# 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 – O předmětu
  - Organizace předmětu
  - Studijní výsledky
- Část 2 – Úvod do programování v C++
  - Neobjektové vlastnosti C++
  - Vstup a výstup
  - Práce se soubory
  - Výjimky

# Část I

## O předmětu

# I. O předmětu

# Předmět a lidé

- Webové stránky předmětu
  https://cw.fel.cvut.cz/wiki/courses/b2b99ppc
- Přednášející
  - Ing. Stanislav Vítek, Ph.D.
    http://mmtg.fel.cvut.cz/personal/vitek/
- Cvičící
  - Ing. Stanislav Vítek, Ph.D.
  - Ing. Ondřej Nentvich
  - Ing. Václav Navrátil, Ph.D.

# Cíle předmětu

- Motivace k programování
  - Programování je klíčová dovednost, která může hrát rozhodující roli na trhu práce
  - Velká většina studentů FEL programování využije během studia
- Aplikace získaných znalostí v praktických úlohách
  - Komunikace s embedded zařízením
  - Kommunikace s webovou službou
  - Desktopová aplikace s GUI
- Další zkušenosti s programováním
  - Programovací jazyk C++
  - Knihovna QT
  - Povědomí o objektovém programování

# Organizace a hodnocení předmětu

- B2B99PPC – Praktické programování v C/C++
- Rozsah: 2p+2c; Zakončení: KZ; Kredity: 6;

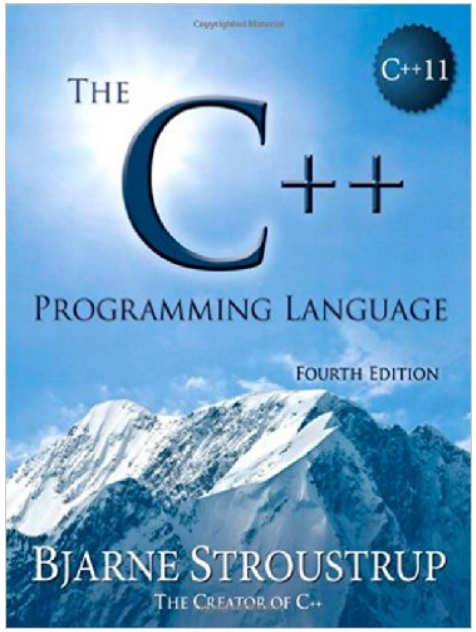
---

- Průběžná práce v semestru – domácí úkoly a test
- Semestrální práce
- Započtový test

---

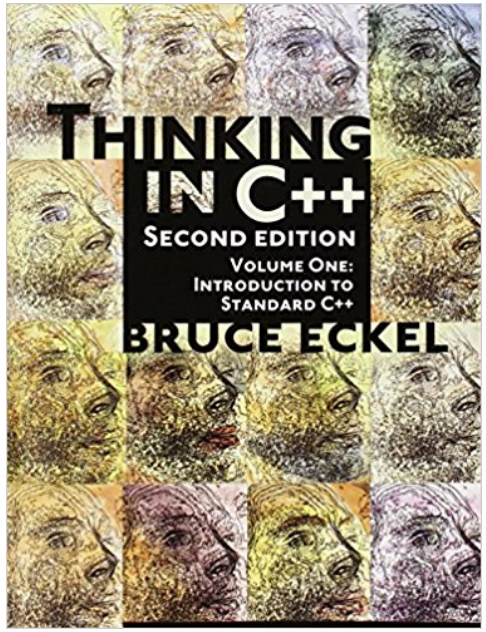
- Docházka na cvičení
  - Cvičení jsou povinná – možné dvě omluvené absence
  - Na cvičení je třeba se připravit, nejlépe návštěvou přednášky a studiem podkladů (příklady, doporučená literatura)

