousak = Pes("Kousak", 200);
```

```
// lze, ale nevhodné → dochází k dvojímu volání destruktoru, dojde ke ↵  
zničení VMT tabulky → přestává fungovat polymorfismus
```

```
kousak.~Pes();
```

Uniform initialization^{C++11}

- C++11
- použití pomocí složených závorek
- volá konstruktor nebo inicializuje veřejné datové složky
- jednotná syntaxe u všech případů!

✓

```
void main() {  
    // bezparametrický konstruktor  
    Pes rafan{};  
    // parametrický konstruktor  
    Pes kousak{"Kousak", 200};  
    // kopie  
    Pes kopieKousaka{kousak};  
}
```

Volání funkcí – předávání parametrů, návratová hodnota

Parametry funkcí



```
void funkce(Pes pes) {  
    pes.stekej();  
}
```

```
Pes rafan{};  
// dojde k vytvoření kopie (kopírovací konstruktor)  
// &pes != &rafan  
funkce(rafan);
```

Parametry funkcí (reference)

- Reference = předání odkazem
 - ▶ „Ukazatelem na pozadí“



```
void funkce(Pes& pes) {  
    pes.stekej();  
}
```

```
Pes rafan{};  
// objekt je předán odkazem  
// &pes == &rafan  
funkce(rafan);
```

Parametry funkcí (ukazatel)



```
void funkce(Pes* pes) {  
    pes->stekej();
```

```
    // nezmění objekt, ani proměnné rafan a ptr
```

```
    // změní pouze lokální proměnnou pes
```

```
    pes = 0xdeadbeef;
```

```
}
```

```
Pes rafan{};
```

```
// objekt je předán pomocí ukazatele (je vytvořena kopie "Pes*" = ↔  
    4(8) bajtové číslo – adresa v paměti)
```

```
Pes* ptr = &rafan;
```

```
// ptr == &rafan
```

```
funkce(ptr);
```

```
// ptr == &rafan
```


Parametry funkcí (ukazatel na ukazatel)



```
void funkce(Pes** pes) {  
    (*pes)->stekej();
```

```
    // nezmění objekt, ani proměnnou rafan
```

```
    // změní ukazatel ptr
```

```
    *pes = 0xdeadbeef;
```

```
}
```

```
Pes rafan{};
```

```
// objekt je předán pomocí ukazatele (je vytvořena kopie "Pes*" = ↔  
    4(8) bajtové číslo – adresa v paměti)
```

```
Pes* ptr = &rafan;
```

```
// ptr == &rafan
```

```
funkce(&ptr);
```

```
// ptr == 0xdeadbeef
```

Návratová hodnota

- Vrácení staticky alokovaného objektu vede ke kopírování
 - ▶ Normálně se ale využije RVO (return value optimization – optimalizace kompilátorem)
 - ▶ Standard C++17 definuje „Guaranteed copy elision“ (= RVO ve standardu)



```
Pes funkce() {  
    Pes rafan{};  
    return rafan;  
    // objekt rafan — zaniká, ven jde kopie  
}  
  
// pes je vytvořen pomocí kopírovacího konstrukturu  
// RVO → kopírování nemusí nastat, objekt je předán "celý"  
Pes pes = funkce();
```

Návratová hodnota (reference)

Referenci lze vracet, pokud objekt nadále existuje

- je dynamicky alokovaný
- je static (pozor u vícevláknových programů)
- je na globálním prostoru (dtto)



```
Pes& funkce() {  
    Pes* rafan = new Pes{};  
    return *rafan;  
    // dynamicky alokovaný objekt je platný, dokud nezavoláme delete  
}  
  
// ok, ale pozor na memory leak!  
Pes& pes = funkce();
```

Návratová hodnota (reference)

- Návrátový typ – reference + staticky alokovaný lokální objekt => nefunguje

×

```
Pes& funkce() {  
    Pes rafan{};  
    return rafan;  
    // objekt rafan – zaniká, ven jde zmetek!  
}
```

// pes ukazuje na rozbitou paměť!

```
Pes& pes = funkce();
```

// MSVC obsahuje rozšíření, které daný příklad korektně provede, my \longleftrightarrow pojedeme dle standardu C++ a nebudeme na to spoléhat

Návratová hodnota (reference)

Prodloužení životnosti dočasných objektů

- ▶ Standard C++ umožňuje prodloužit životnost objektu v návratové hodnotě
 - ▶ pro konstantní L-reference
 - ▶ pro R-reference

```
const Pes& funkce() {  
    Pes rafan;  
    return rafan;  
}
```

```
// ok – uplatněno prodloužení životnosti objektu dle standardu  
const Pes& pes = funkce();
```

Dynamicky alokované objekty

Dynamicky alokované objekty

- Vznikají a zanikají na náš explicitní příkaz (`new` / `delete`)

Objekt



```
// vytvoření proměnné typu ukazatel
Pes* rafan = nullptr;
// alokace paměti, volání konstruktoru
rafan = new Pes{};

rafan->stekej();
(*rafan).stekej();

// volání destruktoru, dealokace paměti
delete rafan;
```


Pole objektů



```
// vytvoření proměnné typu ukazatel
Pes* rafani = nullptr;
// alokace paměti (5 * sizeof(Pes)), 5x volání konstruktoru
rafani = new Pes[5];

rafani[0].stekej();
(rafani + 0)->stekej();
(*(rafani + 0)).stekej();

// volání destruktorků (5x), dealokace paměti
delete[] rafani;
```

Pole ukazatelů na objekt



```
Pes** rafani = nullptr;
rafani = new Pes*[5]; // alokace paměti (5 * sizeof(Pes*)), nevolá ↵
    konstruktor!

for (int i = 0; i < 5; i++)
    rafani[i] = new Pes{}; // alokace paměti objektu, konstruktor

rafani[0] -> stekej();
(*rafani[0]).stekej();
**(rafani+0).stekej();

for (int i = 0; i < 5; i++)
    delete rafani[i]; // destruktory, dealokace paměti objektů

delete[] rafani; // dealokace paměti pole
```



- ▶ malloc, free – neumí volat konstruktor a destruktor
 - ▶ nepoužívejte je

```
// vytvoření proměnné typu ukazatel
Pes* rafan = nullptr;
// alokace paměti
rafan = (Pes*)malloc(sizeof(Pes));
// konstrukce
new(rafan) Pes;

// ...

// destrukce
rafan->~Pes();
// uvolnění paměti
free(rafan);
rafan = nullptr;
```

×

// nekorektní kombinace new—delete[] / new[]—delete

```
Pes* pes = new Pes;
```

```
delete[] pes;
```

```
pes = new Pes[2];
```

```
delete pes;
```

×

// dvojí volání destrukturu

```
Pes* pes = new Pes;
```

```
pes->~Pes();
```

```
delete pes;
```

Konstatní a nekonstantní objekty

- Nekonstantní objekt – lze měnit jeho stav (atributy)
- Konstantní objekt – nelze měnit jeho stav (atributy)

- Nekonstantní metoda
 - ▶ **this** odpovídá typu Třída*
 - ▶ může měnit stav
- Konstantní metoda (za názvem **const**)
 - ▶ **this** odpovídá typu **const** Třída*
 - ▶ nemůže měnit stav

- Nekonstantní objekt
 - ▶ Umí všechno!



```
struct Pes {  
  
    void stekej() {  
        cout << "Haf";  
        _stekano = true;  
    };  
  
    bool byloStekano() const {  
        return _stekano;  
    }  
  
    private:  
        bool _stekano = false;  
}
```



```
Pes pes{};  
  
pes.stekej();  
cout << pes.bylStekano();
```

- Konstantní objekt – neměnný
 - ▶ Lze volat jen const metody!



```
struct Pes {  
  
    void stekej() {  
        cout << "Haf";  
        _stekano = true;  
    };  
  
    bool byloStekano() const {  
        return _stekano;  
    }  
  
private:  
    bool _stekano = false;  
}
```



```
const Pes pes{};  
  
pes.stekej(); // nejde
```



```
const Pes pes{};  
  
//pes.stekej(); – nejde  
cout << pes.bylStekano();
```


Více pohledů na jeden objekt...



```
Pes pes{};
```

```
Pes& nekonstRefPes = pes;  
nekonstRefPes.stekej();
```

```
const Pes& konstRefPes = pes;  
// konstRefPes.stekej(); – nejde
```

```
Pes* nekonstPtrPes = &pes;  
nekonstPtrPes->stekej();
```

```
const Pes* konstPtrPes = &pes;  
// konstPtrPes->stekej(); – nejde
```

Není const jako const...



```
struct Pes {  
  
    // nekonstatní metoda, vrací int  
    int stekej1();  
  
    // nekonstatní metoda, vrací const int  
    const int stekej2();  
  
    // konstatní metoda, vrací int  
    int stekej3() const;  
  
    // konstatní metoda, vrací const int  
    const int stekej4() const;  
}
```

konstantní ukazatel

- **const** u ukazatele může značit konstantní ukazatel (neplést s konstantním objektem)

```
// ukazatel na konstantní objekt
```

```
const Pes* pes1 = &pes;
```

```
// pes1->stekej() // NE
```

```
pes1 = nullptr; // OK
```

```
// konstantní ukazatel na nekonstantní objekt
```

```
Pes* const pes2 = &pes;
```

```
pes2->stekej() // OK
```

```
// pes2 = nullptr; // NE
```

```
// konstantní ukazatel na konstantní objekt
```

```
const Pes* const pes3 = &pes;
```

```
// pes3->stekej() // NE
```

```
// pes3 = nullptr; // NE
```

mutable, volatile, constexpr, const_cast

- ▶ modifikátor **mutable** (pro atributy)
 - ▶ umožňuje měnit stav této proměnné i v konstantním objektu
 - ▶ odporuje obvyklé logice – nebudeme používat
- ▶ modifikátor **volatile** (pro proměnné, atributy)
 - ▶ „nestálé instance“ – zakazuje optimalizace
 - ▶ stejný efekt jako konstantní-nekonstantní objekty
- ▶ modifikátor **constexpr** (pro funkce, metody)
 - ▶ C++11
 - ▶ označuje metody a funkce jako „compile time constant“
 - ▶ lze je pak použít pro definici velikosti pole, šablonového parametru, ...
- ▶ operátor **const_cast**
 - ▶ umožňuje odstraňovat **const** a **volatile** modifikátory
 - ▶ nehlídá, jestli je to korektní!

C++ - Dědičnost, polymorfismus

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Dědičnost

2 Polymorfizmus

3 Implementace dědičnosti v C++

- Implementace polymorfizmu (pozdní vazba, VMT)

Dědičnost

Dědičnost v C++

- podporována vícenásobná dědičnost
 - ▶ třída může dědit z několika tříd
- neexistují rozhraní
 - ▶ ale existují abstraktní třídy
 - ▶ uzavřené (sealed/final) třídy
- u zděděných složek (atributů a metod) lze ovlivnit viditelnost
 - ▶ lze hromadně omezit přístup k těmto složkám
 - ▶ lze individuálně obnovit nebo omezit přístup
- metody lze přetížit/přepsat v potomcích
 - ▶ C++ rozlišuje časnou a pozdní vazbu

Dědičnost

```
struct|class nazevDatovehoTypu [final] [dědičnost] {  
    [složky – atributy, metody, vnořené typy]  
}  
[objekty];
```

- [**final**]^{C++11} – zakazuje dědit z této třídy
- [dědičnost] – specifikace předků
 - ▶ : [viditelnost] [**virtual**] předek1, ...



```
struct Pes : public virtual Zvire, public IIdentifikovatelný, ↵  
    IKopirovatelný {  
    ...  
};
```

Dědičnost...

- pokud viditelnost není uvedena využije se výchozí dle typu
 - ▶ **class** – **private**
 - ▶ **struct** – **public**
- viditelnost udává s jakou viditelností jsou zděděny složky z daného předka, pokud uvedeme
 - ▶ **public** – nic se nemění
 - ▶ **protected** – **public** složky z předka jsou převedeny na **protected**
 - ▶ **private** – **public** a **protected** složky z předka jsou převedeny na **private**

Aby se to chovalo á la Java:

- ▶ používejte **struct** a nic tam nepište
- ▶ používejte **struct** i **class** a vždy uvádějte viditelnost **public**

Obnovení viditelnosti

- ▶ viditelnost je možné obnovit uvedením složky předka ve tvaru `TřídaPředka::složka` v bloku s danou viditelností (rovněž lze využít i `using TřídaPředka::složka`)

```
class TA {
```

```
    protected:
```

```
        int x, y;
```

```
    public:
```

```
        int GetX() const;
```

```
        // ...
```

```
};
```

```
class TB : private TA {
```

```
    protected:
```

```
        TA::x;
```

```
    public:
```

```
        TA::GetX();
```

```
        // ...
```

```
};
```

- potomek může zastoupit předka

- ▶ je korektní přiřadit ukazatel na potomka do ukazatele na předka
- ▶ je korektní přiřadit referenci na potomka do reference na předka
- ▶ NENÍ korektní přiřadit objekt potomka do objektu předka
 - ★ dojde k oříznutí objektu!



```
Potomek potomek{};
```

```
Predek& ref = potomek;
```

```
Predek* ptr = &potomek;
```

```
...
```



```
Predek predek = potomek; // oříznutí objektu!!!
```

Volání konstruktoru předka



```
struct Predek {  
    Predek() : _a(0) { }  
    Predek(int a) : _a(a) { }
```

```
private:  
    int _a;  
}
```

```
struct Potomek : Predek {  
    Potomek() : _b(0) {  
        // kompilátor implicitně volá Predek::Predek();  
    }  
  
    Potomek(int a, int b) : Predek(a), _b(0) {  
        // kompilátor explicitně volá Predek::Predek(int);  
    }
```

```
private:  
    int _b;  
}
```



```
struct Predek {  
    //Predek() : _a(0) { }  
    Predek(int a) : _a(a) { }  
  
private:  
    int _a;  
}
```



```
struct Potomek : Predek {  
    Potomek() : _b(0) {  
        // kompilátor implicitně volá Predek::Predek();  
        // konstruktor neexistuje !!! chyba kompilace  
    }  
  
    Potomek(int a, int b) : Predek(a), _b(0) {  
        // kompilátor explicitně volá Predek::Predek(int);  
        // ok!  
    }  
  
private:  
    int _b;  
}
```

Polymorfizmus

- jedna funkce/metoda může nabývat více různých podob
- metoda `pracuj()` z rozhraní `IPracujici`
 - ▶ sekačka – seká
 - ▶ policista – uděluje pokuty
 - ▶ pes – štěká

Způsoby volání metod v C++

- ▶ časná vazba – výchozí, kompilátor řeší v době překladu
- ▶ pozdní vazba – na vyžádání (**virtual**), řeší se v době vykonávání programu
- ▶ Java – automaticky vždy uplatňuje pozdní vazbu! chování v C++ a C# je tedy odlišné

Časná vazba



```
struct Zvire {  
    void vydejZvuk() {  
        cout << "err";  
    }  
};
```

```
struct Kralik : Zvire {  
    void vydejZvuk() {  
        cout << "Bzzzm!";  
    }  
};
```



```
Zvire z{};  
z.vydejZvuk(); // err
```

```
Kralik k{};  
k.vydejZvuk(); // Bzzzm!
```

Časná vazba...



```
struct Zvire {  
    void vydejZvuk() {  
        cout << "err";  
    }  
};
```

```
struct Kralik : Zvire {  
    void vydejZvuk() {  
        cout << "Bzzzm!";  
    }  
};
```



```
Zvire& refz = k;  
refz.vydejZvuk(); // err
```

```
Zvire* ptrz = &k;  
ptrz->vydejZvuk(); // err
```

Pozdní vazba



```
struct Zvire {  
    virtual void vydejZvuk() {  
        cout << "err";  
    }  
};
```

```
struct Kralik : Zvire {  
    void vydejZvuk() {  
        cout << "Bzzzm!";  
    }  
};
```



```
Zvire& refz = k;  
refz.vydejZvuk(); // Bzzzm!
```

```
Zvire* ptrz = &k;  
ptrz->vydejZvuk(); // Bzzzm!
```

Pozdní vazba...

- metoda musí být označena **virtual** v předkovi
- reference a ukazatele poté za běhu dohledají správnou metodu
- využívá se VMT (virtual method table), kompilátor ji automaticky vytváří a vkládá na ni ukazatel do každého objektu

Pozdní vazba...



```
struct Zvire {  
    virtual void vydejZvuk() {  
        cout << "err";  
    }  
};
```

```
struct Kralik : Zvire {  
    void vydejZvuk() {  
        cout << "Bzzzm!";  
    }  
};
```



```
struct Zvire {  
    virtual void vydejZvuk() {  
        cout << "err";  
    }  
};
```

```
struct Kralik : Zvire {  
    virtual void vydejZvuk() {  
        cout << "Bzzzm!";  
    }  
};
```



```
struct Zvire {  
    virtual void vydejZvuk() {  
        cout << "err";  
    }  
};
```

```
struct Kralik : Zvire {  
    virtual void vydejZvuk() ↵  
        override {  
        cout << "Bzzzm!";  
    }  
};
```

Pozdní vazba...

Doporučené použití:

- předek – **virtual**
 - ▶ povinné, jinak nebude fungovat!
- potomek – **virtual** + **override**
 - ▶ **virtual** – pro přehlednost, že se jedná o virtuální metodu
 - ▶ **override**^{C++11} – zajistí kontrolu kompilátorem, že předpis odpovídá virtuální metodě z předka

Pozdní vazba...

Destruktor:

- časná/pozdní vazba se uplatňuje stejným způsobem na destruktory!
 - ▶ aby bylo zajištěno, že se zavolá správný destruktory, musí být v předkovi virtuální destruktory
 - ▶ **každá třída ze které se dědí by měla mít virtuální destruktory**

Čistě virtuální metody

- virtuální metoda (**virtual**) – uplatnění pozdní vazby
- čistě virtuální metoda (abstraktní, **virtual** ... = 0;) – pozdní vazba + chybí tělo (nutno doplnit v potomkovi)
 - ▶ nelze vytvářet objekty od třídy obsahující čistě virtuální metody!



```
struct Zvire {  
    virtual void vydejZvuk() = 0; // čistě virtuální / pure virtual  
};  
  
struct Kralik : Zvire {  
    virtual void vydejZvuk() override {  
        cout << "Bzzzm!";  
    }  
};
```


Čistě virtuální metody...



```
struct Zvire {  
    virtual void vydejZvuk() = 0; // čistě virtuální / pure virtual  
};
```

```
Zvire z{}; // nelze — Zvire je abstraktní třída
```

- ▶ čistě virtuální metoda může mít i tělo (definici)
 - ▶ stále je pak čistě virtuální (abstraktní)
 - ▶ tělo lze explicitně volat z potomka
Predek::cisteVirtualniMetoda()

Rozhraní

- čistě virtuální metody lze použít pro realizaci typu „rozhraní“
 - ▶ nezapomínejte na virtuální destruktory!



