Seminář C++

Petr Peringer
peringer AT fit.vutbr.cz

Vysoké učení technické v Brně,
Fakulta informačních technologií,
Božetěchova 1, 612 66 Brno

(Verze 2024-02)
Přeloženo: 8. února 2024

Úvod

Tyto slajdy jsou určeny pro předmět ICP na FIT VUT v Brně. Obsahují základní popis jazyka C++ vhodný pro studenty, kteří již zvládli jazyk C. Obsah slajdů je velmi stručný, podrobnější informace jsou součástí výkladu.
Doporučuji zkoušet si krátké programy pro seznámení s různými vlastnostmi C++. Zdroje příkladů viz literatura.

Zdroje informací

- Odkazy na stránce předmětu:
  `http://www.fit.vutbr.cz/study/courses/ICP/public/`
- WWW: FAQ, přehledy knihoven, nástroje (překladače, IDE, ...)
- `http://en.cppreference.com/`
- `http://www.stroustrup.com/`
- Stroustrup B., Sutter H.: C++ Core Guidelines.
  `https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines`
- `cppman` — viz github
- ...

Literatura

- Stroustrup, B.: *The C++ programming language*, 4th edition, Addison-Wesley, 2013
- Stroustrup, B.: *Programming: Principles and Practice Using C++*, 2nd edition, Addison-Wesley, 2014
- C++20: *Draft, Standard for Programming Language C++*, `n4861.pdf` [2020-04-01]
- C++23: *Working Draft, Standard for Programming Language C++*, `n4950.pdf` [2023-05-10]
- Gamma E. et al.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994

Programovací jazyk C++

Historie

1980	vznik ”C with classes”, později C++: Bjarne Stroustrup
1998	ISO C++: mezinárodní norma (C++98)
2003	TC1: aktualizace normy C++ (C++03)
2011	ISO C++: norma C++ (C++11)
2014	ISO C++: C++14
2017	ISO C++: C++17
2020	ISO C++: aktuální norma C++ (C++20)
2023	C++23 (bude ISO/IEC 14882:2024)

Původní definicí je kniha *Stroustrup: The C++ Programming Language (Addison-Wesley 1985, 1991, 1997, 2013)*.
Platí norma *ISO/IEC 14882:2020*.

Poznámka: *Tento text předpokládá základní znalost jazyka ISO C*

Překladače C++

Existuje celá řada překladačů a vývojových prostředí pro jazyk C++. Aktuální přehled najdete na WWW stránce.

Doporučené prostředí
GNU C++ (Existuje pro všechny běžné platformy. Je třeba používat novější verzi, protože ty starší se příliš odchylují od normy.)
Je možné nastavit chování podle revize jazyka (`g++ -std=c++20`)
Makro `__cplusplus` obsahuje rok a měsíc schválení příslušné normy (např: `202002L`)

Poznámka: Integrovaná prostředí (Code::Blocks, Eclipse, KDevelop, ...) nebo editory (VIM, Emacs) + program `make`, ...

Charakteristika jazyka C++

- Obecně využitelný programovací jazyk vyšší úrovně.
- Je standardizovaný (ISO).
- Více nezávislých implementací překladačů.
- Podporuje abstraktní datové typy a vytváření knihoven.
- Nástupce jazyka C (zpětná kompatibilita).
- Vysoká efektivita.
- Objektová orientace (třídy, dědičnost).
- Generické třídy a funkce (šablony, koncepty).
- Možnost přetěžovat operátory, obsluha výjimek, moduly, korutiny, ...
- Std. knihovna (kontejnery, vlákna, regexp, random, ...).
- Množství dalších knihoven (Boost, Qt, ...).

Nevýhody C++

- Je *podstatně* větší a složitější než C
- Není zcela kompatibilní s jazykem C (complex, ...)
- Není čistě objektově orientovaný (např. typ `int` není třída a nelze dědit)
- Zdědil některé problémy jazyka C (indexování polí se nekontroluje, manuální správa paměti, ...), ale řadu z nich lze řešit (viz RAII, `shared_ptr`, kontejnery, ...).
- Ne všechny překladače dostatečně vyhovují normě.

Překlad a sestavení programu

Zdrojový soubor `ahoj.cc`:

```
#include <iostream>    // bez přípony
// import std;         // od C++23, údajně 10x rychlejší
int main() {
   std::cout << "Ahoj!\n";  // tisk C řetězce
}
```

Způsob zpracování (UNIX, překladač GNU C++):

```
g++ -o ahoj ahoj.cc      # překlad+sestavení GCC
clang++ -o ahoj ahoj.cc  # překlad+sestavení clang/LLVM
./ahoj                   # spuštění (Unix/Linux)
```

Poznámka:
Možné přípony: `.cc`, `.cpp`, `.C`, `.c++`, `.cxx` (moduly C++20)

Optimalizace a ladění

Optimalizace kódu překladačem (UNIX, GNU C++):

```
g++ -O2 -o prog  prog.cc
```

Poznámka: Nejlepší úroveň optimalizace je třeba vyzkoušet

Ladění (Linux, GNU C++, GNU debugger):

```
g++ -g -o prog prog.cc   # +ladicí informace
gdb prog                 # ladění v příkazové řádce
```

Poznámka: GUI nadstavby nad gdb

Příklad: čtení a tisk C++ řetězce

```
#include <iostream>     // std::cin, cout
#include <string>       // std::string

int main() {
  using namespace std;  // nemusíme psát std::
  cout << "C++ string I/O test\n";
  string s;
  cout << "s = " << s << "\n";
  cout << "string s (libovolná délka): " << flush;
  cin >> s ;  // sledujte jak funguje (čte slova)
  cout << "s = " << s << "\n";
}
```

NePříklad: čtení a tisk C řetězce v C++ (NEVHODNÉ)

```
// hrozí chyba typu "buffer overflow" [NEPOUŽÍVAT]

#include <iostream>
#include <string>

using namespace std;    // - nemusíme psát std::

int main() {
  cout << "C string\n";
  char s[100] = "";     // pole s inicializací
  cout << "s = " << s << "\n";
  cout << "string s (max 99 zn): " << flush;
  cin >> s ; // - čte slovo, možnost "buffer overflow"
  cout << "s = " << s << "\n";
}
```

Příklad: četnost slov

```
// g++ -std=c++11
#include <iostream>
#include <string>
#include <map>           // kontejner std::map

int main() {
  std::string word;
  std::map<std::string,int> m;  // asociativní pole

  while( std::cin >> word )     // čte po slovech
    m[word]++;

  for(auto &x: m)       // iterace přes m, seřazeno
    std::cout << x.first <<"\t"<< x.second <<"\n";
}
```

Příklad: řazení čísel

```
// g++ -std=c++11
#include <iostream>
#include <vector>       // kontejner vector
#include <algorithm>    // sort, copy
#include <iterator>     // ostream_iterator

int main() {
  using namespace std;
  vector<int> v{4, 2, 5, 1, 3};  // inicializace
  sort(begin(v), end(v));        // algoritmus řazení
  ostream_iterator<int> o(cout,"\n"); // iterátor
  copy(begin(v), end(v), o);     // tisk kopírováním
}
```

Příklad: řazení čísel — varianta 2

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>

int main() {
  using namespace std;
  vector<int> v;        // prázdný vektor
  int i;
  while ( cin >> i )    // čtení čísel ze vstupu
    v.push_back(i);     // vektor se zvětšuje
  sort(v.begin(), v.end());
  ostream_iterator<int> o(cout, "\n");
  copy(v.begin(), v.end(), o);
}
```

Příklad: lambda funkce

```
// g++ -std=c++11
#include <iostream>
#include <vector>
#include <algorithm>   // sort, for_each

int main() {
  using namespace std;
  vector<int> v{ 4, 2, 5, 1, 3 };
  sort(begin(v), end(v),
       [](int x, int y){return x>y;} ); // jiné řazení
  int a = 10;
  for_each(begin(v), end(v),    // pro celý kontejner
           [&a](int &x){ x += a++; });  // "closure"
  for(auto x: v)
    cout << x << "\n";          // tisk
}
```

Příklad: `std::span` (`g++ -std=c++20`)

```
#include <iostream>
#include <algorithm>   // sort
#include <span>
#include <ranges>

int main() {
  int v[] = { 4, 2, 5, 1, 3, 8, 6, 7, 0 };
  std::ranges::sort(std::span{v+3,4});
  for(auto x: v)
    std::cout << x << ' ';      // tisk
  std::cout << '\n';
  auto is_odd = [](int i) { return i%2 != 0; };
  for(int i: v | std::views::filter(is_odd))
    std::cout << i << ' ';
  std::cout << '\n';
}
```

Rozdíly mezi C a C++

- C++ vychází z jazyka C
- Dobře napsané C programy jsou též C++ programy (s několika výjimkami: nová klíčová slova, povinné prototypy funkcí, silnější typová kontrola, ...)
- C++ umožňuje psát programy ”bezpečněji” (”chytré” ukazatele, kontejnery, ...)

Rozdíly mezi C a C++ zjistí překladač, kromě několika výjimek:

- Znakové literály jsou typu `char`

```
sizeof('a') == sizeof(int)  // C
sizeof('a') == sizeof(char) // C++
```

- Globální `const T = x;` nejsou `extern` jako v C
- Výčtový typ obecně není ekvivalentní typu `int`

```
enum e { A };
sizeof(A) == sizeof(int) // pro starší C, nyní je jako C++
sizeof(A) == sizeof(e)   // C++: obecně != sizeof(int)
```

- Jméno struktury v C++ může překrýt jméno objektu, funkce, výčtu nebo typu v nadřazeném bloku:

```
int x[100];
size_t f()  {
  struct x { double a,b; };
  return sizeof(x); // x je pole v C, struktura v C++
}
```

Rozšíření C++ proti ISO C

- typ reference
- přetěžování funkcí a operátorů
- třídy (`class`), abstraktní třídy
  - automatická inicializace (konstruktory, destruktory)
  - zapouzdření (`private`, `public`, `protected`)
  - dědičnost, násobná dědičnost
  - polymorfismus (virtuální funkce)
  - uživatelem definované konverze
- generické datové typy – šablony (`template`, `typename`)
- prostory jmen (`namespace`, `using`)

Rozšíření C++ — pokračování

- obsluha výjimek (`try`, `catch`, `throw`)
- jméno třídy a výčtu je jméno typu
- operátory `new`, `delete`, `new[]` a `delete[]`
- ukazatele na členy tříd
- v inicializaci statických objektů je dovolen obecný výraz
- nové způsoby přetypování (`static_cast`, `const_cast`, `reinterpret_cast`, `dynamic_cast`)
- informace o typu za běhu programu (`typeid`, `type_info`)
- klíčové slovo `mutable`

C++11

- *"range for"* – cyklus přes rozsah kontejneru
- Lambda funkce a výrazy
- R-hodnotové reference, *"move"* konstruktory
- Inference typů (`auto`, `decltype`)
- Sjednocený zápis inicializace (`int a{5};`)
- Inicializační seznamy (`initializer_list<T>`)
- `nullptr`
- `constexpr`
- `extern template`
- Alternativní syntaxe funkcí `[]f(int x)->int{return x;}`
- Speciální identifikátory `override`, `final`
- Změny ve specifikaci dat: POD (*Plain Old Data*)

C++11 – pokračování

- Nové, silně typované výčty
- `template<vector<T>>` (>> byl problém C++98)
- `explicit` pro konverzní operátory
- alias šablony (`using`)
- Šablony s proměnným počtem parametrů (*"variadic templates"*)
- Nové řetězcové literály
- Uživatelem definované literály (`10kg`, `12345_bignum`)
- Podpora vláken (`thread_local`, ...)
- `=default` a `=delete` konstruktory atd.
- `long long` (převzato z C99)
- `static_assert`
- Možnost implementovat *"garbage collector"*, ...

C++14

Malé změny proti C++11 a zrušení některých omezení:

- `auto f() { .... return x; }`
- `auto` parametry lambda funkcí
- šablony proměnných (`pi<double>`)
- inicializace `{ .field=value, .... }`
- binární literály `0b0110`
- oddělovače v numerických literálech `10'000'000`
- vylepšení a rozšíření std knihovny
- ...

C++17

Řada menších změn jazyka:

- zrušení *"trigraphs"* (`??/`)
- float literály v šestnáckové soustavě
- lepší optimalizace

Vylepšení a rozšíření std knihovny:

- práce se soubory a adresáři (viz `boost::filesystem`)
- `std::string_view`
- Další matematické funkce
- `std::variant`
- `std::byte`
- ...

Poznámka: Závislost na C11

C++20

Velké změny jazyka:

- Moduly: `import`, `module` (identifikátory), `export`
- Korutiny (*coroutines*): `co_await`, `co_return`, `co_yield`
- Koncepty (*concepts*): `concept`, `requires`
- (*three-way comparison operator*): `operator <=>`
- Funkce prováděné při překladu (*immediate functions*): `consteval`
- Inicializace proměnné ”při překladu”: `constinit`
- ...

Knihovna: rozsahy (*ranges*), `std::span`, `std::format`, ...
Makra: `__has_cpp_attribute(a)`, `__cpp_concepts`, ...

C++23

- Modulární standardní knihovna: `import std;`
- Operátor indexování s více parametry: `p[i,j,k]`
- Podpora pro korutiny: `std::generator`
- Formátovaný výstup: `std::print`, `std::println`
- Dedukce `this` pro metody (`void m(this Typ& self)`)
- `std::mdspan`
- `if consteval`
- Zápis znaků: `\u{999}`, `\o{77}`, `\x{FF}`, `\N{GREEK SMALL LETTER MU}`
- ...

C++

Poznámky

```
/* text poznámky */
// text poznámky až do konce řádku
```

Vyhrazené identifikátory

- vše co obsahuje dvojité podtržení __ na libovolné pozici
- vše co začíná podtržením _ následovaným velkým písmenem
- globální symboly začínající podtržením _

Speciální identifikátory (C++11, C++20)

`override`	`final`	`import`	`module`

Klíčová slova C++

```
alignas *       constexpr *     int             static_assert *
alignof *       constinit **    long            static_cast
and             continue        mutable         struct
and_eq          co_return **    namespace       switch
asm             co_yield **     new             template
auto            decltype *      noexcept *      this
bitand          default         not             thread_local *
bitor           delete          not_eq          throw
bool            do              nullptr *       true
break           double          operator        try
case            dynamic_cast    or              typedef
catch           else            or_eq           typeid
char            enum            private         typename
char16_t *      explicit        protected       union
char32_t *      export          public          unsigned
char8_t **      extern          register        using
class           false           reinterpret_cast virtual
co_await **     float           requires **     void
compl           for             return          volatile
concept **      friend          short           wchar_t
const           goto            signed          while
const_cast      if              sizeof          xor
consteval **    inline          static          xor_eq
```

(* C++11, ** C++20)

Alternativní reprezentace — ”digraphs”

`<%`	`{`	`and`	`&&`	`and_eq`	`&=`
`%>`	`}`	`bitand`	`&`	`bitor`	`|`
`<:`	`[`	`compl`	`~`	`not`	`!`
`:>`	`]`	`not_eq`	`!=`	`or`	`||`
`%:`	`#`	`or_eq`	`|=`	`xor`	`^`
`%:%:`	`##`	`xor_eq`	`^=`		

Poznámka: Trigraphs (např. `??/`) (Pozor: zrušeno v C++17)

```
// Provede se následující příkaz??/
i++;
```

Literály

Syntaxe běžných číselných literálů je stejná jako v C.

Od C++11: uživatelem definované literály (`operator ""`)

Příklad:

```
BigNumber operator "" _big(const char * literal_string);
BigNumber some_variable = 12345_big;
```

Znakové literály (např. `'$'`) jsou typu:

- `char` v C++
- `int` v C, v C++ pouze víceznakové (mbc)

Poznámka: V C++ existují 3 různé znakové typy:
`char`, `unsigned char` a `signed char`

Typová kontrola

v C++ je silnější než v C:

- Deklarace:

  ```
  void (*funptr)();
  ```

  je ukazatel na fci vracející `void` v C, ukazatel na fci vracející `void` bez parametrů v C++
- Ukazatel na konstantní objekt nelze přiřadit do ukazatele na nekonstantní objekt.
- Typová kontrola při sestavování programu rozliší funkce s různými parametry

Výčtové typy:

- lze přiřadit pouze konstantu daného typu
- lze vynechat klíčové slovo `enum` při použití
- `sizeof` výčtového typu závisí na hodnotách prvků

Poznámky

- Rozsah deklarace proměnné cyklu ve `for`

```
for(int i=1; i<10; i++) { /* zde platí i */ }
```

- Je chybou, když je příkazem skoku přeskočena inicializace proměnné (tj. volání konstruktoru)
- Pozor na `setjmp` a `longjmp`
- C99 lokální pole s nekonstantní velikostí (VLA) nejsou v C++
- Komplexní čísla se liší v C99 a C++

Typ reference

Definice:

```
T & x = Lhodnota_typu_T;
```

- Blízké ukazatelům (ale neexistuje obdoba NULL)
- Použitelné pro předávání parametrů odkazem
- Nelze vytvořit:
  - referenci na referenci (např. `T & & r`),
    Pozor: v šablonách je dovoleno, ale jen nepřímo
  - referenci na bitová pole,
  - ukazatele na reference,
  - pole referencí.

Výhodou referencí je jednoduchost použití (na rozdíl od `*ptr`)

Typ reference – příklady

```
double x = 1.23456;
double & xref = x;  // Typické použití

double & yref;      // CHYBA! chybí inicializace
extern int & zref;  // extern může být bez inicializace

// Předání parametru odkazem:
void Transpose(Matrix & m);

// Vracení reference:
int & f(param);     // Pozor na to _co_ se vrací!

f(p) = 1;           // Volání funkce
o[5] = 0;           // operator [] vrací referenci
```

Reference — chybná nebo netypická použití

```
const int & i = 7;  // Vytvoří pomocnou proměnnou
int & i = 7;  // CHYBA! nelze pro nekonst. referenci

float f = 3.14;
const int & ir = f; // pomocná_proměnná = 3
ir = 5;             // CHYBA! konstantu nelze změnit
```

Poznámka:

Proč nelze použít R-hodnotu s nekonstantní referencí:

```
void incr( int& refint )  { refint++; }
void g() {          // POZOR - toto není C++
    double d = 1;
    incr(d);        // záludná chyba: nezmění d !
}
```

R-hodnotové reference (*R-value references*, C++11)

Definice:

```
T && x = výraz;
```

- `výraz` nemusí být L-hodnota
  (Definice pojmu L-hodnota: `&(výraz)` funguje.)
- Použitelné pro optimalizaci (omezení kopírování)
  viz například ”move” konstruktory
- *Perfect forwarding* při správném použití (dedukce typu)

```
template<typename T>
void f( T && x ); // R-value OR L-value reference

auto && x = y;    // univerzální reference (podle y)
```

Pojmy: *value; glvalue, rvalue; lvalue, xvalue, prvalue*

Typ `bool`

Booleovské literály: `false` a `true`
Implicitní konverze `bool ---> int`

```
true  ---> 1
false ---> 0
```

Konverze čísel, výčtů a ukazatelů na `bool`

```
0     ---> false
jinak ---> true
```

Výsledek relační operace je typu `bool`

Příklad:

```
bool test = false;
test = (a > b); // bool
test = 5;       // int(5) ---> bool(true)
```

Přehled operátorů C++ (podle priority)

operátory	asociativita
`::`	←
`() [] -> . T() T{} x++ x--`	→
`! ~ + - ++x --x & * (T) sizeof co_await new delete`	←
`.* ->*`	→
`* / %`	→
`+ -`	→
`<< >>`	→
`<=>`	→
`< <= > >=`	→
`== !=`	→
`&`	→
`^`	→
`|`	→
`&&`	→
`||`	→
`?: throw co_yield = OP=`	←
`,`	→

Operátory C++

operátor	popis
`::`	kvalifikátor
`.*`	dereference ukazatele na člen třídy přes objekt (`o.*mptr`)
`->*`	dereference ukazatele na člen třídy přes ukazatel na objekt (`ptr->*mptr`)
`new`	dynamické vytvoření objektu
`delete`	zrušení objektu
`static_cast` `reinterpret_cast` `const_cast` `dynamic_cast`	operátory přetypování C++
`alignof`	zarovnání; od C++11
`noexcept`	specifikace výjimek; od C++11
`co_await co_yield`	operace korutin; od C++20
`<=>`	univerzální porovnání; od C++20

Operátory — příklady

```
/// Alokace paměti operátorem new  (chyba: bad_alloc)
T *p = new T[10*n];     // dynamická alokace pole
T *p2 = new T(5);       // dynamická alokace objektu

/// Uvolnění paměti operátorem delete:
delete [] p;            // uvolnění paměti pole
delete p2;              // uvolnění paměti objektu

/// Alokace a rušení pole bajtů (POD):
char *s = new char[100];
delete [] s;
// char *s2 = static_cast<char*>(std::malloc(100));
// if (s2==nullptr) throw std::bad_alloc();
// std::free(s2);
```

Operátor : :

- Přístup ke globální proměnné:

```
double x;
void f() {
   int x;               // lokální x
   ::x = 3.1415926;     // globální x
}
```

- Explicitní specifikace třídy:

```
class T {
  public:
    int metoda();       // deklarace metody
};
int T::metoda() { }     // definice mimo třídu
```

- Specifikace prostoru jmen:

```
prostor::identifikátor
prostor::podprostor::identifikátor
```

Standardní konverze I

Implicitní konverze probíhají automaticky (jsou-li nutné) při vyhodnocování binárních operací:

1. Každý 'malý' celočíselný typ se konvertuje takto:

typ	konverze na	metoda
`char`	`int`	podle nastavení
`unsigned char`	`int`	doplní nuly
`signed char`	`int`	rozšíří znaménko
`short`	`int`	stejná hodnota
`unsigned short`	`unsigned int`	stejná hodnota
`enum`	`int+`	stejná hodnota
`bool`	`int`	hodnota 0 nebo 1

Potom je každá hodnota operandu buď `int` (včetně `long` a `unsigned` modifikátorů) `double`, `float` nebo `long double`.

Standardní konverze II

2. Je-li některý operand `long double`, je druhý konvertován na `long double`
3. Jinak, je-li operand `double`, konvertuje druhý na `double`
4. Jinak, je-li operand `float`, konvertuje druhý na `float`
5. ...
6. Jinak, je-li operand `unsigned long`, konvertuje druhý na `unsigned long`
7. Jinak, je-li operand `long`, konvertuje druhý na `long`
8. Jinak, je-li operand `unsigned`, konvertuje druhý na `unsigned`
9. Jinak, jsou oba operandy typu `int`

Výsledek odpovídá typu obou operandů po konverzi.

Konverze — příklad

Poznámka:
Porovnávání čísla `int` s číslem `unsigned` může vést k (pro někoho neočekávaným) výsledkům:

```
int      i = -1;
unsigned u = 123U;

/// následuje implicitní konverze i na unsigned
if(i<u)                // sledujte varování překladače
   printf(" i < u "); // Pozor - nevytiskne nic!
```

Explicitní konverze

Explicitní konverze uvádí programátor do textu programu (a přebírá za ně veškerou odpovědnost):

```
(typ) výraz             // C kompatibilita, nepoužívat
typ(výraz)
typ{výraz}              // od C++11
static_cast<typ>(výraz)
```

Příklady: Explicitní přetypování

```
double(int1)/int2               int{}
complex(3.14)                   int('c')
static_cast<char*>(ptr)
reinterpret_cast<intptr_t>(ptr)
```

extern "C"

Dovoluje použití funkcí z knihoven jazyka C, případně z jiných jazyků:

```
extern "C" int f(int);

extern "C" {
   int iii;
   int g(int);
   int h(int);
}
```

Poznámky: prostory jmen, *"name mangling"*, ABI

Preprocesor

Je stejný jako v ISO C, je vhodné minimalizovat jeho používání, protože máme lepší prostředky:

- `#define K1 10` lze nahradit za:

```
const int K1 = 10;
```

- `#define f(x) (výraz_x)` lze většinou nahradit za:

```
inline int f(int x) { return výraz_x; }
```

případně lze použít generické funkce:

```
template<typename T>
inline T f(T x) { return výraz_x; }
```

Základní principy OO přístupu

- Tradiční přístup

  `program = data + algoritmy(podprogramy)`
- Modulární přístup

  `program = moduly`
  `modul = data + algoritmy(podprogramy)`
- Objektově orientovaný přístup

  `program = objekty + komunikace`
  - každý objekt patří do nějaké třídy (klasifikace)
  - hierarchie objektů (skládání)
  - hierarchie tříd (dědičnost)

OO vývoj programů

analýza – návrh – implementace – testování – údržba

- Objektově orientovaná analýza (OOA)
  - zkoumání požadavků z hlediska tříd a objektů
- Objektově orientovaný návrh (OOD)
  - dekompozice
  - popis systému a jeho charakteristik
  - notace (grafy)
- Objektově orientované programování (OOP)
  - program je skupina spolupracujících objektů
  - každý objekt reprezentuje instanci nějaké třídy
  - třídy jsou navzájem v relaci dědičnosti
  - volba implementačního jazyka

Objektový model, notace

- Hlavní principy
  - Abstrakce
  - Zapouzdření
  - Modularita (fyz.)
  - Hierarchie
- Vedlejší principy
  - typování
  - paralelismus
  - persistence

Poznámky: nic nového; dědičnost=hierarchie abstrakcí

Notace:

- diagram tříd
- diagram objektů
- diagram komunikace objektů
- ...

Viz UML (ISO/IEC 19501) `http://www.omg.org/spec/UML/`

Třídy

třída = data + funkce + zapouzdření

- Rozšíření systému typů — reprezentuje množinu objektů
- Definuje rozhraní a chování objektů
- Klíčová slova: `class`, `struct` a `union`
- C++ povoluje nekompletní deklarace tříd:

```
class X;          // => omezení při použití
```

Příklad: Třída `T`

```
class T {
    int i;     // data (i jiné objekty)
  public:      // specifikace přístupových práv
    void m(); // metody
    typedef int typ; // vnořené typy, atd.
};
```

Třídy a objekty — příklad

Příklad: Použití třídy `T`

```
T  x;             // objekt třídy T
T &refx = x;      // reference na objekt T
T *ptrx = &x;     // ukazatel na objekt T
T  xarr[20];      // pole objektů třídy T
T  f();           // funkce vracející T
T &&rref = f();   // C++11: R-value reference
```

Poznámky:

- Přístup ke složkám třídy je stejný jako u struktur
- Platí ODR *"One definition rule"*

Datové členy a vnořené objekty

Datové složky třídy

- Běžné datové složky (jsou v každém objektu)
- Statické datové složky
  - pouze jedna instance pro jednu třídu
    (tj. všechny objekty třídy sdílí tuto instanci)
  - jsou přístupné i když neexistuje žádný objekt
  - musí se definovat a inicializovat vně definice třídy
    (kromě konstant)

Použití statických datových složek:

- Společný bank pro všechny objekty
  (např. počet vytvořených objektů dané třídy)
- Zapouzdření (ochrana proti neoprávněnému přístupu)

Datové členy a vnořené objekty — příklad

Příklad různých datových složek třídy

```
class X {
  static const int N = 100; // konstanta
  static int count; // jen jedna instance pro třídu

  int i;            // je v každém objektu
  string s;         // v každém objektu
  double data[3];   // v každém objektu

  int j = 5;        // od C++11: inicializace
  enum TE : int { A=10, B, C, D } e; // C++11
};

int X::count = 0; // Pozor! je nutná definice v modulu
```

Kontrola přístupu ke členům tříd

Specifikace přístupu

`public`	může být použit libovolnou funkcí
`private`	pouze pro metody a friend funkce dané třídy
`protected`	jako `private`, ale navíc je dostupné v metodách a friend funkcích tříd odvozených z této třídy

Implicitní nastavení:

pro definici	implicitně platí	lze předefinovat?
`class`	`private`	ano
`struct`	`public`	ano
`union`	`public`	ne

Kontrola přístupu ke členům tříd — příklad

Přístupové specifikace mohou být umístěny libovolně:

```
class T {
    int i; // class => implicitně private
  public:
    int j; // public
  protected:
    int k; // protected
  public:
    int l; // public
    int m;
  private:
    int n;
};
```

Friend funkce a třídy

- klíčové slovo `friend`
- mají plná práva přístupu ke všem členům třídy
- vlastnost friend se nedědí
- vlastnost friend není tranzitivní

Poznámky:

- používat jen výjimečně
- narušuje ochranu dat
- použitelné pro vytváření množin příbuzných tříd (například kontejner + iterátor)

Friend funkce a třídy — příklad

```
class Y;              // nekompletní deklarace třídy Y
class X {
  int data;           // PRIVÁTNÍ data
  friend Y;           // friend třída Y
  friend int f(X*);// deklarace friend funkce
};
class Y {             // definice třídy Y
  void m(X &o) {
    o.data = 0;       // přístup do X je možný
  }
};

int f(X *ptr) {       // definice friend funkce
  ptr->data = 1;      // přístup povolen
}
```

Metody

Kategorie metod

- Konstruktory (vznik objektu)
- Destruktor (zánik objektu)
- Statické metody
- Běžné metody (ne statické)
- Virtuální metody (pro polymorfismus)
- Operátory (různé operace: `+ - * / new`)
- Konverze (přetypování objektu na jiný typ)

Poznámka: + šablony metod

Inline metody

- optimalizace (ale je možné i volání)
- vhodné pro krátké funkce (jinak *"code bloat"*)
- musí být definována stejně pro každý modul
- metoda definovaná uvnitř třídy je automaticky inline:

```
class X {
  char *i;
public:
  char *f() const { // implicitně inline
     return i;
  }
};
```

- s explicitním uvedením klíčového slova `inline`:

```
inline char *X::f() const { return i; }
```

Jednoduchý příklad: třída interval

```
// třída čísel od jedné do deseti
class Int_1_10 {
  int n;
 public:
  Int_1_10(): n{1} { }  // implicitní konstruktor
  Int_1_10(int n)  { SetValue(n); }
  void SetValue(int x); // deklarace metody
  int GetValue() const { return n; }
  void Print() const;
};

void Int_1_10::SetValue(int x) { // definice metody
  if(x<1 || x>10) error("Range error");
  n = x;
}
```

Klíčové slovo `this`

- Implicitní parametr nestatických metod
- Ukazatel na objekt se kterým metoda pracuje
- Pro metodu třídy `T` je `this` typu:
  `T *const` pro nekonstantní objekty
  `const T *const` pro konstantní objekty
- V šablonách se ve speciálních případech musí používat `this->člen`
- `this` lze použít pouze uvnitř nestatické metody (například pro předání odkazu na objekt do jiné funkce).

Příklad použití `this`

```
class T {
 public:
  void f()        { std::cout << "T\n"; }
  void f() const  { std::cout << "const T\n"; }
  T clone() const { return *this; } // kopie
};

T       o1;     // nekonstantní objekt
const T o2{};   // konstantní objekt

int main() {
  o1.f();       //       T *const this = &o1
  o2.f();       // const T *const this = &o2
  T o = o1.clone();
}
```

Statické metody

- nemají `this`
- nesmí být virtuální
- chovají se jako obyčejné funkce, ale mají přístup k private složkám třídy
- použití mimo metody dané třídy se musí kvalifikovat `T::staticka_metoda(parametry)`
- lze je volat i když neexistuje žádný objekt
- vhodné např. po pomocné funkce pro třídu

Statické metody — příklad

