

1. Úvod do programování v C++

B2B99PPC – Praktické programování v C/C++

Stanislav Vítek

Katedra radioelektroniky
Fakulta elektrotechnická
České vysoké učení v Praze

Přehled témat

- Část 1 – Úvod do programování v C++

Neobjektové vlastnosti C++

Vstup a výstup

Třídy

Objekty

Práce se soubory

Část I

Úvod do programování v C++

O C++

- autorem je Bjarne Stroustrup z Bellových Laboratoří
- původně znám jako **C with Classes**
- aktuální specifikace jazyka ISO/IEC 14882:2017(E)
- imperativní, staticky typovaný
- objektově orientovaný, s funkcionálními prvky
- generické programování a metaprogramování (šablony)
- udržuje efektivitu jazyka C, částečná zpětná kompatibilita
- aplikační domény:
 - systémové i aplikační programování,
 - ovladače zařízení,
 - embedded software,
 - výkonné serverové a klientské aplikace,
 - videohry a zábavní průmysl,
 - nativní kód aplikací pro Android.

na C++ přejmenován v roce 1983

neformálně známa jako C++17, další verze v roce 2020

Programovací paradigmatata

Procedurální programování

- Program popisuje krok z krokem, jak dospět k řešení dané úlohy
- Hodí se spíše pro řešení procesních problémů

Objektové programování

- Data a metody sloužící k manipulaci s těmito daty uloženy v jednotkách zvaných **objekty**
 - Objekt jsou vytvořen podle předpisu, kterému říkáme **třída**
 - Třídy mohou vzájemně **dědit** svoje vlastnosti
- K datům lze přistupovat přes **metody** objektu
 - Tzv. **zapouštění**, někdy je efektivnější se mu vyhnout
 - Aby nemusela každá třída implementovat všechny funkce, je zaveden **polymorfismus**
- Vhodný pro řešení různých informačních systémů, které uvažujeme jako síť komunikujících objektů

I. Úvod do programování v C++

Neobjektové vlastnosti C++

Vstup a výstup

Třídy

Objekty

Práce se soubory

První program v C++

```
1 // C
2 #include <stdio.h>
4 int main () {
5     printf("Ahoj PPC!\n");
6     return 0;
7 }
```

lec03/00-hello.c

```
1 // C++
2 #include <iostream>
4 int main () {
5     std::cout << "Ahoj PPC!\n";
6     return 0;
7 }
```

lec03/00-hello.cpp

Datové typy

- Základní datové typy jsou stejné, jako v **C**
- `char`, `signed char` a `unsigned char` jsou považovány za rozdílné datové typy
`std::is_same<char, signed char>::value`
- Podpora vícebajtových kódování
 - `char16_t` – UTF-16, literály s prefixem `u'a'`
 - `char32_t` – UTF-32, literály s prefixem `U'a'`
 - `wchar_t` – implementačně závislá velikost, prefix `L'a'`
- Primitivní datový typ `bool` pro logické hodnoty
`V C lze použít stdbool.h nebo _Bool (C99)`
 - Literály typu `bool` jsou pouze dva: `true` (= 1) a `false` (= 0)
 - Pro další operace je typ `bool` kompatibilní s celočíselnými typy
- Standardní knihovna zavádí řadu dalších datových typů
 - `std::array`, `std::vector` – náhrada pole
 - `std::string` – textové řetězce

Těmto datovým typům se budeme věnovat v následujících přednáškách.

Řetězce v C++ – `std::string` (<string>)

- sám si alokuje a dealokuje paměť podle potřeby, indexování funguje jako v C

```
1 | std::string s = "Hello!";  
2 | s[0] = 'J';  
3 | std::cout << s;
```

- navíc se umí přiřazovat

```
1 | std::string s = "Hello!";  
2 | std::string t;  
3 | t = s;
```

- umí se řetězit pomocí operátoru `+` (a operátoru `+=`)

```
1 | std::string h = "Hello";  
2 | std::string w = "world";  
3 | std::string s = h + ", " + "!";
```

- dynamické (rozšiřitelné) pole

```
1  std::vector<int> v;  
2  v.push_back(1); // vložení prvku za konec pole  
3  v.push_back(2);  
4  v.push_back(3);  
  
6  for (int i = 0; i < v.size(); ++i) {  
7      std::cout << v[i];  
8  }  
  
10 for (int x : v) { // od C++11: range-for  
11     std::cout << x;  
12 }
```

