Úvod do C++

Prof. Ing. Aleš Čepek, CSc.

Vydavatelství ČVUT
2004

Lektor: Prof. Dr. Ing. Leoš Mervart, DrSc.

ISBN – – –

Obsah

Předmluva

Skriptum „Úvod do C++“ je určeno studentům oboru geodézie a kartografie, stavební fakulty ČVUT v Praze.

Jazyk C++ je dnes nejrozšířenějším objektově orientovaným programovacím jazykem. Autorem programovacího jazyka C++ je Bjarne Stroustrup [1]. Původní návrh jazyka C++ vycházel z programovacího jazyka C [7] a až na drobné výjimky je jazyk C jeho podmnožinou. Bjarne Stroustrup v [1] uvádí, že každý program ze slavné knihy [7] Kernighana a Ritchieho *The C Programming Language (2nd Edition)* je zároveň C++ programem.

Ve srovnání s jazykem C je programovací jazyk C++ nesrovnatelně mohutnější a to jak z hlediska syntaxe tak i možností, které poskytuje. Rozhodně se jej nenaučíte za týden a popravdě je nutno přiznat, že jazyk C++ obsahuje řadu temných zákoutí, ve kterých dokáže zabloudit nejen začátečník.

Existují různé přístupy k výuce jazyka C++. Jedním z nich je, že studenti se nejprve seznámí s jazykem C a teprve pak s C++ a jeho objektovými prostředky — tento přístup k výuce C++ je ale dnes již spíše výjimkou. Znalost C pro zvládnutí C++ není nezbytně nutná, především pokud neklademe důraz na ty vlastnosti, které jsou charakteristické právě pro jazyk C, tj. jazyk relativně blízký asembleru.

Síla a význam C++ je dána mimo jiné jeho rozsáhlou standardní knihovnou a úrovní abstrakce. Pojmy a nástroje jako jsou kontejnery, iterátory a generické algoritmy jsou pro pochopení moderního pojetí C++ mnohem důležitější něž různé triky, které například umožňují makra preprocesoru a pod. C++ je živý jazyk a jeho vývoj bude proto i nadále pokračovat. Jakým směrem se tento vývoj bude ubírat, naznačuje projekt zdrojových knihoven C++ Boost [4].

V našem kurzu nepředpokládáme žádné předchozí znalosti programování. Vyjdeme od elementárních konstrukcí a postupně se zaměříme na to, co je na C++ nejdůležitější, to jest na jeho objektové vlastnosti.

Skriptum je pomůckou pro první seznámení s jazykem C++, slouží jako doplněk přednášek a cvičení. První kapitola je určena studentům, kteří neměli možnost se na střední škole seznámit s žádným programovacím jazykem a lze ji proto při studiu vynechat. Nové pojmy a prvky jazyka C++ jsou demonstrovány na jednoduchých příkladech. Doporučuji vyzkoušet si prakticky na počítači všechny příklady, modifikovat je a experimentovat, dříve než budete pokračovat. Žádný programovací jazyk se nenaučíte pouhou četbou.

Všechny uvedené příklady byly přeloženy GNU kompilátorem `g++` verze `3.4.1`

`http://www.gnu.org/`

pod operačním systémem Debian GNU/Linux 3.1

`http://www.debian.org/`

Skriptum je vysázeno v systému LaTeX a v textu skripta jsou použity následující typografické konvence:

`int main() {}` neproporcionální písmo je použito pro tisk ukázek a v textu pro klíčová slova C++ a identifikátory.

třída (class) kurzívou jsou tištěny nově zaváděné pojmy, obvykle i s příslušným anglickým ekvivalentem uvedeným v závorce. Kromě toho je kurzíva používána i pro zvýraznění vybraných částí textu a také pro jména souborů.

$výraz_{vol}$ označuje v popisu syntaxe nepovinný výraz, tj. výraz, který může být vynechán.

Skriptum jsem psal během letních prázdnin 2004 s využitím starších učebních textů. Na řadu překlepů, nepřesností a námětů pro doplnění mne upozornil kolega Ing. Jan Pytel a poslal mi též několik zajímavých příkladů, rád bych mu proto na tomto místě za jeho pomoc poděkoval.

Prosím čtenáře, aby se nezdráhali upozornit mě na všechny chyby, které jsem přehlédl. Právě tak přivítám všechny připomínky a náměty a pokusím se zodpovědět případné dotazy. Text skripta ve formátu PDF, zdrojové texty příkladů a errata jsou umístěny na adrese

`http://gama.fsv.cvut.cz/~cepek/uvodc++`

Na závěr chci poděkovat lektoru skripta Prof. Dr. Ing. Leoši Mervartovi, DrSc. za pečlivé přečtení rukopisu a řadu připomínek.

Prof. Ing. Aleš Čepek, CSc.
`cepek@fsv.cvut.cz`

Kapitola 1

Začínáme od nuly

Tato kapitola je určena především těm studentům, kteří se na střední škole neměli možnost seznámit s žádným programovacím jazykem. Pokud již znáte například Pascal a umíte napsat a odladit jednoduchý *prográmek*, můžete text této kapitoly jen zběžně prolistovat nebo ji celou přeskočit. Ke všem pojmům a prvkům jazyka C++, se kterými se zde setkáme, se v dalších kapitolách znovu vrátíme.

Na následující ukázce programu si vysvětlíme některé elementární prvky jazyka C++

```
#include <iostream>

int main()
{
  std::cout << "Hello, world!" << std::endl;
}
```

Pokud bychom uvedený program přeložili a spustili, vypsal by nám na obrazovku text

```
Hello, world!
```

tj. *Nazdar, světe!*, nebo libovolný jiný text, který bychom zapsali do uvozovek na předposledním řádku. Takovému textu v uvozovkách říkáme *řetězec*. Na naší ukázce je důležité, že jde o kompletní C++ program. Jde o variaci na patrně nejslavnější program světa, kterým začínají svoji učebnici Kernighan a Ritchie [7].

Náš program obsahuje jediný příkaz ukončený středníkem, který zapisuje konstantní řetězec na standardní výstupní proud `std::cout`, který je obvykle implicitně směrován na monitor počítače. Výstup (tisk) je realizován operátorem výstupu `<<` a symbol `std::endl` označuje nový řádek. Operátor `<<` je v ukázce použit dvakrát. Příkaz výstupu bychom se stejným výsledkem mohli zapsat třeba takto

```
std::cout << "Hello,"
          << " world!" << std::endl;
```

Datové proudy nejsou přímo součástí jazyka ale jsou definovány ve standardní C++ knihovně a musíme je proto nejprve řádně deklarovat, dříve než je můžeme použít. Standardní výstupní proud `std::cout` je objekt, který je deklarován v hlavičce `<iostream>`. Pokud se explicitně odkazujeme na nějaké *jméno* ze standardní knihovny, vyjádříme to takto `std::`*jméno* (více viz 3.9 a 3.9.2).

První řádek našeho programu je *direktiva preprocesoru*

```
#include <iostream>
```

Překladač ji zpracuje tak, že na její místo vloží text souboru, který je jejím argumentem, resp. zdrojový text uvedené systémové hlavičky. Zde jde o standardní hlavičku `<iostream>` ze standardní knihovny.

Program dále obsahuje funkci `main()`, která musí být součástí každého C++ programu. Tělo funkce `main()` tvoří *blok* ohraničený dvojicí složených závorek a obsahuje posloupnost příkazů, které se začnou provádět při spuštění programu.

Předpokládejme, že zdrojový text programu je uložen v souboru pojmenovaném *hello.cpp*. Přípona *.cpp* označuje, že jde o zdrojový kód v jazyce C++ (obdobně přípona *.h* označuje hlavičkové soubory). Na různých systémech se můžete setkat případně i s jinými konvencemi. Abychom mohli náš program spustit, musíme jej nejprve přeložit programem, kterému se říká *překladač* nebo *kompilátor* (z anglického označení *compiler*). Na mém počítači pod operačním systémem Debian GNU/Linux 3.1 vypadá překlad a spuštění programu třeba takto

```
bash$
bash$ g++ -o hello hello.cpp
bash$ ./hello
Hello, world!
bash$
```

V ukázce je `bash$` výzva systému pro vložení příkazu, `g++ -o hello hello.cpp` volání GNU C++ kompilátoru pro překlad a sestavení programu a konečně `./hello` je volání přeloženého programu.

Kompilátor `g++` je jako všechny GNU produkty *Free Software* a je volně šiřitelný včetně zdrojových kódů pod licencí GPL. Projekt GNU byl zahájen v roce 1984 s cílem vyvinout kompletní operační systém podobný Unixu, který by byl *Svobodným Softwarem;* jméno projektu GNU je rekurzivní akronym *GNU's not Unix (GNU není Unix).* Podrobné informace o GNU projektu najdete na adrese

`http://www.gnu.org/`

Všechny ukázky uvedené v následujícím textu byly přeloženy GNU překladačem `g++` verze 3.4.1.

1.1 Minimální C++ program

Minimální C++ program obsahuje pouhých 12 znaků

```
int main(){}
```

Nic nedělá, lze jej ale přeložit a spustit. Znak *mezera,* který odděluje klíčová slova `int` a `main`, je zde významný a nelze jej vypustit.

Uvedených dvanáct znaků musí být obsaženo v každém C++ programu — jinak řečeno, každý C++ program musí obsahovat funkci `main`.

1.2 Základní typy proměnných

Každý reálný program musí nějak komunikovat s vnějším okolím, tj. musí mít možnost přijímat data, ukládat je, manipulovat s nimi a předávat zpracované informace uživateli. Elementární nástroj pro ukládání dat představují proměnné. V této kapitole prozatím vystačíme s proměnnými základních typů:

`int` představuje počítačovou realizaci celých čísel,

`double` obdobně slouží pro reprezentaci reálných čísel,

`char` je základní typ určený pro ukládání jednotlivých znaků,

`bool` je logický typ, který může nabývat pouze hodnot `true` a `false`.

Deklarace proměnné je příkaz, který tvoří typ, jméno proměnné (identifikátor) a je ukončen středníkem. Například

```
double a1;
```

je deklarace proměnné typu `double`, kterou jsme pojmenovali `a1`.