```
class X {
  static int count;       // toto není definice!
 public:
  static int getCount(); // vrací počet objektů
  static void func(int i, X* ptr); // nevhodné?
  void g();
};

int X::count = 0;         // definice a inicializace

void g() {
  int i = X::getCount(); // nepotřebuje objekt
  X obj;                 // definice objektu třídy
  X::func(1, &obj);      // objekt předán explicitně
  obj.g();               // obyčejná metoda
}
```

Vnořené typy

- typy deklarované uvnitř třídy
  - vnořená třída (*nested class*)
  - `using` (nebo `typedef`)
- souvislost s prostory jmen
- lze použít neúplné deklarace vnořené třídy
- použití pro ukrytí implementačně závislých tříd (například iterátory v kontejnerech)

Vnořené typy — příklad

```
struct A {
  using muj_typ = int; // vnořený typ
  struct B {           // definice vnořené třídy
    void metodaB(int); // deklarace metody
  };
  class C;             // deklarace vnořené třídy C
};
class A::C { };        // definice C mimo třídu A
void A::B::metodaB(int i) { } // definice metody

int main() {
  A::muj_typ i = 5;    // public
  A::B       o;
  o.metodaB(i);
}
```

Operátory `new` a `delete`

Umožňují dynamické vytváření a rušení

- jednotlivých objektů (`new`, `delete`)
- polí objektů (`new[]`, `delete[]`)

Poznámka: Lze je předefinovat, případně přetížit (vhodné například pro použití s jinou správou volné paměti).

Alokace objektů

`new T`

1. alokuje paměť pro objekt
   - použije `T::operator new()`, pokud existuje
   - jinak použije `::operator new()`

   `operator new` při chybě vyvolá výjimku `bad_alloc`
2. po úspěšné alokaci je proveden konstruktor

Příklad: (C++11)

```
auto ptr = new T{1,2,3};
```

Alokace polí

`new T[velikost]`

1. alokuje paměť pro pole
   - použije `T::operator new[]()`, pokud je definován
   - není-li použije `::operator new[]()`

   při chybě vyvolá výjimku `bad_alloc`
2. provede konstruktor pro každý prvek pole v rostoucím pořadí indexů

Poznámka: Používejte raději kontejnery, např. `std::vector`

Příklady:

```
T *pt = new T[10]; // pole 10 objektů
auto p = new int[5]{1,2,3,};
```

Operátor `delete`

Uvolnění paměti

```
delete ptr
delete[] ptr
```

1. volá destruktory v obráceném pořadí než při `new`
2. uvolní paměť voláním `T::operator delete()` nebo `::operator delete()`
   (případně varianta s `delete[]` pro pole)

Příklady:

```
delete ukazatel; // ruší objekt alokovaný  new T
delete [] ptr;   // ruší pole alokované  new T[n]
```

Speciální alokace paměti

Operátor `new` (někdy i `delete`) může mít více parametrů. Toho se využívá například pro alokaci na konkrétní adresu (*placement new*) a při alokaci extra zarovnaných objektů (viz `std::align_val_t`).

`new` musí mít první parametr typu `size_t`
`delete` musí mít první parametr typu `void*`

Příklady volání operátoru `new`

```
new T          volá operator new(sizeof(T))
new(adr) T          operator new(sizeof(T),adr)
new T[5]            operator new[](sizeof(T)*5 + x)
new(22) T[5]        operator new[](sizeof(T)*5 + y, 22)
```

(`x` a `y` jsou implementací definované hodnoty)

Speciální alokace paměti – `nothrow`

Identifikátor `nothrow` pro speciální (`noexcept`) variantu operátoru `new(nothrow)` vracející při nedostatku paměti hodnotu `nullptr` místo vyvolání výjimky `bad_alloc`.

Použitelné pro systémy, kde je režie výjimek nežádoucí.

Příklad: Použití `nothrow`

```
T *p = new(nothrow) T;
if( p==nullptr )
    errorexit("Málo paměti");
// ...
delete p;
```

Definice `new` a `delete` pro třídu

Pro třídu `T` lze definovat funkce:

```
void * T::operator new(size_t);     // jeden objekt
void * T::operator new[](size_t);   // pole objektů

void T::operator delete(void* ptr);
void T::operator delete[](void* ptr);
```

Nejsou-li výše uvedené operace definovány, použijí se standardní (globální) operátory.

Speciální alokace objektů třídy — příklad

```
class T {
 public:
  void * operator new (size_t s);
  void operator delete (void* ptr);
};
void * T::operator new (size_t s) {
  char* ptr = new char[s];     // volá ::new
  //=== zde můžeme provést potřebné operace ===
  return static_cast<void*>(ptr);
}
void T::operator delete (void* ptr) {
  //=== zde můžeme provést potřebné operace,
  // např. vynulování z bezpečnostních důvodů ===
  delete[] static_cast<char*>(ptr); // ::delete
}
```

Konstruktory

Speciální metody pro inicializaci objektů při vzniku

- lze definovat více konstruktorů (viz. přetěžování funkcí)
- jsou volány automaticky při vytváření objektů
- nesmí být `static` ani `virtual`
- lze je volat i pro `const` a `volatile` objekty
- `=delete` (nebo `private:`) – zákaz vytváření objektů
- nelze získat adresu konstruktoru (`&ctr`)
- C++11: *delegating constructor* – volá jiný konstruktor

Poznámka: *Implicitní konstruktor* je každý konstruktor, který lze zavolat bez parametrů (a může být pouze jeden pro jednu třídu).

Konstruktory – příklad

```
class X {
   int i;
 public:
   X(): i{} {}         // implicitní konstruktor
   X(int x): i{x} {}; // konstruktor s parametrem
   X(T x): X{convtoint(x)} {}; // delegace
   X(const X&) =default; // kopírovací konstruktor
   X(X&&) =default;      // "move" konstruktor
};

X o;          // volá se X::X()
X o2{55};             // X::X(int)
X o3 = 55;            // X::X(int) a X::X(X&&)
X o4{o2};             // X::X(const X&)
X *px = new X;        // X::X()
X *px2 = new X{666}; // X::X(int)
```

Implicitní vytváření metod překladačem 1

Pokud není definován explicitně, překladač může vytvořit:

- implicitní konstruktor `X::X()`
- kopírovací konstruktor `X::X(const X&)`
- ”move” konstruktor `X::X(X&&)`, C++11
- operátor přiřazení `X &X::operator = (const X&)`
- ”move” operátor přiřazení `X &X::operator = (X&&)`
- destruktor `X::~X()`

Omezení: Např. pokud je definován jiný konstruktor, pak nevytvoří `X::X()`

- Překladačem vytvořené metody jsou vždy `public: inline`
- Pokud některý člen třídy nelze implicitně odpovídajícím způsobem zkonstruovat, je program chybně vytvořen.
- Někdy (typicky u tříd obsahujících ukazatel) implicitní verze operací nevyhovují a musíme je definovat.

Implicitní vytváření metod překladačem 2

C++11:

- Specifikace `= default` vytoří implicitní verzi.
- Specifikace `= delete` zakáže implicitní vytvoření.

Příklad:

```
struct X {
    X() = default;  // Bez tohoto se nevytvoří,
    X(int);         // protože je tu jiný konstruktor.

    // Zákaz kopírovacího konstruktoru a přiřazení:
    X(const X&) = delete;
    X& operator=(const X&) = delete;
    ....
};
```

Kopírovací konstruktor (*Copy constructor*)

- má jeden parametr typu `const X&` (nebo nevhodně `X&`),
- je volán při vzniku objektu kopírováním:
  - definice s inicializací jiným objektem,
  - předávání parametru hodnotou,
  - vracení hodnoty z funkce,
  - vyvolání a zpracování výjimek.
- Pokud je vytvořen překladačem, kopíruje objekty po složkách (jejich kopírovacím konstruktorem), např.
  `X::X(const X& o): s1{o.s1}, s2{o.s2} {}`

Poznámky:

- Nezaměňovat s operátorem přiřazení!
- Možnost optimalizace (*copy elision*), viz změny v C++17

Kopírovací konstruktor – příklad

Ukázka jaké konstruktory se použijí v různých situacích:

```
X f(X p) {     // == předání parametru hodnotou
  X a;         // implicitní konstruktor
  X b{a};      // kopírovací konstruktor
  X b2(a);     // kopírovací konstruktor -
  X c = a;     // kopírovací konstruktor -
  return a;    // * kopírovací konstruktor
  f(a);        // kopírovací konstruktor
  throw a;     // * kopírovací konstruktor
}
```

Poznámky:

∗ – možná optimalizace (RVO, *copy elision*)

\- – raději nepoužívat

”Stěhovací” konstruktor (*Move constructor*, C++11)

- má jeden parametr typu `X&&` (`const` tady nedává smysl)
- je volán při vzniku objektu z R-hodnoty:
  - deklarace s inicializací a použitím `std::move()`
  - vracení hodnoty z funkce s použitím `std::move()`
- Pokud je vytvořen překladačem (např. `=default`), použije ”move” konstruktory složek

```
X::X(X && o): s1{std::move(o.s1)} {}
```

Poznámky:

- Nezaměňovat s ”move” operátorem přiřazení!
- Umožňuje explicitně omezit zbytečné kopírování – viz např. všechny standardní kontejnery.

Inicializace vnořených objektů

- Konstruktory vnořených objektů se volají automaticky před provedením těla konstruktoru třídy.
- Pořadí volání je dáno jejich pořadím v definici třídy.
- Případné parametry lze předat explicitně — viz příklad:

```
class X {
   complex c1;
   complex c2;
  public:
   X() {} // implicitní konstruktory c1 a c2
   X(double a) : c2{a}, c1{5} {} // pořadí c1 c2
};
```

(Pozor na pořadí volání konstruktorů c1 a c2!)

- Automaticky vytvořený (nekopírovací) konstruktor inicializuje vnořené objekty implicitními konstruktory.

Inicializace pole objektů

Konstruktory pro elementy pole jsou volány v rostoucím pořadí indexů

Příklad:

```
X a[10];                          // X::X()
X *ptr = new X[10];               // X::X()
X p[10] { X{1}, X{2}, };          // X::X(const X&) ?
X *p2 = new X[10]{X{1},X{2},};    // X::X(const X&)
```

Poznámka:
Jsou možné optimalizace kopírovacích konstruktorů (*copy elision*)

Inicializace dočasných objektů

Překladač může vytvářet dočasné objekty:

- Implicitně (ve výrazech, inicializaci referencí, ...)
- Explicitně zápisem:

```
X(22);  // použije se X::X(int)
^^^^ funguje jako konverze  int ---> X
```

Poznámky:
Dočasné objekty překladač automaticky ruší — viz destruktory.

Na dočasné objekty vytvořené *implicitní* konverzí se vztahuje omezení — nelze volat metody (příklad viz přetěžování operátorů, `5+obj`).

Destruktor

- jméno je složeno ze znaku ~ (tilda) a jména třídy
- je automaticky vyvolán když končí rozsah platnosti objektu (konec bloku, rušení nadřazeného objektu, konec vlákna, konec programu, ...)
- nesmí mít parametry a nesmí vracet hodnotu
  (je pouze jeden pro jednu třídu)
- může být virtuální (někdy *musí* být virtuální)
- může být implicitně generován překladačem

```
X::~X() {} // destruktor pro třídu X
```

- destruktory vnořených objektů jsou volány v přesně obráceném pořadí než jim odpovídající konstruktory
- u polí se volají v klesajícím pořadí indexů

Destruktor – pokračování

Poznámky:

- Končí-li rozsah platnosti *ukazatele* na objekt, cílový objekt se neruší (zrušení provede operátor `delete`).
- Destruktor lze *ve speciálních případech* volat explicitně:

```
X *p = new X;
p->~X(); // destrukce bez uvolnění paměti
// potom už nelze použít  delete p;
::operator delete(static_cast<void*>(p));
```

- Destruktory polymorfních tříd mají být virtuální.
- Destruktor je `noexcept` a nesmí vyhodit výjimku. Všechny uvnitř vzniklé výjimky musí obsloužit, jinak provede `terminate()` (při *stack-unwind*).

Rušení dočasných objektů

Dočasné (*temporary*) objekty jsou automaticky rušeny vždy v obráceném pořadí jejich vzniku:

- na konci výrazu ve kterém vznikly,
- jde-li o inicializaci, potom až po dokončení inicializace,
- odkazuje-li se na ně reference, pak
  - při zániku reference (Pozor na speciální případy!),
  - na konci konstruktoru v jehož inicializační sekci vznikly,
  - na konci funkce v případě jejich vzniku v příkazu `return`

Poznámka: Pozor — vrácení reference na lokální proměnnou je vždy chyba. (Podrobnosti viz norma: ”class.temporary”)

Konverze

Používají se pro změnu typu výrazu na jiný typ (`T1` $\rightarrow$ `T2`)

Kategorie konverzních operací

- Standardní konverze (součást jazyka C++)
- Uživatelské konverze (definované v programu):

- Jsou aplikovány jen když jsou jednoznačné a přístupné.
- Ke kontrole jednoznačnosti dochází ještě před kontrolou přístupových práv.
- Implicitně může být aplikována pouze jedna *uživatelská* konverze na jednu hodnotu (tj. neprovede se konverze `A` $\rightarrow$ `B` $\rightarrow$ `C`, když neexistuje přímo `A` $\rightarrow$ `C`)

Poznámka: `0-1 std` `0-1 user` `0-1 std`

Konverze — pokračování

Uživatelské konverze lze specifikovat dvěma způsoby:

Konverzní konstruktory (typ → třída)

- konstruktory, které lze volat s jedním argumentem
- specifikace `explicit` zabrání implicitnímu použití

Konverzní operátory (třída → typ)

- metody pojmenované `operator T`
- C++11: je možná specifikace `explicit`
- nemají návratový typ a jsou bez parametrů. Např:

```
Trida::operator T() { return vyraz_typu_T; }
```

- tyto operátory se dědí a mohou být virtuální

Konverze – příklady

```
struct T {
  T(int);                   // int --> T
  T(const char*, int =0); // const char* --> T
  explicit T(U);            // explicitní U --> T
  operator int();           // T --> int
};
void f(T t, U u) {
  T b = "text";  // T("text",0), kopie
  T c{u};        // OK: explicitní konverze
//  T d = u;     // Chyba: implicitní konverze U --> T
  b = 5;         // b = T(5)
  b = T{u};      // OK: U --> T
  int i = c;     // T --> int  implicitní konverze
  i = int(c);    // T --> int  explicitní
  i = static_cast<int>(c);  // T --> int
  if(i<1) f(66,u); // int --> T  implicitní konverze
}
```

Přetěžování operátorů

Přetěžování = přisouzení více významů jednomu symbolu

- Cíl: zpřehlednění zápisu programů
- Rozlišení operací se provádí podle kontextu
- Lze přetěžovat všechny operátory C++ kromě:
  `.` `.*` `::` `?:` `sizeof` a preprocesorových `#` `##`
- Nelze změnit počet operandů operátorů ani jejich prioritu a pravidla pro asociativitu
- Alespoň jeden parametr musí být třída nebo výčet
- Je vhodné dodržovat jistá pravidla:
  - zachovávat smysl operátorů,
  - pokud vestavěná operace (například `int + int`) nemění žádný operand, nemají to dělat ani přetížené varianty.

Přetěžování operátorů 2

Některé operátory jsou definovány implicitně:

- přiřazení (`X & operator = (const X&);`)
- *"move"* přiřazení (`X & operator = (X&&);`)
- získání adresy objektu (unární `&`)
- výběr členu struktury (`.`)
- výběr členu struktury (`->`)
- velikost objektu `sizeof`

a pouze některé z nich můžeme přetížit.

Přetěžování operátorů 3

Operátory je možné definovat jako (nestatické) metody,
nebo jako funkce (znak `@` reprezentuje libovolný operátor):

výraz	metoda	funkce
`@a`	`(a).operator@ ()`	`operator@ (a)`
`a@b`	`(a).operator@ (b)`	`operator@ (a, b)`
`a=b`	`(a).operator= (b)`	
`a[b]`	`(a).operator[](b)`	
`a->`	`(a).operator->()`	
`a@`	`(a).operator@ (0)`	`operator@ (a, 0)`

- Překladač vždy převede zápis operátoru na volání odpovídající operátorové funkce nebo metody.
- Jsou-li deklarovány obě varianty, je aplikováno standardní testování argumentů pro řešení nejednoznačnosti.

Přetěžování unárních operátorů

Možné formy deklarace unárních operátorů

- nestatická metoda bez parametrů
- funkce (ne metoda) s jedním parametrem

Když znak `@` reprezentuje unární operátor, potom:

```
@x  a  x@
```

může být obojí interpretováno buď jako volání metody:

```
x.operator @()
```

nebo jako volání funkce:

```
operator @(x)
```

podle formy deklarace operátoru.

Přetěžování unárních operátorů 2

Prefixové a postfixové operátory ++ -- lze rozlišit pomocí dodatečného fiktivního parametru operátoru:

```
class Y {
  // ...
 public:
  Y operator ++()    { /* inkrementace ++x */ }
  Y operator ++(int) { /* inkrementace x++ */ }
};

Y x,y;
y = ++x;   // volá  Y::operator ++()
y = x++;   // volá  Y::operator ++(int)
```

Přetěžování binárních operátorů

Možné formy deklarace binárních operátorů

- nestatická metoda s jedním parametrem
- funkce (ne metoda) se dvěma parametry

Výraz:

```
x@y
```

je interpretován buď jako volání metody:

```
x.operator @(y)
```

nebo jako volání funkce:

```
operator @(x,y)
```

Přetěžování binárních operátorů — příklad

```
class Complex {
   double re, im;
public:
   Complex(): re(0), im(0) {}
   Complex(double r, double i=0): re(r), im(i) {}
   friend Complex operator +(Complex c1, Complex c2);
   // použití funkce řeší problém:  1 + Complex(1,2)
};
```

```
Complex operator +(Complex c1, Complex c2) {
   return Complex(c1.re + c2.re, c1.im + c2.im);
}
```

```
Complex a(0,1), b(1,0), c;
c = a + b;           // přetížený operátor +
c = operator+(a,b);  // operátorová funkce
```

Přetěžování operátoru přiřazení

```
T& T::operator= (const T&);
```

- není-li definován kopírovací `operator =`, překladač jej vytvoří jako přiřazení po jednotlivých složkách
- nedědí se (na rozdíl od ostatních operátorových funkcí)
- může být virtuální

Poznámky:

- při implementaci je třeba uvažovat i možnost přiřazení typu `a = a;` (Např: `p[i] = p[j];` pro `i==j`)
- specifikací `=delete` zabráníme vytvoření operátoru
- specifikací `private` můžeme zabránit přiřazování objektů

"move" operátor přiřazení

```
T& T::operator= (T&&) noexcept;
```

- není-li definován *"move"* `operator=`, překladač jej vytvoří jako *"move"* přiřazení po jednotlivých složkách (ale jen když nedefinujete *"copy/move"* konstruktor ani *"copy"* přiřazení)
- nedědí se
- může být virtuální

Poznámky:

- implementuje se obvykle jako operace *swap*
- specifikací `=delete` zabráníme vytvoření operátoru
- specifikací `private` můžeme zabránit jeho použití

Přetěžování operátoru přiřazení — příklad

```
class String { public:
  // copy-assignment:
  String & operator = (const String & str) {
     if(&str!=this)
        // ... přiřazení dat (delete,new,copy)
     return *this;
  }
  // move-assignment:
  String & operator = (String && s);
  String & operator = (const char *cstr) {
     *this = String(cstr); // použije "move"=
     return *this;
  }
  // ... další metody/data
};
```

Přetěžování operátoru volání funkce

```
typ T::operator() (parametry);
```

- `operator()` musí být nestatická metoda
- může mít libovolný počet parametrů
- Například volání `x(a1,a2)`, kde x je objekt je interpretováno jako `x.operator()(a1,a2)`.

Použití: funkční objekty v STL

```
std::less<int>()
```

Přetěžování operátoru indexování pole

```
typ_prvku& T::operator[] (typ_indexu);
```

- může být definován pouze jako nestatická metoda
- Binární operátor indexování `a[b]`
  je interpretován jako `a.operator[](b)`
- vícenásobné indexování:
  - pomocná třída (více tříd) s operátorem indexování
  - operátor indexování základní třídy vrací referenci na objekt pomocné třídy
  - pomocný objekt je opět indexovatelný
  - výsledkem musí být reference na prvek vícerozměrného pole

Poznámka: C++23 dovoluje více parametrů

Přetěžování přístupového operátoru ->

```
typ T::operator-> ();
```

- `operator->` je považován za unární operátor a musí být definován jako nestatická metoda
- Výraz `x->m`, je interpretován jako `(x.operator->())->m`
- `operator->()` musí vracet buď:
  - ukazatel na objekt třídy nebo
  - objekt jiné třídy pro kterou je také definován `operator->`

Poznámka: *"smart pointers"* — `unique_ptr<T>, shared_ptr<T>`

Omezení pro dočasné objekty — příklad

Nelze volat metody dočasných objektů vytvořených implicitní konverzí:

```
class X {
    int i;
  public:
    X(int _i=0) : i(_i) {}
    X operator + (const X &b) const {
      return X(i + b.i);
    }
};

int main() {
  X a, b;
  a = 5 + b;      // Chyba! (b+5 by fungovalo)
  a = X(5) + b;   // O.K. (explicitní konverze)
}
```

Přetěžování funkcí

Možnost deklarovat více funkcí se stejným jménem

- rozlišují se podle počtu a typu argumentů při volání
- souvislost s typovou kontrolou
- pravidla pro vyhledání funkcí (ADL)
- problém nejednoznačnosti (*ambiguity*)
- funkce nelze rozlišovat jen podle návratového typu
- přístupová práva se berou v úvahu až po kontrole jednoznačnosti

Poznámky:
`typedef` nedefinuje samostatný typ
možné problémy v souvislosti s prostory jmen a `friend` funkcemi

Přetěžování funkcí — příklad

```
void print(double);
void print(long);

void F() {
  print(1L);      // tiskne long
  print(1.0);     // tiskne double
  print(1);       // CHYBA: nejednoznačnost
}
```

Poznámka:
Řešení nejednoznačnosti: přetypování nebo definice další funkce
`void print(int);`

Rozlišení typů

Při přetěžování nelze rozlišit některé typy:

```
void f(int);
void f(const int);  // CHYBA - nelze rozlišit

void g(int*);
void g(const int*); // O.K. - ukazatele lze rozlišit

void h(int&);
void h(const int&); // O.K. - reference lze rozlišit
```

Pravidla pro prohledávání prostorů jmen

ADL = *Argument Dependent Lookup* (*"Koenig lookup"*)

Při nekvalifikovaném volání funkce `f` s případnými argumenty se hledá odpovídající množina *kandidátských* funkcí takto:

1. Pro vyhledávání funkce v prostoru jmen se použije pouze jméno funkce.
2. V hierarchii prostorů jmen hledá se od aktuálního směrem ke globálnímu (třída je prostorem jmen); navíc se uvažují i další související prostory podle typu argumentů (ADL).
3. Najde-li se funkce/metoda stejného jména, provádí se rozlišení funkce podle typu a počtu parametrů (viz *"viable functions"*)
4. Nenajde-li se odpovídající funkce, hlásí se chyba.

Pravidla pro výběr odpovídající funkce (zjednodušeno)

Z množiny kandidátů se vyberou *"viable"* funkce podle:

- počtu argumentů (včetně ...)
- možných implicitních typových konverzí argumentů

Z těchto funkcí se vybere *nejlépe vyhovující* funkce v pořadí:

1. Shoda typů (kromě nevyhnutelných konverzí: pole → ukazatel, jméno funkce → ukazatel na funkci, `T` → `const T`)
2. Shoda typů po konverzi na `int` (`char` → `int`, `short` → `int`, atd.) a `float` → `double`
3. Shoda po provedení *standardních* konverzí (`int` → `double`, `odvozená_třída *` → `bázová_třída *`)
4. Shoda po provedení *uživatelem definovaných* konverzí
5. Použití nespecifikovaného počtu argumentů (...)

Nalezení dvou a více stejně dobrých funkcí vede k chybě — jde o nejednoznačnost (*"ambiguity"*).

```
void f(int);        // globální funkce
class T {
 public:
  T(int);           // konverze    int ---> T
  operator int();   // konverze      T ---> int
  void f(long);     // --- kandidát 1
  void f(double);   // --- kandidát 2
  void m() {
     f(5);      // chyba --- nejednoznačnost
     ::f(5);    // O.K.  --- globální funkce
  }
  friend int operator + (T x, T y);
};
T x, y;
x = 5 + y; // nejednoznačnost: 5+int(y) nebo T(5)+y ?
```

Možné řešení: zrušení jedné z konverzí, případně použití specifikace `explicit`

Vstup/výstupní operace s využitím streamů

stream (proud) — objekt, zobecnění pojmu soubor

```
#include <iostream>
#include <fstream>
```

- standardní třídy `ios`, `istream`, `ostream`, `iostream`, `ifstream`, `ofstream`, `fstream` a další
- standardní streamy: `cin`, `cout`, `cerr`
- použití operátorů << a >> pro vstup/výstupní operace (jsou definovány pro všechny vestavěné typy)
- možnost doplňovat operátory << a >> pro uživatelem definované typy (bez zásahu do implementace streamů)
- pozor na priority operátorů!

Definice a použití vstup/výstupních operací

Zápis

```
cerr << "x=" << x << '\n';
```

je ekvivalentní zápisu

```
operator<<(operator<<(operator<<(cerr,"x="),x),'\n');
```

a tiskne `x` způsobem, který je definován pro typ objektu x

```
int x = 123;              tiskne:  x=123
Complex x(1,2.4)          tiskne:  x=(1,2.4)
```

Poznámky: C++20: `std::format` C++23: `std::print`

Příklad definice výstupního operátoru

Příklad pro jednoduchou třídu `Complex`

```
// výstupní formát:  (f,f)
ostream& operator<< (ostream& s, Complex z) {
  return s <<'('<< real(z) <<','<< imag(z) <<')';
}
```

Poznámka: Možnost tisku čísla `f` místo `(f,0)`

Příklad definice vstupního operátoru

```
// možné vstupní formáty:   f    (f)   (f,f)

istream& operator>>(istream& s, Complex& a) {
  double re = 0, im = 0;
  char c = 0;
  s >> c;      // čte první nemezerový znak
  if( c == '(' ) {
    s >> re >> c;
    if( c == ',' )
      s >> im >> c;
    if( c != ')' )
      s.setstate(ios::failbit); // chybový stav
  }
  else {
```

(pokračování)

Příklad definice vstupního operátoru – pokračování

```
  else {
    s.putback(c);
    s >> re;
  }
  if( s ) a = Complex(re,im);    // O.K.
  return s;
}
```

Poznámky:

- Vstup je složitější, protože je nutné ošetřit *všechny* varianty
- `if(s)` je totéž jako `if(!s.fail())` a podmínka neplatí, když je nastaven příznak `ios::failbit` (chybné formátování vstupu, pokus o čtení za EOF) nebo `ios::badbit` (stream je vážněji porušen).

Chybové příznaky

`ios::eofbit`	pokus o čtení za koncem souboru
`ios::failbit`	chyba formátování, neočekávaný znak, ...
`ios::badbit`	vážná chyba, nelze zotavit

Metody:

`bool good()`	žádný chybový bit není nastaven
`bool eof()`	nastaven `eofbit`
`bool fail()`	nastaven alespoň jeden z `failbit`, `badbit`
`bool bad()`	nastaven `badbit`
`explicit bool()`	C++11: vrací `!fail()`
`operator void*()`	C++98: vrací null když platí `fail()`
`void clear()`	nuluje příznaky

Manipulátory

`flush`	vyprázdní vyrovnávací paměť
`ws`	přeskočí oddělovače na vstupu
`hex`	použije šestnáctkovou notaci
`dec`	použije desítkovou soustavu
`boolalpha`	použije ”true”/”false”
...	...

Příklad: použití manipulátorů

```
cout << "login: " << flush;
cin >> noskipws >> c1 >> ws >> c2 >> c3 >> skipws;
cout << 1234 << ' ' << uppercase << hex << 1234 << endl;
```

Manipulátory s parametry

```
#include <iomanip>
```

`setfill(char)`	výplňový znak
`setprecision(int)`	přesnost
`setw(int)`	šířka tisku

Příklad: použití manipulátorů

```
cout <<setw(4)<<setfill('#')<< 33 << "+" << 22;
výstup:  ##33+22
```

Poznámka: Nastavení manipulátoru `setw()` platí pouze pro jednu tisknutou položku, ostatní manipulátory platí až do další změny

Práce se soubory

Pro práci s diskovými soubory jsou definovány třídy

- `fstream`
- `ifstream`
- `ofstream`

Navázání streamu na soubor lze provést při vzniku objektu nebo voláním metody `open`.

Metodou `clear()` je možné zrušit chybový stav streamu.

Poznámka: Kombinace streamů/C-funkcí (např. `printf`) a vláken může vést k problémům v pořadí výstupů.

```
std::ios::sync_with_stdio(false);   // =zrychlení
```

Práce se soubory — příklad

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

ifstream from("soubor1.txt");
ofstream to("soubor2.txt");
fstream  f("soubor3.txt", ios::in | ios::out);

void copy(ifstream &from, ofstream &to) {
  char ch;  // může být char (EOF se testuje jinak)
  while(from.get(ch))  // kopie souboru (pomalé!)
    to.put(ch);
  if(!from.eof() || to.fail())  // test chyb
    error("chyba!");
}
```

Dědičnost

- Odvozování nových tříd z již existujících
- Jednoduchá a násobná dědičnost
- Bázové (”base”) a odvozené (”derived”) třídy

Příklad

```
class D : public B { // třída D dědí třídu B
   // nové položky, nové a modifikované metody
};
```

- Sdílení kódu — znovupoužitelnost (reusability)
- Sdílení rozhraní — polymorfismus
- Třída dědí všechny datové složky bázové třídy a běžné metody (ne konstruktory, destruktor, `operator=`).
- Ze zděděných členů třídy lze používat pouze ty, které byly `public` nebo `protected`

Dědění `private`, `public`, `protected`

Specifikace přístupových práv u všech zděděných prvků:

Příklad

```
class D : public B1, protected B2, B3 {};
```

Implicitní je `private` pro `class D`, a `public` pro `struct D`

Odvozená třída dědí atributy přístupu takto:

Způsob dědění	Změna přístupových práv zděděných členů z B v odvozené třídě D
`public B`	`public` i `protected` nezměněno
`protected B`	`public` a `protected` bude `protected`
`private B`	všechny členy budou `private`

Lze změnit přístupová práva u jednotlivých zděděných položek

- použitím `using` deklarace (viz. prostory jmen)
- lze obnovit max. na původní úroveň práv před děděním

Pořadí volání konstruktorů a dědičnost

Konstruktor odvozené třídy je složen z:

1. volání (inline rozvoje) konstruktoru bázové třídy,
2. volání případných konstruktorů lokálních objektů (v pořadí jak jsou uvedeny ve třídě) a
3. nakonec se provádí tělo konstruktoru.

Předání parametrů vyvolaným konstruktorům je možné za dvojtečkou v seznamu inicializátorů konstruktoru (viz následující příklad).

