C++

1. realizace myšlenek objektového programování - pojem třídy, členu,

2. konstruktory a destruktory,

z isocpp
Co dělají konstruktory?
“Constructors build objects from dust.” -- isocpp

Mnoho programátorů zvyklích na odlišné jazyky zaskočí jak se v C++ vytváření objekty, uvažme:

```
1.  void f()
2.  {
3.   List x;     // Local object named x (of class List)
4.   // ...
5.  }
6.  void g()
7.  {
8.   List x();   // Function named x (that returns a List)
9.   // ...
10. }
```

Na řádku 3 vytváříme instanci třídy List, na řádku 8 je deklarace funkce, jejíž jméno je x a vrací List!

V C++ nejde aby jeden konstruktor pro vytvoření instance volal druhý. Uvažme:

```
1.  class Foo {
2.  public:
3.   Foo(char x);
4.   Foo(char x, int y);
5.   // ...
6.  };
7.
8.  Foo::Foo(char x)
9.  {
10.  // ...
11.  Foo(x, 0);  // Does NOT help initialize the this object!!
```

12. // ...
13. }

Kód na řádku 11 vytvoří dočasný další objekt bez jména, který je hned dealokován. Výsledek je tedy takový jako kdyby v těle konstruktoru nebyl žádný řádek (ještě horší, protože děláme zbytečnou věc, pokud teda optimalizace kompilátoru toto nevyřeší, což vyřeší).

V tomto případě by však nejlepší řešení bylo následující:

```
1. class Foo {
2. public:
3.  Foo(char x, int y = 0);  // Has the effect of combining the two constructors
4.  // ...
5. };
```

Tedy využití default iniciátoru.

V ostatních případech je dobré využít nějakou init metodu, kde bude společný kód.

Výchozí konstruktor nemusí vždy být Foo::Foo() viz

```
1. class Foo {
2. public:
3.  Foo(int i=3, int j=5);   // Default constructor: can be called with no args
4.  // ...
5. };
```

Pokud vytvořím instanci, aniž bych předal jakékoliv argumenty, **vždy se zavolá default constructor**. Pokud třída nemá bezparametrický konstruktor, dojde ke compile chybě.

Je povoleno inicializovat hodnotu členské proměnné pomocí nějaké jiné **již inicializované** členské proměnné

Je standardem definované, že použití **this** pointeru uvnitř konstruktoru je validní.

V těle konstruktoru je garantováno, že všichni předci jsou již nainicializováni, tedy je bezpečné přistupovat do jejich proměnných.

Avšak **potomci nejsou**, tedy žádné použití virtuálních metod.

Named constructor idiom
Technika umožňující bezpečné a intuitivní vytvoření objektů.

Problém je ten, že konstruktory mají vždy stejný název jako třída, tedy jediný způsob jak se orientovat mezi konstruktory je rozlišovat parametry metod. Pokud máme mnoho konstruktorů, těžko se mezi nimi vyznává.

Uvažme následující příklad, který by se ani nezkompiloval:

```
1.  class Point {
2.  public:
3.   Point(float x, float y);     // Rectangular coordinates
4.   Point(float r, float a);     // Polar coordinates (radius and angle)
5.   // ERROR: Overload is Ambiguous: Point::Point(float,float)
6.  };
7.
8.  int main()
9.  {
10.  Point p = Point(5.7, 1.2);   // Ambiguous: Which coordinate system?
11.  // ...
12. }
```

Problém je v tom, že souřadnice bodu můžeme definovat pomocí (x,y) dvojce nebo jako (r,a), radius, angle. Všechno jsou **floaty** a tedy ani typy konstruktorů se neliší. V tomto případě potřebujeme využít named constructor idiom.

```
1.  #include <cmath>              // To get std::sin() and std::cos()
2.
3.  class Point {
4.  public:
5.   static Point rectangular(float x, float y);      // Rectangular coord's
6.   static Point polar(float radius, float angle);   // Polar coordinates
7.   // These static methods are the so-called "named constructors"
8.   // ...
9.  private:
10.  Point(float x, float y);     // Rectangular coordinates
11.  float x_, y_;
12. };
13.
14. inline Point::Point(float x, float y)
15.  : x_(x), y_(y) { }
16.
17. inline Point Point::rectangular(float x, float y)
18. { return Point(x, y); }
19.
20. inline Point Point::polar(float radius, float angle)
```

```
21. { return Point(radius*std::cos(angle), radius*std::sin(angle)); }
```

Vytvoříme si statické veřejné factory funkce, které budou vyrábět objekty. Ty budou volat **privátní** konstruktor, a tedy jediná možnost jak vyrobit instanci bude pomocí těchto factory funkcí.