Identifikátor je v C++ posloupnost písmen anglické abecedy a číslic začínající písmenem. Za písmeno je považován i znak podtržítko (_) a rozlišují se velká a malá písmena. Proměnné mohou být navíc při deklaraci inicializovány, jak ukazuje následující příklad

```
#include <iostream>

int main()
{
  int i_prom = 12;
  std::cout << "promenna i_prom  = " << i_prom << std::endl;

  int octal  = 010;
  std::cout << "oktalova hodnota 010 je " << octal
            << " v desitkove soustave" << std::endl;

  int x;
  std::cout << "hodnota promenne x neni definovana! x = "
            << x << std::endl;

  double Q = 12.3;
  char   q = 'P';
  std::cout << "Q je " << Q << " a promenna q obsahuje "
            << q << std::endl;
}
```

Výstup programu by mohl vypadat třeba takto

```
promenna i_prom  = 12
oktalova hodnota 010 je 8 v desitkove soustave
hodnota promenne x neni definovana! x = 1076159824
Q je 12.3 a promenna q obsahuje P
```

Kondicionál v předchozí větě je použit proto, že výpis hodnoty proměnné x je zcela náhodný. Při inicializaci proměnných jsme použili konstant daného typu, ale obecně může jít o výrazy. V této souvislosti si povšimněte, že oktalové konstanty se v C++ uvozují nulou a konstanty typu `double` píšeme s desetinnou tečkou (pravidla pro zápis konstant upřesníme později).

Pro číselné proměnné jsou definovány obvyklé operátory jako

```
+   -   *   /   =
```

a řada dalších. Hvězdička je operátor násobení a rovnítko je operátor přiřazení.

V následujícím fragmentu programu jsou jednotlivé příkazy vysvětleny v *komentářích* (text za dvojicí lomítek do konce řádku):

```
int k = 5;          // deklarujeme proměnnou k a inicializujeme ji na hodnotu 5
double pom;         // deklarujeme proměnnou pom základního typu double
k = k * 2;          // do proměnné k uložíme novou hodnotu 5 × 2
pom = k + 0.364;    // do proměnné pom přiřadíme hodnotu výrazu 10 + 0.364
```

Výsledná hodnota proměnné `pom` bude tedy `10.364`. Před přiřazením se nejprve vyhodnotí výraz vpravo od operátoru přiřazení. Hodnota `k` je proto rovna 10 (nejprve se vypočte 5 krát 2 a pak se teprve provede přiřazení nové hodnoty). Příkazy se v C++ ukončují středníkem. Za příkaz jsou v C++ považovány i deklarace (první dva řádky), poslední dva řádky jsou přiřazovací příkazy.

Při počítání s celočíselnými proměnnými platí zvláštní pravidla. Výsledkem je vždy celé číslo. Výsledek operátoru dělení (`/`) je číslo, které vznikne po zanedbání desetinného rozvoje. Pokud je alespoň jeden z operandů typu `double`, získáme výsledek včetně desetinné části. Program

```
#include <iostream>

int main()
{
  int a = 7;               // celociselna promenna
  int b = 3;
  double c = 7.0;          // 'realna' promenna
  double d = 3.0;
  std::cout << a/b << " "  // celociselne deleni  7/3 == 2
            << b/a << " "  // decociselne deleni  3/7 == 0
            << c/b << " "  // deleni pro typ double  7.0/3 == 2.333...
            << d/a         // deleni pro typ double  3.0/7 == 0.428...
            << std::endl;
}
```

proto vypíše tento výsledek

```
2 0 2.33333 0.428571
```

Předchozí příklad obsahuje funkci `main`, ve které jsme zapsali celkem pět příkazů. Po spuštění programu se provádí postupně příkazy funkce `main` v pořadí jak byly zapsány, pokud toto pořadí nezměníme příkazy pro řízení chodu programu o kterých si řekneme dále. Prozatím si zapamatujte, že funkce `main` určuje, kde náš program má začít pracovat a co bude dělat.

1.3 Komentáře

Ve zdrojových textech C++ můžeme používat dva typy komentářů. První z nich je uvozen dvojicí znaků `/*` a je ukončen dvojicí znaků `*/`. Tento komentář může pokračovat i na více řádcích, komentáře `/* ... */` se ale nesmí vnořovat. Druhý typ je uvozen dvojicí znaků lomítko `//`. Veškerý následující text do konce řádku je kompilátorem ignorován.

```
#include <iostream>

/*
  Toto je ukazka komentare, ktery je zapsan na dvou radcich.
  Komentare se mesmi vnorovat ale mohou obsahovat znaky //
*/

int main()      // zde je dalsi ukazka komentare
{               // tento typ komentare muze obsahovat /* ... */

  std::cout << "Ukazka pouziti komentaru" << std::endl;
}
```

Komentáře jsou určeny pro usnadnění čtení zdrojového textu a pro zaznamenávání údajů, jako jsou například číslo verze, datum poslední úpravy, jméno autora a podobně. Obecně platí, že dobře napsaný program je i dobře čitelný. Vhodně zvolené komentáře k tomu mohou podstatně přispět. S tím souvisí i grafická úprava zdrojového textu. Naučíte-li se důsledně odsazovat jednotlivé programové struktury jako jsou bloky, podmínky a cykly (o těch si řekneme v následujícím odstavci), rozhodně se vám to vyplatí. I celkem jednoduchý program lze nedbalou grafickou úpravou zatemnit tak, že se v něm nevyzná ani samotný autor. V programování platí obdobně jako v životě, že kdo se neumí řádně vyjadřovat, neumí obvykle ani řádně myslet.

Komentáře nemají opodstatnění v případech, kdy je smysl zřejmý ze zápisu C++. Tuto zásadu budeme v našich ukázkách často porušovat a psát komentáře i tam, kde se za normálních okolností neuvádějí. Naše ukázky jsou ale určeny začátečníkům a tomu jsou přizpůsobeny i komentáře. Stručný komentář ve zdrojovém textu mnohdy řekne víc než celý odstavec textu popisujícího daný program.

Čitelnost programu je také závislá na vhodně volených identifikátorech. Naučte se volit jména proměnných (a dalších objektů) tak, aby identifikátory jasně napovídaly, co označují.

Neexistuje univerzální návod, jak správně graficky formátovat C++ texty. Základní grafickou úpravu podle syntaxe často provádí přímo textový editor daného vývojového prostředí. Jednu zásadu lze ale rozhodně doporučit. Pište své programy jednotným dobře čitelným stylem.

Pokusíme se uvedené zásady doložit jednoduchým příkladem funkce pro výpočet druhé odmocniny.

```
double odmocnina(double x)
{
   if (x <= 0)  return 0;

   double xm, N = x;
   do {
      xm = x;
      x = 0.5 * (x + N/x);
   } while (std::abs(x - xm) > x*1e-6);
```

```
   return x;
}
```

Pojmenujeme-li dostatečně zvrhle všechny identifikátory (a použijeme-li navíc ještě *trigrafy* uvedené v tabulce 2.3), lze výše uvedených třináct řádků zapsat i takto

```
double q0(double q1)??<if(q1<=
0)return 0;double q2,q3=q1;do
??<q2=q1;q1=0.5*(q1+q3/q1);??>
while(std::abs(q1-q2)>q1*1e-6)
;return q1;??>
```

Obě uvedené funkce přeloží kompilátor stejně. Závěr nechť si každý učiní sám. Tento odstrašující příklad byl vygenerovány programem `cobfusc` z Debian `cutils`.

1.4 Vstupní a výstupní proudy

Obdobně jako jsme používali standardní výstupní proud `std::cout` pro jednoduchý výstup údajů, slouží pro vstup standardní vstupní proud `std::cin` a operátor vstupu `>>`

```
#include <iostream>

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

Po výpisu zprávy očekává program vložení celočíselné hodnoty a pak ji opíše na `std::cout`. Formálně jsou `std::cin` a `std::cout` objektové proměnné a v C++ jsou výjimečné tím, že je nemusíme explicitně deklarovat.

Začátečníci, především pak ti, kteří znají jiný jazyk, často projevují nevoli při prvním seznámení s datovými proudy a argumentují, že jde o nesrozumitelnou konstrukci (ve srovnání s příkazy vstupu a výstupu v jiných jazycích) a podobně. Pokud používáme operátory a metody pro práci s proudy bez znalosti objektových vlastností C++, je to do jisté míry i pravda. Prozatím se proto naučte jen mechanicky používat několik základních konstrukcí.

V C++ programu můžeme kromě standardních proudů `std::cin` a `std::cout` definovat vlastní objektové proměnné a číst a zapisovat soubory obdobně jako čteme vstup z klávesnice a vypisujeme výstup na obrazovku (prozatím se omezíme pouze na textové soubory). Předpokládejme, že máme soubor *vstup1.txt*, který obsahuje text:

```
10  101.43254

15.676378
24.9
```

Oddělovači údajů jsou tzv. *bílé znaky* (mezery, tabelátory a znaky ukončující textový řádek). Tento soubor bude číst program, který načte ze vstupního souboru jedno celé číslo a tři reálná čísla

```
#include <fstream>     // prace se souborovymi proudy
#include <iomanip>     // manipulatory setw, setprecision, ...

int main()
{
  // instance 'vstup'  otevira soubor 'vstup1.txt'  pro cteni
  //          'vystup' otevira soubor 'vystup1.txt' pro zapis
  std::ifstream vstup ("vstup1.txt");
  std::ofstream vystup("vystup1.txt");

  int index;
  vstup  >> index;
  vystup << "Nacteno cele cislo " << index << std::endl;

  double a, b, c;
  vstup  >> a >> b >> c;

  // promenna 'a' bude vypsana podle implicitnich pravidel
  vystup << "Nactena tri cisla typu double " << a;

  // pro promenne 'b' a 'c' nastavime vystupni formatovani explicitne
  vystup.setf(std::ios_base::fixed);
  vystup.precision(4);
  vystup << " " << b << std::setw(14)
         << std::setprecision(7) << c << std::endl;
}
```

a vytvoří výstupní soubor *vystup1.txt*

```
Nacteno cele cislo 10
Nactena tri cisla typu double 101.433 15.6764    24.9000000
```

V programu jsme nahradili doposud užívanou hlavičku `<iostream>` hlavičkou `<fstream>`, která nám umožňuje pracovat se souborovými proudy. Dále jsme doplnili hlavičku `<iomanip>`, ve které jsou definovány tzv. manipulátory.

První dva příkazy funkce `main()` deklarují objektové proměnné `vstup` a `vystup` a zajišťují jejich spojení s příslušnými soubory. Namísto objektové proměnné se často říká *instance*. V našem programu jsme tedy vytvořili instanci `vstup` pro čtení dat ze souboru *vstup1.txt* a instanci `vystup` pro zápis výsledků do souboru *vystup1.txt*.

Standardní souborové proudy deklarované v hlavičce `<fstream>` se jmenují `fstream`, kde písmeno *f* je odvozeno z anglického slova *file (soubor)* a *stream* překládáme jako *proud*. Pro vstupní proudy začíná jméno typu písmenem *i* podle anglického slova *input (vstup)*, pro výstupní proudy písmenem *o* podle *output (výstup)*.

Předposlední dva příkazy