Poznámka: C++11 dovoluje explicitní volání jiného konstruktoru stejné třídy. (*delegation*)

Poznámka: Destruktory jsou vždy volány v přesně obráceném pořadí než jim odpovídající konstruktory.

Pořadí volání konstruktorů — příklad s chybou

```
class Base {    // POZOR - toto je příklad hrubé chyby
  int x;
 public:
  Base (int i): x(i) {}
};

class Derived : Base {
  int a;
 public:
  Derived(int i) : a(i*10), Base(a) {} // chyba!
  // do Base je předána neinicializovaná hodnota a !
  // konstruktor provede:
  //  1) Base::Base(a)   neinicializovaná hodnota a
  //  2) a(i*10)         inicializace členu a (pozdě)
};
```

Pořadí volání konstruktorů – příklad 2

```
struct base {  //....
  base(int i=0) { }
};
class derived : base {
  complex x;   // vnořené objekty
  complex y;
  int z;
 public:
  derived() : x(2,1) { f(); }
  // Konstruktor provede:
  //  1) base::base()
  //  2) complex::complex(2,1) pro složku x
  //  3) complex::complex() pro y
  //  z  není inicializováno, POZOR na nekorektní kód
  //  4) volání f()
};
```

Virtuální metody

Metody (`public`) označené klíčovým slovem `virtual` umožňují *dynamický polymorfismus*:

- v bázové třídě definují *rozhraní* společné všem odvozeným třídám
- v odvozených třídách je definována odpovídající *implementace* metod
- při správném použití společného rozhraní není třeba znát přesně třídu objektu (ještě nemusí existovat) a přesto je při běhu programu zajištěno volání odpovídajících metod – tzv. *pozdní vazba* ("*late binding*").

Polymorfní volání má typicky tvar:

```
ukazatel->vmetoda(parametry);
```

Virtuální metody 2

Lze definovat virtuální destruktory, které umožňují korektně rušit prvky dynamických datových struktur.

Poznámka: Polymorfní třídy by *vždy* měly mít virtuální destruktor.

Virtuální metody se mohou lišit ve vraceném typu jestliže splňují následující podmínky:

- předefinovaná virtuální metoda vrací ukazatel nebo referenci na bázovou třídu
- přepisující funkce vrací ukazatel nebo referenci na odvozenou třídu (tzv. *kovariantní typ*)

Virtuální metody – příklad

```
class B {
 public:
  virtual B* vf1() { cout << "B"; return this; }
  void f() { cout << "B"; /* vf1(); this->vf1(); */ }
};
class C : public B {
 public:
  C* vf1() { cout << "C"; return this; } // virtual
  void f() { cout << "C"; }
};
class D : public B {
 public:
  virtual D* vf1() { cout << "D"; return this; }
  void f()         { cout << "D"; }
};
```

Virtuální metody – příklad – pokračování

```
int main() {
  B b; C c; D d;
  b.f();     // "B" - obyčejná metoda
  c.f();     // "C"
  d.f();     // "D"
  b.vf1();   // "B" - stejné
  c.vf1();   // "C"
  d.vf1();   // "D"

  B *bp = &c;// uk. na bázovou třídu - rozhraní
  bp->vf1(); // "C"-podle skutečné třídy objektu c
  bp->f();   // "B"-podle typu ukazatele bp
  bp = &d;   // ukazuje na objekt třídy D
  bp->vf1(); // "D"-podle skutečné třídy objektu d
  bp->f();   // "B"-podle typu ukazatele bp
}
```

Virtuální metody — implementace

Pro pochopení virtuálních metod je vhodné vědět, jak je uvedený mechanismus obvykle implementován:

- Každá třída s virtuálními metodami má tzv. tabulku virtuálních metod (VMT – *Virtual Method Table*), ve které jsou odkazy na všechny virtuální metody třídy.
- Při dědění (při překladu) se převezme obsah VMT bázové třídy, odkazy na virtuální metody, které byly předefinovány se nahradí novými a na konec VMT se doplní odkazy na případné nové virtuální metody.
- Každý objekt třídy s virtuálními metodami obsahuje odkaz na tabulku virtuálních metod
- Polymorfní volání použije ukazatel v objektu, vybere odpovídající položku z VMT a zavolá metodu.

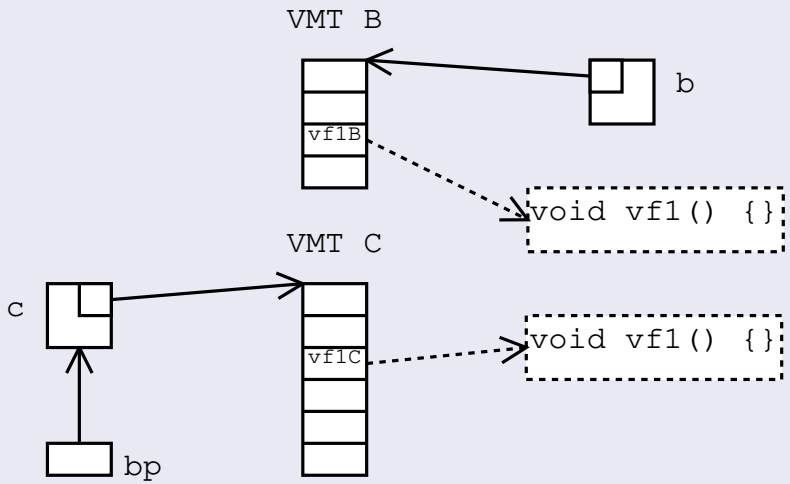
Příklad implementace polymorfismu

```
class B {
 public:
  virtual void vf1(); // rozhraní
  // ...
} b;

class C : public B {  // musí být public
 public:
  virtual void vf1();
  // ...
} c;

B *bp = &c;  // polymorfní rozhraní: bp->vf1()
```

Příklad implementace polymorfismu — obrázek



Komentář k polymorfismu

Voláme-li virtuální metodu, je volána *nepřímo* přes VMT – to umožňuje volat odpovídající metodu i když třída objektu není překladači známa (máme pouze ukazatel na objekt bázové třídy). Informace o třídě objektu je vlastně ukryta v odkazu na VMT této třídy, který je součástí každého objektu.

Máme-li ukazatel na objekt bázové třídy B, můžeme do něj přiřadit ukazatel na třídu z ní odvozenou. Pomocí tohoto ukazatele můžeme potom volat metody definované ve třídě B. Pokud jsou tyto metody virtuální, budou volány metody, které přísluší skutečné třídě objektu na který se ukazuje.

Můžeme definovat bázovou třídu tak, aby poskytovala vhodné rozhraní pro celou skupinu odvozených tříd (které mohou být definovány později) a pracovat potom s objekty (přes toto rozhraní) nezávisle na jejich třídě.

Komentář k polymorfismu — pokračování

Příkladem může být bázová třída `PrvekSeznamu`, která definuje obecné vlastnosti jednoho prvku seznamu, se kterým umí pracovat třída `Seznam`. Potom lze ze třídy `PrvekSeznamu` odvodit jakoukoli novou třídu a její objekty ukládat do seznamu, i když všechny jejich vlastnosti nebyly ještě známy v době psaní implementace třídy `Seznam` (bylo známo pouze rozhraní definované třídou `PrvekSeznamu`).

Poznámka: Dynamický vs statický polymorfismus

Poznámka: Možnost optimalizace (inline rozvoj, ...)

Volání virtuálních metod třídy v konstruktoru

Při provádění konstruktorů (a destruktorů) se mechanismus volání virtuálních metod neuplatňuje na metody konstruovaného objektu — konstruktor vždy nastaví svoji tabulku virtuálních metod a nelze tedy volat metody odvozených tříd.
Důvodem jsou možné problémy při volání virtuální metody, která může pracovat s ještě neinicializovanými složkami objektu.

Poznámka: Předpokládáme polymorfní volání — viz následující příklad.

Poznámka: Podobně se chová `typeid` — viz dále.

Volání virtuálních metod v konstruktoru – příklad

```
struct A {
  A() { m(this); }
  void m(A *bp) { bp->V(); }
  virtual void V() { cout << 'A' ; }
};
class B : public A {
  int i;
 public:
  B(): i(9) { m(this); }
  virtual void V() { cout << 'B' << i << ' ' ; }
};
int main() {
  B b;            // vytiskne AB9 a nikoli B?B9
  A *bp = &b;
  bp->V();        // vytiskne B9
}
```

Čistě virtuální metody

Virtuální metoda nemusí být definována. Potom musí být deklarována jako tzv. *čistě virtuální metoda* (*"pure virtual function"*):

Příklad

```
class B {
  virtual void pvf(int) = 0;
};
```

- Ve všech odvozených třídách se tato metoda dědí jako čistě virtuální, dokud není definována.
- Jsou vhodné pro definování rozhraní v bázových třídách.

Poznámka: Ve speciálních případech lze doplnit definici.

Abstraktní třídy

= třídy s minimálně jednou čistě virtuální metodou

- použitelné pouze jako bázové třídy při dědění
- nelze vytvářet žádné objekty
- lze použít ukazatel/referenci na abstraktní třídu

Příklad:

```
class shape {       // abstraktní třída - obrazec
  point center;
 public:            // definice rozhraní
  virtual void rotate(int)   =0;
  virtual void move(int,int) =0;
  virtual void draw()        =0;
  //...
};
...(pokračování)
```

Abstraktní třídy – příklad

```
class Circle : public shape { // kružnice
  int radius;
 public:
  virtual void rotate(int) {} // definice
  virtual void move(int,int);
  virtual void draw();
  //...
};

shape s;         // nelze vytvořit objekt - chyba
Circle c;        // třída Circle není abstraktní
shape *sp=&c;    // ukazatel na abstraktní třídu

sp->rotate(90); // volání přes polymorfní rozhraní
```

Specifikace `override` a `final`

C++11: speciální identifikátory (ne klíčová slova)

- `override` zajišťuje kontrolu, zda opravdu jde o předefinování virt. metody (musí existovat v bázové třídě). Chyba nastane například když dojde k přetížení (jiné parametry).
- `final` zakazuje předefinování virt. metody v odvozených třídách nebo zakazuje dědění z dané třídy.

Příklad:

```
class D final : public B {
  void sone_func() override;     // CHYBA: překlep
  void vf(int) override;         // OK: virtual B::vf(int)
  virtual void h(char *) final;
};
```

Operátor `typeid`

```
typeid ( výraz )
typeid ( typ )
```

- Operátor získá informace o typu objektu za běhu programu.
- Překladač generuje typové informace a ty jsou dostupné při běhu aplikace (RTTI = *Run-Time Type Information*)
- Výsledek `typeid(výraz)` je typu `const type_info&`
  - `type_info` reprezentuje popis typu výrazu.
  - `výraz` je reference na polymorfní typ nebo `*ukazatel` (je-li `ukazatel==0`, pak dojde k výjimce `bad_typeid`).

Operátor `typeid` — příklad

```
class X {      // polymorfní třída
   // ...
   virtual void f();
   // ...
};

void g(X *p) {
  const type_info &a = typeid(p);   // X*
  const type_info &b = typeid(*p);  // třída X
                                    // nebo odvozená
}
```

Třída `type_info`

Deklarováno v `<typeinfo>` například takto:

```
class type_info {
    // ... implementačně závislá reprezentace
  public:
    virtual ~type_info();
    bool operator==(const type_info&) const noexcept;
    bool operator!=(const type_info&) const noexcept;
    bool before(const type_info&) const noexcept;
    const char *name() const noexcept;
    size_t hash_code() const noexcept;
    // zakázané operace:
    type_info(const type_info&) = delete;
    type_info & operator= (const type_info&) = delete;
};
```

Třída `type_info` — pokračování

- Účelem `before` je možnost řadit objekty `type_info`.
- Není definován žádný vztah mezi dědičností a `before`.
- Metoda `name` vrací řetězec jednoznačně reprezentující typ.

  Poznámka: Obsah řetězce je implementačně závislý (nemusí to být přesné jméno typu).

Operátory pro přetypování

```
const_cast < TYP > ( výraz )
static_cast < TYP > ( výraz )
reinterpret_cast < TYP > ( výraz )
dynamic_cast < TYP > ( výraz )
```

Doplněk k `TYP(výraz)` (nepoužívejte `(TYP)výraz`)

Operátory `static_cast`, `reinterpret_cast`, `const_cast` představují specifické přetypování — vyjadřují totéž co `(TYP)výraz`, kromě přetypování na `private` bázovou třídu.
Jejich účelem je explicitně zvýraznit úmysl přetypování.

Operátor `const_cast`

```
const_cast<T>(e)
```

- Přetypuje `e` na typ `T` přesně jako `(T)e` za předpokladu, že `T` a typ výrazu `e` se liší pouze v `const` a `volatile` modifikátorech a jde o ukazatel nebo referenci, jinak dojde k chybě.
- Nefunguje pro ukazatele na funkce a metody.
- Použití výsledku pro práci s původně konstantními daty může vést k nedefinovanému chování.
- Ostatní operátory přetypování respektují 'konstantnost' a nemohou ji měnit.

Operátor `static_cast`

```
static_cast<T>(e)
```

přetypuje `e` na typ `T` pokud:

- můžeme deklarovat dočasnou proměnnou `T pom(e);`
- typ `T` je `void`
- jde o přetypování reference/ukazatele na (ne virtuální) bázovou třídu na referenci/ukazatel na odvozenou třídu
- existuje inverzní standardní konverze,
  `int`→`enum`, `void*`→`Cls*`, `D::*p`→`B::*p`

jinak dojde k chybě.

Operátor `reinterpret_cast`

```
reinterpret_cast<T>(e)
```

přetypuje `e` na typ `T` když jde o konverzi ukazatelů na:

- celočíselný typ nebo enum a naopak
- funkce různého typu
- objekty různého typu (i reference)
- členy tříd různého typu

jinak dojde k chybě.
Ukazatele a reference uvažuje jako neúplné typy (vztahy bázová/odvozená třída neovlivní význam přetypování).

Poznámka: Výsledky `reinterpret_cast` se obvykle musí dále přetypovávat na svůj originální typ, aby byly použitelné.

Operátor `reinterpret_cast` — příklad

```
void f(char *p) { *p = 'x'; }
typedef void (*FP) (const char*);

int main() {
  FP p = reinterpret_cast<FP>(&f);
  p("text"); // pozor - nemusí fungovat (coredump)
  int *x=static_cast<int*>(malloc(9*sizeof(int)));
  long *y = reinterpret_cast<long*>(0x1FF0);
//  long *z = static_cast<long*>(0x1FF0); // chyba
}
```

Poznámka: Minimalizovat použití

Operátor `dynamic_cast`

```
dynamic_cast<T>(v)
```

speciální přetypování s kontrolou při běhu programu.
Typ `T` musí být ukazatel nebo reference na třídu nebo `void*`.

- Je-li `T` ukazatel a `v` je ukazatel na odvozenou třídu, potom je výsledkem ukazatel na unikátní podobjekt.
- Je-li `T` reference a `v` objekt třídy odvozené z `T`, potom je výsledkem reference na unikátní podobjekt.
- Když `T` je typ `void*`, potom `v` musí být ukazatel a výsledek je ukazatel na kompletní objekt.
- Jinak `v` musí být ukazatel nebo reference na polymorfní typ a je provedena kontrola za běhu programu, zda `v` může být konvertován na typ `T`. Když ne, operace skončí neúspěšně – výsledná hodnota po přetypování bude `0`, v případě reference je vyvolána výjimka `bad_cast`.