- Inicializace

```
1 | std::vector<int> a = {1, 2, 7, 17, 42}; // od C++11
2 | std::vector<int> b{1, 2, 7, 17, 42}; // totéž
3 | std::vector<int> c(10);
4 | std::vector<int> d(10, 17);
```

```
// c: 0 0 0 0 0 0 0 0 0 0
// d: 17 17 17 17 17 17 17 17 17 17
```

- pozor na rozdíl mezi `()` a `{}`
- pozor na rozdíl mezi `resize` a `reserve`
- od C++17 si typ objektů *uvnitř* vektoru umí kompilátor v některých případech domyslet:

```
1 | std::vector v = {1, 2, 3, 4, 5};
```

Automatická dedukce typů – auto

- Klíčové slovo `auto` může nahrazovat jméno datového typu
- V určitých situacích tedy musí kompilátor datový typ odhadnout

```
1 | auto i = 3;      /* i je typu int    */
2 | auto j = i;     /* j je typu int    */
3 | auto x = 3.14; /* x je typu double */
```

- velmi užitečné v generickém programování, kde někdy není jednoduché datový typ určit, může zpřehlednit kód

Deklarace proměnných

```
1 | auto i {0}; /* -> int i = 0; */
```

- `{}` může zabránit nechtěným typovým konverzím

```
1 | int i = 1.1; /* -> funguje  */
2 | int i {1.1}; /* -> nefunguje */
```

Volání funkcí

```
1 void f1(std::string s) {
2     // s je kopie skutečného argumentu
3     // změny s se navenek nijak neprojeví
4 }
5
6 void f2(std::string& s) {
7     // s je reference na skutečný argument
8     // změny s se navenek projeví
9 }
10
11 void f3(const std::string& s) {
12     // s je reference na skutečný argument
13     // je zakázáno s měnit
14 }
```

Reference

- Reference v C++ umožňuje vytvořit odkaz na již existující proměnnou:
 - odkaz má všechny vlastnosti původní proměnné,
 - reference musí být při vytvoření inicializovaná proměnnou, na kterou odkazuje,
 - odkazovanou proměnnou nelze po dobu existence reference změnit,
 - reference se nejčastěji vytváří a inicializuje při volání funkce (parametry funkce v ukázce na následujícím slide).

Příklad

```
1  int a = 1;
2  int &b = a, &c = b;
3  int *d = &a;    // mozne, ale neni bezpecne
5  b = 5;         // a == 5
6  c = 10;        // a == 10
7  *d = 15;       // a == 15
```

Reference – předávání parametrů funkci

```
1 void swapC (int *px, int *py) { // C
2     int tmp = *px;
3     *px = *py;
4     *py = tmp;
5 }
7 void swapCPP (int& x, int& y) { // C++
8     int tmp = x;
9     x = y;
10    y = tmp;
11 }
12 // ...
13 int a, b;
14 swapC (&a, &b);
15 swapCPP (a, b);
```

Range-for

```
1  std::vector<std::string> names;
2  // ...
3  for (std::string s : names) {
4      // s je kopie položky vektoru
5  }
7  for (std::string& s : names) {
8      // s se odkazuje na položku vektoru
9      // můžeme jej měnit
10 }
12 for (const std::string& s : names) {
13     // s se odkazuje na položku vektoru
14     // nesmíme jej měnit
15 }
```


- U velkých projektů může docházet ke kolizím identifikátorů
- C++ umožňuje třídit identifikátory do jmenných prostorů

```
1 int main () {
2     int value = 10;
3     // ..
4     int value = 20;
5 }
```

```
$ g++ main.c
error: redeclaration of
'int value'
```

```
1 namespace A {
2     int value;
3 };
4
5 namespace B {
6     int value;
7 }
8
9 int main () {
10     A::value = 10;
11     B::value = 20;
12 }
13
```

- Kvalifikátor umožňuje přístup ke globálnímu jmennému prostoru

```
1 | ::global = 3;
```

- Funkce standardní knihovny jsou ve jmenném prostoru `std`
- Jmenné prostory mohou být i vnořené
- Jmenným prostorem je i třída
 - Třída může být ve jmenném prostoru
- Pro zkrácení zápisu je možné využít direktivu `using`
 - Využívání direktivy se považuje za nevhodnou techniku a je lépe funkce plně kvalifikovat

Příklad

```
1 | #include <iostream>
3 | using namespace std;
5 | int main () {
6 |     cout << "Ahoj PPC!" << endl;
7 | }
```

- Počet a typy parametrů mohou být využity pro odlišení funkcí stejného jména.