```
vystup.setf(ios_base:fixed);
vystup.precision(4)
```

specifikují požadavek na výstup číselných hodnot v pevném formátu a nastavují počet vypisovaných cifer za desetinnou tečkou.

V posledním příkazu udává `setw(14)` minimální počet znaků následujícího údaje na výstupu a `setprecision(7)` je jiný způsob zadání požadovaného počtu cifer za desetinnou tečkou. Těchto několik operací se výstupními proudy by vám pro začátek mělo postačit, pokud budete ve svých cvičných programech chtít načítat data ze souborů (obecně ze standardních proudů) a nejen ze standardního vstupního zařízení.

1.5 Testy a cykly

Každý C++ program musí obsahovat funkci `main()`. Po spuštění programu se začnou postupně provádět příkazy v pořadí, ve kterém byly zapsány v těle funkce `main()`. Je-li daným příkazem volání funkce, předá se řízení této funkci a začnou se analogicky provádět příkazy zapsané v těle této funkce. Po návratu z funkce pokračuje program dalším příkazem.

Pro změnu posloupnosti prováděných příkazů slouží příkazy, o kterých se nyní stručně zmíníme.

1.5.1 Příkazy `while`, `do` a `for`

Příkazy `while`, `do` a `for` slouží pro zápis cyklů a iterací. První dva z uvedených příkazů mají syntax

```
while ( výraz ) příkaz
do příkaz while ( výraz ) ;
```

V příkazu `while` se provádí *příkaz* dokud je zadaný *výraz* pravdivý. To znamená, že v příkazu `while` se daný příkaz nemusí vůbec provést. Protože se v příkazu `do` testuje podmínka až nakonec, provede se daný příkaz vždy alespoň jednou.

Aby program neuvázl v nikdy nekončícím cyklu, je většinou nutné, aby *příkaz* modifikoval *výraz*. Můžete se sice setkat s programy, ve kterých je záměrně naprogramován nekonečný cyklus (který je ukončen jinými prostředky), to ale nejsou techniky vhodné pro začátečníka. Cyklus `while` v následujícím fragmentu programu je jistě chybný.

```
double a = 2;
int K10 = 10;
while ( a < K10)       // invariantni vyraz
   std::cout << "!!! chyba !!!" << std::endl;
```

V C++ je za pravdivý považován každý numerický výraz různý od nuly. Ve výrazu můžeme užít relačních operátorů

```
<   >   <=   >=   ==   !=
```

s významem *je menší, je větší, menší nebo rovno, větší nebo rovno, rovná se a nerovná se*. Z logických operátorů uveďme prozatím jen logické operátory konjunkce (and) a disjunkce (or)

```
&&   ||
```

Ve výrazech lze používat závorky a pokud si nejste jisti, raději používejte závorky více než méně.

V popisu syntaxe příkazů `while` a `do` může být příkazem *blok*. Blok, nebo též složený příkaz, je v C++ příkaz, který začíná otevírací složenou závorkou `{` a je ukončen uzavírací složenou závorkou `}`. V bloku se obvykle uvádí posloupnost několika příkazů. Protože je středník v C++ součástí syntaxe (a ne oddělovačem příkazů jako třeba v Pascalu), není v následujícím fragmentu za příkazem `while` středník uveden.

```
double p = 1;
int n = 0;
while ( p > 0.00001 ) {
  n++;
  p *= 0.5;
  std::cout << n << " " << p << std::endl;
}

std::cout << "cyklus while ukoncen po " << n << " iteracich\n";
```

V příkladu jsou uvedeny dva nové operátory. Postfixový operátor `++` zvýší hodnotu svého operandu o `1` (totéž jako `n = n + 1;`) a operátor `*=` vynásobí hodnotu prvního operandu druhým operandem (totéž jako `p = p * 0.5;`).

V bloku můžeme deklarovat nové proměnné, které jsou pak v něm lokální a přestávají existovat s ukončením bloku. Pokud v bloku deklarujeme proměnnou se stejným identifikátorem jaký má existující proměnná deklarovaná vně bloku, zastíní nová deklarace původní proměnnou, která pak není implicitně v bloku vidět. Po ukončení bloku skončí existence lokální proměnné a původní proměnná je opět viditelná.

Zjednodušená syntax příkazu cyklu `for` je

`for (`*příkaz 1* `;` *podmínka* `;` *výraz* `)` *příkaz 2*

Popis příkazu `for` není zcela korektní, jeho přesnou syntax uvedeme později. Činnost příkazu `for` můžeme popsat s využitím příkazu `while` takto

příkaz 1 `;`
`while (` *podmínka* `) {`
  příkaz 2
  výraz `;`
`}`

Nejprve se tedy provede *příkaz 1*. Je-li splněna *podmínka*, provede se *příkaz 2*. Vyhodnotí se *výraz* (obvykle mění *podmínku*) a znovu se testuje *podmínka* cyklu.

Jako *příkaz 1* často vystupuje příkaz deklarace numerické proměnné s inicializací, jako například

```
for (int i=1; i <= 10; i++) {
  // blok se provede 10 krat,   i = 1, ..., 10
}
```

Proměnná `i` je v příkazu cyklu lokální a po ukončení cyklu přestává existovat. Některé starší překladače tuto vlastnost nepodporují a proměnná existuje až do konce bloku, ve kterém se nachází příkaz `for`.

V následující ukázce je použit příkaz `while` pro numerický výpočet odmocniny $x = \sqrt{N}$ podle rekurentního vzorce

$$x_{i+1} = \frac{1}{2}\left(x_i + \frac{N}{x_i}\right)$$

```
#include <iostream>
#include <cmath>

int main()
{
  double N = 3;
  double x = N/2;

  while (std::abs(x*x - N)/N > 1e-6)
    x = 0.5 * ( x + N/x);

  std::cout << x << " " << x*x - N << std::endl;
}
```

V hlavičce `<cmath>` je deklarována funkce `std::abs()` pro výpočet absolutní hodnoty pro typ `double`. Konstanta `1e-6` je C++ zápis čísla 1×10^{-6}.

1.5.2 Příkaz `if`

Příkaz `if` umožňuje větvení programu. Kromě příkazu se syntaxí

`if (` *výraz* `)` *příkaz*

je v C++ k dispozici i příkaz

`if (` *výraz* `)` *příkaz* `else` *příkaz*

Kromě těchto dvou příkazů pro větvení programu má C++ ještě příkaz `case`, o kterém ale v této kapitole hovořit nebudeme.

Jako příklad použití příkazu `if` uvedeme program, který počítá kořeny kvadratické rovnice

$$ax^2 + bx + c = 0$$

podle známého vzorce

$$x = \frac{-b \pm \sqrt{D}}{2a}, \qquad D = b^2 - 4ac.$$

```
#include <iostream>
#include <cmath>

int main()
{
  double a, b, c;
  while (std::cin >> a >> b >> c  &&  a != 0)
    {
      double D = b*b - 4*a*c;
      if (D >= 0) {  // reseni v realnem oboru
        D = std::sqrt(D);
        std::cout << "x1  = " << (-b + D) / (2*a) << "  "
                  << "x2  = " << (-b - D) / (2*a)
                  << std::endl;
```

```
        }
        else {         // reseni v komplexnim oboru
          D = std::sqrt(-D);
          std::cout << "x12 = " << -b / (2*a) << " +-i*" << D / (2*a)
                    << std::endl;
        }
      }
    }
```

V ukázce je použit operátor vstupu `>>` přímo v podmínce příkazu `while`. V C++ je to výraz, který se vyhodnotí jako pravdivý, pokud daná operace úspěšně proběhla. V našem případě to bude pokud nebyl dosažen konec dat (navíc požadujeme, aby byl koeficient kvadratického členu nenulový). Pro výpočet odmocniny je volána funkce `std::sqrt()` definovaná ve standardní hlavičce `<cmath>`. Jazyk C++ nemá operátory pro výpočet mocniny a výrazy jako b^2 se obvykle rozepisují jako `b*b` (pro výpočet obecné mocniny slouží funkce `std::pow(double, double)` definované v hlavičce `<cmath>`).

V odstavci 1.3 jsme hovořili o grafické úpravě zdrojového textu. Předchozí příklad je i první elementární ukázkou zarovnávání jednotlivých příkazů podle řídících struktur. Formátování textu dnes obvykle zařídí přímo textový editor, v tomto případě GNU Emacs.

1.6 Funkce

Funkce jsou základním nástrojem programátora v C++. Chceme-li vypočítat v programu odmocninu, zavoláme funkci `sqrt()`. V tomto a podobných případech z oblasti základních numerických výpočtů se pojem *funkce* kryje s pojmem funkce, jak jej známe z matematiky.

V C++ představuje funkce algoritmus. Volání funkce je požadavek na provedení tohoto algoritmu. Opodstatnění mají nejen funkce používané opakovaně, ale i uživatelské funkce, které v programu použijeme jen jednou. Pokud dokážeme program rozdělit na dílčí kroky, které naprogramujeme jako funkce, bude text programu přehlednější a snáze jej odladíme — i v programování platí stará římská zkušenost *rozděl a panuj*.

Zjednodušená syntax definice funkce je

typ identifikátor `(` *seznam parametrů*$_{vol}$ `)` `{` *příkazy*$_{vol}$ `}`

Seznam parametrů a *příkazy* jsou volitelné, tj. mohou případně chybět. Abychom mohli funkci použít, musíme ji nejprve deklarovat. Definice funkce je přitom zároveň deklarací. Deklarace funkce je podobná její definici, místo těla funkce je ale ukončena středníkem. V programu může být funkce deklarována vícekrát, definice funkce smí být ale jen jedna.

Volání funkce je příkaz ve kterém zapíšeme její identifikátor, seznam argumentů a středník. Rozdíl mezi definicí a deklarací funkce a příklad volání dvou funkcí je demonstrován v následující ukázce. Typ funkce `void` znamená, že funkce nevrací hodnotu.