```
struct IZvire {  
  
    virtual ~IZvire() { }  
  
    virtual void vydejZvuk() = 0;  
    virtual void nakrm(Potravina* potravina) = 0;  
    virtual void zautoc(IZvire* zvire) = 0;  
};
```

Implementace dědičnosti v C++

Následující část přednášky obsahuje informace k bližšímu pochopení, jak interně funguje vícenásobná dědičnost a polymorfismus.

Podrobně viz https://www.usenix.org/legacy/publications/compsystems/1989/fall_stroustrup.pdf

Nejedná se o téma potřebné do zkoušky/zápočtu!

Objekt

- ▶ objekt je spojitá část paměti, kde jsou uloženy atributy daného objektu
- ▶ atributy jsou uloženy v pořadí dle pořadí definování
- ▶ pokud třída dědí z jiné třídy → v paměti jsou v souvislém bloku uloženy hodnoty atributů ze všech předků i z dané třídy
- ▶ metoda je realizována jako vylepšená funkce
 - ▶ na pozadí od kompilátoru obdrží ukazatel **this**

Varianta A) Jednoduchá dědičnost

```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbb;  
  
    int getA() const { return a; }  
    int getB() const { return b; }  
};
```

```
struct B : A {  
    int c = 0xcccccccc;  
    int d = 0xdddddddd;  
  
    int getC() const { return c; }  
    int getD() const { return d; }  
};
```

```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbbb;  
  
    int getA() const { return a; }  
    int getB() const { return b; }  
};
```

```
struct B : A {  
    int c = 0xccccccccc;  
    int d = 0xddddddddd;  
  
    int getC() const { return c; }  
    int getD() const { return d; }  
};
```



```
A objA{};  
B objB{};
```

```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;

    int getA() const { return a; }
    int getB() const { return b; }
};

```

```

struct B : A {
    int c = 0xcccccccc;
    int d = 0xdddddddd;

    int getC() const { return c; }
    int getD() const { return d; }
};

```



```

A objA{};
B objB{};

```

| | objA |
|------|------------|
| 0x00 | 0xaaaaaaaa |
| 0x04 | 0xbbbbbbbb |


```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;

    int getA() const { return a; }
    int getB() const { return b; }
};

```

```

struct B : A {
    int c = 0xcccccccc;
    int d = 0xdddddddd;

    int getC() const { return c; }
    int getD() const { return d; }
};

```

✓

```

A objA{};
B objB{};

```

| | objA |
|------|------------|
| 0x00 | 0xaaaaaaaa |
| 0x04 | 0xbbbbbbbb |

| | objB |
|------|------------|
| 0x00 | 0xaaaaaaaa |
| 0x04 | 0xbbbbbbbb |
| 0x08 | 0xcccccccc |
| 0x0c | 0xdddddddd |

```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbbb;

    int getA() const { return a; }
    int getB() const { return b; }
};

```

```

struct B : A {
    int c = 0xccccccccc;
    int d = 0xddddddddd;

    int getC() const { return c; }
    int getD() const { return d; }
};


```

✓

```

A objA{};
B objB{};

```

|  this | objA | | | |
|--|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |

✓

```

int getA() const {
    return * (int*) ( ((char*)this) + 0x00);
}

```

```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;

    int getA() const { return a; }
    int getB() const { return b; }
};

```

```

struct B : A {
    int c = 0xcccccccc;
    int d = 0xdddddddd;

    int getC() const { return c; }
    int getD() const { return d; }
};


```

✓

```

A objA{};
B objB{};

```

|  this | objA | | | |
|--|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |

✓

```

int getB() const {
    return * (int*) ( ((char*)this) + 0x04);
}

```

```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbbbb;

    int getA() const { return a; }
    int getB() const { return b; }
};

```

```

struct B : A {
    int c = 0xccccccccc;
    int d = 0xddddddddd;

    int getC() const { return c; }
    int getD() const { return d; }
};


```

✓

```

A objA{};
B objB{};

```

|  this | objB | | | |
|--|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |

✓

```

int getC() const {
    return * (int*) ( ((char*)this) + 0x08);
}

```

```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbbbb;

    int getA() const { return a; }
    int getB() const { return b; }
};

```

```

struct B : A {
    int c = 0xccccccccc;
    int d = 0xddddddddd;

    int getC() const { return c; }
    int getD() const { return d; }
};


```

✓

```

A objA{};
B objB{};

```

|  this | objB | | | |
|--|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |

✓

```

int getD() const {
    return * (int*) ( ((char*)this) + 0x0c);
}

```

Varianta B) Vícenásobná dědičnost

```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbb;  
    ...  
};
```

```
struct B {  
    int c = 0xcccccccc;  
    int d = 0xdddddddd;  
    ...  
};
```

```
struct C : A, B {  
    int e = 0xeeeeeeee;  
    int f = 0xffffffff;  
    ...  
};
```

```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbb;  
    ...  
};
```

```
struct B {  
    int c = 0xcccccccc;  
    int d = 0xdddddddd;  
    ...  
};
```

```
struct C : A, B {  
    int e = 0xeeeeeeee;  
    int f = 0xffffffff;  
    ...  
};
```



```
C objC{};
```



```
struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;
    ...
};
```

```
struct B {
    int c = 0xcccccccc;
    int d = 0xdddddddd;
    ...
};
```

```
struct C : A, B {
    int e = 0xeeeeeeee;
    int f = 0xffffffff;
    ...
};
```



```
C objC{};
```

| | objC | |
|------|------------|-----|
| 0x00 | 0xaaaaaaaa | ← A |
| 0x04 | 0xbbbbbbbb | |
| 0x08 | 0xcccccccc | ← B |
| 0x0c | 0xdddddddd | |
| 0x10 | 0xeeeeeeee | ← C |
| 0x14 | 0xffffffff | |

```
struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;
    ...
};
```

```
struct B {
    int c = 0xcccccccc;
    int d = 0xdddddddd;
    ...
};
```

```
struct C : A, B {
    int e = 0xeeeeeeee;
    int f = 0xffffffff;
    ...
};
```



```
C objC{};
```

| | objC | | | |
|------|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |
| 0x10 | ee | ee | ee | ee |
| 0x14 | ff | ff | ff | ff |

```
struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;
    ...
};
```

```
struct B {
    int c = 0xcccccccc;
    int d = 0xdddddddd;
    ...
};
```

```
struct C : A, B {
    int e = 0xeeeeeeee;
    int f = 0xffffffff;
    ...
};
```



```
C objC{};
```

| this ↘ | objC | | | |
|---------------|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |
| 0x10 | ee | ee | ee | ee |
| 0x14 | ff | ff | ff | ff |



```
getA() -> * (int*) (((char*)this) + 0x00)
getB() -> * (int*) (((char*)this) + 0x04)
```

```
struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;
    ...
};
```

```
struct B {
    int c = 0xcccccccc;
    int d = 0xdddddddd;
    ...
};
```

```
struct C : A, B {
    int e = 0xeeeeeeee;
    int f = 0xffffffff;
    ...
};
```



```
C objC{};
```

| this ↘ | objC | | | |
|---------------|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |
| 0x10 | ee | ee | ee | ee |
| 0x14 | ff | ff | ff | ff |



```
getC() -> * (int*) (((char*)this) + 0x00)
```

```
getD() -> * (int*) (((char*)this) + 0x04)
```

```
struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;
    ...
};
```

```
struct B {
    int c = 0xcccccccc;
    int d = 0xdddddddd;
    ...
};
```

```
struct C : A, B {
    int e = 0xeeeeeeee;
    int f = 0xffffffff;
    ...
};
```



```
C objC{};
```

| <i>this</i> ↘ | objC | | | |
|---------------|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |
| 0x10 | ee | ee | ee | ee |
| 0x14 | ff | ff | ff | ff |



```
getE() -> * (int*) (((char*)this) + 0x10)
```

```
getF() -> * (int*) (((char*)this) + 0x14)
```

```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbb;  
    ...  
};
```

```
struct B {  
    int c = 0xcccccccc;  
    int d = 0xdddddddd;  
    ...  
};
```

```
struct C : A, B {  
    int e = 0xeeeeeeee;  
    int f = 0xffffffff;  
    ...  
};
```

Původní předpoklad selhal u druhého předka v objektu objC...

- reálná implementace ale funguje
- kompilátor umí transformovat začátek objektu dle potřeby
- $(C*)\&objC == (B*)\&objC$
- $(char*)((C*)\&objC) != (char*)((B*)\&objC)$

Ve skutečnosti tedy...

```

struct A {
    int a = 0xaaaaaaaa;
    int b = 0xbbbbbbbb;
    ...
};

```

```

struct B {
    int c = 0xcccccccc;
    int d = 0xdddddddd;
    ...
};

```

```

struct C : A, B {
    int e = 0xeeeeeeee;
    int f = 0xffffffff;
    ...
};

```

| | objC | | | |
|--------------------|------|----|----|----|
| 0x00 | aa | aa | aa | aa |
| 0x04 | bb | bb | bb | bb |
| 0x08 <i>this</i> → | cc | cc | cc | cc |
| 0x0c | dd | dd | dd | dd |
| 0x10 | ee | ee | ee | ee |
| 0x14 | ff | ff | ff | ff |



```

int impl_getC_classC(C* object) {
    B* temporary = (B*)((char*)object + 0x08);
    return temporary->getC();
}

```

Varianta C) Diamond problem – virtuální dědičnost


```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbb;  
};
```

```
struct B : A {  
    int c = 0xcccccccc;  
    int d = 0xdddddddd;  
}
```

```
struct C : A {  
    int e = 0xeeeeeeee;  
    int f = 0xffffffff;  
}
```

```
struct D : B, C {  
    int x = 0x99999999;  
}
```



D objD{};

| | objD | |
|------|------------|----------------------|
| 0x00 | 0xaaaaaaaa | $\leftarrow B, A_B$ |
| 0x04 | 0xbbbbbbbb | |
| 0x08 | 0xcccccccc | $\leftarrow B \dots$ |
| 0x0c | 0xdddddddd | |
| 0x10 | 0xaaaaaaaa | $\leftarrow C, A_C$ |
| 0x14 | 0xbbbbbbbb | |
| 0x18 | 0xeeeeeeee | $\leftarrow C \dots$ |
| 0x1c | 0xffffffff | |
| 0x20 | 0x99999999 | $\leftarrow D$ |



D objD{};

| | objD | |
|------|------------|----------------------|
| 0x00 | 0xaaaaaaaa | $\leftarrow B, A_B$ |
| 0x04 | 0xbbbbbbbb | |
| 0x08 | 0xcccccccc | $\leftarrow B \dots$ |
| 0x0c | 0xdddddddd | |
| 0x10 | 0xaaaaaaaa | $\leftarrow C, A_C$ |
| 0x14 | 0xbbbbbbbb | |
| 0x18 | 0xeeeeeeee | $\leftarrow C \dots$ |
| 0x1c | 0xffffffff | |
| 0x20 | 0x99999999 | $\leftarrow D$ |



```
D objD{};  
D* d = &objD;
```



```
A* a = (A*)d; // nelze — které A?
```



```
A* a = (A*)(B*)d; // lze
```

```
struct A {  
    int a = 0xaaaaaaaa;  
    int b = 0xbbbbbbbb;  
};
```

```
struct B : virtual A {  
    int c = 0xcccccccc;  
    int d = 0xdddddddd;  
}
```

```
struct C : virtual A {  
    int e = 0xeeeeeeee;  
    int f = 0xffffffff;  
}
```

```
struct D : B, C {  
    int x = 0x99999999;  
}
```



```
D objD{};
```

| | objD | |
|------|------|----------------|
| 0x00 | 0x1c | $\leftarrow B$ |
| 0x04 | 0xc | |
| 0x08 | 0xd | |
| 0x0c | 0x1c | $\leftarrow C$ |
| 0x10 | 0xe | |
| 0x14 | 0xf | |
| 0x18 | 0x9 | $\leftarrow D$ |
| 0x1c | 0xa | $\leftarrow A$ |
| 0x20 | 0xb | |



D objD{};

| | objD | |
|------|-----------------------------------|-----|
| 0x00 | 0x1c (<i>ptr_{B→A}</i>) | ← B |
| 0x04 | 0xc | |
| 0x08 | 0xd | |
| 0x0c | 0x1c (<i>ptr_{C→A}</i>) | ← C |
| 0x10 | 0xe | |
| 0x14 | 0xf | |
| 0x18 | 0x99999999 | ← D |
| 0x1c | 0xa | ← A |
| 0x20 | 0xb | |

Dědičnost

- ▶ vícenásobná dědičnost využívá automatické transformace počátku objektu
 - ▶ kód metody neřeší s jakým typem objektu pracuje a prostě přistupuje k atributům
 - ▶ pokud je objekt složitý je potřeba přepočítat jeho počátek před voláním metody předka
 - ▶ provádí kompilátor automaticky na pozadí
- ▶ virtuální dědičnost využívá odkazu na předka na začátku každého objektu
 - ▶ tak je možné vložit společného předka pouze jednou
 - ▶ metody využijí odkazu k dohledání předka a poté mohou pracovat s jeho atributy

VMT (virtual method table)

VMT

- ▶ VMT je vytvořena pro každou třídu, která obsahuje virtuální metody
- ▶ ukazatel na VMT je automaticky vložen do každého objektu
- ▶ VMT obsahuje:
 - ▶ ukazatel na implementaci dané metody pro danou třídu
 - ▶ offset začátku objektu (viz problémy s vícenásobnou dědičností)

Časná/pozdní vazba

- ▶ při časně vazbě se kompilátor podívá na typ objektu a dosadí instrukci CALL s voláním příslušné metody
- ▶ při pozdní vazbě kompilátor načte příslušný záznam z tabulky VMT (kterou má u sebe objekt) a zavolá zde definovanou metodu



```
struct A {  
    virtual void method1() { }  
    virtual void method2() { }  
  
    int _a = 0xaaaaaaaa;  
};
```

```
struct B : A {  
    virtual void method1() override { }  
    virtual void method2() override { }  
  
    int _b = 0xbbbbbbbb;  
};
```



```
struct A {  
    virtual void method1() { }  
    virtual void method2() { }  
  
    int _a = 0xaaaaaaaa;  
};
```

```
struct B : A {  
    virtual void method1() override { }  
    virtual void method2() override { }  
  
    int _b = 0xbbbbbbbb;  
};
```

| | objA | |
|------|------------|--|
| 0x00 | A_{vmt} | |
| 0x04 | 0xaaaaaaaa | |

| | A_{vmt} | |
|------|------------|--|
| 0x00 | A::method1 | |
| 0x04 | A::method2 | |



```
struct A {  
    virtual void method1() { }  
    virtual void method2() { }  
  
    int _a = 0xaaaaaaaa;  
};
```

```
struct B : A {  
    virtual void method1() override { }  
    virtual void method2() override { }  
  
    int _b = 0xbbbbbbbb;  
};
```

| | objA | |
|------|------------|--|
| 0x00 | A_{vmt} | |
| 0x04 | 0xaaaaaaaa | |

| | objB | |
|------|------------|--|
| 0x00 | B_{vmt} | |
| 0x04 | 0xaaaaaaaa | |
| 0x08 | 0xbbbbbbbb | |

| | A_{vmt} | |
|------|------------|--|
| 0x00 | A::method1 | |
| 0x04 | A::method2 | |

| | B_{vmt} | |
|------|------------|--|
| 0x00 | B::method1 | |
| 0x04 | B::method2 | |



```
struct W {  
    virtual void f() { }  
    virtual void g() { }  
    virtual void h() { }  
    virtual void k() { }  
}  
  
struct AW : virtual W {  
    virtual void g() override { }  
}  
  
struct BW : virtual W {  
    virtual void h() override { }  
}  
  
struct CW : AW, BW {  
    virtual void k() override { }  
}
```

| | | |
|--|--------------------------|-----------------|
| | objCW | |
| | $ptr_{AW \rightarrow W}$ | $\leftarrow AW$ |
| | AW data | |
| | ... | |
| | $ptr_{BW \rightarrow W}$ | $\leftarrow BW$ |
| | BW data | |
| | ... | |
| | CW data | $\leftarrow CW$ |
| | ... | |
| | CW_{vmt} | $\leftarrow W$ |
| | W data | |
| | ... | |

| | | |
|------|------------|--|
| | CW_{vmt} | |
| 0x00 | W::f | |
| 0x04 | AW::g | |
| 0x08 | BW::h | |
| 0x0c | CW::k | |

Realita dle MSVC 2017

0x001A59C8 a8 7b 0f 00 bb bb bb bb b4 7b 0f 00 cc cc cc cc dd dd dd dd 94 7b 0f 00 aa aa aa aa

| | |
|--------------|---|
| objCw | 0x001a59c8 {_d=0xdddddddd } |
| AW | {_b=0xbbbbbbbb } |
| W | {_a=0xaaaaaaaa } |
| _b | 0xbbbbbbbb |
| BW | {_c=0xcccccccc } |
| W | {_a=0xaaaaaaaa } |
| _c | 0xcccccccc |
| W | {_a=0xaaaaaaaa } |
| __vfptr | 0x000f7b94 {polym.exe!void(* CW::`vftable'[5])()} {0x000f1344 |
| [0x00000000] | 0x000f1343 {polym.exe!W::f(void)} |
| [0x00000001] | 0x000f1163 {polym.exe!{thunk}:AW::g`adjustor{12}' (void)} |
| [0x00000002] | 0x000f134d {polym.exe!{thunk}:BW::h`adjustor{4}' (void)} |
| [0x00000003] | 0x000f11a9 {polym.exe!CW::k(void)} |
| _a | 0xaaaaaaaa |
| _d | 0xdddddddd |

C++ - Výjimky, jmenné prostory

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Výjimky

- Výjimky v C++ knihovně

2 Jmenné prostory

Výjimky

Ošetřování chybových stavů

- `abort()`
- návratová hodnota funkce
- technika dlouhých skoků (long jumps) v jazyce C
- `goto`

Výjimky

- výjimky slouží pro řešení chybových stavů v průběhu programu
 - ▶ pokusný blok **try**, kde může vzniknout výjimka
 - ▶ zachycující bloky **catch**, které ošetřují výjimky
 - ▶ příkaz **throw** vyvolávající výjimku
- výjimkou v C++ může být prakticky cokoliv (int, char, string, objekt, dynamicky alokovaný objekt, ...)
 - ▶ obecně se nedoporučuje používat dynamickou alokaci paměti pro objekt výjimky; je pak nutné řešit jeho dealokaci
- pokud výjimka není zachycena na úrovni kde vznikla šíří se postupně dále
 - ▶ neošetřená výjimka způsobí pád programu
- v C++ není obdoba bloku **finally** z Javy/C#
- C++ podporuje označování funkcí, kde výjimka nevzniká (**noexcept**^{C++11})
 - ▶ původní systém označoval konkrétní druhy výjimek, které mohly vznikat; jeho špatná definice a implementace způsobily jeho zrušení

Základní použití výjimek

```
try {  
    [kód, kde může vzniknout výjimka]  
} catchHandler...
```

catchHandler:

```
catch(datovýTypVýjimky názevProměnné) { [blok příkazů handleru] }  
catch(...) { [blok příkazů handleru] }
```



```
try {  
    ...  
    throw 100;  
    ...  
} catch (int i) {  
    cout << "Zachyceno " << i;  
}
```

Základní použití výjimek. . .

- **catch** handlerů může následovat více za sebou
 - ▶ záleží na pořadí!
 - ▶ výjimka je zachycena do prvního vyhovujícího handleru



```
try {  
    ...  
    throw 3.14;  
    ...  
}  
catch (int i) {  
    cout << "Zachyceno int " << i;  
}  
catch (double d) {  
    cout << "Zachyceno double " << d;  
}
```

Základní použití výjimek. . .

- lze vytvořit univerzální **catch** handler
 - ▶ nelze pak pracovat s vlastní hodnotou výjimky



```
try {  
    ...  
    throw 'E';  
    ...  
  