Příklad

```
1  int cube (int x) {
2      return x * x * x;
3  }
4
5  double cube (double x) {
6      return x * x * x;
7  }
8
9  // ..
10 int a = 2;
11 float b = 3.14;
12
13 std::cout << a << "^3 = " << cube(a) << std::endl;
14 std::cout << b << "^3 = " << cube(b) << std::endl;
```

- Při volání přetížené funkce se kompilátor rozhoduje podle nejlepší shody parametrů.
- Porovnání parametrů má čtyři úrovně:
 - **přesná shoda** – typy skutečných a formálních parametrů jsou stejné,
 - **roztahení (promotion)** – zachová rozsah i přesnost:
char → int, enum → int, enum → int, float → double
 - **standardní konverze** – přesnost či rozsah mohou být ztraceny:
int → double, double → int, unsigned → int, int → long, ...
 - **uživatelská konverze** – konverze zavedená uživatelem definovaným konstruktorem nebo přetíženým operátorem přetypování (cast).

Přetěžování funkcí

- Pro výběr přetížené funkce se porovnávají všichni kandidáti:
 - kandidáty jsou všechny funkce daného jména volatelné s daným počtem parametrů.
- Vítězná funkce musí mít porovnávací kategorii stejnou nebo lepší, než ostatní kandidáti. To musí platit pro všechny parametry.
- Pokud neexistuje právě jeden vítěz (funkce s nejlepší shodou ve všech parametrech), porovnávací algoritmus ohlásí chybu.
- Takto nastavená pravidla jsou striktní (vítěz musí mít nejlepší konverzi ve všech parametrech), přesto dokáží překvapit.
- Je rozumné se vyhnout nadměrnému přetěžování funkcí.

- V deklaraci funkce může být uvedena implicitní hodnota parametru.
- Odpovídající parametr může být při volání funkce vynechán.
- Protože v C/C++ jsou poziční parametry, lze implicitní parametry deklarovat jen "na konci" seznamu parametrů.

Příklad

```
1 void print (int x, int y = 0, int z = 0) {
2     std::cout << "x=" << x << "\n";
3     std::cout << "y=" << y << "\n";
4     std::cout << "z=" << z << std::endl;
5 }
6 // ..
7 print (10, 20, 30); // 10, 20, 30
8 print (10, 20);    // 10, 20, 0
9 print (10);        // 10, 0, 0
```

- Nedovolené použití implicitních parametrů:

```
1 | void f (int x = 1, int y); // error
```

- Kombinace přetížení a implicitních parametrů

```
1 | void g (int x, int y = 10);
```

```
2 | void g (int x);
```

- Přetížení funkce povoleno, ale ne tímto způsobem
- Neexistuje způsob, jak zavolat druhou funkci

```
1 | g (20); // viceznacne
```

```
2 | g (10, 40); // ok
```

Struktury

- V C++ je identifikátor struktury zároveň jménem typu (třídy), takže není třeba používat `typedef` nebo doplňovat `struct`

```
1 // C -- pojmenovaná struktura
2 struct List {int val; struct List *next;};
3 struct List *head;
4
5 // C -- nový datový typ
6 typedef struct List {
7     int val; struct List *next;
8 } LIST;
9 LIST *head;
10
11 // C++
12 struct LIST {int val; LIST *next;};
13 LIST *head;
```


Přetížení operátorů

- C++ zavádí klíčové slovo `operator`, který umožňuje definovat funkcionalitu operátoru, který následuje za klíčovým slovem
- Přetížit lze řadu operátorů, nelze měnit význam operátorů vestavěných typů

Více o přetížení operátorů dále v semestru

Příklad

```
1  struct cplx {float im; float re;}; // vlastní typ
3  cplx operator+(cplx &a, cplx &b) { // přetížení +
4      cplx tmp;
5      tmp.re = a.re + b.re; tmp.im = a.im + b.im;
6      return tmp;
7  }
9  cplx c, d, e;
10 e = c + d; // zkrácené použití operátoru
11 e = operator+(c, d); // funkční použití operátoru
```