Nyní můžeme vytvářet instance následovně:

```
1.  int main()
2.  {
3.   Point p1 = Point::rectangular(5.7, 1.2);   // Obviously rectangular
4.   Point p2 = Point::polar(5.7, 1.2);         // Obviously polar
5.   // ...
6.  }
```

Pozn. Je dobrý zvyk dávat spíše privátní konstruktory do sekce **protected**, případní dědicové této třídy mají možnost inicializace předka, pokud bychom dali konstruktory do private sekce, dědičnost by vůbec nefungovala.

Předchozí příklad vybízí k otázce, zda-li named constructory nemají overhead. Zejména proto, že výsledný objekt vrací z factory funkce **by-value**. Odpověď je následující: K žádnému overheadu nedojde, kompilátory implementují tzv. **copy ellision** - vytvoří objekt přímo až uživatelovi do proměnné (místo vytvoření, kopírování a dealokace)

Named parameter idiom
V C++ prostě nejde nastavit hodnotu parametru podle jména, parametr je určen pozicí. Toho však jde dosáhnout např. takhle

```
1.  File f = OpenFile("foo.txt")
2.           .readonly()
3.           .createIfNotExist()
4.           .appendWhenWriting()
5.           .blockSize(1024)
6.           .unbuffered()
7.           .exclusiveAccess();
```

Každou z těchto funkcí si můžeme představit jako jeden parametr konstruktoru. Každá z volaných funkcí vrací *this a tedy &File, který je akorát upravený příslušnou akcí. Teď už nezáleží na pořadí volání funkcí, popř. můžeme nějakou vynechat.

Pro ukázku implementace jedné funkce:

```
inline OpenFile& OpenFile::readonly()
{ readonly_ = true; return *this; }
```

Potřebujeme však ještě konstruktor třídy File přijímající OpenFile.

```
1. class File {
2. public:
3.  File(const OpenFile& params);
4.  // ...
5. };
```

most vexing parse (Scott Meyers)
viz Effective STL (strana 26)

V C++ existuje klíčové slovíčko **explicit**, které můžeme nalepit před jméno konstruktoru. Takový to konstruktor nemůže být využit při implicitních konverzích typů. Často se toto hodí, protože chceme zabránit nějaké drahé nechtěné konverzi mezi typy.

TODO: delete, default, copy constructory, move constructory, rozdíl mezi assigment operátory a construktory

copy and swap idiom
https://stackoverflow.com/questions/3279543/what-is-the-copy-and-swap-idiom

Destruktory uvolňují zdroje objektu, které objekt alokoval.
Obvykle (spíš dneska už trochu historie):

- destruktor v těle obsahuje **delete**
- konstruktor v těle obsahuje **new**

Pořadí uvolňování objektu:

```
1. void userCode()
2. {
3.  Fred a;
4.  Fred b;
5.  // ...
6. } // zde konci scope, tady se budou volat dtory (pořadí??)
```

První je uvolněn **b** a pak **a** (přesně obráceně, než co byli vytvořeni)

- Jedna třída = jeden dtor
- Nikdy nepřijímá parametry a nikdy nic nevrací.
- Je **noexcept**
- dtor pro objekt zavolá dtory všech member variables

Parametry a návratovou hodnotu nemá, protože bychom ho nikdy ani neměli volat sami (až na výjimky) - dtor objektu se volá při dosažení konce scopu jeho životnosti.