```
#include <iostream>

void A()     // definice funkce A()
{
  std::cout << "Provadi se funkce A" << std::endl;
```

```
}
void B();    // funkce B() je zde pouze deklarovana

int main()
{
   for (int i=1; i<7; i++)
     if (i%2 != 0)     // operator % znamena zbytek po deleni
       A();            // volani funkce A()
     else
       B();            // volani funkce B()
}

// nasleduje definice funkce B
void B()
{
  std::cout << "Provadi se funkce B";
  std::cout << std::endl;
}
```

V ukázce je použit celočíselný operátor %, který počítá zbytek po dělení levého operandu pravým operandem. V naší ukázce je tedy pro liché hodnoty `i` volána funkce `A`, pro sudé funkce `B`. Protože definice funkce `B()` je uvedena až za funkcí `main()`, museli jsme ji deklarovat před jejím prvním použitím.

Deklarace funkce může být uvedena i uvnitř jiné funkce. Nelze ale definovat *interní* funkce (jako například v Pascalu).

1.6.1 Návratová hodnota funkce

Některé funkce nevracejí hodnotu, jako například `A()` a `B()` z předchozí ukázky. Typ takových funkcí v C++ je `void`. Typickým reprezentantem funkcí vracejících hodnotu je například funkce `sqrt()` pro výpočet odmocniny. Její deklarace je v hlavičce `<cmath>` a je

```
double sqrt(double);
```

Funkce, které vracejí hodnotu, musí být ukončeny příkazem

```
return výraz ;
```

kde *výraz* je předávaná návratová hodnota. Příklad uvedeme v následujícím odstavci. Funkce typu `void` mohou být také ukončeny příkazem `return`, který ale v takovém případě neobsahuje *výraz*. Příkaz `return` může být ve funkci uveden i vícekrát (v různých větvích daného algoritmu).

1.6.2 Předávání argumentů hodnotou

V C++ máme dvě možnosti jak předávat funkci argumenty

- hodnotou,
- referencí.

Na mechanismus předávání argumentů hodnotou se můžeme dívat tak, jako by parametry byly lokální proměnné funkce a jako by byly při volání funkce inicializovány hodnotami argumentů (provede se přiřazení). Argumenty přitom mohou být i výrazy, např. `sin(0.1+x)` je volání funkce *sinus*, kde argumentem je výraz `0.1+x`. Veškeré změny takto definovaných parametrů jsou lokální ve funkci a neprojeví se ve volajícím programu. Příkladem je funkce `odmocnina()`, která počítá odmocninu ze zadaného argumentu podle algoritmu z ukázky na straně 20. Výsledek předává funkce jako návratovou hodnotu.

```
#include <iostream>
#include <cmath>

double odmocnina(double x);                    // deklarace funkce

int main()
{
  for (double i=0; i<10; i++)
    std::cout << odmocnina(i) << std::endl;    // volani funkce
}

double odmocnina(double x)                     // definice funkce
{
  if (x <= 0)
    return 0;
  double xm, N = x;
  do {
    xm = x;                                    // predchozi hodnota
    x = 0.5 * (x + N/x);                       // nova hodnota
  } while (std::abs(x - xm) > x*1e-6);
  return x;
}
```

1.6.3 Předávání argumentů referencí

Druhou možností jak předávat argumenty, je předávání referencí (někdy se používá termín předávání odkazem). Má-li být argument funkce předáván referencí, uvedeme v definici funkce za specifikací typu parametru znak *ampersand* &. Funkce může mít parametry volané hodnotou i referencí, jak ukazuje v následujícím příkladu funkce `prevod()`, která počítá kartézské souřadnice (x, y) ze zadaných polárních souřadnice (ϱ, φ).

```
#include <iostream>
#include <cmath>

void prevod(double rho, double phi, double& sour_x, double& sour_y)
{
  sour_x = rho * std::cos(phi);
  sour_y = rho * std::sin(phi);
}

int main()
{
  double x, y;
  for (double r = 0, p = 0; r < 10; r++) {
```

```
    prevod(r, p, x, y);
    std::cout << "( " << r << ", " << p << " ) ==> "
              << "( " << x << ", " << y << " ) " << std::endl;
    p += 0.05;
  }
}
```

Příklad zároveň demonstruje, že není žádná souvislost mezi jmény parametrů funkce (`rho`, `phi`, `sour_x`, `sour_y`) a jmény proměnných, které použijeme jako argumenty při volání dané funkce (`r, p, x, y`).

1.7 Od struktur ke třídám

Na závěr této kapitoly se pokusíme naznačit co jsou to struktury a třídy jazyka C++. Třídy představují nástroj, pomocí kterého můžeme definovat vlastní datové typy, jsou základem objektového programování a budeme se jim věnovat v kapitolách 5 a 6. Jaký je smysl tříd a motivace pro jejich používání si ukážeme na návrhu třídy `Polygon`.

Struktury v C++ slouží pro definování *agregovaných typů*, jinak řečeno struktury umožňují spojit logicky dohromady dva nebo více objektů, resp. proměnných, existujících typů a tento složený typ pojmenovat. Definice struktury začíná klíčovým slovem `struct`, následuje jméno struktury a blok s deklaracemi členů struktury. Definice struktury musí být ukončena středníkem.

V C++ se struktury (`struct`) a třídy (`class`) liší pouze pravidly pro přístup k jejich členům. Struktury mají implicitně všechny členy přístupné (`public`), třídy mají implicitně všechny členy soukromé (`private`).

Naším úkolem je tedy definovat nový typ, který by nám umožnil snadnou práci s objekty typu *polygon*, kde polygon (mnohoúhelník) je určen seznamem rovinných bodů. Nejprve tedy musíme začít definicí typu bod.

Pro naši potřebu můžeme bod popsat jako strukturu obsahující dva datové členy `sx` a `sy` popisující souřadnice a příznak `test`, který určuje, zda jsou souřadnice daného bodu definovány nebo ne:

```
struct Bod {
  double  sx;
  double  sy;
  bool    test;
};
```

Tím jsme definovali velmi jednoduchý nový typ a můžeme nyní vytvářet proměnné typu `Bod`, obdobně jako proměnné základních typů. Pro přístup k datovým členům instancí typu `Bod` je určen operátor *tečka* (.), viz například:

```
Bod a, b, c;   // a, b, c jsou objektové proměnné typu Bod
               //          objektovým proměnným se obvykle říká instance

// ...
std::cout << a.sx << " " << a.sy;     // výstup souřadnic bodu a
```

S objekty typu `Bod` toho nemůžeme dělat mnoho. Můžeme pracovat s jejich složkami, můžeme je vzájemně přiřazovat, předávat je jako parametry funkcí anebo jako návratovou hodnou funkce.

Předchozí příklad navíc skrývá jeden poměrně závažný problém: jaká je hodnota atributu `test` po vytvoření dané instance? ... tak jako v případě základních typů je atribut `test` po vytvoření instance v nedefinovaném stavu. To ale můžeme snadno ošetřit doplněním speciální členské funkce, které říkáme *implicitní konstruktor.*

```
struct Bod {

  Bod() { test = false; }      // implicitní konstruktor (nemá typ)

  double  sx;
  double  sy;
  bool    test;
};
```

Složkami struktury tedy mohou být nejen datové členy ale i funkce. Implicitní konstruktor, pokud je definován, je volán vždy při vytvoření každé instance a v našem případě nastavuje atribut `test` na implicitní hodnotu `false` (bod při vytvoření nemá souřadnice definovány).

Zavedením implicitního konstruktoru jsme ale problém s nastavením atributu `test` vyřešili jen částečně. S instancemi typu `Bod` budeme totiž provádět dvě základní operace: a) budeme do nich ukládat souřadnice a b) budeme tyto souřadnice používat v dalších výpočtech. Není přitom nikterak zaručeno, že programátor, který pracuje s instancí typu `Bod` bude vždy udržovat nastavení jejího atributu `test` v konsistentním stavu. Jinak řečeno, není zaručeno, že programátor po té co nastaví hodnoty souřadnic x a y zároveň také nastaví atribut `test` na hodnotu *true.*

Řešení našeho problému poskytuje mechanismus, kterému se říká *zapouzdření (encapsulation).* Přepíšeme strukturu `Bod` na třídu (klíčové slovo `struct` zaměníme za klíčové slovo `class`) a doplníme členské funkce pro ukládání a výběr souřadnic. Datové členy `sx`, `sy` a `test` budou nyní privátní a přístupné pouze prostřednictvím veřejných metod třídy `Bod`. Zapouzdření atributů zaručuje, že instance třídy `Bod` budou vždy v konsistentním stavu.

```
class Bod {
public:

  Bod() { test = false; }

  void vloz_xy(double x, double y) { sx = x;  sy = y; test = true; }
  void zrus_xy() { test = false; }

  double x() const { return sx; }
  double y() const { return sy; }
  bool ma_souradnice() const { return test; }

private:

  double  sx;
  double  sy;
  bool    test;
};
```

Dříve než přistoupíme k návrhu třídy `Polygon`, musíme rozhodnout, zda polygon bude udržovat přímo seznam bodů nebo jen odkazy na jednotlivé body. Druhá možnost je vhodnější, pokud předpokládáme, že budeme pracovat s více než s jedním polygonem a různé polygony budou sdílet hraniční body.

Abychom mohli realizovat druhou z uvedených alternativ, budeme používat pro všechny polygony společný seznam bodů, který budeme realizovat jako standardní kontejner *vector* definovaný v hlavičce `<vector>`:

```
std::vector<Bod> seznam(1000);    // maximální počet bodů v seznamu je 1000
```

Jednotlivé body budeme označovat čísly 0 až N a v seznamu k nim budeme přistupovat pomocí operátoru indexování `[]`. Jedním z atributů třídy polygon bude reference na společný seznam bodů. Seznam čísel bodů bude dalším atributem třídy `Polygon` a budeme jej realizovat jako standardní celočíselný seznam `list<int>`, kontejner *list* je deklarován ve standardní hlavičce `<list>`.

Třída `Polygon` má jeden konstruktor, který při vytvoření instancí vytváří referenci na společný seznam bodů.

```
class Polygon {
private:

  std::vector<Bod>& seznam;
  std::list  <int>  sb;

public:

  Polygon(std::vector<Bod>& s) : seznam(s) {}

  void   pridej_bod(int b) { sb.push_back(b); }
  double plocha () const;
  Bod    teziste() const;
  // ...
};
```

V definici třídy `Polygon` jsou uvedeny pouze deklarace členských funkcí pro výpočet plochy a těžiště polygonu a jejich definice budou typicky zapsány v jiném souboru.