Operátor dynamic_cast — příklad

```
class B {
    // min. jedna virtuální metoda
    virtual void m() {}
};
class D : public B {
    // ...
};

int main() {
  B *pb1 = new B;
  B *pb2 = new D;  // implicitní přetypování

  D *pd1 = dynamic_cast<D*>(pb1); // 0  (nelze)
  D *pd2 = dynamic_cast<D*>(pb2); // uk. na objekt D
}
```

Šablony – základní pojmy

šablona (*template*) = typ parametrizovaný jinými typy/hodnotami

- *Generické programování*
- Klíčová slova: `template`, `typename`
- Šablony tříd
- Šablony funkcí a metod
- Šablony proměnných(C++14)
- Alias deklarace šablon (`using`, C++11)
- Instanciace, specializace a částečná specializace
- Metaprogramování, koncepty(C++20), ...

Poznámky: Definice v hlavičkových souborech, implicitní/explicitní generování instancí, typová kontrola, různé optimalizace, možnost *"code bloat"*, ...

Šablony – příklady

Deklarace

```
template<typename R, class T>  R f(T);
template<typename T>           class Vector;
```

Použití šablon, dedukce, instanciace

```
int i = f<int>(3.14);            // f<int,double>
double d = f<double,double>(3);

Vector<int> vec1;
Vector<Vector<int>> vec2; // Pozor na >> v C++98

std::map<std::string,int> m;
std::basic_string<unsigned char> us;
```

Parametry šablon

Parametrem šablony může být:

- typ — například `typename T`, `class T` nebo koncept (C++20)
- hodnota — např. `int N` nebo `auto N`
  Lze použít pouze celočíselný typ, výčet, ukazatel/referenci na objekt/funkci nebo ukazatel na člen třídy (a také float od C++20).
  Odpovídající argument musí být konstantní výraz.
- šablona — např. `template<typename T> class K`
- *"parameter pack"* — např. `template<typename ... Args>`

Šablona může mít více parametrů:

```
template<typename T, T v, typename U> class S;

template<typename T0, typename ... T1_N>
  void sink(T0 p_0, T1_N ... p_i) {}
```

Parametry šablon — pokračování

Lze definovat implicitní hodnoty parametrů šablon:

```
template< typename T = int, int N = 10 > class S;
template< template<typename T> class C = S > class X;
```

Příklad: šablona `Buffer`

```
template< typename T = char, int Size = 256 >
class Buffer {
    // Definice...
};

// Použití šablony:
Buffer<> buf;              // Buffer<char,256>
Buffer<long,1024> buf1;
Buffer<std::string> buf2; // Size=256
```

Šablony funkcí

Příklad: šablona `max`

```
template <typename T>
T max(T x, T y) {        // definice
  return (x>y)?x:y;
}
```

Při prvním použití překladač generuje odpovídající funkci (instanci šablony) podle typu argumentů:

```
int j = max(5,0);       // dedukce: max<int>(5,0)
MyClass a, b;
MyClass m = max(a,b);   //          max<MyClass>(a,b)
```

Omezení: Šablonu `max` lze použít pro libovolný typ, který má definován `operator>`, kopírovací konstruktor a destruktor.

Šablony funkcí a funkce

Je možné definovat další funkce `max`, například:

```
const char *max(const char *x, const char *y) {
  return (strcmp(x,y)>0) ? x : y;
}

const char *s = max("a", "aaa");
```

Potom má tato funkce přednost před šablonou.

Poznámka: Přetěžování funkcí

Pro generické funkce se neprovádí žádné implicitní konverze argumentů (pouze triviální: pole → ukazatel, ...):

```
void f(int i, char c) {
  max(i,i);  // volá max<int>(i,i)
  max(c,c);  // volá max<char>(c,c)
  max(i,c);  // není funkce max(int,char) =chyba!
  max(c,i);  // není        max(char,int) =chyba!
}
```

Pokud bychom předem explicitně deklarovali funkci

```
int max(int,int);
```

k chybě by nedošlo, protože v takovém případě se použije konverze `char` na `int`.

Poznámka: Parametry šablony lze dedukovat pouze podle argumentů (ne podle návratového typu funkce).

Šablony tříd — příklad `Vector`

```
template<typename T>  class Vector {
    T * data;
    int size;
    int capacity;
  public:
    explicit Vector(int size);
    // ... operator[] atd.
};
template<typename T>  Vector<T>::Vector(int  size) {
    // definice konstruktoru šablony
}
int main() {
    Vector<int> x(5); // generuje instanci vektoru
    for(int i = 0; i<5; i++)
      x[i] = i;
}
```

Specializace šablon – příklady

= explicitní definice speciálních případů:

```
// Obecná šablona funkce:
template<typename T> T max(T x, T y) {return (x>y)?x:y;}
// Specializace šablon funkcí:
template<typename T> T* max(T*a, T*b)      { /*...*/ }
template<> int*  max<int*>(int *a, int *b) { /*...*/ }
template<> char* max<>(char *a, char *b)   { /*...*/ }
template<> void* max(void *a, void *b)     { /*...*/ }
```

```
// Obecná šablona Vector - viz předcházející slajd
// Částečná (partial) specializace:
template<typename T> class Vector<T*>    { /*...*/ };
// Specializace:
template<>           class Vector<char*> { /*...*/ };
```

Specializace šablon

- Specializace musí následovat až po definici obecné šablony
- Definované specializace zabrání automatickému vytváření instancí z obecné šablony
- Použije se vždy nejvíce specializovaný případ:

```
// kód:              |  co se použije:
//-----------------------------------------
Vector<int>   v1; // obecná šablona Vector<T>
Vector<int*>  v2; // částečná specializace <T*>
Vector<char*> v3; // specializace Vector<char*>
```

Identifikátor šablony (`Vector`) nelze použít bez specifikace parametru šablony <>, kromě některých případů uvnitř popisu šablony. (Změna: CTAD)

Příklad: Seznam s typovou kontrolou 1

Problémem následující ”generické” implementace seznamu je nemožnost typové kontroly vkládaných prvků.

```
class Glist {
    // data
  public:
    void insert(void *);
    void *get();
    // ... další operace
};
```

Poznámky:

- Pouze *odkazy* přes ukazatel (na rozdíl od `std::vector` atd.)
- Heterogenní seznam (`void*`)

Příklad: Seznam s typovou kontrolou 2

Doplnění typové kontroly šablonou `List`:

```
template<class T>
class List : public Glist {
  public:
    void insert(T *t) { Glist::insert(t); }
    T *get()          { return (T *)Glist::get(); }
    // ...
};
```

Využití tohoto přístupu zkracuje kód programu a je přitom velmi efektivní (v tomto případě typová kontrola nic nestojí, protože na přetypování se negeneruje žádný kód a metody jsou inline).

Poznámky

- Šablony lze definovat jen v hlavičkových souborech (`export` zrušeno v C++11) nebo modulech (C++20).
- Každá instance šablony má svoje statické prvky.
- Je možné definovat šablony metod (včetně konstruktorů).
- Pro jména závislá na parametru šablony (*dependent types*) se používají klíčová slova `typename` a `template`:

```
template<class T>  class X {
  typedef typename T::TypeU U;   // vnořený typ
  using UU = typename T::TypeU;  // lepší, C++11
 public:
  int m(U o) {
    T::template m<10>(); // šablona statické metody
  } //           ^ zde by byl problém: operator<
};
```

Poznámky

- Explicitní instanciace šablony pro zadaný typ:

```
template class std::vector<int>;
```

- Externí deklarace šablony (nové v C++11):

```
extern template class std::vector<int>;
```

  Překladač neprovádí instanciaci v daném modulu (zrychlení překladu).

- *Alias templates* (C++11) řeší nedostatky `typedef` definic.

  Příklad:

```
template<typename T1, typename T2, int N3> class S;

template<typename T2>
using ExtraS = S<int, T2, 5>; // toto typedef neumí
```

Poznámky

- Použití šablon umožňuje nové přístupy — metaprogramování, různé optimalizace, atd.
- STL *(Standard Template Library)* použita jako základ standardní knihovny jazyka ISO C++.
- **SFINAE** (*Substitution Failure Is Not An Error*)
  Pokud nelze rozvinout šablonu funkce pro zadaný typ parametru, ale existuje jiná použitelná alternativa, *nejde o chybu*.
- Šablony s proměnným počtem parametrů (C++11 *Variadic templates*) dovolují elegantní zápis — viz např. `std::tuple`

Příklad: *fold expression*

C++17 dovoluje tzv. ”fold expression” pro binární operace nad libovolným počtem parametrů funkce:

```
// šablona funkce, binární operace + nad parametry:
template<typename ... Tpar>     // typy parametrů
auto sum1(Tpar ... args) {      // např. (T1 p1, T2 p2, T3 p3)
  // "binary left fold" např. pro sečtení ((0+p1)+p2)+p3:
  return (0.0 + ... + args);    // závorky nutné, asoc. ->
}
template<typename ... Tpar>
auto sum(Tpar ... args) {
  return (... + args);          // "unary left fold"
}
int main() {
  auto x = sum1(11.0, 2, 3.14, 4); // libovolné typy s operací +
  return sum(1, 2, 3, 4);          // ==10
}
```

Příklad: *fold expression 2*

Typově bezpečný tisk podobný `std::printf`:

```
#include <iostream>

// binary left fold:
template<typename... Tparams>
void print(Tparams... args) {
    ((std::cout << args), ...);
}

// test:
int main(int argc, char *argv[]) {
    print(11.0, ' ', argc, " pi=", 3.14, '\n');
}
```

Obsluha výjimek

Obsluha chyb vznikajících při běhu programu

Tradiční přístupy:

1. Ukončení programu
2. Vrácení hodnoty, která reprezentuje chybu (např. `fopen`)
3. Vrácení legální hodnoty a ponechání programu v nesprávném stavu (nevyhovující)
4. Volání speciální funkce, která chyby ošetří (např. `matherr`)

C++ umožňuje strukturované řešení výjimečných situací:

Klíčová slova:

`try`	—	vymezení obsluhované oblasti
`throw`	—	generování výjimky
`catch`	—	zachycení výjimky

Příklad: třída `Vector` s kontrolou mezí indexu

```
class Vector {
  int *p;
  int sz;
 public:

  class Range {};  // typ výjimky

  int &operator[] (int i) {
    if(i>=0 && i<sz) return p[i];
    else             throw Range(); // vznik výjimky
  }

  // ...
};
```

Příklad: třída `Vector` s kontrolou mezí indexu 2a

```
void f(Vector &v) {
  // ...
  try {  // "hlídaná" oblast je tento blok
    v[x] = 5;
    f2();
    // ...
  }
  catch(Vector::Range) {
    // obsluha výjimky  Vector::Range
    // tento kód se provede jen když se
    // v bloku try vyskytne index mimo rozsah
  }
  // ...
}
```

Příklad: třída `Vector` s kontrolou mezí indexu 2b

Kontrolujeme celou funkci na výskyt výjimky `Vector::Range`:

```
void f(Vector &v) try {
    // "hlídaná" oblast je celá funkce
    v[x] = 5;
    f2();
    // ...
}
catch(Vector::Range) {
    // obsluha výjimky  Vector::Range
    // tento kód se provede jen když se
    // v sekci try vyskytne index mimo rozsah
}
```

Rozlišení výjimek

Příkaz `catch` může mít parametr typu

`T`, `const T`, `T&`, nebo `const T&`

Takový příkaz zachytí výjimky:

- stejného typu
- typu pro který je `T` public bázovou třídou
- je-li `T` ukazatel, výjimka musí být typu ukazatel, který lze zkonvertovat standardní konverzí na `T`

Příklad: třída `Vector` s kontrolou indexu a rozměru

```
class Vector {
  int *p;
  int sz;
 public:
  static const int max = 10000;
  class Range {};  // výjimka - rozsah indexu
  class Size {};   // výjimka - chybná velikost
  Vector(int size) {
    if(size<0 || size>max) throw Size();
    // ...
  }
  int &operator[] (int i);
};
```

Příklad: třída `Vector` s kontrolou indexu a rozměru

```
void f() try {
    use_vectors(); // použití třídy Vector
}
catch(Vector::Range) {
    // obsluha výjimky  Vector::Range
}
catch(Vector::Size) {
    // obsluha výjimky  Vector::Size
}
```

Poznámky:

- Funkce `f` nemusí obsluhovat všechny možné výjimky — jejich zpracování je možné provést ve funkcích, které ji zavolaly.
- Obsluhu výjimek lze vnořovat, tj. v obsluze výjimky může být další blok `try`.

Výjimky s parametry

Výjimky mohou obsahovat užitečné informace:

```
class Vector {
    // ...
 public:
  struct Range {  // třída výjimky
    int index;    // hodnota chybného indexu
    Range(int i): index(i) {}
  };
  int &operator[] (int i) {
    if(i>=0 && i<sz) return p[i];
    throw Range(i); // předání hodnoty
  }
  // ....
};
```

....pokračování

Výjimky s parametry — pokračování

```
void f(Vector &v) try {
    // zde mohou vzniknout lokální objekty
    // ....
    v[x] = 5;   // pokud vznikne výjimka zde,
                // tak se následující kód neprovede
    // .... kód
    // ale destrukce lokálních objektů proběhne
}
catch(Vector::Range r) {
    // zde se pokračuje, pokud vznikla výjimka
    // (r se předá podobně jako parametr funkce)
    cerr << "chybný index" << r.index << '\n' ;
    // ....
}
```

Sdružování výjimek

Je možné definovat hierarchie výjimek — např:

```
class Matherr {};                       // bázová třída
class Overflow: public Matherr {};      // přetečení
class Underflow: public Matherr {};     // podtečení
class Zerodivide: public Matherr {}; // dělení nulou

void f() try {
  // ...
}
catch(Overflow) {
  // obsluha výjimek typu Overflow a odvozených
}
catch(Matherr) {
  // všechny ostatní odvozené z Matherr
}
```

Sdružování výjimek

- Vyhodnocování obsluhy výjimek probíhá v pořadí, ve kterém jsou uvedeny příkazy `catch`.
- Obsluha výjimky typu `T` zpracuje i všechny výjimky odvozené z `T`
- Všechny ještě neobsloužené výjimky je možno zpracovat v sekci `catch(...)`, která bude poslední.

Přidělování zdrojů

Tradiční (nevhodné) řešení:

```
void use_file(const char *jmeno) { // použití souboru
   FILE *f = fopen(jmeno,"r");      // otevření
   // použití souboru f
   // zde může vzniknout výjimka
   // a potom se následující kód neprovede
   // ....
   fclose(f);                       // uzavření
}
```

V případě výskytu výjimky uvnitř funkce je přeskočen příkaz `fclose` a soubor zůstane otevřen!

Přidělování zdrojů 2

Triviální (a nevhodné) řešení:

```
void use_file(const char *jmeno) {
   FILE *f = fopen(jmeno,"r");
   try {
      // použití souboru f
   }
   catch(...) {    // pro všechny výjimky ...
      fclose(f);   // uzavře soubor při výjimce
      throw;       // znovu vyvolá tutéž výjimku
                   // (pošle ji volající funkci)
   }
   fclose(f); // provede se jen v případě bez výjimky
}
```

Řešení přidělování zdrojů inicializací (RAII)

RAII (*Resource Acquisition Is Initialization*)

Zdroje zapouzdříme do lokálních objektů:

- přidělení zdroje = inicializace (volání konstruktoru)
- uvolnění při automatickém volání destruktoru na konci bloku

Funguje správně i při výskytu výjimky (viz dále)

Poznámka: Použití — viz např. `std::unique_ptr<T>`

Přidělování zdrojů = inicializace (příklad)

```
class FilePtr { // chová se stejně jako FILE*
   FILE *p;
 public:
   FilePtr(const char *name, const char *atr) :
     p(fopen(name,atr)) { if(!p) throw FileError(); }
   ~FilePtr()           { fclose(p); }
   operator FILE*()     { return p; }
}

void use_file(const char *jmeno) {
   FilePtr f(jmeno,"r");  // otevření souboru
   // použití souboru f
}  // automatické uzavření souboru
```

Konstruktory, destruktory a výjimky

Při vzniku výjimky v konstruktoru jsou volány destruktory všech objektů, jejichž konstruktory byly bezchybně dokončeny.