} catch (int i) {  
    cout << "Zachyceno int " << i;  
}  
} catch (double d) {  
    cout << "Zachyceno double " << d;  
}  
} catch (...) {  
    cou << "Zachyceno vše ostatní";  
}
```


Základní použití výjimek. . .

- neošetřená výjimka se šíří dále

```
✓  
  
void funkce() {  
    throw "Vyjimka";  
}  
  
try {  
    ...  
    funkce();  
    ...  
}  
catch (const char* str) {  
    cout << "Zachyceno " << str;  
}
```

Základní použití výjimek. . .

- dynamicky alokované objekty výjimky jsou možné
 - ▶ ale nevhodné
 - ▶ alokace je další místo, kde může vzniknout výjimka!



```
try {  
    ...  
    throw new int(-99);  
    ...  
} catch (int* i) {  
    cout << "Zachyceno " << *i;  
    delete i;  
}
```

Základní použití výjimek. . .

- objekt výjimky je při vyvolání nakopírován někam do paměti
- **catch** handler přijímá kopii objektu výjimky
 - ▶ pokud není použita reference!



```
try {  
    ...  
    throw error_object{"My error"};  
    ...  
}  
catch (error_object& errobj) {  
    cout << "Zachyceno " << errobj.getMessage();  
}
```

throw()



- ▶ **throw** také sloužilo k definici, jaké výjimky může funkce vyvolat
 - ▶ obdoba `throws` z Javy
 - ▶ nikdy to pořádně nefungovalo, kompilátory to ignorovaly
 - ▶ deprecated v C++11
 - ▶ zcela odstraněno v C++17

```
void func1() throw() { ... } // nevyvolává výjimky
void func2() throw(int) { ... } // vyvolává int
void func3() throw(int, char) { ... } // vyvolává int i char
```

noexcept^{C++11}

- **noexcept**^{C++11} – nově zjednodušuje předchozí systém a slouží k označení funkcí a metod, které nevyvolávají výjimky
 - ▶ vyvolání výjimky z označené funkce by vedlo k vyvolání `terminate` a ukončení programu



```
void func() noexcept { ... }
```

Výjimky v C++ knihovně

std::exception

- `std::exception` je třída ze které dědí všechny standardní výjimky definované v knihovně C++
- objekt výjimky obsahuje pouze textovou informaci o druhu chyby (**virtual const char*** `what()`**const**)
- definováno v hl. s. `<exception>`
 - ▶ potomci definováni v hl. s. `<stdexcept>`

Potomci `std::exception`

- `logic_error`
 - ▶ `invalid_argument`
 - ▶ `domain_error`
 - ▶ `length_error`
 - ▶ `out_of_range`
 - ▶ `future_error`^{C++11}
 - ▶ `bad_optional_access`^{C++17}
- `runtime_error`
 - ▶ `range_error`
 - ▶ `overflow_error`
 - ▶ `underflow_error`
 - ▶ `regex_error`^{C++11}
 - ▶ `system_error`^{C++11}
 - ★ `ios_base::failure`^{C++11}
 - ★ `filesystem::filesystem_error`^{C++17}

Potomci `std::exception...`

- `bad_typeid`
- `bad_cast`
 - ▶ `bad_any_cast`^{C++17}
- `bad_weak_ptr`^{C++11}
- `bad_function_call`^{C++11}
- `bad_alloc`
 - ▶ `bad_array_new_length`^{C++11}
- `bad_exception`
- `bad_variant_access`^{C++17}

Function try

- ▶ celé tělo funkce nebo metody lze zabalit do bloku try
 - ▶ funguje i na konstruktory a inicializační část

```
struct Trida {  
  
    Trida(const std::string& atr) try : _atr(atr) {  
        // ...  
    } catch (const std::exception& ex) {  
        std::cerr << "oops";  
    }  
  
    std::string _atr;  
}
```

Jmenné prostory

Jmenné prostory

- řeší konflikty jmen
 - ▶ v C existuje jediný globální prostor
 - ▶ knihovní funkce se prefixují, aby nedošlo ke shodě (`glutCreateWindow`, `png_image_begin_read_from_file`)
- jmenné prostory jsou pouze balíkem funkcí, typů, proměnných, ...
 - ▶ neřeší se zde viditelnost (neexistuje obdoba `package-private`)
 - ▶ lze vytvořit anonymní jmenný prostor pro ukrytí členů jen v rámci jednoho souboru
- není definováno striktní fyzické uspořádání souborů
 - ▶ Java – balíček = složka, obsahuje pouze soubory v této složce
 - ▶ C++ – jmenný prostor může být použit kdekoliv
 - ▶ C++ – do existujícího jmenného prostoru je možno kdykoliv cokoliv přidat
- jmenné prostory je možné vnořovat

Vytvoření jmenného prostoru

```
namespace názevJmennéhoProstoru {  
    [deklarace/definice složek přidávaných do jm. prostoru]  
}
```

- jm. p. je možné vytvořit na globální úrovni nebo ve jmenném prostoru
- jm. p. je možné opakovaně definovat – vždy se přidají nové prvky do prostoru
- přístup k prvkům jm. p. je pomocí operátoru ::

Přístup k prvkům jmenného prostoru z vnitřku



```
namespace System {  
  
    void vymazKonzoli() { ... }  
  
    void ukonciProgram() {  
        vymazKonzoli();  
    }  
}
```

Přístup k prvkům jmenného prostoru zvenčí



```
namespace System {  
  
    void vymazKonzoli() { ... }  
  
    void ukonciProgram() { ... }  
}
```



```
System::vymazKonzoli();  
System::ukonciProgram();
```

Přístup k prvkům jmenného prostoru zvenčí – direktiva using



```
namespace System {  
  
    void vymazKonzoli() { ... }  
  
    void ukonciProgram() { ... }  
}
```



```
using namespace System;  
  
vymazKonzoli();  
ukonciProgram();
```


Přístup k prvkům jmenného prostoru zvenčí – deklarace using



```
namespace System {  
  
    void vymazKonzoli() { ... }  
  
    void ukonciProgram() { ... }  
}
```



```
using System::vymazKonzoli;  
  
vymazKonzoli();  
System::ukonciProgram(); // nelze "ukonciProgram()!"
```



- deklarace/direktiva **using** by neměla být používána v hlavičkových souborech!!!
 - ▶ každý, kdo provede **#include** na daný soubor bude ovlivněn usingy
 - ★ pozor na řetězení závislostí (hrac.h → pohyblivy_objekt.h → objekt.h)
 - ▶ může způsobit opět konflikty jmen

Otevřenost jmenných prostorů

- do jmenného prostoru jde vždy přidávat nové složky



```
namespace Objekty {  
    Kocka micka;  
}
```

```
// ...
```

```
namespace Objekty {  
    Kocka mourek;  
}
```

```
Objekty::micka.mnoukni();  
Objekty::mourek.mnoukni();
```

Vnořování jmenných prostorů

- jmenné prostory lze libovolně vnořovat



```
namespace Hra {  
    namespace Objekty {  
        namespace ZiveObjekty {  
            struct NPC { ... };  
        }  
    }  
}
```



```
Hra::Objekty::ZiveObjekty::NPC obchodnik{}
```

Vnořování jmenných prostorů...

- od C++17 podpora pro zkrácený zápis vnořených jm. p.



```
namespace Hra::Objekty::ZiveObjekty {  
    struct NPC { ... };  
}
```



```
Hra::Objekty::ZiveObjekty::NPC obchodnik{}
```

Anonymní jmenné prostory

- lze použít pro skrytí jeho složek jen pro daný soubor .cpp

```
namespace {  
    int _anonymous = 123;  
}
```

```
// dále ve stejném souboru  
cout << _anonymous;
```

C++ - Šablony

Ing. Roman Diviš

UPCE/FEI/KST

1 Šablony

- Generické programování
- Šablona a její parametry
- Použití šablon
 - Šablony funkcí
 - Šablony objektových typů
- Vnořené šablony
- Závislá jména
- Specializace šablon
 - Explicitní specializace
 - Parciální specializace

Šablony

Generické programování

```
void Vypis(int hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(int hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(std::string ↵  
    hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(int hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(std::string ↵  
    hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(Kocka& hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(int hodnota) {  
    cout << hodnota << endl;  
}
```

```
struct PoleIntu {  
    int* _pole;  
};
```

```
void Vypis(std::string ↵  
    hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(Kocka& hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(int hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(std::string ↵  
    hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(Kocka& hodnota) {  
    cout << hodnota << endl;  
}
```

```
struct PoleIntu {  
    int* _pole;  
};
```

```
struct PoleStringu {  
    std::string* _pole;  
};
```

```
void Vypis(int hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(std::string ↵  
    hodnota) {  
    cout << hodnota << endl;  
}
```

```
void Vypis(Kocka& hodnota) {  
    cout << hodnota << endl;  
}
```

```
struct PoleIntu {  
    int* _pole;  
};
```

```
struct PoleStringu {  
    std::string* _pole;  
};
```

```
struct PoleKoccek {  
    Kocka* _pole;  
};
```



```

void Vypis(int hodnota) {
    cout << hodnota << endl;
}
void Vypis(std::string ↵
    hodnota) {
    cout << hodnota << endl;
}
void Vypis(Kocka& hodnota) {
    cout << hodnota << endl;
}
void Vypis(double hodnota) {
    cout << hodnota << endl;
}
void Vypis(bool hodnota) {
    cout << hodnota << endl;
}
void Vypis(Pes& hodnota) {
    cout << hodnota << endl;
}
void Vypis(Kralik& hodnota) ↵
    {
        cout << hodnota << endl;
    }
}

```

```

struct PoleIntu {
    int* _pole;
};
struct PoleStringu {
    std::string* _pole;
};
struct PoleKoccek {
    Kocka* _pole;
};
struct PoleDoublu {
    double* _pole;
};
struct PoleBoolu {
    bool* _pole;
};
struct PolePsu {
    Pes* _pole;
};
struct PoleKraliku {
    Kralik* _pole;
};

```

```

void Vypis(int hodnota) {
    cout << hodnota << endl;
}
void Vypis(std::string hodnota) {
    cout << hodnota << endl;
}
void Vypis(Kocka& hodnota) {
    cout << hodnota << endl;
}
void Vypis(double hodnota) {
    cout << hodnota << endl;
}
void Vypis(bool hodnota) {
    cout << hodnota << endl;
}
void Vypis(Pes& hodnota) {
    cout << hodnota << endl;
}
void Vypis(Kralik& hodnota) {
    cout << hodnota << endl;
}
void Vypis(unsigned int hodnota) {
    cout << hodnota << endl;
}
void Vypis(long long hodnota) {
    cout << hodnota << endl;
}
void Vypis(exception& hodnota) {
    cout << hodnota << endl;
}

```

```

struct PoleIntu {
    int* _pole;
};
struct PoleStringu {
    std::string* _pole;
};
struct PoleKoccek {
    Kocka* _pole;
};
struct PoleDoublu {
    double* _pole;
};
struct PoleBoolu {
    bool* _pole;
};
struct PolePsu {
    Pes* _pole;
};
struct PoleKraliku {
    Kralik* _pole;
};
struct PoleUIntu {
    unsigned int* _pole;
};
struct PoleLongLong {
    long long* _pole;
};
struct PoleException {
    exception* _pole;
};

```

Generické programování

- využívá abstraktních vzorů funkcí a tříd
- základem jsou šablony (templates)

Šablony

- podobné makrům preprocesoru
- zpracovává je kompilátor
- umožňují pracovat s parametry, které jsou obecného datového typu
- jedna šablona může definovat celou množinu funkcí/tříd



```
template<typename T>
void Vypis(T& hodnota) {
    cout << hodnota << endl;
}
```

```
template<typename T>
struct Pole {
    T* _pole;
};
```

Šablona třídy?

- šablona
 - ▶ to co programátor vytvoří
 - ▶ kus zdrojového kódu, který se vlastně ani použít nemusí
- instance šablony
 - ▶ konkrétní datový typ
 - ▶ vytvoří kompilátor (automaticky/na vyžádání)
 - ▶ kus binárního kódu, který se pak používá
- objekt
 - ▶ typu „instance šablony“

Programátor

Kompilátor

Šablona

```
template<typename T>  
struct Pole {  
    T* _pole;  
};
```

Šablona

```
template<typename T>
struct Pole {
    T* _pole;
};
```

Instance šablony

```
// vytvořena automaticky na pozadí
```

```
struct .?AU?$Pole@H@@ {
    int* _pole;
};
```


Šablona

```
template<typename T>
struct Pole {
    T* _pole;
};
```

Instance šablony

```
// vytvořena automaticky na pozadí
```

```
struct .?AU?$Pole@H@@ {
    int* _pole;
};
```

Objekt

```
Pole<int> poleIntu{};
```

```
.?AU?$Pole@H@@ poleIntu{};
```

Šablona a její parametry

- je generický předpis funkce, typu nebo metody
- definuje seznam parametrů šablony (typové, hodnotové)
 - ▶ uvnitř šablony jsou to konstanty
- pro jednu funkci, typ nebo metodu může existovat více variant šablon
 - ▶ základní šablona
 - ▶ explicitní specializace šablony
 - ▶ parciální specializace šablony (pouze šablony typů)

Šablona

```
template<parametrŠablony... , ...>  
deklaraceNeboDefiniceFunkceTypuMetody;
```

parametrŠablony:

typename|**class** názevParametru

typHodnotovéhoTypu názevParametru

template<...> **typename** názevParametru

Parametry šablony

- uvnitř instance šablony představují **konstantní hodnoty**
- mohou představovat typ, konkrétní hodnotu (číslo, ukazatel, ...) nebo vnořenou šablonu

Typové parametry

- představují libovolný datový typ (primitivní i objektový)
- jakmile je zvolen je uvnitř šablony uplatňována standardní striktní typová kontrola
- používá se zástupné slovo **typename** nebo **class** (starší, stále dostupné z důvodu zpětné kompatibility)



```
template<typename ParametrSablony>
struct Pole {
    ParametrSablony _array[10];
};

Pole<int> poleIntu{};
Pole<Kocka> poleKocek{};
```

Hodnotové parametry

- představují konkrétní (číslnou) hodnotu
 - ▶ celočíselný parametr
 - ▶ výčtový
 - ▶ ukazatel na objekt
 - ▶ reference na objekt
 - ▶ ukazatel na funkci
 - ▶ třídní ukazatel
- uvnitř šablony představují neměnné konstanty

✓

```
template<int CiselnParametr>
struct Pole {
    int _array[CiselnParametr];
};
```

```
Pole<10> poleDesetiIntu{};
Pole<50> polePadesatiIntu{};
```

Hodnotové parametry...



```
enum TypObjektu { Kamen, Strom, Ker, Trava };
```

```
template<TypObjektu typ>
```

```
struct Pole {  
    void* _array[10];  
};
```

```
Pole<Kamen> poleSutru{};
```

```
Pole<Strom> poleStromecku{};
```



```
void func1(Object* ptr) { ... }
```

```
void func2(Object* ptr) { ... }
```

```
template<void (*function) (Object*)>
```

```
struct Pole {  
    void* _array[10];  
};
```

```
Pole<func1> poleSFunkci1{};
```

```
Pole<func2> poleSFunkci2{};
```

Parametrů může být více



```
template<typename Typ, int VelikostPole>
struct Pole {
    Typ _array[VelikostPole];
};
```

```
Pole<Kamen, 10> poleDesetiSutru{};
Pole<Strom, 3> poleTriStromecku{};
```

Variadic templates^{C++11}

- ▶ C++11 zavedlo šablony s proměnným počtem parametrů
- ▶ **template**<**typename**... args> **struct** StructWithVarArgs { };
- ▶ část parametrů může být specifikována konkrétně
 - ▶ **template**<**typename** First, **typename**... Rest> **struct**...

Instance šablony

- instance šablony představuje konkrétní datový typ nebo kód funkce, který překladač vytvoří po dosazení parametrů šablony
- kompilátor ji vytváří automaticky (při použití typu/funkce)
 - ▶ lze také vynutit explicitní vytvoření
- **pokud chcete, aby kompilátor zkontroloval „obsah“ šablony, je potřeba instance šablony a její použití!**
- **aby mohla vzniknout instance šablony, je nutné znát kompletní definici šablonové funkce/typu (proto se šablony celé píšou do hlavičkových souborů)**

Instance šablony...