```
double Polygon::plocha() const
{
  // zaporna plocha indikuje chybny vypocet (mene nez 3 body)
  if (sb.size() < 3) return -1;

  double s = 0;                          // sumace pro vypocet plochy
  Bod z = seznam[sb.back()];             // posledni bod

  for (std::list<int>::const_iterator    // pruchod kontejnerem
         i=sb.begin(), e=sb.end(); i!=e; ++i)
    {
      int k = *i;                        // cislo bezneho bodu
      const Bod& b = seznam[k];          // reference na bezny bod

      if (!b.ma_souradnice()) return -2;  // nedefinovane souradnice
```

```
      s += (b.x() - z.x()) * (b.y() + z.y());
      z  =  b;                            // predchozi bod
    }

  return std::abs(s/2);
}

Bod Polygon::teziste() const
{
  Bod    t;
  double x = 0;
  double y = 0;
  int    N = 0;

  for (std::list<int>::const_iterator
         i=sb.begin(), e=sb.end(); i!=e; ++i)
    {
      const Bod& b = seznam[*i];
      if (!b.ma_souradnice()) return t;

      x += b.x();
      y += b.y();
      N++;
    }

  if (N > 0)
    {
      x /= N;
      y /= N;
      t.vloz_xy(x, y);
    }

  return t;
}
```

Použití třídy `Polygon` demonstruje následující příklad:

```
int main()
{
  std::vector<Bod> seznam(1000);
  seznam[12].vloz_xy(102.23, 133.22);
  seznam[33].vloz_xy(197.38,  42.94);
  seznam[16].vloz_xy(311.25, 143.65);
  seznam[47].vloz_xy(389.26, 250.17);
  seznam[67].vloz_xy(223.53, 308.99);
  seznam[29].vloz_xy(148.83, 223.87);

  Polygon p(seznam);
  p.pridej_bod(12);   p.pridej_bod(33);   p.pridej_bod(16);
  p.pridej_bod(47);   p.pridej_bod(67);   p.pridej_bod(29);

  double d = p.plocha();
  std::cout << std::fixed << std::setprecision(2)
```

```
            << "plocha  : " << d << std::endl;

  Bod t = p.teziste();
  std::cout << "teziste : " << t.x() << " " << t.y() << std::endl;
}

plocha  : 40192.05
teziste : 228.75 183.81
```

Kapitola 2

Některé elementární pojmy

Jazyk C++ je obecný programovací jazyk, který při svém vzniku vycházel z programovacího jazyka C [7]. Až na několik výjimek je jazyk C podmnožinou C++. Oproti jazyku C poskytuje C++ další datové typy, třídy, šablony (templates), výjimky (exceptions), prostory jmen, vložené funkce (inline), přetížení operátorů, přetížení funkcí, reference, operátory pro správu paměti a další knihovní nástroje a bohatou standardní knihovnu.

Autorem jazyka C++ je Bjarne Stroustrup. Počátkem 80. let Bjarne Stroustrup navrhl a realizoval jazyk "C with Classes", který obohatil jazyk C o třídy a některé další rysy a stal se základem pro jazyk C++. Roku 1986 publikoval Bjarne Stroustrup popis jazyka C++ v knize "The C++ Programming Language" [1].

Pro jazyk C++ a jeho uplatnění měl nesmírný význam proces jeho standardizace (ANSI/ISO), který zpětně ovlivnil i standard jazyka C. V současné době je proces standardizace jazyka C++ dovršen [2]. C++ je živý jazyk a proto jeho vývoj bude i nadále pokračovat.

Jazyk C++ podporuje hlavní rysy objektové programování

zapouzdření	(encapsulation)
dědičnost	(inheritance)
polymorfismus	(polymorphism)

Spektrum nasazení jazyka C++ je velice široké. Jazyk C++ byl navržen tak, aby vyhověl požadavkům kladeným na systémové programování a přeložený C++ kód může konkurovat kódu psanému v asembleru. Na druhé straně poskytuje C++ bohaté a účinné nástroje abstrakce jako jsou třídy a šablony. K tomu přistupuje mohutná a efektivní C++ standardní knihovna.

2.1 Překlad

C++ program tvoří zdrojový text, který musíme přeložit a sestavit, abychom jej mohli na daném operačním systému spouštět (procesu sestavení se česky běžně říká *linkování*). Zdrojový text C++ programu sestává obvykle z jednoho nebo více *souborů (file)*. Soubor obsahuje C++ zdrojový text a direktivy preprocesoru. Soubor (soubory) musíme přeložit překladačem jazyka C++ (běžně se též užívá hovorové

synonymum kompilátor z anglického *compiler*). Výsledkem překladu souboru je tzv. relativní modul *(object file)*. Relativní moduly musíme dále sestavit s knihovními soubory a vytvořit z nich spustitelný program. Pokud je text programu uložen ve více souborech, které překládáme samostatně, obvykle definujeme pro jeho překlad/sestavení tzv. *projekt* (např. Borland C++Builder) nebo vytvoříme *makefile* (GNU).

Při překladu souboru jsou nejprve zpracovány direktivy preprocesoru (vložení souborů a expanze maker). Výsledkem je zdrojový text označovaný jako *jednotka překladu (translation unit)*.

Existují systémy, ve kterých nejsou zdrojové C++ kódy ukládány jako soubory, my se ale omezíme na tradiční řešení, kdy zdrojové texty jsou realizovány jako textové soubory, které vytváříme a upravujeme textovým editorem.

2.1.1 Direktivy preprocesoru

Direktivy preprocesoru začínají znakem `#` (mohou před ním být *bílé znaky* — mezera, tabelátor, konec řádku) a musí být ukončeny znakem *konec řádku*. Direktiva preprocesoru může pokračovat na dalším řádku, pokud je posledním znakem na běžném řádku znak \.

Vkládání souborů

Direktiva preprocesoru

```
#include <jméno>
```

je při překladu nahrazena obsahem standardní hlavičky (viz tabulky 10.1 a 10.2), resp. obsahem souboru, jehož jméno je uvedeno jako argument direktivy. Daná implementace definuje, ve kterých adresářích bude soubor hledán. Připomeňme ale, že standardní hlavičky nemusí nutně být realizovány jako zdrojové soubory, viz odstavec 2.1.2.

Podobně je při překladu direktiva

```
#include "jméno-souboru"
```

nahrazena obsahem souboru, jehož jméno je uvedeno jako argument. V tomto případě je obvykle soubor nejprve hledán v běžném adresáři a pokud není nalezen, pokračuje hledání ve stejných adresářích jako v předchozím případě. V obou případech může vkládaný soubor opět obsahovat direktivy `#include`.

Direktiva `#include` se používá především pro vkládání hlavičkových souborů, které obsahují různé deklarace (tříd, funkcí, konstant, template deklarací a pod.), které jsou sdíleny mezi více zdrojovými soubory nebo jsou uloženy v knihovně.

Jistou pozornost musíme věnovat vkládání hlavičkových souborů v programech, které mají být nezávislé na zvolené platformě. Konkrétně v systému Microsoft Windows se nerozlišují velká a malá písmena ve jménech souborů a v direktivě `#include` řetězce `"Test.h"` a `"test.h"` označují jediný soubor, podle konvencí systému Unix jde ale o dva různé soubory.

Dalším potenciálním zdrojem problémů jsou různé symboly používané jako oddělovače souborů anebo adresářů v různých systémech. Například v systémech Microsoft Windows je tímto oddělovačem znak *zpětné lomítko* a pokud chceme direktivou `#include` vložit soubor `test.h`, který je umístěn v podadresáři `adr` můžeme psát

```
#include "adr\test.h"
```

Mapování řetězce specifikujícího jméno souboru v direktivě `#include` na fyzické jméno souboru v daném operačním systému je definováno implementací. Bíle znaky jsou ale významné a předchozí příklad proto může být interpretován jako řetězec *adr* následovaný znakem tabelátor (`'\t'`) a dále řetězcem *est.h* (celkem tedy 9 znaků). Aby bylo možno snadno zamezit podobným rozporům, lze i v systémech Microsoft Windows používat jako oddělovač adresářů symbol lomítko stejně jako například v Unixu a předchozí příklad zapsat ekvivalentně takto

```
#include "adr/test.h"
```

Podmíněný překlad

Direktivy pro podmíněný překlad jsou

```
#if konstantní výraz
#ifdef identifikátor
#ifndef identifikátor

#elif konstantní výraz
#else

#endif
```

V *konstantním výrazu* direktiv `#if` a `#elif` může být použit unární operátor `defined` (česky *je definován*) ve formátu

```
defined identifikátor
```

nebo

```
defined (identifikátor)
```

Direktiva	`#ifdef` *identifikátor* `#ifndef` *identifikátor*	má stejný význam jako	`#if defined` *identifikátor* `#if !defined` *identifikátor*

Je-li v direktivě `#ifdef` použit *identifikátor*, je jeho hodnota `1`, pokud byl definován direktivou

```
#define identifikátor
```

Platnost *identifikátoru* může být zrušena direktivou

```
#undef identifikátor
```

Kromě toho můžeme definovat *identifikátor* pomocí parametru překladače.[1] Překladače také poskytují vlastní implicitně deklarované *identifikátory*. To lze využít pro psaní programů, ve kterých jsou vybrané partie překládány různě, podle toho, jaký použijeme překladač.

[1] Parametr `-D`*identifikátor* pro překladač GNU i Borland. V IDE prostředí překladače Borland pak volby *Options — Project Options — Compiler — Defines*.

```
#include <iostream>

int main()
{
  std::cout << "Ukazka podmineneho prekladu \n";

  #ifdef IDENT_TEST
    std::cout << "Identifikator IDENT_TEST je definovan \n";
  #else
    std::cout << "Identifikator IDENT_TEST neni definovan \n";
  #endif

  #ifdef __GNUC__
    std::cout << "Program byl prelozen kompilatorem GNU C++ \n";
  #elif defined __BORLANDC__
    std::cout << "Program byl prelozen kompilatorem Borland C++ \n";
  #else
    std::cout << "Program byl prelozen jinym kompilatorem C++ \n";
  #endif

  std::cout << "... konec ukazky \n";
}
```

Dalším příkladem uplatnění techniky podmíněného překladu je zápis různých ladicích příkazů v textu programu, které slouží pouze ve fázi ladění programu (kdy explicitně požadujeme jejich překlad) a v konečné fázi se nepoužijí.

Další direktivy preprocesoru jsou

```
#line   řetězec
#error  řetězec
#pragma řetězec
#
```

Direktiva `#` je prázdná direktiva, která nic nedělá. Direktiva `#pragma` určuje implementačně závislé chování, není-li v dané implementaci direktiva `#pragma` definována, je překladačem ignorována. Direktiva `#error` při překladu vypíše zadanou chybovou zprávu a označí danou jednotku překladu za chybnou *(ill-formed)*. Direktiva `#line` umožňuje přenastavit běžné číslo řádku anebo jméno daného zdrojového souboru, viz dále.

Makra

Direktivou `#define` můžeme definovat makra bez parametrů nebo makra s parametry. Jejich význam v C++ je ve srovnání s jazykem C výrazně menší. V jazyce C++ je většinou vhodnější nahradit je ekvivalentní `inline` funkcí (vloženou funkcí) nebo pojmenovanou konstantou.

Příkladem makra bez parametru je