Příklad přidělování paměti s chybou

```
class X {
  int *p;
 public:
  X(int s): p(new int[s])  { init(); }
  ~X()  { delete [] p; }
  // ...
};
```

Použití této třídy není vhodné, protože při výjimce vzniklé v `init()` se nezavolá destruktor, protože nebyl kompletně dokončen konstruktor.

Bezpečné a univerzální řešení:

```
template<class T> class MemPtr {
   T *p;
 public:
   MemPtr(size_t s): p(new T[s]) {}
   ~MemPtr()         { delete [] p; }
   // TODO: delete copy-ctr, copy-operator=, ...
   operator T* ()    { return p; }
};
class X {
   MemPtr<int> cp;  // ukazatel - objekt
 public:
   X(int s): cp(s) { init(); }
};
```

Výjimka v `init()` vždy vede ke korektnímu uvolnění paměti.

Poznámka: Porovnat `std::unique_ptr<T>` a `shared_ptr<T>`

Specifikace výjimek

Specifikace, zda funkce (ne)může vyprodukovat výjimku.

Příklad specifikace

```
void f() noexcept;
void g() noexcept(true);
```

funkce `f` a `g` nemohou vyvolat výjimku.

Funkce bez specifikace může vyvolat jakoukoli výjimku:

```
int h();  // může vyvolat libovolnou výjimku
```

Poznámky:

Specifikace výjimek je součástí typu funkce (od C++17), při přetěžování se ale neuvažuje podobně jako návratový typ.

Zrušeno (C++98...14): Dynamické specifikace `T f() throw(v1,v2);`

Specifikace výjimek — poznámky

Poznámky:

- Destruktory jsou typicky `noexcept`
- U `noexcept` virtuální metody musí být všechny metody, které ji předefinovávají v odvozených třídách také `noexcept`.
- Do ukazatele na `noexcept` funkci lze přiřadit pouze ukazatel na `noexcept` funkci.
- Pozor na některé operace (např. `operator=`, copy/move)
- pokud se použije `throw;` mimo sekci `catch(...)`, dojde k vyvolání funkce `terminate`

Pomocné funkce

```
#include <exception>

int std::uncaught_exceptions() noexcept; // unwinding in progress

std::exception_ptr current_exception() noexcept;

[[noreturn]] void terminate() noexcept;

std::terminate_handler set_terminate( std::terminate_handler f )
```

Poznámky: vlákna

Neobsloužené výjimky

Není-li výjimka obsloužena žádným příkazem `catch`

- je volána funkce `terminate`, která ukončí program.
- Efekt volání `terminate()` lze předefinovat funkcí `set_terminate`:

```
typedef void(*PF_t)(); // ukazatel na funkci
PF_t set_terminate(PF_t);
```

- Implicitně platí, že `terminate()` volá `abort()`

Poznámka: Použití výjimek vede na větší kód, vlastní vyvolání výjimky je pomalé.

Standardní výjimky

```
<exception>:
class exception              // báze pro std výjimky
  class bad_exception

<new>:
  class bad_alloc

<typeinfo>:
  class bad_cast
  class bad_typeid
```

Standardní výjimky — přehled

```
<stdexcept>:
  class logic_error          logické chyby
    class domain_error
    class invalid_argument
    class length_error
    class out_of_range
  class runtime_error        chyby při běhu programu
    class range_error
    class overflow_error
    class underflow_error
```

Všechny tyto třídy mají definovánu metodu která vrací řetězec s popisem výjimky:

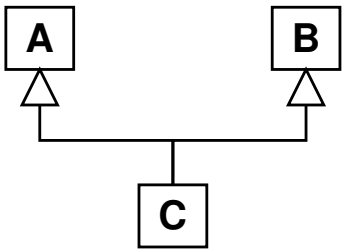
```
virtual const char* what() const noexcept;
```

Násobná dědičnost

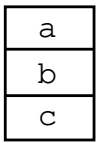
Dědění z několika bázových tříd současně:

```
class C : public A, public B {
   // členy třídy
};
```

hierarchie tříd:



objekt:



Násobná dědičnost

Poznámky:

- Anglicky: "*multiple inheritance*"
- Pořadí tříd v deklaraci není významné kromě pořadí volání konstruktorů, destruktorů a uložení v paměti.
- Použití násobné dědičnosti:
  loď + letadlo = hydroplán
- Pozor na konverze ukazatelů! (*thunk*, ==)
- Souvislosti: Java interface, `private`, ...

Násobná dědičnost — možné problémy

Možná nejednoznačnost:

- dvě bázové třídy mají stejně pojmenovaný člen — jejich použití lze rozlišit kvalifikací.
  Příklad `A::i` nebo `B::i`

Dědit dvakrát tutéž třídu lze jen nepřímo:

```
class X  { .... };        // bázová třída
class A : X  { .... };
class B : X  { .... };
class C : A,B  { .... };
```

Třída C obsahuje dvě instance třídy X.

Násobná dědičnost — virtuální báze

```
class X  { .... };
class A : virtual X  { .... };
class B : virtual X  { .... };
class C : A,B  { .... };
```

Třída C obsahuje pouze jednu instanci třídy X.

Poznámka: Možnost parametrizace konstruktoru virtuální bázové třídy X: `C::C(): X(1), A(2), B(3) { .... }`

Pořadí volání konstruktorů a násobná dědičnost

- Konstruktory bázových tříd jsou volány před provedením konstruktoru dané třídy v pořadí deklarace bázových tříd v hlavičce třídy.
- Konstruktory virtuálních bázových tříd jsou vyvolány před konstruktory nevirtuálních bázových tříd (je-li jich více volají se v pořadí jejich uvedení v deklaraci).
- Je-li v hierarchii tříd více instancí téže virtuální bázové třídy je její konstruktor volán pouze jednou.
- Jsou-li instance bázové třídy virtuální i nevirtuální, potom je konstruktor volán jednou pro všechny instance virtuální a jednou pro každou instanci nevirtuální — destruktory jsou volány v přesně obráceném pořadí než jim odpovídající konstruktory.

Ukazatele na členy tříd

Nejde o běžné ukazatele, nelze na ně aplikovat běžná pravidla pro přetypování, nelze je konvertovat na `void *`

operátor	první operand	druhý operand
`.*`	objekt třídy T nebo odvozené třídy	ukazatel na člen třídy T
`->*`	ukazatel na objekt třídy T nebo odvozené	ukazatel na člen třídy T

Příklad deklarace ukazatele na metodu

```
void (T::*ptr)(int) = &T::metoda2; // inicializace
```

Metoda musí být kompatibilní s ukazatelem.

Ukazatele na členy tříd — příklad

```
T objekt;
T *ukazatel_na_objekt = &objekt;
int main() {
   int (T::*p1)();
   p1 = &T::Metoda1; // přiřazení
   (objekt.*p1)();
   p1 = &T::Metoda5; // přiřazení
   (objekt.*p1)();

   void (T::*const p2)(int) = &T::Metoda2; // init
   (ukazatel_na_objekt->*p2)(parametr);
}
```

Poznámka: Vzhledem k prioritám operátorů je nutné používat závorky. Operátor `&` nelze vynechat.

Ukazatele na členy tříd — poznámky

Poznámky:

- Jestliže výsledkem operátoru `.*` nebo `->*` je metoda, potom jej lze použít pouze jako operand pro operátor `()`.
- Použití — například předávání ukazatele na metodu jako parametru funkce.

Základy použití operátorů .* a ->* — příklad

```
struct A {
  int i;
  void M() { i = 0; }
};
void (A::*u1)();
int (A::*u2);
int main() {
  A a;
  A *u = &a;
  u1 = &A::M;  // ukazatel na metodu M
  u2 = &A::i;
  (a.*u1)();   // volání metody objektu a
  (u->*u1)();  // volání přes ukazatel na objekt
  a.*u2 = 5;   // přiřazení do a.i
}
```

Prostory jmen (”namespaces”)

```
definice-prostoru-jmen:
  namespace identifikátor_opt { seznam-deklarací }

alias-definice-prostoru-jmen:
  namespace identifikátor = jméno-prostoru ;

jméno-prostoru:
  ::opt identifikátor
  jméno-prostoru :: identifikátor
```

Standardní prostory: `std`, `std::literals::string_literals`, ...
TODO: inline namespace

Prostory jmen (namespaces)

- možnost definovat prostor po částech
- jméno musí být unikátní (nesmí se shodovat se jménem třídy, objektu, typu, ...)
- definice prostoru jmen je deklarací
- prostory jmen lze vnořovat (`std::chrono`)
- existuje nepojmenovaný prostor jmen
- funkce deklarovaná jako `extern "C"` je svázána s C funkcí bez kvalifikace:

```
namespace X {
  extern "C" void f();   // C-funkce f()
}
```

Definice a použití prostoru jmen

```
namespace A {
   class String { /* ... */ };
   void f(String);
}

namespace B {
   class String { /* ... */ };
   void f(String);
}

void g() {
   A::String s; // explicitní kvalifikace
   A::f(s);
}
```

Definice a použití prostoru jmen

Členy prostoru jmen lze definovat vně tohoto prostoru:

```
void A::f(String s)        // A::String
{
   String ss = "aaa";      // A::String
   // ...
}
```

Poznámka: `extern`, `friend` funkce

Using deklarace

```
using-deklarace:
  using jméno-prostoru :: identifikátor ;
  using typename jméno-prostoru :: identifikátor ;
```

using deklarace není definicí, jsou dovoleny redundantní deklarace (ne pro lokální objekty ve funkci nebo třídě)

```
void h() {
   using A::String;
   using A::f;

   String s;          // A::String
   f(s);              // A::f
}
```

Using direktiva

```
using-direktiva:
  using namespace jméno-prostoru ;
```

using direktiva nedeklaruje žádná nová jména (pouze je zpřístupní)

```
void k() {
   using namespace A;
   String s;          // A::String
   f(s);              // A::f
}
```

- Pokud jsou nalezeny stejné identifikátory různých objektů (funkce se přetěžují) ve více prostorech jde o chybu.
- using direktiva je tranzitivní.

Třídy a prostory jmen

- pro účely hledání jmen je třída prostorem jmen
- using lze aplikovat podobně jako u prostoru jmen:

```
class B {
 public:
   virtual void f(int);
   virtual void f(double);
   // ...
};
class C : protected B {
 public:
   using B::f; // B::f(int), B::f(double) dostupné
   virtual void f(int);    // překrytí B::f(int)
   virtual void f(char);   // nové f(char)
   // ...
};
```

Třídy a prostory jmen

- using deklarace použitá uvnitř třídy jako členská deklarace musí odkazovat na přístupné členy bázových tříd
- using direktiva nemůže být použita jako členská deklarace
- uvnitř třídy nelze definovat prostor jmen
- prostor jmen tříd nelze rozdělit do více deklarací

Nepojmenovaný prostor jmen

Symboly nejsou dostupné z jiných modulů (jako `static` v C).

```
namespace {
  void f();  // přístupné pouze z aktuálního modulu
}

void g() {   // externí funkce přístupná všude
  f();
}
```

Poznámky:

- V C++ nepoužívat globální třídu paměti `static`, ale nepojmenovaný prostor jmen
- Nepoužívat zastaralé deklarace přístupu uvnitř tříd, ale using deklarace

Klíčové slovo `mutable`

`mutable` — může být aplikováno na členy třídy bez specifikace `const` nebo `static`. Takto označený člen třídy není nikdy konstantní — a to ani uvnitř konstantního objektu.

```
class T {
  mutable unsigned refcount;
  int data;
 public:
  T(): refcount(0), data(0) {}
  const T * ref() const { refcount++; return this; }
  void setdata(int n) { data = n; }
};
const T o;
const T * g() {
  o.setdata(4);   // chyba: nelze modifikovat const
  return o.ref(); // modifikace konstantního objektu
}
```

Standardní knihovny C++

C++ zahrnuje část standardní knihovny C (`<c*>`, `<*.h>`):

rozhraní	co obsahuje
`<cassert>`	Makro assert pro ladění.
`<cctype>`	Makra pro klasifikaci znaků.
`<cerrno>`	konstanty – chybové kódy
`<cfloat>`	Parametry a meze pro floating-point
`<cfenv>`	funkce pro floating-point prostředí
`<cinttypes>`	`intmax_t` atd
`<climits>`	rozsahy celých čísel
`<clocale>`	národní/jazyková podpora
`<cmath>`	Matematické funkce.
`<csetjmp>`	Typy pro `longjmp()` a `setjmp()`.
`<csignal>`	deklarace pro `signal()` a `raise()`

Standardní knihovny

rozhraní	co obsahuje
`<cstdarg>`	práce s proměnným počtem argumentů
`<cstddef>`	Některá makra a datové typy.
`<cstdio>`	Definice pro standardní vstup/výstup
`<cstdint>`	celočíselné typy a meze
`<cstdlib>`	Obecně použitelné funkce
`<cstring>`	Funkce pro práci s řetězci a pamětí.
`<ctime>`	Typy a funkce pro práci s časem.
`<cuchar>`	práce s Unicode znaky
`<cwchar>`	práce s `wchar_t`
`<cwctype>`	makra pro klasifikaci `wchar_t`

Definované symboly jsou umístěny v prostoru jmen `std` a některé i v globálním prostoru jmen.

Přehled standardní knihovny C++ 1/6

`<algorithm>`	algoritmy nad kontejnery
`<any>`	třída std::any (C++17)
`<array>`	`std::array` pole pevné velikosti
`<atomic>`	atomické operace
`<barrier>`	bariéra (C++20)
`<bit>`	funkce pro práci s bity (C++20)
`<bitset>`	posloupnost bitů pevné délky
`<charconv>`	`std::to_chars`, `std::from_chars` (C++17)
`<chrono>`	práce s časem
`<codecvt>`	Unicode konverze (nepoužívat)
`<compare>`	podpora operátoru <=> (C++20)
`<complex>`	komplexní čísla
`<concepts>`	koncepty (C++20)
`<condition_variable>`	podmínky pro čekání vláken

Přehled standardní knihovny C++ 2/6

`<coroutine>`	podpora korutin (C++20)
`<deque>`	obousměrná fronta
`<exception>`	obsluha výjimek, terminate
`<execution>`	podpora paralelizace algoritmů (C++17)
`<filesystem>`	třída `std::path` a podpůrné funkce (C++17)
`<format>`	podpora formátování, `std::format` (C++20)
`<forward_list>`	jednosměrně vázaný seznam
`<fstream>`	soubory - vstup/výstupní streamy
`<functional>`	funkční objekty (unární, binární)
`<future>`	nástroje pro asynchronní operace
`<initializer_list>`	inicializační seznam
`<iomanip>`	manipulátory pro streamy
`<ios>`	bázová třída pro streamy
`<iosfwd>`	forward deklarace pro streamy

Přehled standardní knihovny C++ 3/6

`<iostream>`	vstup/výstupní streamy
`<istream>`	vstupní streamy
`<iterator>`	iterátory
`<latch>`	synchronizační nástroj `latch` (C++20)
`<limits>`	implementační limity (mezní hodnoty)
`<list>`	seznam (DLL)
`<locale>`	lokalizace
`<map>`	asociativní kontejner (klíč–hodnota)
`<memory_resource>`	polymorfní alokátory atd. (C++17)
`<memory>`	správa paměti - alokátory, ...
`<mutex>`	nástroje pro vzájemné vyloučení
`<new>`	správa dynamické paměti
`<numbers>`	matematické konstanty (C++20)
`<numeric>`	numerické operace (`accumulate`,...)