- Dynamickou alokaci provádí operátor `new`:
 - výsledek operace `new` má správný typ, nemusí se přetypovávat (`cast`),
 - velikost je dána v počtu prvků (nikoli v bajtech),
 - pro objektové datové typy volá operátor `new` s konstruktorem.
- Paměť alokovaná použitím operátoru `new` musí být uvolněna pomocí operátoru `delete`.
- Nelze mixovat C a C++ alokaci a uvolňování paměti:
 - blok alokovaný použitím `malloc` musí být uvolněn použitím `free`,
 - objekt alokovaný použitím `new` musí být uvolněn použitím `delete`,
 - pole alokované použitím `new []` musí být uvolněno použitím `delete []`.

```
1  int *p = new int; // alokuje proměnnou typu int
3  struct S {
4      int a;
5      char b;
6  };
8  S *q = new S; // alokuje strukturu typu S
9              // C++ nevyžaduje klíčové slovo struct
10 int *a = new int[1000]; // alokuje pole
12 delete p;     // uvolňuje jednoduchou proměnnou
13 delete q;
14 delete [] a;  // uvolňuje pole, bez [] je to chybně
16 a = p + 1;
17 delete a;     // chybně, uvolnit lze jen to,
18              // co bylo vytvořeno pomocí new
```

I. Úvod do programování v C++

Neobjektové vlastnosti C++

Vstup a výstup

Třídy

Objekty

Práce se soubory

Vstup a výstup v C

```
1  #include <stdio.h>
3  int main ( void )
4  {
5      int x;
6      printf ("Napis cislo:\n");
7      scanf ("%d", &x);
8      printf ("Vstup byl: %d\n", x);
9      return 0;
10 }
```

Co se stane při změně deklarace `x` na `float`?

Vstup a výstup v C++

```
1  #include <iostream>
3  int main ( )
4  {
5      int x;
6      std::cout << "Napis cislo: ";
7      std::cin >> x;
8      std::cout << "Vstup byl: " << x << std::endl;
9      return 0;
10 }
```

Co se stane při změně deklarace `x` na `float`?

Vstup a výstup v C++

- Formátovaný vstup/výstup v C spoléhá na správný formátovací řetězec.

Každá neshoda konverze ve formátovacím řetězci s typem parametru může způsobit chybu (pád programu).

- Proudý v C++ jsou bezpečné, neboť není třeba žádný formátovací řetězec, způsob konverze je vybrán kompilátorem podle typu parametru.
- Proudý v C++ mohou být snadno modifikovány:
 - vstup/výstup nových (uživatelských) datových typů,
 - různé zdroje/cíle proudů (soubory, buffery v paměti, sockety, ...)
- Standardní proudý, deklarovány v `<iostream>`:
 - `std::cout` – standardní výstup (stdout, bufferovaný)
 - `std::cerr` – standardní chybový výstup (stderr, nebufferovaný)
 - `std::clog` – standardní chybový výstup (stderr, bufferovaný).
 - `std::cin` – standardní vstup (stdin)

Výstupní manipulátory a funkce

- Řídí formátování výstupu, deklarovány v `<iomanip>`:
 - `endl` – nový řádek + flush,
 - `flush()` – synchronizace bufferu proudu s fyzickým výstupem,
 - `setw(x)` – šířka výstupního pole,
 - `setfill(c)` – výplňkový znak,
 - `right/left` – zarovnání doprava / doleva,
 - `setprecision(x)` – počet desetinných míst,
 - `fixed/scientific` – formát bez / s exponentem (semilog),
 - `hex/oct/dec` – základ číselné soustavy 16, 8, 10,

Příklad

```
1  int x = 10;
2  std::cout << "dekadicky " << x << std::endl;
3  std::cout << "sirka 10 znaku " << setw(10) << x << std::endl;
4  std::cout << "sestnactkove " << hex << x << std::endl;
5  std::cout << "opet dekadicky " << dec << x << std::endl;
```