```
#define MAXDIM 80
```

Ve zdrojovém textu následujícím tuto direktivu nahradí preprocesor každý výskyt identifikátoru `MAXDIM` literálem `80`. Stejného efektu ale dosáhneme C++ deklarací

```
const int MAXDIM = 80;
```

Navíc bude tato hodnota přístupná *debuggeru*, zatímco symboly definované v direktivách preprocesoru obecně nejsou.

Příklad makra s parametry je

```
#define max(x,y) ((x) > (y) ? (x) : (y))
```

které pro typ `int` může být nahrazeno bez ztráty efektivnosti C++ vloženou funkcí

```
inline int max(int x, int y)
{
  return (x > y ? x : y);
}
```

Deklarace `template` umožňuje napsat podobné funkce pro libovolný typ T (v daném konkrétním případě musí mít typ T definován operátor >).

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

Obecně jsou makra zdrojem potenciálních problémů. Jestliže například

```
const int MAXDIM = 80;

void f() {
   const int MAXDIM = 100;
   // ...
}
```

funguje podle očekávání, pak varianta

```
#define MAXDIM 80

void f() {
   const int MAXDIM = 100;
   // ...
}
```

skončí při překladu chybou, protože preprocesor pouze mechanicky nahradí jméno lokální proměnné s výsledkem `const int 80 = 100;`.

Protože preprocesor probíhá před vlastním předkladem, možňuje navíc definovat i různé zvrácenosti, jako je příklad následujícího makra, které otevírá nekontrolovaný přístup k privátním členům.

```
#define private public
```

Předdefinovaná makra

Následující předdefinovaná makra poskytují informace o dané jednotce překladu

`__LINE__`	běžné číslo řádku v daném souboru
`__FILE__`	jméno zdrojového textu
`__DATE__`	datum překladu daného zdrojového textu
`__TIME__`	čas překladu daného souboru
`__STDC__`	implementačně závislá hodnota; nemusí být definována
`__cplusplus`	nastaveno při překladu C++ zdrojového textu

Příklad:

```
#include <iostream>

int main()
{
  std::cout << "__LINE__ : " << __LINE__ << "\n";
  std::cout << "__FILE__ : " << __FILE__ << "\n";
  std::cout << "__DATE__ : " << __DATE__ << "\n";
  std::cout << "__TIME__ : " << __TIME__ << "\n\n";

  #line 100 "abc... "
  std::cout << "__LINE__ : " << __LINE__ << "\n";
  std::cout << "__FILE__ : " << __FILE__ << "\n";
  std::cout << "__DATE__ : " << __DATE__ << "\n";
  std::cout << "__TIME__ : " << __TIME__ << "\n\n";

  #ifdef __STDC__
  std::cout << "makro __STDC__ \n";
  #endif

  #ifdef __cplusplus
  std::cout << "makro __cplusplus \n";
  #endif
}
```

```
__LINE__ : 5
__FILE__ : preproc2.cpp
__DATE__ : Aug 14 2004
__TIME__ : 12:07:30

__LINE__ : 100
__FILE__ : abc...
__DATE__ : Aug 14 2004
__TIME__ : 12:07:30

makro __STDC__
makro __cplusplus
```

Pravidlo jediné definice

C++ program nesmí obsahovat více než jednu definici proměnné, funkce, třídy, výčtového typu nebo šablony. Například definice dané třídy se vyskytuje pouze jedinkrát v nějakém souboru. Protože taková definice ale může být zařazena direktivou `#include` do dalších souborů, říká *pravidlo jediné definice (One Definition rule)*, že dvě definice třídy, šablony nebo vložené funkce jsou přípustné

- pokud se nacházejí v různých jednotkách překladu a
- pokud jsou z pohledu překladače identické, tj. všechny *symboly (tokens)* jsou identické a mají stejný význam v obou jednotkách překladu.

Direktivu `#define` budeme převážně používat pro definování symbolů pro řízení podmíněného překladu, především při psaní hlavičkových souborů pro zajištění pravidla jediné definice *(include guards)*. Běžný postup je ten, že hlavičkový soubor začíná direktivou `#ifndef` s vhodným identifikátorem (odvozeným od jména hlavičkového souboru), následuje direktiva `#define`, vlastní text hlavičkového souboru a končí `#endif`

```
#ifndef ukazka_h____verze_2004_07_06__18_08___
#define ukazka_h____verze_2004_07_06__18_08___

/*
  zde je uveden text hlavickoveho souboru "ukazka.h"

  identifikator ukazka_h____verze_2004_07_06__18_08___ nesmi byt jinde
  definovan a casto se proto pouzivaji velmi dlouhe identifikatory,
  ktere obsahuji informace jako je datum, cas a pod.
*/
#endif
```

2.1.2 Standardní C++ hlavičky

Prvky standardní C++ knihovny jsou deklarovány ve 32 *hlavičkách (headers)*. Jejich přehled uvádí tabulka 10.1. Pro přístup ke standardním funkcím jazyka C je dále k dispozici 18 hlaviček uvedených v tabulce 10.2. Standardní hlavičky mohou být realizovány jako *hlavičkové soubory (header files)*. Podle standardu C++ [2] hlavičky ale nemusí nutně být zdrojové soubory a pokud jsou, nemusí jejich jména přímo korespondovat se jmény příslušných souborů.

V jazyce C++ se na hlavičky odkazujeme direktivou preprocesoru

```
#include <hlavička>
```

Obsah hlavičky *cjméno* je stejný jako obsah korespondující hlavičky *jméno.h* programovacího jazyka C. Rozdíl mezi nimi je v tom, že deklarace z hlavičky *cjméno* mají prostor jmen (namespace) `std`. Prostory jmen jsou popsány v odstavci 3.9.

Často se ještě setkáme s C++ programy, které pracují s direktivami `#include` s explicitně uvedenými příponami hlavičkových souborů, například

```
#include <math.h>
```

Pokud to ale není nezbytné (například při překladu starších programů), měli bychom používat pouze standardní C++ hlavičky odkazující na prostor jmen `std` (tj. hlavičky bez přípony `.h`).

2.2 Komentáře

Dvojice znaků `/*` zahajuje komentář, který je ukončen dvojicí znaků `*/`. Tyto komentáře mohou pokračovat na více řádcích, ale nesmí být vnořovány.

```
/* Zde zacina komentar
      /* !!! Toto je vnoreny komentar: v C++ nelze !!! */
*/
```

Přestože některé kompilátory vnořené komentáře připouštějí, je vhodné se držet standardu jazyka C++ a nepoužívat je.

Dvojice znaků `//` uvozuje druhý typ komentáře, který je ukončen koncem řádku (znak *new-line*, který v C++ označuje znakový literál `'\n'`).

Komentářové znaky `//` nemají žádný zvláštní význam uvnitř komentáře `/* ... */`. Obdobně v komentáři uvozeném dvojicí lomítek `//` nemají žádný zvláštní význam znaky `/*` a `*/`.

2.3 Identifikátory, klíčová slova a oddělovače

Identifikátor *(identifier)* je posloupnost písmen a číslic začínající písmenem. Písmeny se přitom obvykle rozumí písmena anglické abecedy, tj. písmena bez diakritiky, ale standard připouští i univerzální znaky v kódování ISO 10646. Za písmeno je považován také znak podtržítko (_). Rozlišují se velká a malá písmena. Délka identifikátoru není obecně omezena, všechny znaky jsou významné (to ale neplatí ve všech implementacích). Jako identifikátor nemůže být použito klíčové slovo.

Klíčová slova *(keywords)* jsou rezervovaná a nemohou být použita jinak. Jejich přehled uvádí tabulka 2.1. Tabulka 2.2 uvádí přehled operátorů, další rezervovaná slova pro jejich alternativní vyjádření a oddělovače *(punctuator)* jazyka C++.

asm	do	inline	short	typeid
auto	double	int	signed	typename
bool	dynamic_cast	long	sizeof	union
break	else	mutable	static	unsigned
case	enum	namespace	static_cast	using
catch	explicit	new	struct	virtual
char	extern	operator	switch	void
class	false	private	template	volatile
const	float	protected	this	wchar_t
const_cast	for	public	throw	while
continue	friend	register	true	
default	goto	reinterpret_cast	try	
delete	if	return	typedef	

Tabulka 2.1: Klíčová slova

Použití identifikátorů obsahujících dvě podtržítka (__) a identifikátorů začínajících podtržítkem za kterým následuje velké písmeno je rezervováno pro implementace C++ a použití ve standardních knihovnách a

{	}	[	]	#	##	(	)	
?	::	.	.*	;	:	...	new	delete
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->

Tabulka 2.2: Operátory a oddělovače

uživatelé by neměli takové identifikátory používat. Právě tak se nedoporučuje používat identifikátory začínající znakem podtržítko, protože jsou rezervovány pro implementace jazyka C.

2.3.1 Možnost alternativního zápisu

Jazyk C++ umožňuje nahradit ve zdrojovém kódu některé speciální znaky jako `[`, `]`, `{`, `}`, `|`, `\`, a vybrané operátory alternativním zápisem pomocí tzv. trigrafů, digrafů a rezervovaných slov. Je to motivováno tím, že pozice zmíněných speciálních ASCII znaků mohou být používány pro zápis národních znaků jako jsou například Dánské znaky Æ, æ, Ø, ø, Å či å a nemusí proto být pro programátora k dispozici. O možnosti alternativního zápisu se zde zmiňujeme pouze pro úplnost, přehled uvádí tabulka 2.3.

klíčová slova		digrafy		trigrafy	
and	&&	<%	{	??=	#
and_eq	&=	%>	}	??(	[
bitand	&	<:	[	??<	{
bitor	|	:>	]	??/	\
compl	~	%:	#	??)	]
not	!	%:%:	##	??>	}
or	||			??'	^
or_eq	|=			??!	|
xor	^			??-	~
xor_eq	^=			???	?
not_eq	!=				

Tabulka 2.3: Možnost alternativního zápisu

2.4 Literály

Literály slouží pro přímý zápis konstantních hodnot základních typů v C++ programu (někdy se jim proto říká konstanty). Literály jsou v C++ těchto typů

- celočíselné literály
- znakové literály

- reálné literály
- řetězcové literály
- boolovské literály

2.4.1 Celočíselné literály

Celočíselné literály (integer literals) mohou být dekadické (decimal), oktalové (octal) a hexadecimální (hexadecimal).

dekadické literály tvoří neprázdná posloupnost dekadických číslic

`0 1 2 3 4 5 6 7 8 9`

která nezačíná nulou.

oktalové literály začínají nulou a tvoří je posloupnost oktalových číslic

`0 1 2 3 4 5 6 7`

hexadecimální literály začínají dvojicí znaků `0x` nebo `0X` za kterou následuje neprázdná posloupnost hexadecimálních číslic

`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

Příklad:

```
20        // dekadicky
024       // oktalově
0x14      // hexadecimálně
```

Celočíselné literály mohou mít sufix (příponu) `u` nebo `U` specifikující explicitně typ `unsigned`, sufix `l` nebo `L` specifikující typ `long` nebo oba sufixy specifikující typ `unsigned long` (nezáleží na pořadí ani na kombinaci velkých a malých písmen). Doporučuje se dávat přednost sufixu `L` před sufixem `l`, který lze při čtení snadnou zaměnit za číslici `1`.

Příklad:

```
128u   1024UL   1L   8Lu
```

Jestliže dekadický literál nemá sufix, je jeho typ první z následujících typů, ve kterém může být reprezentován:

`int, long int, unsigned long int.`

Dekadický celočíselný literál bez sufixu tedy nikdy není `unsigned int`.

Jestliže oktalový nebo hexadecimální literál nemá sufix, je jeho typ první z následujících typů, ve kterém může být reprezentován:

`int, unsigned int, long int, unsigned long int.`

Jestliže má celočíselný literál sufix `u` nebo `U`, je jeho typ první z následujících typů, ve kterém může být reprezentován:

`unsigned int, unsigned long int.`

Jestliže má celočíselný literál sufix `l` nebo `L`, je jeho typ první z následujících typů ve kterém může být reprezentován:

`long int, unsigned long int.`

Jestliže má celočíselný literál sufix některý ze sufixů `ul, lu, uL, Lu, Ul, lU, UL,` nebo `LU`, je typu `unsigned long int`.

2.4.2 Znakové literály

Znakové literály (character literals) tvoří znaky zapsané mezi dvojici apostrofů. Například

`'a' 'G' '2' '=' '#' ' '`

poslední z uvedených literálů je znak mezera. Takovéto znakové literály jsou typu `char`.

V apostrofech může být uvedeno i více znaků. V tom případě jde o víceznakové literály (multicharacter literals), které jsou typu `int` a jejichž hodnota je implementačně závislá.

Znakový literál může začínat písmenem `L`, například `L'x'`. V takovém případě jde o *široký znak (wide-character)* a jeho typ je `wchar_t`. Tyto znakové literály jsou určeny pro znakové sady, kde jeden znak nelze uložit do jednoho bajtu. Jejich hodnota je implementačně závislá.

Některé nezobrazitelné znaky, apostrof, uvozovky, otazník a zpětné lomítko (backslash) zapisujeme tak, že mezi dvojici apostrofů zapíšeme *řídící sekvenci (escape sequence)* znaků. Přehled řídících sekvencí C++ uvádí tabulka 2.4. Znaky uvozovky a otazník mohou být vyjádřeny přímo jako `'"'` a `'?'` nebo pomocí řídící sekvence `'\"'` a `'\?'`. Řídící sekvence také umožňují zapsat znak jeho oktalovou nebo hexadecimální hodnotou.

2.4.3 Reálné literály

Reálné literály (floating literals) se skládají z celočíselné části, desetinné tečky, zlomkové části, písmene `e` nebo `E`, celočíselného exponentu (může mít znaménko) a volitelného sufixu.

Celočíselnou a zlomkovou část tvoří posloupnost dekadických číslic. Celočíselná nebo zlomková část může chybět (ale ne obě). Dále může chybět desetinná tečka nebo písmeno `e`, resp. `E` a exponent (ale ne obojí).

nový řádek	new-line	NL (LF)	\n
horizontální tabelátor	horizontal tab	HT	\t
vertikální tabelátor	vertical tab	VT	\v
posun o jeden znak zpět	backspace	BS	\b
návrat vozíku (kurzoru)	carriage return	CR	\r
nová stránka	form feed	FF	\f
zvukový signál	alert	BEL	\a
zpětné lomítko	backslash	\	\\
otazník	question mark	?	\?
apostrof	single quote	'	\'
uvozovky	double quote	"	\"
oktalová hodnota	octal number	ooo	\ooo
hexadecimální hodnota	hex number	hhh	\xhhh

Tabulka 2.4: C++ znakové řídící sekvence

Exponent vyjadřuje mocninu čísla 10, kterou je vynásobena hodnota tvořená celočíselnou a desetinnou částí literálu.

Typ reálného literálu bez sufixu je `double`. Reálné literály se sufixem `f` nebo `F` jsou typu `float`. Reálné literály se sufixem `l` nebo `L` jsou typu `long double`.

Příklad:

```
43.463   70.0   1.   .278   54.2e4   1.23e-4   4.0f   1.3L
```

2.4.4 Řetězcové literály

Řetězcové literály (string literals) tvoří posloupnost znaků uzavřená dvojicí uvozovek. Tato posloupnost může být i prázdná. Typ řetězcového literálu je *pole o n složkách typu `const char`* a řetězcový literál je vždy ukončen nulovým bajtem.

```
"ukazka bezneho retezce"
```

Volitelně může řetězcový literál předcházet písmeno `L`. Jeho typ pak je *pole o n složkách typu `const wchar_t`*.

```
L"ukazka retezce znaku typu wchar_t"
```

Sousední řetězce oddělené pouze bílými znaky jsou při překladu spojeny do jediného řetězce.

```
"Toto je jediny retezec"
" zapsany na dvou radcich"
```

Každý řetězcový literál je ukončen znakem `'\0'`. Tento nulový bajt doplňuje do řetězcového literálu automaticky překladač. Pro práci s C-řetězci ukončenými znakem `'\0'` jsou určeny funkce deklarované ve standardní hlavičce `<cstring>`. Velikost běžného řetězce (bez prefixu `L`) je tedy *počet znaků* $+ 1$.

V řetězcích můžeme používat stejné řídící sekvence jako ve znakových literálech, viz tabulka 2.4. Apostrof v řetězci může být zapsán přímo `"'"` nebo pomocí řídící sekvence `"\'"`.

```
cout << "Priklad retezce,\n\tkterý je vypsan na dva radky\n";
```

2.4.5 Boolovské literály

Boolovské literály jsou `false` a `true`. Jsou typu `bool`.

2.5 Paměť

Paměť počítače je logicky členěna na bajty (byte). Jednotlivé bajty jsou v paměti očíslovány, těmto číslům říkáme *adresy*. Bajt obvykle obsahuje osm bitů (dvojkových číslic) a je nejmenší adresovatelnou jednotkou paměti.

Všechny objekty (proměnné, funkce) přímo dostupné programu musí být uloženy v paměti počítače. Objekt je v paměti uložen v prostoru n bajtů (minimálně zabírá jeden bajt). Každý objekt je tedy jednoznačně identifikován svojí adresou (adresou prvního bajtu, od kterého je uložen a svým typem). Kromě toho mohou mít objekty (proměnné) přiděleno v programu jméno. C++ literály jméno nemají a v C++ programu nelze určit ani jejich adresu.

Žádný objekt v C++ nemá adresu `0`.

Obsah paměti sám o sobě nemá žádnou interpretaci, můžeme se na ni dívat jako na velmi dlouhou posloupnost dvojkových číslic 0 a 1 (resp. jako na posloupnost binárních stavů). Pro daný objekt tedy nestačí mít pouze jeho adresu, ale musíme znát také jeho typ. Typ určuje, kolik bajtů objekt v paměti zabírá a jaká je jejich interpretace. Běžně totiž například typ `int` zabírá čtyři bajty stejně jako typ `float`, ale interpretace jednotlivých bajtů je různá.

V následující ukázce funkce `bt()` vypisuje hodnoty zadaných literálů (typu `int`, `float` a `char`) a po bitech obsah bajtů, ve kterých jsou uloženy. Prozatím pro nás není důležité, jak je funkce `bt()` napsána, ale výstup programu.

```
#include <iostream>
#include <iomanip>

template <typename T> void bt(T x)
{
  std::cout.setf(std::ios_base::fixed);
  std::cout.precision(5);
  std::cout << std::setw(14) << x << " ";

  int poc_b = sizeof(x);
  unsigned char* pb = (unsigned char*)&x;

  do {
    std::cout << " ";
    unsigned char m = '\001'; m <<= 7;
    for (long int i = 1; i <= 8; i++, m >>= 1)
      if (*pb & m) std::cout << "1"; else std::cout << "0";
    pb++;
  } while (--poc_b);

  std::cout << std::endl;
}
```

```
int main()
{
  bt(0); bt(-1); bt(1); bt(2); bt(3); bt(-3); bt(1053); bt(-1053);
  std::cout << std::endl;
  bt(0.0f); bt(1.0f); bt(1.5f); bt(1.25f); bt(1.125f);
  bt(20.1f); bt(-20.1f);
  std::cout << std::endl;
  bt('a'); bt('A'); bt('B'); bt('0'); bt('1'); bt('2');
}
```

V první skupině jsou zobrazeny bitové reprezentace celočíselných literálů typu `int` (na čtyřech bajtech jsou hodnoty reprezentovány v tzv. *dvojkovém doplňkovém kódu + 1*). Druhá skupina zobrazuje reprezentaci literálů typu `float` (ve čtyřech bajtech je uloženo znaménko, exponent a normalizovaná mantisa). Poslední skupiny tvoří výpis literálů typu `char`.