Přehled standardní knihovny C++ 4/6

`<optional>`	šablona `std::optional` (C++17)
`<ostream>`	výstupní streamy
`<queue>`	fronta
`<random>`	(pseudo)náhodné generátory
`<ranges>`	podpora ”rozsahů”(C++20)
`<ratio>`	zlomky (compile-time)
`<regex>`	regulární výrazy
`<scoped_allocator>`	speciální alokátor (C++11)
`<semaphore>`	synchronizační nástroj semafor (C++20)
`<set>`	(multi)množina - asociativní kontejner
`<shared_mutex>`	vzájemné vyloučení (více ”čtenářů”)
`<source_location>`	podpora pro ladění (C++20)
`<span>`	`std::span` (C++20)
`<sstream>`	řetězcové streamy

Přehled standardní knihovny C++ 5/6

`<stack>`	zásobník
`<stdexcept>`	standardní typy výjimek
`<stop_token>`	podpora `std::jthread` (C++20)
`<streambuf>`	vyrovnávací paměť pro streamy
`<string>`	obecné řetězce (`basic_string`, ...)
`<string_view>`	šablona `std::basic_string_view` (C++17)
`<syncstream>`	`std::basic_osyncstream`, ... (C++20)
`<system_error>`	zpracování chyb
`<thread>`	vlákna
`<tuple>`	n-tice
`<typeindex>`	adaptér pro `type_info`
`<typeinfo>`	typové informace, `type_info`
`<type_traits>`	`is_class`, `is_array`, ...
`<unordered_map>`	”hash”-tabulka

Přehled standardní knihovny C++ 6/6

`<unordered_set>`	”hash”-množina
`<utility>`	relační operace
`<valarray>`	pole hodnot
`<variant>`	šablona `std::variant` (C++17)
`<vector>`	vektor
`<version>`	informace o implementaci knihovny (C++20)

Všechny definice jsou umístěny v prostoru jmen `std`.

Knihovny pro C++

Poznámka: STL — Standard Template Library — knihovna, která posloužila jako základ pro definice kontejnerů, iterátorů, algoritmů, atd. ve standardní knihovně C++98. STL kontejnery nejsou vhodné pro dědění (nemají virtuální destruktor). Iterátory mají podobné nevýhody jako ukazatele (mohou být zneplatněny např. modifikací kontejneru).

Na Internetu jsou dostupné další knihovny, které doplňují to, co není ve standardní knihovně ISO C++:

- Grafika a GUI (Qt, wxWidgets)
- Síťová rozhraní (Boost/asio, omniORB)
- Numerické metody (Blitz++, Eigen)
- Zpracování textu (Boost/spirit, Boost/JSON)
- Testování (Boost/Test, Googletest)
- ...

Poznámka: + různé nadstavby nad C knihovnami (GTKmm)

Boost

`http://www.boost.org/`

- Cíl: testovat nové knihovny aby se mohly zařadit do příští verze normy ISO C++.
- Zahrnuje celou řadu rozšiřujících knihoven pro C++:
  - regulární výrazy (regex), LL parser (spirit)
  - vlákna (thread), síťová komunikace (asio, MPI)
  - pseudonáhodná čísla (random), různá rozložení
  - matematika (math, uBLAS, quaternion, rational, ...)
  - zpracování obrazu (GIL), (geometry)
  - datové struktury (graph, bimap, circular buffer, dynamic bitset, ...)
  - n-tice, smart ptr, souborový systém, datum a čas, intervaly, lambda, dim. analýza (units)
  - návrhové vzory (flyweight), metaprogramování (MPL), serializace, coroutine
  - ...

Boost jako základ pro standardní knihovny

Například následující knihovny jsou v ISO C++:

- `<array>` — obálka pro pole (ve stylu STL)
- `<unordered_*>` — ”hash”funkce, kontejnery
- `<random>` — pseudonáhodná čísla, rozložení
- `<ratio>` — zlomky (*compile-time*)
- `<regex>` — regulární výrazy
- `<thread>` — vlákna
- `<tuple>` — n-tice
- `<type_traits>` — šablonové predikáty (`is_class`, ...)
- `<filesystem>` — souborový systém, adresáře, ...
- ...

Boost – shrnutí

- Další příklady viz WWW
- Boost je průběžně doplňován o další knihovny
- Knihovny v Boost jsou velmi dobře použitelné, je vhodné se s nimi seznámit i když ne všechny budou standardizovány.
- Nevýhody: závislosti knihoven

Přehled změn: C++11

- Změny jazyka:
  - R-hodnotové reference (výkon)
  - Lambda, vlákna, vylepšení šablon, inicializace, ...
  - Opravy chyb (`explicit`, ...)
  - Různá vylepšení syntaxe (`for`, `initializer_list`,...)
  - ...
- Podstatné vylepšení C++98 knihoven
- Nové C++11 knihovny

C++11 — R-hodnotové reference

```
T && rr = rvalue_expression;
```

Použití:

- reference na dočasné proměnné (nepojmenované),
- ”perfect function forwarding” v šablonách

Příklad: explicitní použití ”move” konstruktoru

```
std::vector<int> && f() {
    std::vector<int> pom(10000);
    // ...
    return std::move(pom); // nekopíruje data vektoru
}
```

Poznámka: není nutné používat explicitně (viz *copy elision*)

C++11 — constexpr

Definice musí být před použitím (stejně jako u `inline`)
Implicitně také `const`

Příklad:

```
constexpr int get(constexpr int x) { return x*2; }
int myarr[ get(2+3)+7];
int arr2[get(myarr[2])]; // CHYBA: myarr není constexpr

constexpr double acceleration_g = 9.8;
constexpr double moon_gravity = acceleration_g / 6;
```

C++11 — Inicializační seznamy

```
class SequenceClass {
  public:
    // initializer list constructor
    SequenceClass(std::initializer_list<int> list);
    // ...
};

SequenceClass x = {1, 2, 3, 4};
```

Inicializační seznam vytváří překladač a je konstantní.

```
void function_name(std::initializer_list<float> list);
function_name({1.0f, -3.45f, -0.4f}); // volání

std::vector<std::string> v = { "aaa", "bbb", "ddd" };
std::vector<std::string> v{ "aaa", "bbb", "ddd" };
```

C++11 – jednotná inicializace kontejnerů

```
struct Struktura1 { // POD = Plain Old Data
    int x;
    double y;
};

struct Struktura2 {
    Struktura2(int x, double y) : x_{x}, y_{y} {}
private:
    int x_;
    double y_;
};

// stejná inicializace:
Struktura1 s1{2, 3.14};
Struktura2 s2{2, 3.14};
```

C++11 – jednotná inicializace kontejnerů

```
struct IdString {
    std::string name;
    int identifier;
};

IdString get_string() {
    return {"SomeName", 4}; // typ není třeba
}

std::vector<int> v{4}; // POZOR! ini seznam
std::vector<int> v(4); // POZOR! 4 prvky
```

C++11 — auto a decltype

```
auto obj = boost::bind(&fce, _2, _1, neco);
auto i = obj;
for (auto i = C.cbegin(); i != C.cend(); ++i) { }

decltype(a+1) val = a + 1;

// decltype a auto se mohou lišit:
const std::vector<int> v(1);
auto a = v[0];        // a typu int
decltype(v[0]) b = 1; // b typu const int&

auto c = 0;           // c typu int
decltype(c) e;        // e typu int
decltype((c)) f = c;  // f typu int&,  (c) je L-hodnota
decltype(0) g;        // g typu int,    0  je R-hodnota
```

C++11 — cyklus `for`

Cyklus přes zadaný rozsah (*range-based for*)

```
int pole[5] = {1, 2, 3, 4, 5};
for (int &x: pole) {
    x *= 2;
}

for (int i: {1, 2, 5, 10, 20, 50} ) {
    // ....
}
```

Funguje pro pole, inicializační seznamy a všechny rozsahy (např. kontejnery s `begin()` a `end()`)

C++11 — lambda funkce

Funkční objekty

```
[](int x, int y) { return x + y; }
```

návratový typ je `decltype(x+y)`, (omezení na jeden příkaz)

Syntaxe pro explicitní návratový typ

```
[](int x, int y) -> int { int z=x+y; return z*z; }
```

"*Closure*"

```
[]      = bez dalších (lokálních) proměnných
[x,&y]  = x hodnotou, y referencí
[&]     = všechny použité další proměnné odkazem
[=]     = všechny použité další proměnné hodnotou
```

C++11 — lambda funkce

Typické použití: parametr algoritmu

```
vector<int> l;
int sum = 0;
for_each(l.begin(),l.end(), [&sum](int x){sum += x;});
```

V metodě je automaticky `friend`

Typ funkčního objektu zná jen překladač:

```
auto lambda1 = [&](int x) { /*...*/ };
auto lambda2 = new auto([=](int x) { /*...*/ });
```

C++11 — alternativní syntaxe funkcí

Je nutné pro šablony. Například:

```
template< typename L, typename R>
auto afunc(const L &lh, const R &rh)->decltype(lh+rh) {
  return lh + rh;
}
```

Nová syntaxe je použitelná obecně:

```
struct Struct {
    auto fun(int x, int y) -> int;
};

auto Struct::fun(int x, int y) -> int {
    return x + y;
}
```

C++11 — delegace a konstruktory

```
class C  {
    int number;
    int number2;
 public:
    C(int n) : number{n}, number2{} {}
    C() : C(42) {} // delegace, možné od C++11
    C(float f): C(static_cast<int>(f)) {}
};
```

Poznámky:
Dědění konstruktorů, změna v dokončení inicializace objektu

C++11 — nullptr

```
void foo(char *);
void foo(int);

  foo(0);                // int
  foo(nullptr);          // ukazatel (problém v C++98)

char *pc = nullptr;      // OK
int  *pi = nullptr;      // OK
bool   b = nullptr;      // OK, hodnota false
int    i = nullptr;      // CHYBA
```

C++11 — silně typované výčty

```
enum class E {
    Val1, Val2, Val3 = 100, Val4 /* = 101 */
};
```

Nejsou kompatibilní s typem int `E::Val4 == 101` je chyba

Konstanty lze použít pouze s kvalifikací:

```
enum class E2 : int { Val1, Val2 };
E2 e = E2::Val1; // jinak chyba
```

na rozdíl od běžných výčtů:

```
enum E3 : unsigned short { ValA = 65, ValB };
E3 ee = ValA;     // totéž jako E3::ValA
```

C++11 — další změny

- >> v šablonách

```
template<bool B> class TTT;
std::vector<TTT<(1>2)>> x1;  // TTT<false>
```

- `explicit` i pro konverzní operátory
- alias šablony řeší nedostatky `typedef`

```
template<typename T1, typename T2, int v3> class S;
template<typename T2>
using MujAliaS = S<int, T2, 5>;

using MujTyp = void (*)(double);  // nová syntaxe
```

- `long long` převzato z C99

C++11 — šablony s proměnným počtem argumentů

"Variadic templates", použitelné např. pro definice n-tic:

```
template<typename... Values> class tuple;
tuple<int,double> my_pair;
tuple<>           empty_tuple;
```

Je možné definovat i typově bezpečný `printf`:

```
template<typename... Params>
void printf(const std::string &format, Params... par);
```

Počet typových argumentů šablony zjistí `sizeof`:

```
template<typename... Args> struct MyStruct {
    static const int size = sizeof...(Args);
};
static_assert(MyStruct<int,float>::size == 2, "Error");
```

C++11 — řetězcové literály

"string literals" — příklady

```
u8"UTF-8 string"       // typ const char[]
u"UTF-16 string"       //      const char16_t[]
U"UTF-32 string"       //      const char32_t[]
```

Zápis (uni)kódové pozice (jen hexadecimálně):

```
u8"Unicode Characters: \u2021  \U0001F596"
```

"raw string literals" — příklady

```
R"( text včetně \ a " bez nutnosti prefixu \ )"
u8R"XXX( cokoli UTF-8 včetně )" atd. )XXX"
```

C++11 — uživatelské literály

Varianta 1

```
MujTyp operator "" _pripona(const char *s);

MujTyp promenna = 1234_pripona; // parametr  s = "1234"
```

Varianta 2 — šablona

```
template<char...> MujTyp operator "" _pripona();

MujTyp promenna = 1234_pripona;
// použije se operator "" _pripona<'1', '2', '3', '4'>()
```

Příklad použití literálů:

```
hmotnost = 10kg + 1lb +100g + 1hrivna;
```

C++11 — default a delete

```
struct NonCopyable {
  NonCopyable & operator=(const NonCopyable&) = delete;
  NonCopyable(const NonCopyable&) = delete; // zákaz
  NonCopyable() = default;  // vygeneruje překladač
};

struct NoInt {
  void NoInt(int) = delete;     // zákaz konstruktoru
};

struct OnlyDouble {
  int f(double d);
  template<class T> int f(T) = delete; // zákaz ostatních f
};
```

C++11 – static assert

Kontrola při překladu:

```
template<class T>
struct Check  {
  static_assert( sizeof(T) >= sizeof(int),
                 "T is not big enough!");
};
```

Podmínka musí platit, jinak chyba překladu a chybové hlášení obsahuje řetězec.

C++14 = menší změny

```
// GCC option "-std=c++14"  (implicitní od GCC 6)

// dedukce návratového typu
auto f() { int x = 1; return x; }

// šablony proměnných:
template<typename T>
constexpr T pi = T(3.141592653589793);

int main() {
  auto f = [](auto x){ return x; }; // auto lambda param.
  int y = 0b0101110111101001101;    // bin. literály
  int z = 123'456'789;              // oddělovače
}
```

C++17 = střední změny

Pro starší překladač použijte `g++ -std=c++17`

- Zrušení *"trigraphs"*
- Závislost na C11 (ne vše z C11 je podporováno)
- `auto & [ x, y ] = nejaka_dvojice;`
- vyhrazeno `std[0-9]*` pro standardní prostory jmen
- Doplnění std knihovny:
  - `filesystem`
  - `string_view`
  - `variant`, `any`, `optional`
  - `byte`
  - ...

C++17: nové hlavičkové soubory

```
<any>
<charconv>
<execution>
<filesystem>
<memory_resource>
<optional>
<string_view>
<variant>
```

C++20 — přehled novinek

- Korutiny (Coroutines)
- Moduly (Modules)
- Koncepty (Concepts)
- Operátor <=> (3-way comparison operator)
- Klíčová slova `consteval`, `constinit`
- Atributy `[[no_unique_address]]`, `[[likely]]`, `[[unlikely]]`
- Makra pro test schopností překladače (*Feature test macros*)
- Inicializátory `.složka=hodnota` (*designated initializers*)
- Doplněn `char8_t`
- Formát znaménkových celých čísel je dvojkový doplněk
- Nevyžaduje `typename` v některých kontextech, lepší `constexpr`, zjednodušení šablon funkcí (`auto`), ...

C++20: nové hlavičkové soubory

```
<concepts>
<coroutine>
<compare>
<version>
<source_location>
<format>
<span>
<ranges>
<bit>
<numbers>
<syncstream>

<stop_token>
<semaphore>
<latch>
<barrier>
```

C++20 — příklady

viz WWW

C++23 — přehled novinek

- Modulární standardní knihovna: `import std;`
- Operátor indexování s více parametry: `p[i,j,k]`
- Podpora pro korutiny: `std::generator`
- Formátovaný výstup: `std::print`, `std::println`
- `std::mdspan`
- `if consteval`
- Zápis znaků: `\u{999}`, `\o{77}`, `\x{FF}`, `\N{Greek Small Letter Mu}`
- ...

C++23 — příklady

viz WWW

Závěr

- Další evoluce jazyka (C++26)
- Nové části std knihovny (sítě, grafika?, ”stacktrace”, ...)
  inspirace viz projekt Boost `http://www.boost.org/`
- Implementace překladačů a std knihoven (GCC, LLVM/clang, ...)
- Implementace knihoven pro GUI, hry, matematiku, multimédia, ...
- Budoucnost C++