# Zdroje a literatura



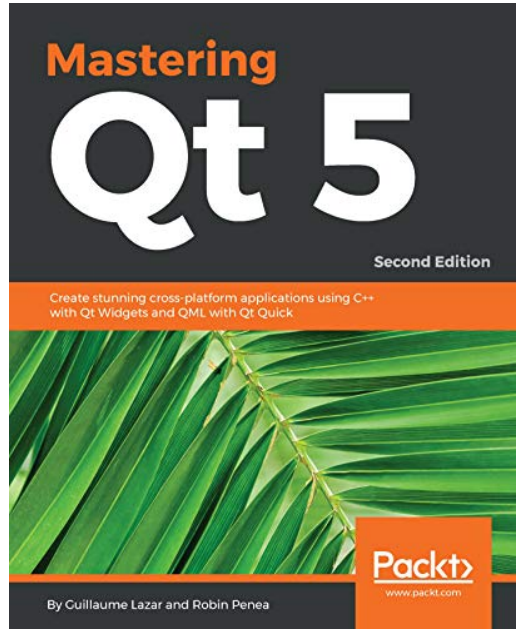

Bjarne Stroustrup
The C++ Programming Language
Addison-Wesley
2014

ISBN 978-0321563842




Bruce Eckel
Thinking in C++
Prentice Hall
2014

ISBN 978-0139798092




G. Lazar, R. Penea
Mastering QT 5
Packt Publishing
2016

ISBN 978-1788995399

# Řešení problémů

- Obracejte se na svého cvičícího
- Pokud komunikujete elektronickou poštou (e-mail)
  - Pište vždy ze své fakultní adresy
  - Do předmětu zprávy uvádějte zkratku predmětu PPC
  - V případe zásadních problémů (napr. týkajících se zápočtu) uvádějte do Cc též přednášejícího

# I. O předmětu

# Přehled úkolů na cvičení

- Domácí úkoly
  1. HW01 – (5b) Vstup a výstup v C++
  2. HW02 – (8b) Třídy a objekty
  3. HW03 – (8b) Standardní šablony
  4. HW04 – (8b) Přetěžování operátorů
  5. HW05 – (8b) Program v MBED – generátor
  6. HW06 – (8b) Program v QT – osciloskop
- Semestrální práce
  - Ucelený program, který vhodným způsobem zapadá do kontextu studia nebo navazuje na samostatnou tvůrčí práci studentů.
- Celkem lze získat
  - za domácí úlohy **45b**,
  - za semestrální práci **25b**.

# Hodnocení

| Zdroj bodů | Maximum | Nutné minimum |
|---|---|---|
| Domácí úkoly | 45 | 25 |
| Semestrální práce | 25 | 10 |
| Závěrečný test | 35 | 15 |
| Součet | 105 | |

- Za práci v semestru je třeba získat nejméně 35 bodů
- Za domácí úkoly je možné získat body nejpozději do 24.5.2020
- Semestrální práce – možno vypracovat během prázdnin
- Klasifikace – standarní stupnice
  - A – $\geq 90$
  - B – 80 – 89
  - C – 70 – 79
  - D – 60 – 69
  - E – 50 – 59
  - F – $< 50$

# Přehled přednášek

| | | | |
|---|---|---|---|
| 1. | Informace o předmětu, úvod do programování v C++ | 18.2. | |
| 2. | Objektově orientované programování v C++ | 25.2. | HW01 |
| 3. | Knihovna standardních šablon STL | 3.3. | HW02 |
| 4. | Šablony funkcí a tříd v C++ | 10.3. | HW03 |
| 5. | STL algoritmy | 17.3. | HW04 |
| 6. | Embedded programování – MBED | 24.3. | HW05 |
| 7. | QT 1. – úvod do frameworku, komponenty, stylování | 31.3. | HW06 |
| 8. | QT 2. – event driven programování | 7.4. | |
| 9. | QT 3. – síťové služby, databáze | 14.4. | |
| 10. | Přesná aritmetika (doc. Dobeš) | 21.4. | |
| 11. | Komunikace mezi procesy | 28.4. | |
| 12. | Páteční výuka | 5.5. | |
| 13. | C++11, C++17 – co je nového | 12.5. | |
| 14. | Zápočtový test | 19.5. | |

# Část II

# Úvod do programování v C++

# O C++

- autorem je Bjarne Stroustrup z Bellových Laboratoří
- původně znám jako **C with Classes**

  na **C++** přejmenován v roce 1983
- aktuální specifikace jazyka ISO/IEC 14882:2017(E)

  neformálně známa jako **C++17**, další verze v roce 2020
- 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 applikace,
  - videohry a zábavní průmysl,
  - nativní kód aplikací pro Android.