```
template<typename T>
struct Pole {
    ...
};
```



```
// explicitní vytvoření instance šablony Pole<int>
template struct Pole<int>;

// implicitní vytvoření instance šablony Pole<string>
Pole<string> globalniPoleStringu{};
```

Použití šablon

Šablony funkcí

- funkce podporují automatické odvození parametrů šablony
 - ▶ parametry musí vhodně odpovídat, nesmí dojít ke sporné situaci



```
template<typename Typ>
void vypis(const Typ& vystup) {
    cout << vystup;
}
```



```
int intyxr = 123;
double dabl = 3.141592;
```

```
vypis<int>(intyxr); // explicitní uvedení parametrů šablony
vypis(dabl); // kompilátor automaticky odvodí parametry šablony
```



```
template<typename Typ>
bool porovnej(const Typ& a, const Typ& b) {
    return a == b;
}
```



```
int intyZrAlfa = 123;
int intyZrBeta = 456;
double dabl = 3.141592;

bool vysledek;
vysledek = porovnej(intyZrAlfa, intyZrBeta);
```



```
vysledek = porovnej(intyZrAlfa, dabl); // int,double?!?
```

Šablony objektových typů

- u struktur a tříd je nutné vždy definovat parametry šablony
 - ▶ při splnění určitých podmínek je automatické odvození parametrů podporováno od C++17



```
template<typename Typ>
struct Pole {
    ...
};
```



```
Pole<int> poleIntu{};
Pole<string> poleStringu{};
```


- složky definované uvnitř typu zapisujeme bez úprav



```
template<typename Typ>
struct Pole {

    const Typ& dej(int index) const {
        return _pole[index];
    }

private:
    Typ _pole[10];
};
```

- složky definované vně typu musí oznámit kompilátoru, že se jedná o šablonu
 - ▶ je nutné definovat s jakými parametry pracujeme



```
template<typename Typ>
struct Pole {

    const Typ& dej(int index) const;

private:
    Typ _pole[10];
};

template<typename Typ> // ← je to šablona a jaké má parametry
//=====
const Typ& Pole<Typ>::dej(int index) const {
    //          ===== ← jaká konkrétní varianta šablony to je
    return _pole[index];
}
```

- statické atributy se definují jako šablony



```
template <typename T>
struct Pole {

    static int PocetPoli;

};

template <typename T>
int Pole<T>::PocetPoli = 0;
```

Vnořené šablony

- představuje šablonu v šabloně
- ve vnořených šablonách je možné pracovat s
 - ▶ aktuálními šablonovými parametry
 - ▶ i s parametry nadřazených šablon
- při vytvoření instance nadřazené šablony nevznikají automaticky instance vnořených šablon

- uvnitř šablony je možné definovat další šablony (vnořené typy, metody)



```
template <typename T>
struct Pole {

    template <typename U>
    U collect(U (*funkce)(U, T*), U init) {
        U tmp = init;
        for (auto&& item : _pole)
            tmp = funkce(tmp, &item);

        return tmp;
    }

private:
    T _pole[10];
};
```

- vnější definice vnořené šablony vyžaduje definování všech šablon



```
template <typename T>
struct Pole {

    template <typename U>
    U collect(U (*funkce)(U, T*), U init);
};

template <typename T>
template <typename U>
U Pole<T>::collect(U (*funkce)(U, T*), U init) {
    U tmp = init;
    for (auto&& item : _pole)
        tmp = funkce(tmp, &item);

    return tmp;
}
```

Závislá jména

- pokud pracujeme s obecným typem a ten představuje komplexní typ může se objevit problém závislých jmen
- nelze přímo pracovat s vnořenými typy pod obecným typem (nebo závislých na obecném typu)
 - ▶ kompilátor je nevidí
- nelze přímo pracovat s vnořenými šablonami pod obecným typem (nebo závislých na obecném typu)
 - ▶ kompilátor je nevidí

×

```
template<typename T>
struct S {
    T::iterator _iter; // ← error — neznám "T::iterator"
};
```

- vnořený typ je nutné označit pomocí **typename**

×

```
template<typename T>
struct S {
    T::iterator _iter; // ← error — neznám "T::iterator"
};
```

✓

```
template<typename T>
struct S {
    typename T::iterator _iter;
};
```

- **typedef** rovněž musí použít **typename**

×

```
template<typename T>
struct S {
    typedef T::iterator TIterator; // ← error — neznám "T::iterator"
    TIterator _iter;
};
```

✓

```
template<typename T>
struct S {
    typedef typename T::iterator TIterator;
    TIterator _iter;
};
```

- pro použití vnořených šablon je potřeba **template**

×

```
struct Usage {  
    template<typename U>  
    void method() {  
        ...  
    }  
};
```

// předpokládejme T = Usage

```
template<typename T>  
struct Template {  
    void m() {  
        T t{};  
        t.method<int>(); // ← error — neznám "t.method"  
    }  
};
```

- pro použití vnořených šablon je potřeba **template**



```
struct Usage {  
    template<typename U>  
    void method() {  
        ...  
    }  
};  
  
// předpokládejme T = Usage  
template<typename T>  
struct Template {  
    void m() {  
        T t{};  
        t.template method<int>();  
    }  
};
```

- oba uvedené případy se mohou i kombinovat...



```
struct Usage {  
    template<typename TT>  
    struct NestedTemplateStruct {  
        struct iterator {  
            int _x = 123;  
        };  
    };  
};
```

// předpokládejme T = Usage

```
template<typename T>  
struct Template {  
  
    void m() {  
        T t{};  
        typename T::template NestedTemplateStruct<int>::iterator iteratorVar;  
        cout << iteratorVar._x << endl;  
    }  
};
```

Specializace šablon

Explicitní specializace

Explicitní specializace

- představuje jinou realizaci šablony (vlastní kód je nově definovaný) pro konkrétní sadu parametrů
 - ▶ s původní šablonou sdílí jenom jméno (kód ne!)
- použitelné pro šablony funkcí i typů
- může existovat mnoho explicitních specializací jedné šablony

- ▶ definuje se po definici základní šablony
- ▶ definuje se jako šablona, ale
 - ▶ nemá obecné šablonové parametry (uvádí se **template**<>)
 - ▶ za názvem typu/funkce se uvádí konkrétní sada parametrů, pro kterou tato šablona bude použita (... sablonaFunkce<**int**>())...

Explicitní specializace funkce



```
// Základní šablona
```

```
template<typename T>
```

```
void printTemplate(string name, T value) {  
    cout << name << ": " << value << endl;  
}
```

```
// Explicitní specializace
```

```
template<>
```

```
// explicitní specializace je šablonou, ale všechny parametry jsou konkrétní ↵  
    hodnoty
```

```
void printTemplate<double>(string name, double value) {  
    cout << name << ": " << fixed << setprecision(3) << value << ↵  
    endl;  
}
```

```
printTemplate("Char", 'x'); // Char: 'x'
```

```
printTemplate("Int", 15); // Int: 15
```

```
printTemplate("Double", 3.14); // Double: 3.140
```

Explicitní specializace třídy



// Základní šablona

```
template<typename T>
```

```
struct Comparator : ComparatorBase {
```

```
    int compare(const T& a, const T& b) const;  
};
```

```
template<typename T>
```

```
int Comparator<T>::compare(const T& a, const T& b) const {  
    return b - a;  
}
```

Explicitní specializace třídy...



```
// Explicitní specializace
```

```
template<>
```

```
struct Comparator<const char*> : ComparatorBase {
```

```
    int compare(const char* a, const char* b) const;
};
```

```
// metody explicitně specializované třídy se definují bez template<>
```

```
int Comparator<const char*>::compare(const char* a, const char* b) ←
    const {
    return strcmp(a, b);
}
```

Explicitní specializace třídy...



```
struct ComparatorBase {  
    constexpr static int FIRST_IS_BIGGER = 1;  
    constexpr static int EQUAL = 0;  
    constexpr static int SECOND_IS_BIGGER = -1;  
};
```



```
Comparator<int> cInt{};  
cout << (cInt.compare(20, 40) == Comparator<int>::FIRST_IS_BIGGER ←  
    ? "20 > 40" : "20 <= 40") << endl;  
  
Comparator<const char*> cCstr{};  
int result = cCstr.compare("hello", "hella");
```

Parciální specializace

Parciální specializace

- představuje jinou realizaci šablony (vlastní kód je nově definovaný) pro částečně specifikované parametry šablony
 - ▶ s původní šablonou sdílí jenom jméno (kód ne!)
- použitelné pouze pro šablony typů
- může existovat mnoho parciálních specializací jedné šablony

- ▶ definuje se po definici základní šablony
- ▶ definuje se jako šablona, ale
 - ▶ má obecné/konkrétní/částečně specifikované šablonové parametry (uvádí se `template<...>`)
 - ▶ za názvem typu/funkce se uvádí konkrétní sada parametrů, pro kterou tato šablona bude použita (... `ŠablonaTridy<int, T>()`...)

Co a jak lze specializovat - parciálně?



```
template<typename T, typename U, int S>  
struct Template { };
```

```
// lze některé parametry přesně definovat (jako u explicitní specializace)  
// zde definujeme S
```

```
template<typename T, typename U>  
struct Template<T, U, 10> { };
```

```
// zde definujeme U a S
```

```
template<typename T>  
struct Template<T, int, 10> { };
```

```
// lze některé typové parametry označit za obecné ukazatele
```

```
// zde definujeme T jako libovolný ukazatel
```

```
template<typename T, typename U, int S>  
struct Template<T*, U, S> { };
```

Jedna věc může být definována jako

- základní šablona
- několik parciálních specializací
- několik explicitních specializací

- kompilátor pak vybírá nejvhodnější sadu parametrů
 - ▶ přesná shoda s explicitní specializací
 - ▶ shoda s parciální specializací
 - ▶ jako nouzovka – základní šablona
- nesmí dojít ke shodě u více parciálních specializací
 - ▶ $\langle T^*, U \rangle$ a $\langle T, U^* \rangle$ – co použít pro $\langle \text{int}^*, \text{int}^* \rangle$

Parciální specializace...



```
template<typename T>
struct Comparator<T*> : ComparatorBase {

    int compare(T* a, T* b) const;
};

template<typename T>
int Comparator<T*>::compare(T* a, T* b) const {
    return *b - *a;
}
```

Parciální specializace. . .



```
Comparator<int*> cInt{};
```

```
int* a = new int{ 20 };
```

```
int* b = new int{ 40 };
```

```
cout << (cInt.compare(a, b) == Comparator<int>::↵  
    FIRST_IS_BIGGER ? "20 > 40" : "20 <= 40") << endl;
```

C++ - Datové proudy

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Datové proudy

- Soubory
- Paměťové proudy
- Praktické problémy – přenositelnost a funkčnost
 - Textové soubory
 - Binární soubory

Datové proudy

string

<string>

- třída `string` je realizována šablonou `basic_string<char>`
- rozhraní zapadá do konceptu STL kontejnerů (viz později), chová se jako kontejner znaků
- umožňuje jednoduše upravovat, vyhledávat, procházet řetězec



```
string s1 = "hello world";  
string s2{ "hello world" };  
string s3 = s2;  
string s4 = s1 + s2;  
  
cout << s4.c_str() << endl;
```


string – základní metody a vlastnosti

- **operator[]** – procházení znaků
- **c_str()** – konverze na **const char***
- **begin()**, **end()** – iterátory
- **empty()**, **size()** – test prázdnoty/test délky řetězce
- **insert()**, **erase()** – vkládání, mazání znaků
- **substr()** – výběr podřetězce
- **find()**, **rfind()** – hledání podřetězce
- **find_first_of()**, **find_first_not_of()** – hledání znaků
- **find_last_of()**, **find_last_not_of()** – hledání znaků
- porovávací operátory, **replace()**, **at()**, **data()**, ...

Datové proudy

- nízkourovňové datové proudy zabaleny do objektů s jednoduchým rozhraním
- proudy pro práci s konzolí, soubory a paměťový proud
- jedná se o součást STL, ale neposkytují iterátorové rozhraní
 - ▶ v knihovně jsou k dispozici adaptéry pro napojení na iterátory
- vstup a výstup dat pomocí metod a přetížených operátorů (<<, >>)

| objekt | instance třídy | popis | lowlevel proud (C) |
|--------|----------------|----------------|--------------------|
| cout | ostream | výstupní proud | stdout |
| cin | istream | vstupní proud | stdin |
| cerr | ostream | chybový výstup | stderr |



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

int main(void)
{
    int cislo;

    // výpis textu na obrazovku s odřádkováním
    cout << "Napis cislo" << endl;
    // načtení celého čísla z klávesnice
    cin >> cislo;

    cout << "Napsal jsi: " << cislo << endl;
    return 0;
}
```

Manipulátory

- formát výstupu lze jednoduše přizpůsobit pomocí manipulátorů
 - ▶ jednoduché objekty, které přenastaví proud
- knihovna obsahuje řadu připravených manipulátorů
 - ▶ lze definovat vlastní manipulátory
- hlavičkový soubor `<iomanip>`



```
int val = 15;
cout << val << " 0x" << hex << val << endl;
// vypíše: 15 0xf
```

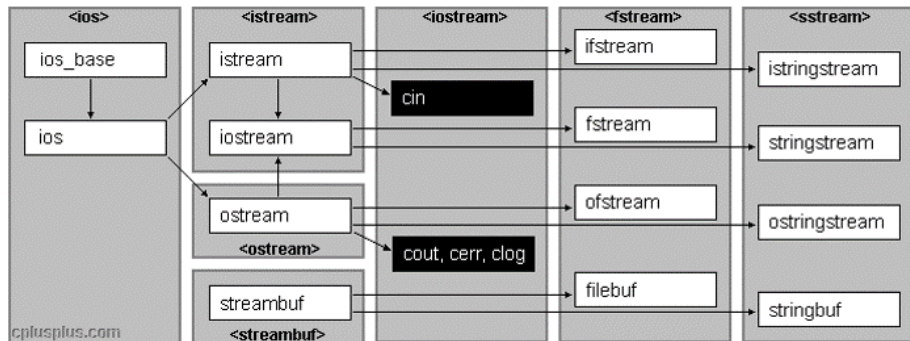
Knihovní manipulátory

- endl – konec řádku a vyprázdnění bufferu
- flush – vyprázdnění bufferu
- dec, hex, oct – výpis čísel v desítkové, šestnáctkové nebo osmičkové soustavě
- setbase(int) – výpis čísel v zadané soustavě
- setw(int) – nastaví šířku vypisované hodnoty (na kolik znaků zarovnávat)
- setfill(char) – nastaví vyplňovací znak
- setprecision(int) – nastaví vypisovanou přesnost reálných čísel



```
int c = 7;
cout << setw(3) << c << endl;
cout << setw(4) << setfill('@') << hex << c << endl;
```

Hierarchie tříd proudů



Přetěžování operátorů <<, >> pro práci s proudy

- musí se přetěžovat jako obyčejné funkce
- musí se zajistit řetězení volání
 - ▶ z funkce se vrací stejný proud pomocí reference
- vypisovaný/načítaný objekt se předává pomocí reference
 - ▶ při načítání už musí existovat instance objektu, její obsah bude přepsán

```
// operátor pro zápis objektu do výstupního proudu  
ostream& operator<<(ostream& os, const TRIDA& obj)
```

```
// operátor pro načtení objektu ze vstupního proudu  
istream& operator>>(istream& is, TRIDA& obj)
```

Přetěžování operátorů <<, >> pro práci s proudy



```
struct KomplexniCislo {  
  
    KomplexniCislo(double r, double i) : re(r), im(i) { }  
  
    double re;  
    double im;  
};  
  
ostream& operator<<(ostream& os, const KomplexniCislo& obj) {  
    os << obj.re << " " << obj.im;  
    return os;  
}  
  
istream& operator>>(istream& is, KomplexniCislo& obj) {  
    is >> obj.re >> obj.im;  
    return is;  
}
```


Přetěžování operátorů <<, >> pro práci s proudy



```
KomplexniCislo kc{10, 2};
```

```
cout << "Komplexni cislo je: " << kc;
```

```
cin >> kc;
```

Soubory

<fstream> třídy

- `ifstream` – čtení ze souboru
- `ofstream` – zápis do souboru
- `fstream` – čtení i zápis do souboru

režimy

- `ascii`
- `binární`

Zápis do souboru



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

void main()
{
    ofstream out{};
    out.open("pokus.txt");

    if (out.is_open())
    {
        out << "radka textu";
        out.close();
    }
    else
        cerr << "Soubor se nepodarilo otevrit...";
}
```

Čtení ze souboru



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

void main()
{
    string slovo{};
    ifstream in{};
    in.open("temp.txt");

    if (in.is_open())
    {
        while(in >> slovo)
            cout << slovo << " ";
        cout << endl;
    }
}
```

Kopie souboru po znacích



```
void main()
{
    ifstream from{"soubor1.txt"};
    ofstream to{"soubor2.txt"};
    char ch;

    while(from.get(ch))
        to.put(ch);

    from.close();
    to.close();
}
```

Metody/funkce pro načítání dat



```
char znak;  
char text[50];  
  
ifstream in{};  
in.open("pokus.txt");  
  
in.get(znak); // načte znak  
in >> text; // načte text po první bílý znak (dle nastavení proudu)  
in.getline(text,50); // načte řádek (max. 50 znaků)  
in.read(text, 50); // načte 50 znaků  
  
while (in.get(znak)){ ... } // čte dokud jsou k dispozici znaky  
  
in.close();
```

Režimy práce se souborem

Režimy jsou bitové flagy, lze je kombinovat:

- `in` – čtení dat ze souboru (výchozí flag pro `ifstream`)
- `out` – zápis dat do souboru (výchozí flag pro `ofstream`)
- `ate` (at the end) – přesuň kurzor na konec souboru
- `app` (append) – přidávání dat na konec souboru
- `trunc` (truncate) – vymaže obsah souboru
- `binary` – binární režim



```
ofstream out{};  
out.open("pokus.txt", ios_base::app | ios_base::out);
```


Binární režim

- data ukládána dle formátu v paměti počítače
 - ▶ úsporné – každé **int** číslo má pevně 4 B (ať je to 0, 1 000 000 nebo 4 miliardy)
 - ▶ rychlé – odpadá konverze na ascii reprezentaci a zpět
 - ▶ pevný formát dat – možnost rychlého prohledávání obsahu souboru
 - ▶ „hůře čitelné“ – nelze použít textový editor
 - ▶ „přenositelné“ – problém přenosu mezi big-endian a little-endian platformami
 - možnost ukládat celé struktury v jedné operaci
 - ▶ pozor na ukazatele! nutno řešit ručně
-
- `stream.write(pointer, size)` – zapíše blok dat do proudu
 - `stream.read(pointer, size)` – přečte blok dat z proudu

Zápis do binárního souboru



```
int pole[4] = {1, 2, 3, 4};
ofstream out{};
out.open("vystup.dat", ios_base::binary);

if (out.is_open()) {
    out.write((char *)pole, sizeof(pole));
    out.close();
}
else
    cerr << "Nepodarilo se otevrit!" << endl;
```

Čtení z binárního souboru



```
char pole[4];
ifstream in{};
in.open("vstup.dat", ios_base::binary);

if (out.is_open()) {
    in.read((char *)pole, sizeof(pole));
    in.close();
}
else
    cerr << "Nepodarilo se otevrit!" << endl;
```

Posun kurzoru v souboru

seekg, seekp

- `seekg(pozice, vychoziBod = ios_base::beg)` (`seek get`) – posun čtecího kurzoru
- `seekp(pozice, vychoziBod = ios_base::beg)` (`seek put`) – posun zapisovacího kurzoru

výchozí bod

- `ios_base::beg` – počet bajtů od počátku souboru
- `ios_base::end` – počet bajtů od konce souboru
- `ios_base::cur` – počet bajtů od aktuální pozice kurzoru



```
inputfile.seekg(20, ios_base::beg);
```

Paměťové proudy

Paměťové proudy

<sstream>

- `ostringstream` – výstupní (lze do něj zapisovat) paměťový proud
- `istringstream` – vstupní (lze z něj číst) paměťový proud



```
ostringstream oss{};
oss << "vystupni" << ' ' << "datovy" << ' ' << "proud" << ' ' ↵
    << 12345;

string s = oss.str();
cout << s << endl;
```

Paměťové proudy...



```
string s = "jan maly 123456";  
istringstream iss{ s };  
string jmeno;  
string prijmeni;  
int id;  
  
iss >> jmeno >> prijmeni >> id;  
  
cout << "J:" << jmeno << " P:" << prijmeni << " I:" << id;
```

Paměťové proudy...