- dtor zavolaný vícekrát může mít undefined behaviour

Pokud chceme dtor zavolat na nějakém určitém místě měli bychom obalit objekt ručně vytvořených novým scopem

```
{
   Lock l;
} // tady konci scope zamku
```

Vytvořme dynamicky alokovaný objekt:

1. Fred* p = new Fred();

Potom jediná správný způsob jak tento objekt dealokovat je zavolat

delete p;

delete udělá:

```
if (p) {    // or "if (p != nullptr)"

 p->~Fred(); //zahodi (znici) objekt

 operator delete(p); //vrati pamet zpatky spravci pameti

}
```

Placement new:

```
1.  #include <new>        // Must #include this to use "placement new"
2.  #include "Fred.h"     // Declaration of class Fred
3.
4.  void someCode()
5.  {
6.   char memory[sizeof(Fred)];     // Line #1
7.   void* place = memory;          // Line #2
8.
9.   Fred* f = new(place) Fred();   // Line #3 (see "DANGER" below)
10.  // The pointers f and place will be equal
11.
12.  // ..
13. f->~Fred() // zde znicime objekt f (nevratime pamet)
14. } // zde se uvolni promenna memory
```

Placement new slouží k tomu, že potřebujeme zkonstruovat objekt na naší předem dané adrese (jsme zodpovědni za správnou alokaci paměti a zarovnání, destrukci objektu a vrácení paměti správci paměti)

Řádek #3 vlastně pouze zkonstruuje objekt na adrese **place**.

Virtuální destruktor:
Pokud máme pointer na BaseClass, z které nějaké třídy dědí, a tedy může ten náš pointer ukazovat na data nějaké odvozené třídy, potřebujeme **virtuální destruktor**, který dispatchne volání destruktoru do správné odvozené třídy.

- virtualita funkcí (tedy i destruktoru) se automaticky dědí

Bez virtuálního destruktoru bychom vždycky zničili pouze BaseClass část objektu a zbytek by odletěl.

3. ochrana přístupu ke členům tříd,

C++ podporuje několik klíčových slov pro řízení přístupu ke členům objektu (class, struct, union).

1. public
    - členové třídy (funkce a proměnné) označené tímto modifikátorem jsou přístupné uživateli objektu (tečkovou nebo šipkovou notací)
2. protected
    - private + přístupnost třídám, které z této třídy dědí

3. private
    - členové jsou přístupné pouze zevnitř třídy, nikoliv navenek

Vyložená specialita C++ je klíčové slovo **friend**. Za frienda nějaké {třídy, struktury, unionu??} můžeme označit metodu nebo {třídu, strukturu, union??}, ta pak má přístup k protected a private členům.

Výchozí přístupový modifikátor pro:

- **class** je **private**
- **struct** je **public**

class B : /* private */ A {}
struct B : /* public */ A {}

Částečně nesouvisející věci:
Je možné nadefinovat const členské metody, ty dostanou **const** pointer na objekt, narozdíl od normálních non-const členských metod. Constantní členské metody pak nemohou měnit hodnoty non-const proměnných a volat non-const metody. Toto je často omezující -- často máme metodu, která je konstantní chováním, ale z hlediska efektivity by se nám hodilo, kdyby třeba mohla cachovat (zapsat něco do nějaké proměnné), proto v C++ existuje klíčové

slovo **mutable**. Pokud proměnnou nadefinujeme mutable, může být její hodnota změněna z const funkce.

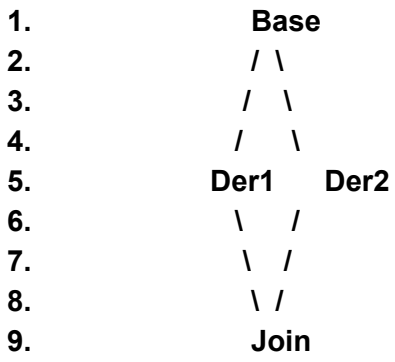
friend
V těle class nebo struct se může objevit klíčové slovíčko **friend** předcházející deklaraci funkce/třídy.
Takto označená metoda nebo třída pak má přístup ke všem členským metodám a proměnným (i těm private).

4. vícenásobná dědičnost,

Je nezbytná (vícenásobná) dědičnost?
Není, můžeme používat C (kde není žádné OOP)

Abstraktní třídy často slouží jako interface v C#.

Dreated diamond



Problém je ten, že třída Join obsahuje 2x třídu Base (přes Der1 a přes Der2).