# Programovací paradigmata

## 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. **zapoudření**, 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

# II. Úvod do programování v C++

# První program v C++

```c
// C
#include <stdio.h>

int main () {
  printf("Ahoj PPC!\n");
  return 0;
}
```

lec01/00-hello.c

```cpp
// C++
#include <iostream>

int main () {
  std::cout << "Ahoj PPC!\n";
  return 0;
}
```

lec01/00-hello.cpp

# Klíčová slova

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

# 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.

# Klíčové slovo `auto`

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

```
auto i = 3;      // i je typu int
auto j = i;      // j je typu int
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

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

- {} může zabránit nechtěným typovým konverzím

```
int i = 1.1;     // -> funguje
int i {1.1};     // -> nefunguje
```

# Reference

- Práce s ukazateli v C může být nepohodlná
  - komplikovaný zápis
  - potenciální nebezpečí přepisu paměti

Lze řešit pomocí konstatních ukazatelů.

- Ukazatel má v C programech různé role:
  - dynamická alokace paměti,
  - realizace výstupních a vstupně-výstupních parametrů funkce,
  - předávání vstupního parametru bez jeho kopírování.

Pole a velké struktury.

## Příklad

```
void readResize (int ** data, int * nr, int * max)
{
  *data = (int*) realloc (*data, *max * sizeof (**data));
  (*data)[(*nr)++] = x; // !!!
}
```

C programátor pomocí ukazatelů říká, jak se má kód přeložit do strojového kódu. Zápis ale neříká, co má ukazatel za roli.

# Reference

- Reference v C++ umožnuje 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

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

lec01/01-reference.cpp

# Reference – předávání parametrů funkci

```
// C
void swapC (int *px, int *py) {
  int tmp = *px;
  *px = *py;
  *py = tmp;
}
// C++
void swapCPP (int& x, int& y) {
  int tmp = x;
  x = y;
  y = tmp;
}
// ...
int a, b;
swapC (&a, &b);
swapCPP (a, b);
```

lec01/02-swap.cpp

# Reference – `rvalue` reference

- Ve většině případů si potřebujeme `lvalue` referenci
- Představme si ale případ, že bychom chtěli např. do funkce místo reference předat hodnotu

```
void funkce (int& a) {...}
// ...
funkce (10); // co na to kompilátor?
```

- Situace je řešitelná pomocí `rvalue` reference

```
void funkce (int&& a) {...}
// ...
funkce (10);
```

# Konstanty

- Primitivní typy s `const` kvalifikátorem jsou v C++ konstanty
- Na rozdíl od C maker se C++ konstanty řídí pravidly pro rozsah platnosti

Je tedy možno deklarovat konstanty lokálně v modulu, funkci nebo třídě

```
const int MAX = 100;
int array[MAX]; // ok

int main(void) {
  int *p;
  const int *q;

  MAX = 10; // error
  p = &MAX; // error
  q = &MAX; // ok
  return 0;
}
```

# Operátor příslušnosti :: – kvalifikátor 1/2

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

```
int main () {
  int value = 10;
  // ..
  int value = 20;
}
```

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

```
namespace A {
  int value;
};
namespace B {
  int value;
}
int main () {
  A::value = 10;
  B::value = 20;
}
```

# Operátor příslušnosti `::` – kvalifikátor 

- Kvalifikátor umožňuje přístup ke globálnímu jmennému prostoru
  `::global = 3;`
- Funkce standarní 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 této direktivy je ale všeobecně považováno spíše za nevhodnou techniku a je lépe funkce plně kvalifikovat

**Příklad**

```
#include <iostream>

using namespace std;

int main () {
  cout << "Ahoj PPC!" << endl;
}
```

# Přetěžování funkcí 1/3

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

**Příklad**

```
int cube (int x) {
  return x * x * x;
}
double cube (double x) {
  return x * x * x;
}
// ..
int a = 2;
float b = 3.14;
std::cout << a << "^3 = " << cube(a) << std::endl;
std::cout << b << "^3 = " << cube(b) << std::endl;
```

lec01/03-pretizeni.cpp

# Přetěžování funkcí



- 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,
  - **roztažení (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.

# Implicitní hodnoty parametrů funkce 1/2

- 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**