- použití pro konverze datových typů, zpracování textu v souborech, ...
- do C++11 jediný způsob dle C++ standardu a knihovny pro konverzi datových typů `string` \leftrightarrow `int`

Konverze string \leftrightarrow int

<string> konverzní funkce^{C++11}

- `std::string to_string(int/long/float/double)` – převod na string
- `stoul()`, `stoull()` – převod na **unsigned** čísla
- `stoil()`, `stol()`, `stoll()` – převod na **signed** čísla
- `stof()`, `stod()`, `stold()` – převod na desetinná čísla

Praktické problémy – přenositelnost a funkčnost

Textové soubory

ofstream - různé formáty textu



```
ofstream outputFile{ "out.txt" };  
// char  
outputFile << "První řetězec 1234567890 ěščřžýáíů" << endl;  
// wchar_t  
outputFile << L"První řetězec 1234567890 ěščřžýáíů" << endl;  
// utf-8 (char)  
outputFile << u8"První řetězec 1234567890 ěščřžýáíů" << endl;  
// utf-16 (char16_t)  
outputFile << u"První řetězec 1234567890 ěščřžýáíů" << endl;  
// utf-32 (char32_t)  
outputFile << U"První řetězec 1234567890 ěščřžýáíů" << endl;  
outputFile.close();
```

Náhled v režimu ANSI (cp1250)

- korektně se zapsaly pouze **char** a u8 řetězce

out.txt outw.txt

```
1 Prvni retezec 1234567890 ěščřžýáíěů
2 009AA5C0
3 Prvni retezec 1234567890 Ä>Í~ÄŤÍ™ÍIÄ~Ä~Ä-Ä©ÍŽ
4 009AA650
5 009AA6A8
6
```

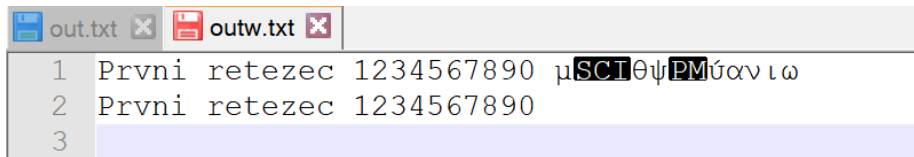
wofstream - různé formáty textu



```
wofstream outputFile{ "outw.txt" };  
// char  
outputFile << "Prvni retezec 1234567890 ěščřžýáíů" << endl;  
// wchar_t  
outputFile << L"Prvni retezec 1234567890 ěščřžýáíů" << endl;  
// utf-8 (char)  
outputFile << u8"Prvni retezec 1234567890 ěščřžýáíů" << endl;  
// utf-16 (char16_t)  
outputFile << u"Prvni retezec 1234567890 ěščřžýáíů" << endl;  
// utf-32 (char32_t)  
outputFile << U"Prvni retezec 1234567890 ěščřžýáíů" << endl;  
outputFile.close();
```

Náhled v režimu ANSI (cp1250)

- korektně není zapsán žádný řetězec!
- i u `wchar_t` došlo k ořezu diakritických znaků



```
out.txt outw.txt
1 První řetězec 1234567890 μSCIθψPMúανιω
2 První řetězec 1234567890
3
```

basic_ofstream<char16_t> - různé formáty textu



```
// utf-16 (char16_t)
std::basic_ofstream<char16_t> outputFile{ "out16.txt" };
outputFile << u"První řetězec 1234567890 ěščřýáíů" << endl;
outputFile.close();
```


- ✗ LNK2001 Nerozpoznaný externí symbol "__declspec(dllimport)"
public: static class std::locale::id std::codecvt<char16_t, char,
struct _Mbstatet>::id" (__imp_?id@?\$codecvt@_SDU_Mbstatet@@@std@@2V0locale@2@A)
- ✗ LNK1120 Pořet nerozpoznaných externích typů: 1

- V rámci paměti lze používat řetězce v kódování nativním kódování, UTF-8, UTF-16, UTF-32
- V prostředí VS lze do souboru pouze zapisovat v nativním kódování nebo v UTF-8

Binární soubory

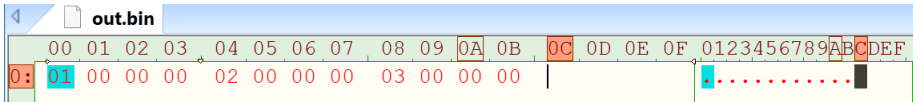
Zápis binárního souboru – struktura



```
struct JednoduchaStruktura {  
    int a;  
    int b;  
    int c;  
};  
  
void testZapisACteniBinarne() {  
    JednoduchaStruktura abc{ 1, 2, 3 };  
  
    ofstream binFile{ "out.bin" };  
    binFile.write((const char*)&abc, sizeof abc);  
    binFile.close();  
}
```

Náhled v HEX editoru

- `sizeof(int) == 4`
- `sizeof(JednoduchaStruktura) == 3*4 == 12`
- OK
- Pozor na přenos souboru na jinou platformu – pořadí bajtů ve skupině je nyní little-endian
 - ▶ Na systému big-endian by byly načteny hodnoty 16 777 216, 33 554 432, 50 331 648



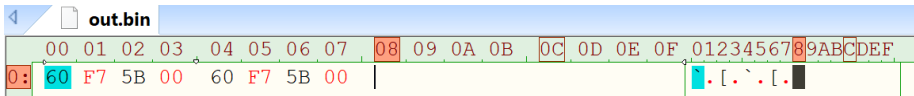
Zápis binárního souboru – struktura



```
struct SlozitejsiStruktura {  
    JednoduchaStruktura& ref;  
    JednoduchaStruktura* ptr;  
};  
  
void testZapisACteniBinarne() {  
    JednoduchaStruktura abc{ 1, 2, 3 };  
    SlozitejsiStruktura structure{ abc, &abc };  
  
    ofstream binFile{ "out.bin" };  
    binFile.write((const char*)&structure, sizeof structure);  
    binFile.close();  
}
```

Náhled v HEX editoru

- `sizeof(JednoduchaStruktura) == 12`
- Velikost souboru == 8
- Není OK
- Reference i ukazatele jsou v paměti reprezentovány stejně
 - ▶ Ukazatel (32/64 bitové číslo) na adresu do paměti
 - ▶ Binární zápis pouze zapíše hodnotu (tj. tu adresu), kterou ve struktuře přečte



Zápis binárního souboru – struktura



```
struct NevyrovnanaStruktura {  
    char a;  
    int b;  
    char c;  
    double d;  
};  
  
void testZapisACteniBinarne() {  
    NevyrovnanaStruktura structure{ 0x99, 0x12345678, 0x99, ↵  
        3.14};  
  
    ofstream binFile{ "out.bin" };  
    binFile.write((const char*)&structure, sizeof structure);  
    binFile.close();  
}
```


Zápis binárního souboru – struktura



```
struct Struktura {  
    string retezec;  
};  
  
void testZapisACteniBinarne() {  
    Struktura structure{ "ahoj svete" };  
  
    ofstream binFile{ "out.bin" };  
    binFile.write((const char*)&structure, sizeof structure);  
    binFile.close();  
}
```

Náhled v HEX editoru

- Velikost souboru == 29
- OK? Proč je to tolik?
- Je to třída a nese nějaké atributy navíc, ale jinak to jde zapsat i načíst

| | | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F 0123456789ABCDEF | | | | | | | | | | | | | | | |
|-----|----|--|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-----------------|
| 00: | B8 | FB | 03 | 00 | 61 | 68 | 6F | 6A | 20 | 73 | 76 | 65 | 74 | 65 | 00 | CC | ...ahoj svete.. |
| 10: | CC | CC | CC | CC | 0D | 0A | 00 | 00 | 00 | 0F | 00 | 00 | 00 | | | | |

Zápis binárního souboru – struktura



```
struct Struktura {  
    string retezec;  
};  
  
void testZapisACteniBinarne() {  
    Struktura structure{ "ahoj svete, dneska je moc pekne!" };  
  
    ofstream binFile{ "out.bin" };  
    binFile.write((const char*)&structure, sizeof structure);  
    binFile.close();  
}
```

Náhled v HEX editoru

- Velikost souboru == 28
- OK? Ne!
- Text není uložen v souboru!

| out.bin | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---------|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|---|
| | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F | 01 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
| 00: | 00 | EB | 1E | 00 | B8 | EC | 1E | 00 | CC | CC | CC | CC | CC | CC | CC | CC | . | | | | | | | | | | | | | | |
| 10: | CC | CC | CC | CC | 20 | 00 | 00 | 00 | 2F | 00 | 00 | 00 | | | | | . | | | | | | | | | | | | | | |

Implementace std::string v knihovně

- Liší se dle autora knihovny
- Někde je využívána přímo dynamicky alokovaná paměť
- Někde je používán hybridní přístup
 - ▶ Pro krátké řetězce (do 16 znaků) je přímo ve třídě staticky alokované pole, kam se uloží znaky
 - ▶ Delší řetězce vyžadují alokaci samostatného paměťového bloku a do struktury je uložen ukazatel
 - ▶ Tento přístup umožňuje předcházet dvojí alokaci paměti pro krátké řetězce
 - ▶ Je využíván i v rámci knihovny od Microsoftu

Zápis struktur do souboru

Lze při splnění podmínek:

- struktura neobsahuje referenční nebo ukazatelové atributy
- program má ošetřenou přenositelnost (little-endian/big-endian) pro případ přesunu na jinou platformu
- program má ošetřené zarovnání struktur pro případ přesunu na jiný kompilátor/platformu

Nelze tedy přímo zapisovat struktury u kterých si nejsme jisti splněním těchto podmínek (`std::vector`, `std::string`, ...)

C++ - Přetěžování operátorů, STL

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Přetěžování operátorů

2 STL

- C++11 a STL, kontejnery, iterátory, algoritmy

Přetěžování operátorů

Přetěžování operátorů

- lze přetížit operátory nad uživatelskými objektovými typy
 - ▶ použití daného operátoru nad objektem vyvolá uživatelem definovanou metodu/funkci
 - ▶ lze přetížit pouze vybrané existující operátory
- je to pouze syntax-sugar pro volání funkcí/metod
 - ▶ operátor `+` lze volat jako `a + b`, ale i jako metodu `a.operator+(b)`
 - ▶ v kombinaci s STL slouží k definování některých základních chování objektů

Přetěžování operátorů

- definování, jak se bude daný operátor chovat pro náš objektový typ
- 4 skupiny operátorů:
 - ▶ nelze přetížit
 - ▶ lze metodou
 - ▶ lze metodou i funkcí
 - ▶ lze funkcí/statickou metodou

× Nelze přetížit

- tečkové operátory – `.`, `.*`, `::`
- ternární operátor – `? :`
- přetypovací operátory – `static_cast`, ...
- další – `typeof`, `sizeof`

Přetížitelné operátory

Metodou

`= -> ->* [] () op. přetypování(typ)`

Metodou i funkcí

`+ - * / ++ -- > < >= <= == >> << & | ~ += -= *= /= ...`

Funkcí/statickou metodou

`new new[] delete delete[]`

návratovýTyp operator@([parametryOperátoru])

uvažujte výraz:

int a = 5, b = 10;

int c = a + b;

platí:

- operátor + je binární operátor
- operandy jsou *a*, *b*
- změní se proměnná *a*? NE → konstanta
- změní se proměnná *b*? NE → konstanta

návratovýTyp operator@([parametryOperátoru])

uvažujte výraz:

```
int a = 5, b = 10;
```

```
int c = a + b;
```



```
int operator+(int a, int b) {  
    return {a->state + b->state};  
}
```



```
struct int {  
    int operator+(int b) {  
        return {this->state + b->state};  
    }  
};
```


návratovýTyp operator@([parametryOperátoru])

uvažujte výraz:

```
int a = 5, b = 10;
```

```
int c = a + b;
```

✓

```
int operator+(const int& a, const int& b) {  
    return {a->state + b->state};  
}
```

✓

```
struct int {  
    int operator+(const int& b) const {  
        return {this->state + b->state};  
    }  
};
```

Reálný příklad – komplexní číslo



```
Komplex a = 5, b = 10;
```

```
Komplex c = a + b;
```

```
Komplex operator+(const Komplex& a, const Komplex& b) {  
    return {a->re + b->re, a->im + b->im};  
}
```

```
struct Komplex {  
    Komplex operator+(const Komplex& b) const {  
        return {this->re + b->re, this->im + b->im};  
    }  
};
```

Kombinace funkce + definice friend



```
struct Komplex {  
    friend Komplex operator+(const Komplex& a, const Komplex& b) {  
        return {a->re + b->re, a->im + b->im};  
    }  
};
```

friend + definice v těle způsobí:

- nejedná se o metodu, ale o funkci!
- má přístup k privátním a chráněným složkám

Operátorové metody – konstantnost

- konstatní metoda operátoru může být vyvolána i nad konstatním objektem!
- nekonstatní metoda operátoru může být vyvolána pouze nad nekonstatním objektem!

Většina operátorů nemění stav objektu

- $+$ $-$ $*$ $/$
- $<<$ $>>$ $\&$ $|$ \sim
- $<$ $>$ $>=$ $<=$ $==$ $!=$

Operátorové metody – konstantnost...

Operátory (potenciálně) měnící stav objektu

- ++, --
- =, +=, -=, *=, /=, ...
- [] – pokud vrací referenci/ukazatel na stav objektu
- () – pokud mění stav objektu

Typ návratové hodnoty

- operátory porovnání – **bool**
- ostatní – typ objektu nebo jiný libovolný typ

Proč vracet staticky alokovaný objekt? Možnosti:

- statický objekt – dochází ke kopírování, vytvoření dočasného objektu a jeho následné destrukci (pokud nedojde k optimalizaci od kompilátoru)
- reference
 - ▶ ale kde ji vezmu?
 - ▶ obyčejná proměnná zanikne s koncem bloku – nelze
 - ▶ statická (nebo globální) proměnná bude existovat dál, ale nebude to fungovat ve vícevláknových aplikacích – nelze
 - ▶ dynamicky alokovaná proměnná – ztratím ukazatel a informaci o alokaci a do konce programu bude paměť zabraná – nelze
- ukazatel (dyn. alokovaná paměť) – změna sémantiky operátoru – nelze

Využití přetížených operátorů

- `< > <= >= == !=`
 - ▶ řazení hodnot, asociativní kontejnery, jednoduché porovnávání
- `[]`
 - ▶ přístup k prvkům kontejneru
- `()`
 - ▶ funkční objekty (objekty určující chování algoritmů)
- `++ -- * --> -->*`
 - ▶ realizace iterátoru
- `++ -- + - * /`
 - ▶ matematické struktury (matice, komplexní čísla, ntice, ...), měnitelné hodnoty

další příklady k nalezení např. v knize Thinking in C++

[http://www.drbio.cornell.edu/pl47/programming/
TICPP-2nd-ed-Vol-one-html/Chapter12.html](http://www.drbio.cornell.edu/pl47/programming/TICPP-2nd-ed-Vol-one-html/Chapter12.html)

další zdroje

<http://en.cppreference.com/w/cpp/language/operators>

STL

STL – Standard Template Library

- představuje generickou knihovnu pro správu a zpracování dat
- jednotlivé komponenty jsou šablony
- základní komponenty jsou
 - ▶ kontejnery – slouží pro ukládání a organizaci dat
 - ▶ algoritmy – obecné algoritmy pro zpracování dat
 - ▶ iterátory – obecné „rozhraní“ mezi kontejnery a algoritmy
 - ▶ pomocné funkční objekty, adaptéry, bindery, ...

Funkční objekt

- vylepšení obyčejné funkce
- objekt s přetíženým operátorem ()
 - ▶ lze volat jako funkci
 - ▶ má atributy (stav)
- od C++11 lze výhodně využívat lambda výrazy
- v STL existuje řada univerzálních funkčních objektů a adaptérů

Kontejnery

- slouží pro ukládání dat
 - standard definuje rozhraní a složitost operací (není definována konkrétní fyzická datová struktura)
-
- posloupnosti – data nemají vlastní identifikátor (klíč)
 - ▶ `arrayC++11`
 - ▶ `vector`
 - ▶ `deque`
 - ▶ `list`, `forward_listC++11`
 - ▶ adaptéry – `stack`, `queue`, `priority_queue`
 - asociativní kontejnery – data mají vlastní identifikátor (klíč)
 - ▶ `set`, `multiset`
 - ▶ `map`, `multimap`
 - ▶ hashovací kontejnery^{C++11}
(`unordered_{set,map,multiset,multimap}`)

Iterátory

- představují jednotné rozhraní pro procházení dat v libovolném kontejneru
- vycházejí z ukazatelů a logiky jejich použití (ukazatel na pole je v zásadě iterátor)
- obvykle realizovány jako objekty s přetíženými operátory
- definováno několik kategorií iterátorů dle požadovaných operací

- InputIterator – čtou data z kontejneru
 - ▶ ForwardIterator
 - ▶ BidirectionalIterator
 - ▶ RandomAccessIterator
 - ▶ ContiguousIterator^{C++17}
- OutputIterator – zapisují data do kontejneru

- zpracovávají data
 - ▶ vyhledávají
 - ▶ mažou
 - ▶ upravují
 - ▶ obecně zpracovávají prvky
 - ▶ řadí
 - ▶ ...
- pracují s obecnými daty a s obecnými zdroji dat
 - ▶ tvar dat je libovolný (int, string, složitý objekt)
 - ▶ zdroj dat je libovolný (používají se iterátory)
 - ▶ konkrétní zpracování dat je definováno pomocí operátorů nebo funkčních objektů

C++11 a STL, kontejnery, iterátory, algoritmy

auto

- **auto** umožňuje místo psaní konkrétního datového typu nechat typ odvodit kompilátor.
- stále se jedná o silné typování, typ proměnné není možné později změnit
- lze s výhodou využít v mnoha situacích v následující práci s STL
- **auto** je možné dále specifikovat modifikátory



```
Object obj;
```

- **auto** v = obj; //Object v
- **const auto** v = obj; //const Object v
- **auto&** v = obj; //Object& v
- **const auto&** v = obj; //const Object& v

initializer_list

- kontejnery je možné inicializovat výčtem hodnot pomocí inicializačních seznamů
- jde o rozšíření syntaxe uniform initialization



- `vector<int> vect {10, 15, 20, 25, 30};`
- `vector<int> vect = {10, 15, 20, 25, 30};`

Lambda výrazy – anonymní funkce

`[] () {}`

Lambda výrazy – anonymní funkce

[zachycení vnějších proměnných] (parametry funkce) { tělo funkce }

Lambda výrazy – zachycení vnějších proměnných



```
int x = 10;
```

```
// auto lambda0 = []() { return x; }; // nejde – x není definováno v ↔  
// anonymní funkci
```

```
// zachycení hodnotou
```

```
auto lambda1 = [x]() { return x; }; // ++x – nejde (je read only)  
cout << lambda1(); // x = 10; out = 10;
```

```
// zachycení referencí (odkazem)
```

```
auto lambda2 = [&x]() { return ++x; };  
cout << lambda2(); // x = 11; out = 11;
```