Vstupní manipulátory a funkce

- Řídí formátování vstupu:
 - `ignore()` – vyprázdní vstupní buffer
 - `ws` – extrahuje bílé znaky
 - `hex/oct/dec` – základ číselné soustavy 16, 8, 10
 - `skipws/noskipws` – přeskokování bílých znaků při dalších operacích
 - `boolalpha/noboolalpha` – vstup true, false / 1, 0
 - `setw(n)` – omezení délky načítaného řetězce

Příklad

```
1  #include <iostream>
2  #include <iomanip>
4  int main() {
5      std::string text;
6      std::cin >> std::setw(3) >> text;
7      std::cout << text << std::endl;
8  }
```

`ostream & put()`

- vloží jeden znak do výstupního proudu
- vrací referenci na proud, takže může být řetězena

```
1 | std::cout.put('A');  
2 | std::cout.put('A').put('p').put('p').put('\n');
```

`int get()`

- přečte ze vstupního proudu znak a vrátí ho jako `int`

```
1 | int i;  
2 | while ((std::cin.get()) != EOF)  
3 |     std::cout.put(i);
```

`istream & get(char& c)`

- přečte znak, uloží ho jako `c` a vrátí referenci na vstupní proud

```
istream & get(char* c, streamsize n, char delim='\n')
```

- přečte `n-1` znaků (nebo znaky před oddělovačem) a uloží je do `c`
- uložený řetězec je ukončen terminátorem `\0`
- oddělovací znak `delim` zůstává ve vstupním proudu

```
istream & getline(char* c, streamsize n, char d='\n')
```

- stejná funkce jako `get`, nenechává oddělovač v proudu

```
1 | char * a;  
2 | while (std::cin.getline(a, 10, ' '))  
3 |     std::cout << a << std::endl;
```

```
int getline(istream& is, string& str, char delim='\n')
```

```
1 | std::string T;  
2 | while (getline(std::cin, T, ' '))  
3 |     std::cout << T << std::endl;
```

```
istream & read (char * c, streamsize n);
```

- Přečte `n` znaků (bytů) ze standardního vstupu a uloží je do pole
- v poli nedoplňuje terminační znak

```
streamsize gcount() const;
```

- Vrací počet znaků načtených posledním voláním některé z funkcí `get()`, `getline()`, `ignore()` nebo `read()`

```
ostream & write (const char * c, streamsize n);
```

- Zapiše do výstupního proudu `n` znaků (bytů) z pole `c`

```
char peek ();
```

- Vrací příští znak ve vstupním bufferu bez jeho načtení

```
istream & putback (char c);
```

- Zapiše znak `c` do vstupního proudu

Stavové bity I/O proudů

- I/O proudy informují o svém stavu (tj. chybách) pomocí stavových bitů (příznaků)
 - **ios::goodbit** – v pořádku
 - **ios::badbit** – vážná chyba (např. se nepodařilo otevřít soubor)
 - **ios::failbit** – méně závažná chyba (např. chyba při konverzi)
 - **ios::eofbit** – dosažení konce souboru
- Program může pracovat přímo s hodnotami bitů, nebo využít funkcí proudu, které jsou součástí standardní knihovny
 - **rdstate()** – stav všech bitů
 - ```
if (is.rdstate() & (ios::badbit||ios::failbit)) ...
```
  - **good()**, **bad()**, **fail()**, **eof()**
  - **clear()** – nulování stavového bitu
- Kromě funkcí standardní knihovny lze využít také systém vyjímek

# Stavové bity I/O proudů

---

```
1 double readDouble() {
2 double d;
3 std::cin >> d;
4
5 if (std::cin.good()) {
6 return d;
7 }
8 else if (std::cin.bad() || std::cin.eof()) {
9 throw std::runtime_error("readDouble() failed");
10 }
11 else {
12 std::cin.clear();
13 std::cin.ignore(1, '\n');
14 return readDouble();
15 }
16 }
```

# I. Úvod do programování v C++

---

Neobjektové vlastnosti C++

Vstup a výstup

Třídy

Objekty

Práce se soubory

# Co je to třída?

---

- Třída je abstrakcí entity reálného světa.
- Příklad: třída Automobil:
  - všechny automobily mají nějaké společné vlastnosti
    - obsah motoru,
    - barvu,
    - ...
  - všechny automobily mají nějaké společné rozhraní (interface)
    - lze je nastartovat,
    - lze je rozjet nebo zastavit,
    - ...
- Objekt je tzv. instance třídy.
- Příklady instance třídy Automobil:
  - Škoda Octavia,
  - VW Passat,
  - ...