```
void print (int x, int y = 0, int z = 0) {
  std::cout << "x=" << x << "\n";
  std::cout << "y=" << y << "\n";
  std::cout << "z=" << z << std::endl;
}
// ..
print (10, 20, 30); // 10, 20, 30
print (10, 20);     // 10, 20, 0
print (10);         // 10, 0, 0
```

lec01/04-parametry.cpp

# Implicitní hodnoty parametrů funkce 2/2

- Nedovolené použití implicitních parametrů:

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

- Kombinace přetížení a implicitních parametrů

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

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

```
g (20);      // viceznacne
g (10, 40); // ok
```

# Struktury

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

```
// C -- pojmenovaná struktura
struct List {int val; struct List *next;};
struct List *head;

// C -- nový datový typ
typedef struct List {
  int val; struct List *next;
} LIST;
LIST *head;

// C++
struct LIST {int val; LIST *next;};
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 na 3. přednášce

**Příklad**

```
struct cplx {float im; float re;};   // vlastní typ
cplx operator+(cplx &a, cplx &b) { // přetížení +
  cplx tmp;
  tmp.re = a.re + b.re; tmp.im = a.im + b.im;
  return tmp;
}
//..
cplx c, d, e;
e = c + d;              // zkrácené požití operátoru
e = operator+(c, d); // funkční použití operátoru
```

# Dynamická alokace paměti 1/2

- 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 []`.

# Dynamická alokace paměti



```
int *p = new int; // alokuje proměnnou typu int
struct S {
  int a;
  char b;
};
S *q = new S; // alokuje strukturu typu S
              // C++ nevyžaduje klíčové slovo struct
int *a = new int[1000]; // alokuje pole
delete p;     // uvolňuje jednoduchou proměnnou
delete q;
delete [] a;  // uvolňuje pole, bez [] je to chybně
a = p + 1;
delete a;     // chybně, uvolnit lze jen to,
              // co bylo vytvořeno pomoci new
```

# II. Úvod do programování v C++

# Vstup a výstup v C

```c
#include <stdio.h>
int main ( void )
{
  int x;
  printf ("Napis cislo:\n");
  scanf ("%d", &x);
  printf ("Vstup byl: %d\n", x);
  return 0;
}
```

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

# Vstup a výstup v C++

```
#include <iostream>
int main ( )
{
  int x;
  std::cout << "Napis cislo: ";
  std::cin >> x;
  std::cout << "Vstup byl: " << x << std::endl;
  return 0;
}
```

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).

- Proudy v C++ jsou bezpečné, neboť není třeba žádný formátovací řetězec, způsob konverze je vybrán kompilátorem podle typu parametru.
- Proudy v C++ mohou být snadno modifikovány:
  - vstup/výstup nových (uživatelských) datových typů,
  - různé zdroje/cíle proudů (soubory, buffery v paměti, sockety, ...)
- Standardní proudy, 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,
  - `noshowbase/showbase` – (ne)vypisovat `0x` - hex, resp. `0` - oct,
  - `boolalpha/noboolalpha` – true, false / 1, 0.

## Příklad

```
int x = 10;
cout << "dekadicky " << x << endl;
cout << "sirka 10 znaku " << setw(10) << x << endl;
cout << "sestnactkove " << hex << x << endl;
cout << "opet dekadicky " << dec << x << 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` – přeskakování bílých znaků při dalších operacích
  - `boolalpha/noboolalpha` – vstup true, false / 1, 0
  - `setw(n)` – omezení délky načítaného řetězce

**Příklad**

```
#include <iostream>
#include <iomanip>
int main() {
  std::string text;
  std::cin >> std::setw(3) >> text;
  std::cout << text << std::endl;
}
```

# Funkce pro neformátovaný I/O 1/3

## ostream & put()

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

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

## int get()

- přečte ze vstupního proudu znak a vrátí ho jako int

```
int i;
while ((std::cin.get()) != EOF) {
  std::cout.put(i);
}
```

## istream & get(char& c)

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

# Funkce pro neformátovaný I/O 2/3

`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

```
char * a;
while (std::cin.getline(a, 10, ' ')) {
  std::cout << a << std::endl;
}
```

---

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

```
std::string T;
while (getline(std::cin, T, ' ')) {
  std::cout << T << std::endl;
}
```

# Funkce pro neformátovaný I/O 3/3