Lambda výrazy – parametry funkce



```
auto lambda0 = []() { return 1; };  
cout << lambda0() << endl; // 1
```

```
auto lambda1 = [](int x) { return 2 * x; };  
cout << lambda1(10) << endl; // 20
```

```
auto lambda2 = [](int x, int y, double t) { return x + y; };  
cout << lambda2(10, 20, 3.1415) << endl; // 30
```

specifikace návratového typu

`[] () -> návratový_typ { }`

mutable

`[] () mutable -> návratový_typ { }`

Umožňuje měnit vnější hodnoty zachycené hodnotou (změna se projeví pouze v těle lambdy).

zachycení vnějších hodnot

- `[=]` – vše hodnotou
- `[&]` – vše odkazem
- `[x, &]` – x hodnotou, zbytek odkazem
- `[this]` – zachycení this v objektu

Předávání / uchování anonymních funkcí



// použití auto

```
auto lambda1 = [](int value){ return value = 1; }
```

// použitím std::function<>

```
std::function<int(int)> = [](int value){ return value = 1; }
```

std::function<>

- šablona s proměnným počtem parametrů
- parametry ve tvaru result(param1, param2, ...)
 - ▶ function<int()> – funkce bez parametrů, vrací int
 - ▶ function<void(int)> – funkce s jedním int parametrem, vrací void
 - ▶ function<int(int)> – funkce s jedním int parametrem, vrací int
 - ▶ function<int(int, int)> – funkce se dvěma int parametry, vrací int

std::function<>

- umožňuje uchovávat i ukazatel na libovolnou funkci nebo metodu



```
std::function<void(Citac&, int)> fptr = &Citac::nastavAtribut↵  
    ;
```

```
Citac citac{};  
fptr(citac, 10);
```

C++ - Kontejnery

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Kontejnery

- Posloupnosti
 - vector
 - deque
 - list
- Asociativní kontejnery
 - set
 - map
- Hashovací asociativní kontejnery^{C++11}

Kontejnery

Kontejnery

- slouží k ukládání dat (koncept podobný s kolekcemi v Javě)
- různé druhy kontejnerů dle požadovaného použití
- dodržují jednotné rozhraní
- dvě základní skupiny:
 - ▶ posloupnosti
 - ▶ asociativní kontejnery

Přehled kontejnerů

- posloupnosti – data nemají vlastní identifikátor (klíč)
 - ▶ `arrayC++11`
 - ▶ `vector`
 - ▶ `deque`
 - ▶ `list`, `forward_listC++11`
 - ▶ adaptéry – `stack`, `queue`, `priority_queue`
- asociativní kontejnery – data mají vlastní identifikátor (klíč)
 - ▶ `set`, `multiset`
 - ▶ `map`, `multimap`
 - ▶ hashovací kontejnery^{C++11}
(`unordered_{set,map,multiset,multimap}`)

Základní vlastnosti kontejnerů

- realizovány jako šablony tříd, pomocí parametrů šablony lze nastavit
 - ▶ typ ukládaných dat,
 - ▶ typ alokátoru (specifikuje práce s pamětí),
 - ▶ způsob porovnání dat u asociativních kontejnerů,
 - ▶ způsob hashování u hashovacích kontejnerů



```
std::vector<Kocka>  
    kontejnerKocek{};
```

```
std::map<string, Kocka>  
    mapaKocekPodleJejichJmena{};
```

```
std::set<Kocka, RazeniKocekPodleBarvy>  
    mnozinaKocekRazenaDleBarvy{};
```

```
std::vector<Kocka, KockoAlokator>  
    zvlastneAlokovanyVektorKocek{};
```

Základní vlastnosti kontejnerů...

- kontejnery poskytují hodnotovou sémantiku
 - ▶ data (objekty/primitivní hodnoty) jsou kopírovány do vnitřního úložiště kontejneru
 - ▶ nejedná se o reference na původní umístění dat

×

```
std::vector<Kocka&> kontejnerReferenciNaKocky{};
```

✓

```
std::vector<Kocka> kontejnerKocek{};  
// od C++11 lze použít std::reference_wrapper  
// kontejner pak obsahuje objekty, které uchovávají reference  
std::vector<std::reference_wrapper<Kocka>>  
    kontejnerReferenciNaKocky{};
```

Základní vlastnosti kontejnerů...

- operace nejsou bezpečné
 - ▶ je nutné hlídat splnění požadavků na parametry jednotlivých operací
 - ▶ v době kompilace není možné otestovat zcela správnost volání
- pro realizaci obecných algoritmů kontejnery zveřejňují několik základních datových typů
 - ▶ `value_type` – typ ukládaných hodnot
 - ▶ `key_type` – typ klíče
 - ▶ `pointer (const_pointer)` – typ ukazatele na typ uložené hodnoty
 - ▶ `reference (const_reference)` – typ reference na typ uložené hodnoty
 - ▶ `iterator (const_iterator)` – typ iterátoru
 - ▶ `reverse_iterator (const_reverse_iterator)` – typ reverzního iterátoru



```
std::vector<int>::value_type intPromenna = ...;  
std::vector<Kocka>::pointer ukazatelNaKockuPromenna = ...;
```

Základní operace kontejnerů

Konstrukce kontejneru

- `Kontejner()` – vytvoření prázdného kontejneru
- `Kontejner(Kontejner k), operator=` – zkopírování obsahu stejného typu kontejneru
- `Kontejner(Iterator begin, Iterator end)` – vytvoření kontejneru z dat dle iterátorů
- `Kontejner(initializer_list il)` – vytvoření pomocí `initializer_list`



```
std::vector<int> prazdnyVektor{};  
std::vector<int> vektorInitializerList = {1, 2, 3, 4, 5};  
std::vector<int> vektorInitializerList2{1, 2, 3, 4, 5};  
std::vector<int> kopieVektoru = vektorInitializerList;
```


Základní operace kontejnerů...

Počet prvků v kontejneru

- `size()` – vrací počet položek v kontejneru
- `empty()` – vrací **true**, pokud je kontejner prázdný



```
std::vector<int> prazdnyVektor{};  
bool jePrazdny = prazdnyVektor.empty();  
  
if (prazdnyVektor.size() > 10)  
    ...
```

Základní operace kontejnerů...

Procházení kontejnerů (iterátory)

- `begin()` – vrací iterátor ukazující na první prvek v kontejneru
- `end()` – vrací iterátor ukazující za poslední prvek v kontejneru
- `cbegin()`, `cend()` – konstantní iterátory
- `rbegin()`, `rend()` – reverzní iterátory



```
std::vector<int> vektor{1, 2, 3, 4, 5};  
  
for (std::vector<int>::iterator it = vektor.begin(); it !=  $\leftarrow$   
    vektor.end(); ++it) {  
    cout << *it << endl;  
}
```

Základní operace kontejnerů...



```
std::vector<int> vektor{1, 2, 3, 4, 5};

for (auto it = vektor.begin(); it != vektor.end(); ++it) {
    cout << *it << endl;
}
```



```
std::vector<int> vektor{1, 2, 3, 4, 5};

// také lze s "auto"
for (int cislo : vektor) {
    cout << cislo << endl;
}
```

Základní operace kontejnerů...

Úpravy prvků v kontejneru

- `insert(pozice, prvek)` – vloží prvek do kontejneru
- `erase(zacatek, konec)` – odstraní vybrané prvky z kontejneru
- `clear()` – odstraní všechny prvky z kontejneru

- functions present in C++03
- functions present since C++11
- functions present since C++17

| | | Sequence containers | | | | | Associative containers | | | |
|----------------|---------------|---------------------|---------------|---------------|----------------|---------------|------------------------|--------------|--------------|--------------|
| Header | | <array> | <vector> | <deque> | <forward_list> | <list> | <set> | | <map> | |
| Container | | array | vector | deque | forward_list | list | set | multiset | map | multimap |
| Iterators | (constructor) | (implicit) | vector | deque | forward_list | list | set | multiset | map | multimap |
| | (destructor) | (implicit) | ~vector | ~deque | ~forward_list | ~list | ~set | ~multiset | ~map | ~multimap |
| | operator= | (implicit) | operator= | operator= | operator= | operator= | operator= | operator= | operator= | operator= |
| | assign | | assign | assign | assign | assign | | | | |
| | begin | begin | begin | begin | begin | begin | begin | begin | begin | begin |
| | cbegin | cbegin | cbegin | cbegin | cbegin | cbegin | cbegin | cbegin | cbegin | cbegin |
| | end | end | end | end | end | end | end | end | end | end |
| | cend | cend | cend | cend | cend | cend | cend | cend | cend | cend |
| | rbegin | rbegin | rbegin | rbegin | | rbegin | rbegin | rbegin | rbegin | rbegin |
| | crbegin | crbegin | crbegin | crbegin | | crbegin | crbegin | crbegin | crbegin | crbegin |
| Element access | rend | rend | rend | rend | | rend | rend | rend | rend | rend |
| | crend | crend | crend | crend | | crend | crend | crend | crend | crend |
| | at | at | at | at | | | | | at | |
| | operator[] | operator[] | operator[] | operator[] | | | | | operator[] | |
| Capacity | front | front | front | front | front | front | | | | |
| | back | back | back | back | | back | | | | |
| | empty | empty | empty | empty | empty | empty | empty | empty | empty | empty |
| | size | size | size | size | size | size | size | size | size | size |
| | max_size | max_size | max_size | max_size | max_size | max_size | max_size | max_size | max_size | max_size |
| | resize | | resize | resize | resize | resize | | | | |
| | capacity | | capacity | | | | | | | |
| Modifiers | reserve | | reserve | | | | | | | |
| | shrink_to_fit | | shrink_to_fit | shrink_to_fit | | | | | | |
| | clear | | clear | clear | clear | clear | clear | clear | clear | clear |
| | insert | | insert | insert | insert_after | insert | insert | insert | insert | insert |
| | emplace | | emplace | emplace | emplace_after | emplace | emplace | emplace | emplace | emplace |
| | emplace_hint | | | | | | emplace_hint | emplace_hint | emplace_hint | emplace_hint |
| | erase | | erase | erase | erase_after | erase | erase | erase | erase | erase |
| | push_front | | | push_front | push_front | push_front | | | | |
| | emplace_front | | | emplace_front | emplace_front | emplace_front | | | | |
| | pop_front | | | pop_front | pop_front | pop_front | | | | |
| | push_back | | push_back | push_back | | push_back | | | | |
| | emplace_back | | emplace_back | emplace_back | | emplace_back | | | | |
| | pop_back | | pop_back | pop_back | | pop_back | | | | |
| | swap | swap | swap | swap | swap | swap | swap | swap | swap | swap |
| | merge | | | | merge | merge | merge | merge | merge | merge |
| | extract | | | | | | extract | extract | extract | extract |

| | | | | | | | | | | | |
|---------------------|---------------|--------|---------------|---------------|---------------|------------------------|---------------|---------------|---------------|---------------|--|
| List operations | splice | | | | splice_after | splice | | | | | |
| | remove | | | | remove | remove | | | | | |
| | remove_if | | | | remove_if | remove_if | | | | | |
| | reverse | | | | reverse | reverse | | | | | |
| | unique | | | | unique | unique | | | | | |
| | sort | | | | sort | sort | | | | | |
| Lookup | count | | | | | | count | count | count | count | |
| | find | | | | | | find | find | find | find | |
| | lower_bound | | | | | | lower_bound | lower_bound | lower_bound | lower_bound | |
| | upper_bound | | | | | | upper_bound | upper_bound | upper_bound | upper_bound | |
| | equal_range | | | | | | equal_range | equal_range | equal_range | equal_range | |
| Observers | key_comp | | | | | | key_comp | key_comp | key_comp | key_comp | |
| | value_comp | | | | | | value_comp | value_comp | value_comp | value_comp | |
| | hash_function | | | | | | | | | | |
| | key_eq | | | | | | | | | | |
| Allocator | get_allocator | | get_allocator | get_allocator | get_allocator | get_allocator | get_allocator | get_allocator | get_allocator | get_allocator | |
| Container | array | vector | deque | forward_list | list | set | multiset | map | multimap | unordered_map | |
| Sequence containers | | | | | | Associative containers | | | | | |

Posloupnosti

Přehled posloupností

- `arrayC++11` – pole statické velikosti
- `vector` – dynamicky alokované pole
- `deque` – oboustranně dynamicky alokované pole
- `list` – obousměrně zřetězený spojový seznam
- `forward_listC++11` – jednosměrně zřetězený spojový seznam
- adaptéry
 - ▶ `stack` – zásobník
 - ▶ `queue` – fronta
 - ▶ `priority_queue` – prioritní fronta

vector

vector

- `#include <vector>`
- šablona `vector<TypDat, Alokator = vychozí>`
- iterátor – random access
- obvykle realizován jako dynamické pole
- poskytuje $O(1)$ složitost čtení libovolného prvku
- rychlé přidávání a odebírání prvků z konce vektoru
- pomalá úprava prvků uprostřed vektoru

vector

- při úpravách pomalé realokace pole – možnost rezervovat kapacitu vnitřního pole
- `vector.capacity()` – vrací velikost interního pole
- `vector.reserve(velikost)` – vyhradí paměť pro velikost prvků
- `vector.resize(velikost)` – změní počet prvků na velikost, způsobuje smazání nebo přidání prvků!
- `vector.shrink_to_fit()`^{C++11} – zmenší vnitřní pole pouze na potřebnou velikost pro aktuální počet prvků

Vkládání hodnot



```
std::vector<int> v{};
```

```
// vložení hodnoty na konec
```

```
v.push_back(10);
```

```
// vložení hodnot na zvolené místo (iterátorem)
```

```
v.insert(v.end(), 20);
```

```
// konstrukce a vložení hodnoty na konci (C++11)
```

```
v.emplace_back(30);
```

```
// konstrukce a vložení hodnoty na zvoleném místě (C++11)
```

```
v.emplace(v.end(), 40);
```

```
for (auto item : v) { // foreach (C++11)
```

```
    cout << item << endl;
```

```
}
```

```
// 10 20 30 40
```

Vkládání hodnot – objekt KomplexniCislo – definice



```
struct KomplexniCislo {  
    int real;  
    int imag;  
  
    KomplexniCislo() : real(0), imag(0) {}  
    KomplexniCislo(int re, int im) : real(re), imag(im) {}  
};  
  
ostream& operator<<(ostream& os, const KomplexniCislo& kc) {  
    os << kc.real << " + " << kc.imag << "i";  
    return os;  
}
```

Vkládání hodnot – objekt KomplexniCislo – použití



```
std::vector<KomplexniCislo> v{};

v.push_back(KomplexniCislo{ 1, 0 });
v.insert(v.end(), KomplexniCislo{ 10, 0 });

v.emplace_back(1, 10);
v.emplace(v.end(), 5, 50);

vypis(v);
// 1 + 0i
// 10 + 0i
// 1 + 10i
// 5 + 50i
```

Inicializace pomocí initializer_list



```
std::vector<KomplexniCislo> v{  
    {1, 0},  
    {10, 0},  
    {1, 10},  
    {5, 50}  
};
```

```
vypis(v);  
// 1 + 0i  
// 10 + 0i  
// 1 + 10i  
// 5 + 50i
```

Mazání prvků pomocí erase



```
std::vector<KomplexniCislo> v{
    {1, 0},
    {10, 0},
    {1, 10},
    {5, 50}
};
```

```
auto it = v.begin() + 1;
v.erase(it);
```

```
vypis(v);
// 1 + 0i
// - 10 + 0i - smazán
// 1 + 10i
// 5 + 50i
```


Mazání prvků pomocí erase – hromadné mazání



```
std::vector<KomplexniCislo> v{  
    {1, 0},  
    {10, 0},  
    {1, 10},  
    {5, 50}  
};
```

```
auto it1 = v.begin() + 1;  
auto it2 = v.begin() + 3;  
v.erase(it1, it2);
```

```
vypis(v);  
// 1 + 0i  
// - 10 + 0i - smazán  
// - 1 + 10i - smazán  
// 5 + 50i
```

Přístup k prvkům



```
std::vector<KomplexniCislo> v{
    { 1, 0 },
    { 10, 0 },
    { 1, 10 },
    { 5, 50 }
};

KomplexniCislo kc1 = v[0]; // 1 + 0i
KomplexniCislo kc2 = *v.begin(); // 1 + 0i
KomplexniCislo kc3 = *(v.begin() + 1); // 10 + 0i
KomplexniCislo kc4 = v.at(2); // 1 + 10i
KomplexniCislo kc5 = v.front(); // 1 + 0i
KomplexniCislo kc6 = v.back(); // 5 + 50i
KomplexniCislo kc7 = *v.rbegin(); // 5 + 50i
```

deque

deque

- `#include <deque>`
- šablona `deque<TypDat, Alokator = vychází>`
- iterátor – random access
- obvykle realizován jako pole polí
- oproti vektoru zrychluje přidávání a odebírání prvků ze začátku kontejneru

Přidávání prvků na začátek/konec



```
std::deque<KomplexniCislo> d{  
    { 1, 0 },  
    { 10, 0 },  
    { 1, 10 },  
    { 5, 50 }  
};
```

```
d.push_front({ 0, 0 });  
// nebo také: d.emplace_front(0, 0);  
d.push_back({ 100, 100 });  
// 0 + 0i  
// 1 + 0i  
// 10 + 0i  
// 1 + 10i  
// 5 + 50i  
// 100 + 100i
```

list

list

- `#include <list>`
- šablona `list<TypDat, Alokator = vychází>`
- iterátor – bidirectional
- obvykle realizován jako obousměrně zřetězený spojový seznam
- $O(1)$ složitost libovolné atomické úpravy spojového seznamu
- pomalé vyhledávání prvku
- iterátor zůstává v platnosti i při změnách v seznamu

Operace nad spojovým seznamem

- `l.remove(hodnota)` – odebere všechny prvky s danou hodnotou
- `l.remove_if(podminka)` – odebere všechny prvky splňující podmínku
- `l.unique()` – odstraňuje po sobě jdoucí duplicitní prvky (==)
- `l.unique(porovnavac)` – dtto., umožňuje specifikovat metodu porovnávání prvků
- `l.splice(pozice, l2)` – přesune všechny prvky z `l2` před pozici
- `l.splice(pozice, l2, pozice2)` – přesune vybraný prvek z `l2` do `l` na pozici `pozice`
- `l.splice(pozice, l2, zacatekl2, konecl2)` – přesune prvky z daného rozsahu před pozici

Operace nad spojovým seznamem...