# Pohled programátora

---

- Třída je popisem datového typu:
  - jméno,
  - data – členské proměnné (položky, atributy),
  - interface – členské funkce (metody).
- Třídy jsou vyvíjeny programátory a jsou kompilovány do spustitelného programu.
- V C++ nelze vytvářet nové třídy za běhu.
- Objekty (instance tříd) jsou proměnné:
  - každý objekt má třídu,
  - v průběhu běhu programu jsou objekty vytvářeny a rušeny,
  - obvykle vytváříme více objektů/instancí stejné třídy,
  - stejně jako jiné datové typy, C++ povoluje staticky alokované objekty, dynamicky alokované objekty, pole objektů, ...

# Deklarace třídy

---

```
1 class T
2 {
3 typ a; // atribut (členská proměnná)
4 typ f (...); // metoda (členská funkce)
5 T (...); // konstruktor
6 ~T (void); // destruktorka
7 };
```

- Definice konstruktoru:

```
T::T (...) { ... }
```

- Definice destruktorky

```
T::~~T (void) { ... }
```

- Definice metody

```
typ T::f (...) { ... }
```

# Zapouzdření

---

- Pro úplnou ochranu atributů objektu je potřeba zabránit jejich modifikaci jinak než přes příslušné metody.
- Atributy a metody třídy jsou vždy přístupné z metod definovaných v této třídě, z jiných metod a funkcí však již přístupné být nemusí.
- Zapouzdření je zároveň prostředek, jak vytvořit a udržovat kontrolovatelné a vysokoúrovňové veřejné rozhraní třídy.
- Zapouzdření umožňuje:
  - nezávisle upravovat implementaci uvnitř třídy (např. zvolit efektivnější algoritmus, jinou reprezentaci dat, ...),
  - kompletně nahradit třídu bez rozbití zbytku programu (pokud nová třída dodrží veřejné rozhraní),
  - opravit chyby uvnitř třídy bez rozbití zbytku programu,
  - pracovat na vývoji tříd nezávisle (např. různými programátory zároveň).

# Řízení přístupu

---

- Přístup k atributům a metodám je řízen pomocí **modifikátorů viditelnosti**:
  - `public` – jsou přístupné komukoli,
  - `protected` – jsou přístupné v třídě samé a v jejích podtřídách,
  - `private` – jsou přístupná jen v třídě samé.

```
1 class T
2 {
3 // metody/atributy s implicitním přístupem
4 // (zde private)
5 public:
6 // metody/atributy přístupné komukoli
7 private:
8 // metody/atributy přístupné jen ve třídě
9 };
```

# Klíčová slova `class` a `struct`

---

- Obě klíčová slova mohou být použita k deklaraci třídy
- Jediný rozdíl je v implicitní viditelnosti:

class: `private`

struct: `public`

```
1 | class T { // impl. private
2 | int a;
3 | public:
4 | void f ();
5 | };
```

```
1 | struct T {
2 | private:
3 | int a;
4 | public:
5 | void f ();
6 | };
```

---

```
1 | class T {
2 | public:
3 | void f ();
4 | private:
5 | int a;
6 | };
```

---

```
1 | struct T { // impl. public
2 | void f ();
3 | private:
4 | int a;
5 | };
```

# I. Úvod do programování v C++

---

Neobjektové vlastnosti C++

Vstup a výstup

Třídy

Objekty

Práce se soubory

# Vytváření objektů

---

- Objekt je proměnná (instance třídy)
- Při vytvoření objektu je zavolán konstruktor
- Konstruktory mohou být přetíženy
- Konstruktor nelze volat explicitně na existující objekt
- Destruktor je volán automaticky, když je objekt rušen

## Příklad

```
1 int main() {
2 T x (a, b); // konstruktor s parametry (a, b)
3 T y; // konstruktor bez parametrů
4 // ..
5 }
```

# Přístup k atributům a metodám

---

- Metody jsou volány prostřednictvím tečkové notace
- Metody mohou být přetíženy. Platí pravidla pro přetěžování funkcí (nejlepší shoda skutečných a formálních parametrů)
- Přístup k atributům je tečkovou notací (jako struct).