Jak vypadá objekt v paměti, vtables
https://stackoverflow.com/questions/12378271/what-does-an-object-look-like-in-memory

Memory layout C++ programu
https://www.geeksforgeeks.org/memory-layout-of-c-program/

5. abstraktní třídy,

ABC = abstract base class

ABC oddělují **interface** od **implementace**

ABC je normální třída, která obsahuje **alespoň jednu** “pure virtual” funkci, tj.
virtual void ahoj() = 0;

Virtuální funkce, vždy pokud jsou deklarované musí být definované, jinak dostaneme compile error. V tomto případě, kdy “nastavíme” hodnotu virtuální funkce na 0, požadujeme C++ aby znemožnilo vytvoření instance dotyčné třídy, a tedy se jedná o tzv. ABC.

Pure virtual funkce **neznamená**:

- že ABC nemá implementaci metody (můžeme klidně mít pure virtual funkci s definicí)

Pure virtual funkce **znamená**:

- že potomci **musí** přetížit tuto funkci

Tedy se dostáváme k tomu, že výchozí definice pro pure virtual funkce nemá moc význam.

- nemůžeme vytvořit instanci třídy ABC (která obsahuje výchozí definici pure virtual funkce)
- potomek musí mít svoji verzi virtual funkce
- Jde však explicitně pomocí BaseClass::jmenoPureVirtualFunkce zavolat výchozí implementaci v ABC

Jak udělat ABC bez pure virtual funkce

1. řešení je vytvořit classu a do ní přidat dummy virtual funkci = 0
2. řešení je definovat konstruktor protected

ABC většinou koresponduje s nějakým abstraktním konceptem. Můžeme mít ABC geometrický tvar, která obsahuje metodu **draw**, není jasné jak se má tvar vykreslit, protože nevíme o jaký tvar jde. Odvozené třídy pak budou třeba trojúhelník, tam už to jasné bude.

Virtual constructor idiom
Máme pointer na Base třidu, z které dědí několik dalších tříd klidně. Potřebujeme deep kopírovat objekt do nového pointeru na Base. Problém je v tom, že nevíme na jaký že objekt ten náš pointer ukazuje.

Rešení:

```
1. class Shape {
2. public:
3.  // ...
4.  virtual Shape* clone() const = 0;   // The Virtual (Copy) Constructor
5.  // ...
6. };
7. class Circle : public Shape {
8. public:
9.  // ...
10. virtual Circle* clone() const;
11. // ...
12. };
13.
```

```
14. Circle* Circle::clone() const
15. {
16.  return new Circle(*this);
17. }
```

6. zpracování výjimek,

Exception handling je mechanismus, který nám umožňuje zpracovat běhovou chybu. Pokud nastane výjimka (chyba) **exception is thrown**. Tato výjimka je zachycena nejbližším **exception handlerem** (catch blok). Propagující exception je objekt, který obsahuje informace o problému. Výjimky je možné vyvolat i ručně -- **throw**.

Doporučuje se používat výjimky narozdíl od starého způsobu, všechno o if elsovat. Základním důvodem je např. to, že používání výjimek odděluje užitečný kód a kód řešící výjimky.

Exceptions v moderních C++ implementacích nepřináší prakticky žádný overhead.

Exception handling code == error handling code, proto nikdy nepoužívat výjimky jako další způsob jak vrátit nějakou hodnotu z funkce. Implementace jsou optimalizovány spíše na případy kdy výjimky nebudou vyhozeny, v těchto případech často nedochází k žádnému overheadu.

Z předchozího plyne, že **throw** by měl být použit pouze tehdy, pokud funkce nemůže vykonat, co se po ní vyžaduje. Nastal **error**.

Používejte **catch**, pokud chcete specifikovat nějaký error handling code.

Nepoužívat throw, pokud dojde k porušení nějakého invariantu v našem programu. Tyto situace by měli být vyeliminovány použitím **assert**, případně můžeme zavolat **std::terminate()** abychom ukončili program.

Pokud konstruktor nemůže vytvořit objekt (žádá o paměť a ta mu nebyla přidělena). Jediné správné řešení je ohlásit výjimku z konstruktoru. **Pozor!** v tomto případě nebyl objekt dokonstruován a nezavolá se destruktor objektu, pokud už konstruktor udělal nějaký nepořádek, **musí po sobě sám uklidit**.

Na druhou stranu, výjimky nikdy nemůžou nastat v destruktoru objektu, to by totiž znamenalo, že selhala dealokace paměti. Přesněji řečeno výjimky mohou nastat v destruktoru, avšak nesmí se dostat mimo něj -- musí se zachytit uvnitř (desktruktory jsou noexcept).