- `l.sort()` – seřadí všechny prvky vzestupně ($<$)
- `l.sort(porovnavac)` – dtto., umožňuje specifikovat metodu porovnávání prvků
- `l.merge(l2)` – sloučí dva seřazené seznamy
- `l.merge(l2, porovnavac)` – dtto., umožňuje specifikovat metodu porovnávání prvků
- `l.reverse()` – obrátí pořadí prvků

Odstranění prvků dle hodnoty



```
std::list<KomplexniCislo> l{  
    { 1, 0 },  
    { 10, 0 },  
    { 1, 10 },  
    { 10, 0 },  
    { 5, 50 }  
};
```

```
l.remove({ 10, 0 });
```

```
// 1 + 0i
```

```
// 1 + 10i
```

```
// 5 + 50i
```

sort() a unique()



```
std::list<KomplexniCislo> l{
    { 10, 0 }, { 1, 10 }, { 1, 0 }, { 10, 0 }, { 5, 50 }
};

l.sort(
    [](KomplexniCislo k1, KomplexniCislo k2) {
        if (k1.real == k2.real)
            return k1.imag < k2.imag;
        return k1.real < k2.real;
    }
);

l.unique();
// 1 + 0i
// 1 + 10i
// 5 + 50i
// 10 + 0i
```

Přesun prvků pomocí splice()



```
std::list<KomplexniCislo> dest{
    { 0, 0 }, { 1, 0 }, { 1, 1 }, { 0, 1 }
};
std::list<KomplexniCislo> src{
    { 5, 0 }, { 5, 0 }, { 5, 5 }, { 0, 5 }
};
```

```
auto destinationIterator = dest.begin(); // *it = {0, 0}
advance(destinationIterator, 1); // posun o 1 – *it = {1, 0}
```

```
auto sourceIterator = src.begin(); // *it = {5, 0}
advance(sourceIterator, 2); // posun o 2 – *it = {5, 5}
```

```
dest.splice(destinationIterator, src, sourceIterator);
// dest = { 0, 0 }, { 5, 5 }, { 1, 0 }, { 1, 1 }, { 0, 1 }
// src = { 5, 0 }, { 5, 0 }, { 0, 5 }
```

Asociativní kontejnery

Přehled asociativních kontejnerů

- struktury realizované nad vyvažovanými stromy
 - ▶ `set` – množina prvků
 - ▶ `multiset` – dtto., může obsahovat duplicitní klíče
 - ▶ `map` – mapa (slovník - obsahuje dvojice klíč-hodnota)
 - ▶ `multimap` – dtto., může obsahovat duplicitní klíče
- struktury realizované pomocí hashování
 - ▶ `unordered_set`^{C++11} – množina prvků
 - ▶ `unordered_multiset`^{C++11} – dtto., může obsahovat duplicitní klíče
 - ▶ `unordered_map`^{C++11} – mapa
 - ▶ `unordered_multimap`^{C++11} – dtto., může obsahovat duplicitní klíče

Základní vlastnosti asociativních kontejnerů

- data jsou identifikována klíčem
 - ▶ musí být definována operace pro porovnávání prvků
 - ▶ stromové struktury využívají standardně operaci **operator<**
 - ▶ hashovací struktury využívají standardně operaci **operator==**
 - ★ také musí být definována hashovací funkce převádějící klíč na celé číslo
 - ★ knihovna obsahuje definici pro primitivní typy, string a některé další typy
- klíč nelze změnit po vložení do struktury
 - ▶ změnu je možné provést odebráním a přidáním prvku

set

set

- **#include** <set>
- šablona `set<TypDat, KomparatorKlice = menšítko, Alokator ↯ = vychozí>`
- iterátor – bidirectional
- obvykle realizován vyvažovaný binární strom (AVL, Red-Black tree, ...)
- neposkytuje přímý přístup k prvkům
- prvky jsou automaticky řazeny

Operace nad množinou

- `s.count(klic)` – vrací počet prvků s daným klíčem
- `s.find(klic)` – vrací iterátor na první prvek s daným klíčem (nebo `s.end()`)
- `s.lower_bound(klic)` – vrací iterátor na první pozici, na kterou byl klíč vložen
- `s.upper_bound(klic)` – vrací iterátor za poslední pozici, na kterou byl klíč vložen
- `s.equal_range(klic)` – kombinuje předchozí dvě metody, vrací oba iterátory zároveň

Vyhledání prvku v množině



```
std::set<int> s{
    2, 5, 3, 1, 6, 9, 8, 7, 0
};

if (s.find(10) != s.end()) {
    cout << "Prvek 10 nalezen" << endl;
}
```

multiset

- `#include <set>`
- šablona `multiset<TypDat, KomparatorKlice = menšítko, ↵
Alokator = vychází>`
- iterátor – bidirectional
- obvykle realizován vyvažovaný binární strom (AVL, Red-Black tree, ...)
- povoluje existenci duplicitních klíčů

Vyhledání prvků v multimnožině



```
std::multiset<int> s{
    2, 2, 1, 2, 3, 4, 5, 4, 4, 6, 7, 0, 9
};

auto iteratorPair = s.equal_range(4);
for (auto it = iteratorPair.first; it != iteratorPair.second; ←
    ++it) {
    cout << *it << endl;
}
// 4
// 4
// 4
```

map

map

- `#include <map>`
- šablona `map<TypKlice, TypDat, KomparatorKlice = menšítko, ↵
Alokator = vychozí>`
- iterátor – bidirectional
- obvykle realizován vyvažovaný binární strom (AVL, Red-Black tree, ...)
- tabulka obsahující dvojice klíč – hodnota, klíče jsou po vložení do mapy neměnné, hodnoty lze měnit
- přístup podle klíčů je rychlý
- obsahuje obdobné operace jako množina pro přístup k datům

Vkládání hodnot do mapy



```
map<string, KomplexniCislo> m{};

m.insert(make_pair("pi", KomplexniCislo{ 3.1415, 0 }));
m.insert(make_pair("zero", KomplexniCislo{ 0, 0 }));

m.emplace("one", KomplexniCislo{ 1, 0 });
m.emplace("two", KomplexniCislo{ 2, 0 })
```


Čtení hodnot z mapy



```
map<string, KomplexniCislo> m{
    { "real", { 1, 0 } },
    { "imag", { 0, 1 } }
};

// nalezení hodnoty — vrací iterátor (proto *)
std::pair<string, KomplexniCislo> p = *m.find("real");
// výpis komplexního čísla
cout << p.second;
```

- prvky je možné číst a zapisovat také pomocí přetíženého **operator[]**
 - ▶ čtení neexistujícího prvku tímto operátorem způsobí, že se prvek vytvoří v mapě!



```
std::map<std::string, KomplexniCislo> komplexniKonstanty{};

komplexniKonstanty["imag"] = KomplexniCislo{0, 1};
komplexniKonstanty["real"] = KomplexniCislo{1, 0};
komplexniKonstanty["pi"] = KomplexniCislo{3.141592, 0};

std::cout << komplexniKonstanty["pi"];
```

multimap

- `#include <map>`
- šablona `multimap<TypKlice, TypDat, KomparatorKlice = menší↔tko, Alokator = vychozí>`
- iterátor – bidirectional
- obvykle realizován vyvažovaný binární strom (AVL, Red-Black tree, ...)
- oproti struktuře `map` umožňuje vkládat duplicitní klíče
- neposkytuje přístup k prvků pomocí `operator[]`

Hashovací asociativní kontejnery^{C++11}

Přehled hashovacích asociativních kontejnerů

- `unordered_setC++11` – množina prvků
- `unordered_multisetC++11` – dtto., může obsahovat duplicitní klíče
- `unordered_mapC++11` – mapa
- `unordered_multimapC++11` – dtto., může obsahovat duplicitní klíče

Hashovací kontejnery

- data nejsou v uspořádaném stromu, ale jsou organizována do „buckets“ dle hodnoty hashe
- kontejnery poskytují operace pro prohlížení jednotlivých „buckets“ nebo pro rehashing kontejneru
- iterátor – forward
- ostatní metody jsou shodné s nehashovacími variantami kontejnerů

Parametry šablon hashovacích kontejnerů – set

- `Key` – typ dat množiny
- `Hash = std::hash<Key>` – hashovací algoritmus
- `KeyEqual = std::equal_to<Key>` – způsob porovnání klíčů
- `Allocator = std::allocator<Key>` – typ alokátoru (pro nás nezajímavé)

`std::hash` a `std::equal_to`

- realizovány jako šablony
- definovány pro běžné primitivní typy, string, automatické ukazatele a další knihovní typy

Použití hashovacích kontejnerů



```
std::unordered_set<int> us{  
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10  
};
```

```
for (auto i : us) {  
    cout << i << endl;  
}
```

```
// 9, 1, 2, 3, 4, 5, 6, 7, 8, 10
```




```
struct IntyHash {
    unsigned operator()(int value) const {
        return value % 3;
    }
};

struct IntyEqualTo {
    bool operator()(int v1, int v2) const {
        return v1 == v2;
    }
};

////////////////////////////////////

std::unordered_set<int, IntyHash, IntyEqualTo> usv{
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

for (auto i : usv) {
    cout << i << endl;
}

// 10, 1, 4, 7, 2, 5, 8, 3, 6, 9
```

C++ - Iterátory

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

- 1 Iterátory
 - Tvorba vlastního vstupního iterátoru

Iterátory

Iterátory

- představují obecné *rozhraní* pro procházení kontejnerů
- vycházejí z logiky práce s ukazateli
- umožňují číst i zapisovat hodnoty

Kategorie iterátorů

- vstupní (input) – slouží k získání dat z kontejneru
- výstupní (output) – slouží k zapsání dat do kontejneru (standardně fungují tak, že přepisují stávající data!)

Detailní popis vlastností k nalezení na:

<http://www.cplusplus.com/reference/iterator/InputIterator>

<http://en.cppreference.com/w/cpp/iterator>

Druhy iterátorů

- vstupní (input) – jednou lze přistoupit k prvku (pak je nutné posun na další prvek)
- výstupní (output) – jednou lze zapsat prvek (pak je nutné posun na další prvek)
- dopředný (forward) – jeden prvek lze číst opakovaně, lze i zapisovat
- obousměrný (bidirectional) – v kontejneru jde pohyb dopředu i dozadu
- s náhodným přístupem (random access) – v kontejneru jde pohyb i o X prvků na obě strany
- spojitý (contiguous)^{C++17} – iterátor nad spojitým paměťovým blokem

Procházení rozsahu

Na projití rozsahu potřebujeme dva iterátory - začátek a konec. Proto všude najdeme dvojice metod: `begin-end`, `cbegin-cend`, `rbegin-rend`, `crbegin-crend`.

- `begin-end` – pohyb dopředu, prvky nekonstatní
- `cbegin-cend` – pohyb dopředu, prvky konstatní
- `rbegin-rend` – pohyb odzadu, prvky nekonstatní
- `crbegin-crend` – pohyb odzadu, prvky konstatní

✓ Použití iterátoru

- `*it` – přístup k hodnotě prvku
- `it->...` – přístup k hodnotě prvku (objekty)
- `++it`, `it++` – posun na další prvky
- `it != it2`, `it != kontejner.end()` – kontrola konce

Použití vstupních iterátorů



```
vector<int> vect;  
  
for(vector<int>::iterator it = vect.begin(); it != vector.end(); it++) {  
    int data = *it;  
}
```


Použití vstupních iterátorů



```
struct osoba { string jmeno; string prijmeni; };  
  
vector<osoba> vect;  
  
for (auto it = vect.begin(); it != vect.end(); it++) {  
    osoba& os = *it;  
    string jmeno = it->jmeno;  
    string prijmeni = (*it).prijmeni;  
}
```

Použití vstupních iterátorů – foreach^{C++11}



```
struct osoba { string jmeno; string prijmeni; };
```

```
vector<osoba> vect;
```

```
for (osoba& it : vect) {
```

```
// také lze
```

```
// for (auto& it : vect) {
```

```
    osoba& os = it;
```

```
    string jmeno = it.jmeno;
```

```
    string prijmeni = it.prijmeni;
```

```
}
```

× Pozor na rušení platnosti iterátorů

```
vector<int> vect;
```

```
for (auto it = vect.begin(); it != vect.end(); it++) {  
    if (*it == 123456)  
        vect.erase(it); // smazani prvku -> zrusi platnost iteratoru -> ↵  
                           crash  
}
```

✓

```
for (auto it = vect.begin(); it != vect.end(); ) {  
    if (*it == 123456)  
        it = vect.erase(it); // ok -> erase vraci platny iterator na ↵  
                               dalsi prvek  
    else  
        it++;  
}
```

× Použití výstupních iterátorů

```
vector<int> vect;  
auto it = vect.begin();  
*it = 123;  
// crash — vektor byl prazdny
```

✓ Použití výstupních iterátorů

```
vector<int> vect;  
vect.resize(2);  
auto it = vect.begin();  
*it = 123;  
it++;  
*it = 456;  
// vect nyní obsahuje [123, 456]
```

Výstupní iterátorové adaptéry

Aby výstupní iterátory **nepřepisovaly**, ale **vkládaly** data - používají se adaptéry (hl. soubor `<iterator>`)

- `insert_iterator` - volá metodu `insert`
 - ▶ adaptér lze snadno vytvořit využitím pomocné funkce `inserter(kontejner, pozice)`
- `back_insert_iterator` - volá metodu `push_back`
 - ▶ `back_inserter(kontejner)`
- `front_insert_iterator` - volá metodu `push_front`
 - ▶ `front_inserter(kontejner)`

✓ Použití výstupních iterátorů – adaptéru back_insert_iterator

```
vector<int> vect;  
auto it = back_inserter(vect);  
*it = 123;  
it++;  
*it = 456;  
it++;  
*it = 789;  
// vect nyní obsahuje [123, 456, 789]
```

✓

```
vector<int> vect;  
auto it = back_inserter(vect);  
for(int i = 0; i < 10; i++, it++) {  
    *it = i;  
}  
// vect = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

✓ Kopírování obsahu pomocí dvou iterátorů

```
vector<int> read = {1, 10, 15, 20};  
vector<int> write;
```

```
auto wit = back_inserter(write);
```

```
for (auto rit = read.begin(); rit != read.end(); rit++, wit++){  
    *wit = *rit;  
}  
// write = [1, 10, 15, 20]
```

Proudové iterátorové adaptéry

Pro přístup k proudům (*stream) se standardně používají operátory << a >>. Existují dva adaptéry, aby se daly zpracovávat jako iterátory:

- ostream_iterator
 - ▶ ostream_iterator(ostream)
 - ▶ ostream_iterator(ostream, oddelovac)
- istream_iterator
 - ▶ istream_iterator(istream)

✓ Proudový iterátorový adaptér

```
ostream_iterator<int> osi{ cout, "--" };
```

```
*osi = 10;
```

```
osi++;
```

```
*osi = 20;
```

```
osi++;
```

```
*osi = 30;
```

```
osi++;
```

```
// na výstupu bude: 10--20--30--
```

Pomocné funkce v <iterator>

- `advance(iterator, posun)` – posune iterátor o "posun" prvků
- `distance(it1, it2)` – vrací počet prvků v rozsahu `it1 – it2`
- `next(it, posun = 1)C++11` – posune iterátor na následující prvek
- `prev(it, posun = 1)C++11` – posune iterátor na předcházející prvek
- `iter_swap(it1, it2)` – prohodí obsah iterátorů `it1` a `it2`

Tvorba vlastního vstupního iterátoru

✓ Šablona kontejneru MyArray

// šablona staticky alokovaného pole obsahující prvku typu T (počet \leftrightarrow prvků je Size)

```
template<typename T, int Size>
```

```
struct MyArray {
```

```
    // pole prvků
```

```
    T _data[Size];
```

```
    // velikost pole
```

```
    const int _size = Size;
```

```
};
```

✓ Základ iterátoru ...

```
template<typename T, int Size>
```

```
struct MyArray {
```

```
    // implementace iterátoru
```

```
    struct iterator {
```

```
};
```

```
    // begin — vrací iterátor ukazující na první prvek pole
```

```
    iterator begin() {
```

```
    }
```

```
    // end — vrací iterátor ukazující za poslední prvek pole
```

```
    iterator end() {
```

```
    }
```

```
    // ...
```

⚠ Definice veřejných typů

```
// do C++17 lze dědit z std::iterator  
struct iterator : std::iterator<std::forward_iterator_tag, T>
```

✓ Definice veřejných typů

```
typedef forward_iterator_tag iterator_category;  
  
typedef T value_type;  
typedef ptrdiff_t difference_type; // int / int64  
  
typedef T* pointer;  
typedef T& reference;
```

✓ Definice iterátoru...

```
struct iterator {  
    public:  
        // konstruktor iterátoru — předáváme ukazatel s prvkem kam bude↔  
        // iterátor ukazovat  
        iterator(T* ptr) : _ptr(ptr) { }  
  
    private:  
        // pro realizaci iterátoru stačí uchovat ukazatel na aktuální prvek  
        T* _ptr;  
};
```

✓ Metody begin() a end() v MyArray

// implementace begin() a end(), aby vracely korektní iterátor

```
iterator begin() {  
    return {_data};  
}
```

```
iterator end() {  
    return {_data + _size};  
}
```


✓ Posun iterátoru na další prvky

// posun iterátoru na další prvek

```
iterator& operator++() {  
    _ptr++;  
    return *this;  
}
```

// postinkrementální varianta

```
iterator operator++(int) {  
    iterator copy{*this};  
    _ptr++;  
    return copy;  
}
```

✓ Přístup k prvkům 1.

```
// vrácení prvku z iterátoru  
reference operator*() const {  
    // stačí dereferencovat ukazatel na aktuální prvek  
    return *_ptr;  
}
```

✓ Přístup k prvkům 2.