## Příklad

```
1 class T {
2 public:
3 void foo () { ... } // metoda
4 int bar; // atribut
5 };
6 // ..
7 T x;
8 x.foo ();
9 x.bar = 10;
```



# Dynamická alokace

- Objekty mohou být alokovány dynamicky – užitím operátoru `new`

Operátor volá odpovídající konstruktor

- Dynamicky alokované objekty je rušeny použitím `delete`

Operátor volá destruktork

- C alokace (`malloc/free`) nemůže být použita

Tyto funkce by nezavolaly konstruktor a destruktork

## Příklad

```
1 struct T
2 {
3 int a;
4 T (int x) {a = x;}
5 ~T ();
6 void f () {cout << a;}
7 };
```

```
1 int main()
2 {
3 T *p = new T(20);
4 p->f();
5 p->a = 10;
6 p->f();
7 delete p;
8 }
```

## Klíčové slovo `this`

---

- Lokální deklarace mohou být v konfliktu se jmény položek.
- Přístup k položkám lze zařídit pomocí klíčového slova `this` nebo plně kvalifikovaným jménem.
- Lepší je vyhnout se konfliktu jmen (např. prefixem).

### Příklad

```
1 struct T {
2 int a;
3 void f (int a);
4 };
5
6 void T::f (int a) { // konflikt se jménem atributu a
7 T::a = a; // T::a - plně kvalifikovaný atribut
8 // a - jméno parametru metody
9 this->a = 10; // this - ukazatel na instanci typu T*
10 }
```

# I. Úvod do programování v C++

---

Neobjektové vlastnosti C++

Vstup a výstup

Třídy

Objekty

Práce se soubory

# Rozhraní pro práci se soubory

---

- C++ umožňuje používat dvě různá API pro práci se soubory
  1. založené na principech z jazyka **C**, definované v `<cstdio>`
  2. čisté C++ rozhraní v knihovně `<fstream>`
- Třída `fstream` slouží pro obousměrnou práci se soubory
  - je podmnožinou třídy `iostream`
- Pro zápis do souboru slouží třída `ofstream`
  - je podmnožinou třídy `ostream`
- Pro čtení ze souboru slouží třída `ifstream`
  - je podmnožinou třídy `istream`
- Pro práci se souborem je třeba provést následující kroky
  1. vytvořit instanci vhodné třídy
  2. připojit se k souboru
  3. provést I/O operaci – `>>`, `get()`, `read()`, ...
  4. uzavřít proud

# Funkce pro otevření a uzavření souboru

---

```
void open (const char* filename, ios::openmode mode);
```

- pracuje s **C** řetězci, při použití `std::string` je třeba použít funkci `c_str()`
- 

```
void close ();
```

- Uzavře soubor, vyprázdní buffer a odpojí proud
- 

```
1 #include <fstream>
2 //..
3 ofstream fout;
4 fout.open(filename, mode);
5 //..
6 fout.close();
7 // nebo lze kombinovat deklaraci a open()
8 ofstream fout(filename, mode);
```

# Módy otevření souboru

---

- Módy jsou definovány v `ios_base`, jsou referencovatelné i z jejích podtříd, jako je např. `ios`
  - `ios::in` – otevření souboru pro zápis
  - `ios::out` – otevření souboru pro čtení
  - `ios::app` – zápis bude proveden na konec souboru
  - `ios::trunc` – vyprázdnit obsah souboru
  - `ios::binary` – binární operace
  - `ios::ate` – nastavení ukazatele na konec souboru
- Módy lze kombinovat pomocí operace logického součinu
  - Default pro zápis: `ios::out | ios::trunc`
  - Zápis na konec existujících dat: `ios::out | ios::app`

# Soubory s náhodným přístupem

---

```
istream & seekg (streampos pos);
ostream & seekp (streampos pos);
```

- Nastaví pozici v souboru

---

```
istream & seekg (streamoff offset, ios::seekdir way);
ostream & seekp (streamoff offset, ios::seekdir way);
```

- Nastaví pozici v souboru relativně k `seekdir`
- `ios::beg` (beginning), `ios::cur` (current), `ios::end` (end)

---

```
streampos tellg ();
streampos tellp ();
```

- Vrací pozici v souboru