Pokud by však došlo k nějaké neočekávané situaci uvnitř destruktoru, měli bychom tuto událost zalogovat, nebo zavolat **std::terminate()**.

Pokud zachytáváme výjimku máme tři možnosti jak exception handler objekt zachytit:

1. catch by value (kopírujeme, objekt se může změnit)
2. catch by reference
3. catch by pointer

Náš výchozí způsob na zachytávání výjimek by měl být **catch by reference**.

Největší problém s catch by pointer je následující:

```
catch (MyException* p) {
        // should we delete p here or not???!?
 }
```

V C++ nebylo, není a nikdy nebude **finally** (narozdíl od Jávy). V C++ existuje idiom **RAII**, idea je taková, že v destruktoru **RAII** objektu se dealokují resources, toto znemožňuje, že by programátor na něco zapomněl.

7. šablony,

Inspirací k šablonám v C++ bylo hlavně:

- parametrizované moduly jazyka CLU
- generika jazyka Ada

Existují 3 typy šablon:

- function template
- class template
- variadic template (C++14)

Obecně jsou templaty velmi silný nástroj jazyka C++, který se často využívá spolu s přetěžováním operátorů a vícenásobnou dědičností.

Function template
Function template reprezentuje množinu funkcí. Za parametry templatů se kompilátor pokusí instanciovat libovolný volaný typ, pokud instanciovaný datový typ neimplementuje potřebný interface, přijde se na to v linkovací fázi ???

Class template
Můžeme vytvořit templatovou třídu (závislou na typech parametrů). Převážně využívané pro implementaci kolekcí.

Instanciuje se obdobně jako function template ve <> se předají template parametry.

V STL se nachází spousta class templatů (vector, list, deque, stack,...)

Od C++14 existuje novinka, která se jmenuje **variable template** (variable jako proměnná, ne že je template nějak proměnný)

template<typename **T**> constexpr **T** pi = T(3.14159265358979323846264383L);

typ proměnné je parametr templatu (to opravdu kdysi něšlo).

Variadic template
Jedná se o druh templatu, který přijímá libovolné množství parametrů, function templaty i class templaty mohou být variadic.

Template aliases (C++11)
Prakticky se jedná o parametrizovaný typedef

template<class T>
using StrMap = std::unordered_map<T, std::string>;

umožňuje používat zkratku StrMap<int> za std::unordered_map<int, std::string>

template specialization
Některé parametry templatu vrazíme za pevné natvrdo. Často se hodí - občas je potřeba řešit pro určitý speciální typ odlišné chování třídy (funkce)

např. vector<bool> je template specialized typ

Rozlišujeme 2 typy template specializace

- partial template specialization
    - class template je specializován pouze vlastní podmnožinou template parametrů
    - function template **nemůže** být partial specializován
- full template specialization
    - function nebo class template je specializován všemi parametry

Porovnání templatů a maker
Šablonovou funkci max je možné definovat pomocí maker z jazyka C.

- makra jsou stejně jako templaty “zpracovány” při kompilaci
    - makra preprocesorem
    - templaty jsou instanciovány kompilátorem
- makra jsou pouze textové nahrazení, které **není type-safe**
- vzhledem k tomu makra neprodukují nikdy žádný overhead
- templaty jsou inlinovány pouze když kompilátor uzná za vhodné, ale vždy jsou type-safe
- avšak pokud se kompilátor rozhodne šablonu neinlinovat často to má rozumný důvod - např. kód půjde procesoru lépe nacpat do cache

Nevýhody templatů

- nějaký kompilátor (MSVC) má obrovské problémy napsat rozumný error při chybě v šabloně
- pro každý typ se instanciuje “kopie” šablony → při volání obrovského množství templatů s různými typy se může stát, že výsledný exe soubor bude mnohem větší než bychom čekali
- templaty se zpracovávají při kompilaci → obrovské množství templatů (nebo template metaprogramming) může způsobit obrovskou dobu kompilace
- templaty jsou v header filech, vždy je nutné je rekompilovat
- prakticky nemožné debuggovat šablony

Ačkoliv “generika” existuje v různých jazycích, tak systém šablon je v C++ jedinečný, jedná se vlastně o vlastní **turing-complete** jazyk žijící uvnitř C++.