```
// přístup k prvkům pomocí operátoru šipky  
pointer operator->() const{  
    return _ptr;  
}
```

✓ Porovnání iterátorů

// porovnání iterátorů – aby bylo možné zjistit konec iterování

```
bool operator==(const iterator& it) const {  
    return _ptr == it._ptr;  
}
```

```
bool operator!=(const iterator& it) const {  
    return !(*this == it);  
}
```

✓ Konstruktor MyArray pro initializer_list

```
#include <initializer_list>
MyArray(std::initializer_list<T> il) {
    int i = 0;
    for (auto it = il.begin(); it != il.end(); it++) {
        _data[i] = *it;
    }
}
```

✓ Test iterátoru

```
// a je hotovo... test kontejneru a iterátoru
// vytvoření pole s prvky [1, 2, 3]
MyArray<int, 3> ary { 1, 2, 3 };

for (auto it = ary.begin(); it != ary.end(); ++it) {
    std::cout << *it << std::endl;
}
```

C++ - Algoritmy

Ing. Roman Diviš

UPCE/FEI/KST

Obsah

1 Algoritmy

- Algoritmus copy
- Algoritmus find
- Funkční objekty
 - bind
- Algoritmus remove
- Přehled algoritmů

Algoritmy

Algoritmy

- obecné algoritmy pro zpracování dat v kontejnerech (nebo i jinde – využívá se iterátorů pro přístup k datům)
- hlavičkové soubory `<algorithm>` a `<numeric>`

Algorithmus copy

Algoritmus copy

- kopíruje prvky ze vstupního rozsahu na cílové umístění
 - ▶ cílem může být jiný kontejner
 - ▶ ale i jiná část zdrojového kontejneru
 - ▶ pomocí adaptérů je možné prvky zapisovat i do proudů (soubor, konzole)
- naivní implementace je jednoduchý cyklus a operátor= pro zkopírování prvků
 - ▶ reálně složité šablony, které na základě typu iterátorů použijí nejvhodnější algoritmus
 - ▶ v případě spojitých oblastí paměti lze využít i `memcpy`
- existuje celá řada variant algoritmu
 - ▶ `copy_n`, `copy_if`, `copy_backward`



```
copy(src.begin(), src.end(), dest.begin());
```

copy - možná implementace

```
template<class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last, ↵
    OutputIterator result)
{
    while (first != last) {
        *result = *first;

        ++result;
        ++first;
    }

    return result;
}
```

copy - možná implementace

```
template<class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last, ←
    OutputIterator result)
{
    // iteruj, dokud nedojdeme na konec
    while (first != last) {
        // z iterátoru first vezmi objekt a ulož ho na místo kam ukazuje it. ←
        result
        *result = *first;

        // posuň oba iterátory na další prvek
        ++result;
        ++first;
    }

    return result;
}
```



```
vector<int> source {0, 1, 20, 4, 50, 60};  
vector<int> destination;  
  
copy(source.begin(), source.end(), destination.begin());
```



```
vector<int> source {0, 1, 20, 4, 50, 60};  
vector<int> destination;  
  
// 1) použití reserve/resize  
destination.resize(source.size());  
copy(source.begin(), source.end(), destination.begin());  
  
// 2) použití vkladacího iterátorového adaptéru  
copy(source.begin(), source.end(), back_inserter(destination));
```

Algorithmus find

Algoritmus find

- vyhledá prvek v zadaném rozsahu
- vrací iterátor na jeho pozici nebo vrací koncový iterátor
- prvek pozná podle shody (==) s jiným prvkem
 - ▶ existuje predikátová verze algoritmu `find_if`

find - možná implementace

```
template<class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, ↵
    const T& val)
{
    while (first != last) {
        if (*first == val)
            return first;

        ++first;
    }

    return last;
}
```



```
vector<int> vect {0, 4, 45, 20, 40, 50};  
  
auto it = find(vect.begin(), vect.end(), 40);  
if (it != vect.end())  
    cout << "prvek 40 nalezen";
```



```
vector<Osoba> vect {"Jan", "Stastny"}, {"Petr", "Mlady"};  
  
auto it = find(vect.begin(), vect.end(), Osoba{"Jiri", "Veliky"});  
if (it != vect.end())  
    cout << "Osoba Jiri Veliky nalezena";
```

Algoritmus find_if

- stejný princip jako `find`, ale využívá predikátovou funkci na porovnání prvků

find_if - možná implementace

```
template<class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator  $\leftarrow$ 
    last, UnaryPredicate pred)
{
    while (first != last) {
        if (pred(*first))
            return first;

        ++first;
    }

    return last;
}
```

Predikát – obyčejná funkce



```
bool jeVetsiNez40(int hodnota) {  
    return hodnota > 40;  
}
```



```
vector<int> vect {0, 4, 45, 20, 40, 50};  
  
auto it = find_if(vect.begin(), vect.end(), jeVetsiNez40);  
if (it != vect.end())  
    cout << "prvek vetsi nez 40 nalezen -> " << *it;
```

Predikát – funkční objekt



```
struct jeVetsiNez40FO {  
    bool operator()(int prvek) const {  
        return prvek > 40;  
    }  
};
```



```
vector<int> vect {0, 4, 45, 20, 40, 50};  
  
auto it = find_if(vect.begin(), vect.end(), jeVetsiNez40FO{});  
if (it != vect.end())  
    cout << "prvek vetsi nez 40 nalezen -> " << *it;
```

Predikát – standardní funkční objekty



```
vector<int> vect {0, 4, 45, 20, 40, 50};
```

```
// 1) std::binder1st, std::binder2nd – deprecated od C++11
```

```
auto it = find_if(vect.begin(), vect.end(),  
    bind2nd(greater<int>(), 40)  
);
```

```
// 2) std::bind (od C++11) – lepší, obecnější, trochu složitější
```

```
using std::placeholders::_1;  
auto it = find_if(vect.begin(), vect.end(),  
    bind(greater<int>(), _1, 40)  
);
```

```
if (it != vect.end())  
    cout << "prvek vetsi nez 40 nalezen -> " << *it;
```

Predikát – lambda výraz



```
vector<int> vect {0, 4, 45, 20, 40, 50};
```

```
auto it = find_if(vect.begin(), vect.end(),  
    [](int prvek) { return prvek > 40; }  
);
```

```
if (it != vect.end())  
    cout << "prvek vetsi nez 40 nalezen -> " << *it;
```


Funkční objekty

Funkční objekty, adaptéry, bindery

- `<functional>`
- umožňují psát obecné funkce pro použití s algoritmy jako predikáty, transformační funkce, ...
- systém adaptérů značně obměněn ve standardu C++11, změny pokračují v C++17

binder1st, binder2nd



- ▶ `std::binder1st`, `std::binder2nd` – převádějí binární funkci na unární
- ▶ jeden z parametrů je nahrazen konstantou, může být vypočtena i v době vytváření binderu
- ▶ od C++11 deprecated, od C++17 odebráno, bylo nahrazeno jednotným `std::bind`
- ▶ pro snadnější zápis se používaly funkce `bind1st()`, `bind2nd()`

```
struct Plus : std::binary_function<int, int, int> {  
    int operator()(int a, int b) const {  
        return a + b;  
    }  
};  
////////////////////////////////////  
auto plus5 = bind2nd(Plus{}, 5);  
  
cout << plus5(1) << endl; // 6  
cout << plus5(5) << endl; // 10
```

Deprecated/removed funkční objekty



- ▶ kromě `std::binder1st`, `std::binder2nd` byla odebrána řada dalších funkčních objektů
- ▶ `std::unary_function`, `std::binary_function` – definuje typy argumentů a návratové hodnoty funkce
- ▶ `std::ptr_fun` – vytváří „objektový“ ukazatel na funkci
- ▶ `std::mem_fun` – dtto na metodu (skrže ukazatel)
- ▶ `std::mem_fun_ref` – dtto na metodu (skrže referenci)
- ▶ a objekty/wrappery s nimi související

Funkční objekty – operátorové – aritmetické

- `std::plus` – $x + y$
- `std::minus` – $x - y$
- `std::multiplies` – $x * y$
- `std::divides` – x / y
- `std::modulus` – $x \% y$
- `std::negate` – $-x$

Funkční objekty – operátorové – porovnávací

- `std::equal_to` – $x == y$
- `std::not_equal_to` – $x != y$
- `std::greater` – $x > y$
- `std::less` – $x < y$
- `std::greater_equal` – $x \geq y$
- `std::less_equal` – $x \leq y$

Funkční objekty – operátorové – logické a bitové

- `std::logical_and` – $x \& \& y$
- `std::logical_or` – $x || y$
- `std::logical_not` – $!x$
- `std::bit_and` – $x \& y$
- `std::bit_or` – $x | y$
- `std::bit_xor` – $x \wedge y$
- `std::bit_not`^{C++14} – $\sim x$

Funkční objekty – příklad



```
std::multiplies<double> multipliesDouble{};
cout << multipliesDouble(10, 3.2) << endl;
// 32

std::equal_to<int> equalToInt{};
cout << (equalToInt(4, 4) ? "T" : "F") << endl;
// T

std::equal_to<string> equalToString{};
cout << (equalToString("hello", "hella") ? "T" : "F") << endl;
// F
```


Funkční objekty

Function wrappers

- `functionC++11` – slouží jako wrapper pro libovolný druh funkce (funkce, metoda, lambda, ...) dle daného předpisu
 - `mem_fnC++11` – vytváří funkční objekt ze specifikované metody

Invoke

- `invokeC++17` – vyvolání funkce/objektu se specifikovanými argumenty

Funkční objekty

Bind

- `bindC++11` – vytvoří fce objekt s možností statického nastavení některých parametrů
- `is_bind_expressionC++11` – indikuje, že daný objekt je výrazem `std::bind`
- `is_placeholderC++11` – indikuje, že daný objekt je standardní placeholder

Negators

- `not_fnC++17` – vytvoří fce objekt negující výsledek specifikované funkce
- Do C++17 (pak deprecated)
 - `not1` – vytváří fce objekt negující výsledek unární funkce
 - `not2` – vytváří fce objekt negující výsledek binární funkce

Funkční objekty

Searchers

- `default_searcher`^{C++17}
- `boyer_moore_searcher`^{C++17}
- `boyer_moore_horspool_searcher`^{C++17}

Reference wrapper

- `reference_wrapper`^{C++11}
- `ref`^{C++11} – konstruuje wrapper pro referenci
- `cref`^{C++11} – konstruuje wrapper pro konstatní referenci

bind

std::bind

- umožňuje zabalit libovolnou funkci, funkční objekt, lambda výraz, metodu do funkčního objektu
- je možné nahradit specifické parametry konkrétní hodnotou nebo referencí
- je možné změnit pořadí parametrů
- je možné u metody specifikovat konkrétní objekt nad kterým má být metoda volána
- volitelné parametry se dosazují pomocí konstant `_1`, `_2`, ... z jmenného prostoru `std::placeholders`

std::bind – základní použití



```
void printVar(string variable, int value) {  
    cout << variable << ": " << value << endl;  
}  
////////////////////////////////////  
auto pv1 = bind(printVar, "V1", 100);  
pv1();  
// V1: 100  
  
using namespace std::placeholders;  
auto pv2 = bind(printVar, "V2", _1);  
pv2(200);  
pv2(300);  
// V2: 200  
// V2: 300  
  
auto pv3 = bind(printVar, _2, _1);  
pv3(400, "V3");  
// V3: 400
```

std::bind – použití na metody



```
using namespace std::placeholders;
```

```
struct Counter {  
    int counter;  
  
    Counter(int c) : counter(c) {}  
  
    void set(int v) { counter = v; }  
    int get() const { return counter; }  
};
```

std::bind – použití na metody...



```
auto setterTo100 = bind(&Counter::set, _1, 100);  
Counter c1{ 0 };  
setterTo100(c1);  
  
Counter c{ 0 };  
auto cSetter = bind(&Counter::set, &c, _1);  
cSetter(20);  
// c.counter == 20  
  
auto cGetter = bind(&Counter::get, &c);  
cout << cGetter() << endl;  
// 20
```


std::bind – použití na reference



```
void increment(int& value) {  
    value++;  
    cout << value << endl;  
}  
////////////////////////////////////
```

```
int v = 0;  
// do objektu i1 se přenese kopie v  
auto i1 = bind(increment, v);  
i1(); // 1  
i1(); // 2  
cout << v << endl; // 0  
  
// do objektu i2 se předá reference_wrapper  
auto i2 = bind(increment, ref(v));  
i2(); // 1  
i2(); // 2  
cout << v << endl; // 2
```

Algorithmus remove

Algoritmus remove

- slouží k odebrání prvků z rozsahu
- odebírá prvky podle shody (==)
- iterátory neumí přidat nebo odebrat prvky!
 - ▶ algoritmus přeuspořádá pořadí prvků
 - ▶ vrátí iterátor na nový konec rozsahu
 - ▶ vlastní odebrání zbylých prvků je nutné provést ručně

remove – možná implementace

```
template <class ForwardIterator, class T>
ForwardIterator remove (ForwardIterator first, ↵
    ForwardIterator last, const T& val)
{
    ForwardIterator result = first;
    while (first != last) {
        if (!(*first == val)) {
            *result = *first;
            ++result;
        }
        ++first;
    }

    return result;
}
```

remove – možná implementace

```
template <class ForwardIterator, class T>
ForwardIterator remove (ForwardIterator first, ↵
    ForwardIterator last, const T& val)
{
    ForwardIterator result = first; // vytvoř kopii iterátoru first
    while (first != last) { // projdi všechny prvky v rozsahu
        if (!(*first == val)) { // pokud nejsou shodné s odebíraným
            *result = *first; // zapiš do result
            ++result; // posuň result
        }
        ++first; // posuň first
    }

    return result;
}
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[FR1, 10, 2, 10, 3, 4, 5, 10, 6, 7]
```

```
first = 1
```

```
[1, FR10, 2, 10, 3, 4, 5, 10, 6, 7]
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, FR10, 2, 10, 3, 4, 5, 10, 6, 7]
```

```
first == 10! nezapisuj
```

```
[1, R10, F2, 10, 3, 4, 5, 10, 6, 7]
```


remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, R10, F2, 10, 3, 4, 5, 10, 6, 7]
```

```
first = 2
```

```
[1, 2, R2, F10, 3, 4, 5, 10, 6, 7]
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

[1, 2, **R**2, **F**10, 3, 4, 5, 10, 6, 7]

first = 10! nezapisuj

[1, 2, **R**2, 10, **F**3, 4, 5, 10, 6, 7]

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, 2, R2, 10, F3, 4, 5, 10, 6, 7]
```

```
first = 3
```

```
[1, 2, 3, R10, 3, F4, 5, 10, 6, 7]
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, 2, 3, R10, 3, F4, 5, 10, 6, 7]
```

```
first = 4
```

```
[1, 2, 3, 4, R3, 4, F5, 10, 6, 7]
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, 2, 3, 4, R3, 4, F5, 10, 6, 7]
```

```
first = 5
```

```
[1, 2, 3, 4, 5, R4, 5, F10, 6, 7]
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

[1, 2, 3, 4, 5, **R**4, 5, **F**10, 6, 7]

first == 10! nezapisuj

[1, 2, 3, 4, 5, **R**4, 5, 10, **F**6, 7]

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, 2, 3, 4, 5, R4, 5, 10, F6, 7]
```

```
first = 6
```

```
[1, 2, 3, 4, 5, 6, R5, 10, 6, F7]
```

remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, 2, 3, 4, 5, 6, R5, 10, 6, F7]
```

```
first = 7
```

```
[1, 2, 3, 4, 5, 6, 7, R10, 6, 7] F
```


remove – postup činnosti

```
remove({1, 10, 2, 10, 3, 4, 5, 10, 6, 7}, 10);
```

```
[1, 2, 3, 4, 5, 6, 7, R10, 6, 7]
```



```
vector<int> vect {1, 10, 2, 10, 3, 4, 5, 10, 6, 7};
```

```
auto r = remove(vect.begin(), vect.end(), 10);
```

```
// vect = [1, 2, 3, 4, 5, 6, 7, 10, 6, 7]
```

```
vect.erase(r, vect.end()); // smazání prvků 10, 6, 7
```

```
// vect = [1, 2, 3, 4, 5, 6, 7]
```

Přehled algoritmů

Nemodifikující algoritmy

`all_of`^{C++11}

`any_of`^{C++11}

testuje splnění podmínky prvky

`none_of`^{C++11}

`for_each`

provede operaci nad každým prvkem

`for_each_n`^{C++17}

dtto + limit pro n prvků

`count`

spočítá počet prvků

`count_if`

`mismatch`

vyhledá pozici neshody mezi dvěma množinami

`equal`

testuje shodu dvou množin

`find`

`find_if`

najde první prvek podle hodnoty/podmínky

`find_if_not`^{C++11}

`adjacent_find`

najde dva po sobě jdoucí shodné prvky
(nebo dle podmínky)

`search`

hledá rozsah prvků

`search_n`

Algoritmy modifikující

| | |
|--|--|
| <code>copy</code> | kopíruje prvky |
| <code>copy_if^{C++11}</code> | |
| <code>copy_n</code> | dtto + specifikovaný počet prvků |
| <code>copy_backward</code> | dtto + otáčí pořadí |
| <code>move^{C++11}</code> | přesouvá prvky |
| <code>move_backward^{C++11}</code> | |
| <code>fill</code> | nastaví hodnotu prvků |
| <code>fill_n</code> | dtto + specifikovaný počet prvků |
| <code>transform</code> | transformuje prvky (aplikuje na ně funkci) |
| <code>generate</code> | nastaví hodnotu prvků dle funkce |
| <code>generate_n</code> | dtto + specifikovaný počet prvků |

Algoritmy modifikující...

| | |
|--|------------------------------|
| <code>remove</code> | odebírání prvků |
| <code>remove_if</code> | |
| <code>remove_copy</code> | kopíruje neodebrané prvky |
| <code>remove_copy_if</code> | |
| <code>replace</code> | nahrazuje hodnoty |
| <code>replace_if</code> | |
| <code>replace_copy</code> | kopíruje a nahrazuje hodnoty |
| <code>replace_copy_if</code> | |
| <code>swap_ranges</code> | prohodí prvky z dvou rozsahů |
| <code>reverse</code> | otáčí pořadí prvků |
| <code>reverse_copy</code> | dtto + výsledek kopíruje |
| <code>rotate</code> | rotuje pořadí prvků |
| <code>rotate_copy</code> | dtto + výsledek kopíruje |
| <code>random_shuffle</code> ^{C++03,11,14,-} | náhodně zamíchá prvky |
| <code>shuffle</code> ^{C++17} | |

Algoritmy modifikující...

| | |
|-------------------------------------|--------------------------|
| <code>sample^{C++17}</code> | vybere náhodný vzorek |
| <code>unique</code> | maže sousedící duplicity |
| <code>unique_copy</code> | dtto + výsledek kopíruje |

Algoritmy dělící na skupiny

| | |
|--|---|
| <code>is_partitioned^{C++11}</code> | je skupinou dle podmínky |
| <code>partition</code> | rozdělí prvky na dvě skupiny |
| <code>partition_copy^{C++11}</code> | dtto + výsledek kopíruje |
| <code>stable_partition</code> | dtto + zachovává relativní pořadí prvků |
| <code>partition_point^{C++11}</code> | vrací dělící místo v rozsahu |

Algoritmy řadící

| | |
|--|---|
| <code>is_sorted^{C++11}</code> | testuje jestli je rozsah seřazený |
| <code>is_sorted_until^{C++11}</code> | vrací největší seřazený rozsah |
| <code>sort</code> | řadí rozsah |
| <code>partial_sort</code> | řadí N prvních prvků |
| <code>partial_sort_copy</code> | dtto + výsledek kopíruje |
| <code>stable_sort</code> | řadí rozsah, zachovává relativní pořadí prvků |
| <code>nth_element</code> | částečně řadí rozsah dle spec. prvku |

Algoritmy vyhledávající na seřazeném rozsahu

lower_bound

upper_bound

binary_search

equal_range

Množinové operace (na seřazeném rozsahu)

`merge`

`inplace_merge`

`includes`

`set_difference`

`set_intersection`

`set_symmetric_difference`

`set_union`

Haldové operace

is_heap

is_heap_until

make_heap

push_heap

pop_heap

sort_heap

Operace pro minimum/maximum

`max`

`max_element`

`min`

`min_element`

`minmax`^{C++11}

`minmax_element`^{C++11}

`clamp`^{C++17}

`lexicographical_compare`

`is_permutation`^{C++11}

`next_permutation`

`prev_permutation`

Numerické operace

`iotaC++11`

`accumulate`

`inner_product`

`adjacent_difference`

`partial_sum`

`reduceC++17`

`...`