```
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);
```

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

---

```
char peek ();
```

- Vrací příští znak ve vstupním bufferu bez jeho načtení

---

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

- Zapíše znak `c` do vstupního proudu

lec01/07-peek.cpp

# 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 standarní knihovny
  - **rdstate()** – stav všech bitů

    ```
    if (is.rdstate() & (ios::badbid||ios::failbit)) ...
    ```

  - **good()**, **bad()**, **fail()**, **eof()**
  - **clear()** – nulování stavového bitu
- Kromě funkcí standarní knihovny lze využít také systém vyjímek

# Stavové bity I/O proudů

```
double readDouble() {
  double d;
  std::cin >> d;
  if (std::cin.good()) {
    return d;
  }
  else if (std::cin.bad() || std::cin.eof()) {
    throw std::runtime_error("readDouble() failed");
  }
  else {
    std::cin.clear();
    std::cin.ignore(1, '\n');
    return readDouble();
  }
}
```

lec01/08-osetreni-chyb.cpp

# II. Úvod do programování v C++

# 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

---

```
#include <fstream>
//..
ofstream fout;
fout.open(filename, mode);
//..
fout.close();
// nebo lze kombinovat deklaraci a open()
ofstream fout(filename, mode);
```

lec01/09-textovy-soubor.cpp

# 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

# II. Úvod do programování v C++

# Výjimky

- V programování se výjimkou rozumí nějaká neočekávaná událost, která může vést i ke zhroucení samotného programu.
- Výjimky v C++
  - kód, ve kterém může nastat výjimka, uzavřeme do bloku `try`
  - pokud bloku `try` dojde k nějaké nečekané situaci
    - řízení programu se předá do bloku `catch`
    - bloků `catch` může být více
  - pokud k žádné výjimce nedojde, blok `catch` se přeskočí

dělení nulou, zápis za konec pole, čtení neexistujícího souboru, ...

```
try {
  // nějaký kód, který může způsobit výjimku
}
catch (typ vyjimky) {
  // kód, který se provede při vyvolání výjimky
}
```

# Výjimky – příklad

```
1  #include <iostream>
2  #include <exception>
4  int main() {
5    double a, b;
7    std::cin >> a >> b;
9    try {
10     if (b == 0) throw "Deleni nulou.\n";
11     std::cout << a / b << std::endl;
12   }
13   catch (const char* exception) {
14     std::cout << "Vyjimka - " << std::exception;
15   }
17   return 0;
18 }
```

lec01/10-vyjimka.cpp

# Ošetření více výjimek

- V programu může dojít k více výjimkám
- Je možné napsat více bloků `catch` pro různé výjimnky.
  - Výjimky jsou rozlišeny svým datovým typem
- Existuje blok `catch(...)`, který dokáže zachytit všechny výjimky.

```
try {
  NejakaNebezpecnaFunkce();
}
catch (int) {
  // zachycení výjimky datového typu int
}
catch (...) {
  // zachycení všech neošetřených výjimek
}
```

# Standardní výjimky v C++

C++ definuje třídu standardních výjimek, včetně interface a datových typů.

- `<exceptions>`
  - `std::exception` – bázová třída
  - `virtual what()` – vrací řetězec s popisem výjimky
  - `terminate()` – ukončuje program, volá `abort()`
- `<stdexcept>`
  - `std::out_of_range` – přístupu mimo rozsah
  - `std::overflow_error` – přetečení
- `<new>`
  - `std::bad_alloc` (memory allocation fails)
  - `std::nothrow`

# Co když není výjimka zachycena?

```
void myFunction()
{
  std::cerr << "Nezachycena vyjimka.\n";
  std::cerr << "Program bude ukoncen.\n";
  exit(1);
}
int main()
{
  std::set_terminate(myFunction);
  throw 1;
  return 0;
}
```

# Výjimka při alokaci dynamické paměti

```
try {
  while (true) {
    // throwing overload
    new int[100000000ul];
  }
} catch (const std::bad_alloc& e) {
  std::cout << e.what() << '\n';
}
while (true) {
  // non-throwing overload
  int* p = new (std::nothrow) int[100000000ul];

  if (p == nullptr) {
    std::cout << "Allocation returned nullptr\n";
    break;
  }
}
```