Template metaprogramming

Obrovskou zajímavostí je, že když Bjarne Stroustrup s týmem pracoval na templatech v C++, nikdo si vlastně neuvědomoval jak mocný nástroj vytvořili, že se templaty dají zneužít k template metaprogramming se vlastně přišlo až později. To má i své nevýhody - syntaxe templatů v C++ není zrovna příjemná.

- zneužití templatů takovým způsobem, aby kompilátor z nich vyrobil zdrojový kód, který potom zkompiluje se zbytkem programu

Nevýhody template metaprogramming

- jedná se o unitkání věc jazyka C++, která však i pro zkušené C++ programátory je obtížná k pochopení (syntaxe není příjemná)
- vše se děje při kompilaci, což je z jedné strany výhoda - věci, které je možné je možné přepsat tak, aby se vypočítali při kompilaci, na druhou stranu to prodlužuje kompilační čas

8. přetěžování operátorů a metod,

Operátory přetěžujeme zejména proto, abychom ulehčili život uživatelům naší třídy, vývojářům to právě komplikuje kód třídy.

Operátory, jež nejdou přetížit:

- .
- ::
- ?: (ternární operátor)

Pokud programujeme třídy pro manipulaci s maticema, jako indexovací operátor se nám hodí spíše **operator()** nežli **operator[]**. Operator[] vždy přijímá právě jeden argument.

Operátory mohou být přetíženy jako **member** nebo **non-member** (ne uplně všechny, jsou nějaké výjimky, např. → nemůže být non-member

Pokud operátor + přetížím jako member (tj. ve třídě Fred)
tak f1 + 10 bude f1.operator+(10). Čili operátor bude vázán na první objekt. V tomto případě by ale nefungovalo 10 + f1, protože 10 není objekt.

Fred operator+(int n, Fred const &f1) (nečlenské přetížení)
Fred& operator+(int n) (členské přetížení)

Všimněte si rozdílu, zatímco nečlenský přetížený operátor vytváří novou proměnnou, členský je vázaný na existující objekt, který upravuje.

Obvykle je žádoucí operátor + definovat nečlensky.

Narozdíl od operátoru +=, kde bychom právě čekali, že je vázán na existující objekt, který upravuje.

Můžeme definovat operátory pro konverzi mezi typy:
operator float() { return 3.14; }
explicit operator float() { return 2.718281728 };

Rozdíl mezi těmito operátory je ten, že zatímco první je implicitní, konverze podle druhého by se provedla pouze při použití static_cast.

Never overload comma operator

Method overloading
Můžeme přetížit metody, tj. mít více metod stejného názvu, avšak žádné dvě nesmí mít typově stejné parametry (které sedí i počtem)

Jak tohle funguje?
Při kompilaci se metody přejmenují na všechny s různým názvem:

- prefix - return type
- jmeno metody
- suffix - parametry

(name mangling)
A výskyty volání se přejmenují na správnou metodu.

Jak se klasifikuje **match**:

- nejlepší je **exact match**
- potom je **match**, kde se zvýší přesnost proměnných (short int → int)
- nejhorší je **match**, kde se provede implicitní konverze

9. srovnání jazyků C a C++

Většina těchto informací pochází přímo od Bjarna Stroustrupa.

C++ je přímý následník C, který zachovává skoro celé C jako podmnožinu.

- C++ má silnější typovou kontrolu než C
    - operator new vrátí pointer na typ, který je operandem new
    - malloc v C vrátí pointer na void, které je nutné (stačí implicitně) přetypovat
- C++ podporuje větší množství programovacích stylů
    - např. objektově orientované programování
- C++ “je lepší C”, že podporuje styl programování jazyka C s lepší typovou kontrolou (vše bez ztráty výkonnosti)
- C nemá namespacy pro logické členění částí kódu velkých projektů
- C nezná reference
- C nemá virtuální funkce ani klíčové slovo friend
- C nemá vestavěné konstrukci pro práci s výjimkami
- dneska má C++ značně bohatší standardní knihovnu oproti C
- C++ má scoped enums
- C++ má přetěžování operátorů
- C++ má templaty
- C++ je možné objekty zapouzdřit třeba jako RAII, aby se paměť spravovala sama (souvisí s OOP)