```
           0  00000000 00000000 00000000 00000000
          -1  11111111 11111111 11111111 11111111
           1  00000001 00000000 00000000 00000000
           2  00000010 00000000 00000000 00000000
           3  00000011 00000000 00000000 00000000
          -3  11111101 11111111 11111111 11111111
        1053  00011101 00000100 00000000 00000000
       -1053  11100011 11111011 11111111 11111111

     0.00000  00000000 00000000 00000000 00000000
     1.00000  00000000 00000000 10000000 00111111
     1.50000  00000000 00000000 11000000 00111111
     1.25000  00000000 00000000 10100000 00111111
     1.12500  00000000 00000000 10010000 00111111
    20.10000  11001101 11001100 10100000 01000001
   -20.10000  11001101 11001100 10100000 11000001

           a  01100001
           A  01000001
           B  01000010
           0  00110000
           1  00110001
           2  00110010
```

Na různých systémech je obecně reprezentace jednotlivých typů (například `int`) implementována různě. Pro nás je především důležité vědět, že reprezentaci číselných hodnot tvoří vždy jistý počet bajtů, tj. číselná hodnota je vyjádřena jistým omezeným počtem dvojkových číslic. Z toho je zřejmé, že přesnost zobrazení vyjádřena v dekadické hodnotě je také omezená (zde například řádově 6–7 platných cifer pro typ `float`).

2.6 Podmínky a cykly

Řízení chodu C++ programu zahajuje první příkaz funkce `main()`; ponechme prozatím stranou skutečnost, že před zahájením funkce `main()` jsou volány konstruktory globálních objektů.

Jednotlivé příkazy se provádějí za chodu sekvenčně, tj. postupně v tom pořadí, jak jsou zapsány ve zdrojovém textu. V C++ jsou příkazy i deklarace proměnných (deklarační příkazy). Pořadí provádění příkazů se mění při volání funkce, kdy je řízení předáno volané funkci. Po jejím ukončení program pokračuje následujícím příkazem. O funkcích budeme hovořit v kapitole 4. Kromě toho může být předáno řízení jinam, pokud daná funkce vyvolá *výjimku (exception)*. Výjimkami se budeme zabývat v kapitole 8.

Implicitní sekvenční posloupnost vykonávání příkazů můžeme změnit pomocí příkazů pro větvení programu `if` a `switch` a příkazy cyklů `for`, `while` a `do`. Přenos řízení umožňují příkazy `break` a `continue` (návrat z funkce zajišťuje příkaz `return`).

Jazyk C++ má příkaz skoku `goto` *návěští;* návěští označuje příkaz, kterému může být předáno řízení příkazem skoku (návěští je od příkazu odděleno dvojtečkou).

Existují výjimečné situace, kdy je použití příkazu skoku oprávněné. Příkaz skoku představuje ale ideální nástroj pro zatemnění smyslu programu a potenciální zdroj komplikací.

Začátečník, který se zoufale pokouší „rozchodit“ svůj program pomocí jednoho či více magických příkazů `goto` vždy demonstruje, že nemá valné ponětí o tom, co právě provádí. Snažte se příkaz skoku nepoužívat.

2.6.1 Příkaz `if`

Příkaz `if` slouží pro výběr z jedné nebo z více cest řízení chodu programu. Jeho syntax je

```
if ( podmínka ) příkaz
if ( podmínka ) příkaz1 else příkaz2
```

V prvním případě nám umožňuje zvolit, zda při splnění podmínky má být proveden *příkaz*. Druhá forma příkazu `if` umožňuje rozhodnout mezi dvěma příkazy; je-li podmínka splněna, provede se první příkaz, v opačném případě se provede druhý příkaz uvedený za klíčovým slovem `else`.

Často potřebujeme zvolit na základě podmínky vykonání více než jednoho příkazu. V takovém případě použijeme jako *podpříkaz* příkazu `if` *blok (block)*. Blok je také označován jako *složený příkaz (compound statement)*. Blok tvoří dvojice složených závorek `{}`, ve složených závorkách zapisujeme posloupnost příkazů a deklarací proměnných. Proměnné deklarované v bloku jsou v něm lokální a přestávají existovat po ukončení bloku. Podrobněji viz odstavec 4.2.1.

Příkaz `if` může jako podpříkaz obsahovat opět příkaz `if`, můžeme tedy zapisovat posloupnost podmínek a jim odpovídajících příkazů jako

```
if ( podmínka-1 )
   příkaz-1
else if ( podmínka-2 )
   příkaz-2
...
else if ( podmínka-n )
   příkaz-n
else
   příkaz
```

Je třeba přitom dávat pozor na středníky ukončující příkazy, složený příkaz se středníkem neukončuje. Zápis

```
{ /* ... */ } ;
```

jsou dva příkazy — první je blok, druhý je *prázdný* příkaz.

V příkazu `if` může *podmínka* být

výraz
specifikace-typu deklarátor `=` *přiřazovací-výraz*

Číselný výraz různý od nuly je chápán jako pravdivý výraz. Deklarátor nesmí být funkce nebo pole; specifikace typu nesmí obsahovat `typedef`, deklaraci nové třídy nebo výčtového typu. Jméno deklarované v *podmínce* má platnost od místa deklarace do konce příkazu `if`.

Hodnota *podmínky,* ve které byla deklarována proměnná, je rovna hodnotě inicializačního výrazu konvertovaného implicitně na typ `bool` (pro příkaz `switch` platí jiná pravidla).

```
if (int x = f()) {
   cout << "hodnota f() je " << x << endl;
   // ...
}
else {
   cout << "funkce f() rovna nule" << endl;
   // ...
}
// ... zde již proměnná x, deklarovaná v podmínce příkazu if, neexistuje
```

Starší překladače jazyka C++ možnost deklarace proměnné v podmínce příkazů `if`, `switch` a `while` nepřipouštěly.

2.6.2 Příkaz `switch`

Syntax příkazu `switch` je

`switch (` *podmínka* `)` *příkaz*

Podmínka příkazu `switch` musí být celočíselného typu, výčtového typu nebo typu třída, pro kterou existuje přímá konverze na celočíselný nebo výčtový typ.

Příkazem je obvykle blok. Kterýkoliv příkaz v bloku může být označen jedním nebo více návěštími ve tvaru

`case` *celočíselný-konstantní-výraz* `:`

Příkaz `switch` předá řízení prvnímu příkazu s návěštím shodným s *podmínkou*. Nanejvýš jeden příkaz může mít návěští

```
default:
```

Pokud je návěští `default` uvedeno, předá se na něj řízení v tom případě, že není nalezena shoda s žádným výrazem uvedeným v návěští `case`.

Návěští `case` v bloku příkazu `switch` označují pouze možný začátek větvení chodu programu, ale ne ukončení řídící struktury. Pro explicitní ukončení vybrané větve programu slouží příkaz `break`.

```
#include <iostream>

int main()
{
  char z;
  std::cin >> z;

  switch (z)
    {
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      std::cout << "zadana cislice" << std::endl;
      break;

    default:
      std::cout << "zadan znak " << z << std::endl;
    }
}
```

Volba mezi příkazy `switch` a `if` často závisí převážně na tom, který zápis bude čitelnější. Použití příkazu `switch` při více podmínkách někdy představuje kompaktnější zápis. Možnost opomenutí zápisu příkazu `break` ale vede na nenápadné chyby, které nemůže při překladu odhalit kompilátor a použití příkazu `switch` v tomto smyslu vyžaduje zvýšenou pozornost. Pomoci může dobrá grafická úprava zdrojového textu.

2.6.3 Příkazy `while`, `do` a `for`

Iterační příkazy `while`, `do` a `for` slouží pro programování cyklů a iterací. Jejich syntax je

`while (` *podmínka* `)` *příkaz*
`do` *příkaz* `while (` *výraz* `) ;`
`for (` *for-init-příkaz* *podmínka*$_{vol}$ `;` *výraz*$_{vol}$ `)` *příkaz*

Podpříkaz iteračního příkazu implicitně definuje lokální oblast platnosti, která je vytvořena a ukončena s každým průchodem cyklu. Lokální nestatická jména deklarovaná v podpříkazu jsou tedy vytvářena a rušena v každé iteraci znovu.

Pokud je podpříkazem iteračního příkazu jednoduchý příkaz a ne složený příkaz, je to totéž, jako by původní podpříkaz byl přepsán do složeného příkazu. Příkaz

```
while (--x >= 0)
    int i;
```

může být ekvivalentně přepsán jako

```
while (--x >= 0) {
    int i;
}
```

Po provedení příkazu `while` přestává proměnná `i` existovat.

Příkaz `while`

Příkaz `while` se provádí, dokud je specifikovaná podmínka splněna, podpříkaz se nemusí provést ani jednou, není-li daná podmínka splněna (nulový počet iterací). Pokud je podmínkou deklarace, je podobně jako v příkazu `if` oblast platnosti deklarované proměnné lokální od místa deklarace do konce příkazu `while`. Příkaz `while` ve tvaru

```
while (T t = x) příkaz
```

je ekvivalentní

```
návěští:
{             // začátek oblasti platnosti podmínky
    T t = x;
    if (t) {
        příkaz
        goto návěští;
    }
}             // konec oblasti platnosti podmínky
```

Objekt deklarovaný v podmínce je vytvořen a zrušen vždy s každým průchodem cyklu.

```
#include <iostream>

class A {
   int val;
public:
   A(int i) : val(i) { std::cout << "A::ctor val = " << val << "\n"; }
   ~A()              { std::cout << "A::dtor val = " << val << "\n"; }
   operator bool()   { return val != 0; }
};

int main()
{
   int i = 1;
   while (A a = i) {
      // ...
      i = 0;
   }
}
```

V příkladu cyklu `while` jsou konstruktor a destruktor volány dvakrát, jednou pro podmínku, která je splněna, a podruhé pro podmínku, která splněna není.

```
A::ctor val = 1
A::dtor val = 1
A::ctor val = 0
A::dtor val = 0
```

Příkaz `while` můžeme použít například pro výpočet odhadu součtu řady

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

```
double tol = 1e-8;
double fakt = 3, dif = x, soucet = x;
while (abs(dif) > tol) {
  dif    *= -x*x / fakt / (fakt-1);
  soucet += dif;
  fakt   += 2;
}
```

Příkaz `do`

Výraz v příkazu

`do` *příkaz* `while (` *výraz* `) ;`

se implicitně převádí na typ `bool`. Příkaz `do` umožňuje programování cyklů podobně jako příkaz `while`, podmínka se ale testuje až po provedení *příkazu*, tj. vždy na konci každé iterace. Podpříkaz příkazu `do` se proto provede vždy alespoň jednou.

Předchozí příklad bychom mohli přepsat s využitím příkazu `do` takto

```
double tol = 1e-8;
double fakt = 3, dif = x, soucet = x;
do {
  dif    *= -x*x / fakt / (fakt-1);
  soucet += dif;
  fakt   += 2;
} while (abs(dif) > tol);
```

Příkaz `for`

Příkaz cyklu

`for (` *for-init-příkaz* $podmínka_{vol}$ `;` $výraz_{vol}$ `)` *příkaz*

bychom mohli nahradit zápisem využívajícím příkaz `while` takto

```
{
   for-init-příkaz
   while ( podmínka ) {
      příkaz
      výraz ;
   }
}
```

For-init-příkaz je příkaz (výraz ukončený středníkem — *expression statement*) nebo deklarační příkaz. Pokud je *for-init-statement* deklarace, je oblast platnosti jména od místa deklarace do konce příkazu `for`.

Jak *podmínka* tak *výraz* mohou v příkazu `for` chybět. Chybějící *podmínka* je ekvivalentní logické konstantě `true`. Následující zápis proto představuje nekonečný cyklus.

```
for (;;) { /* ... */ }
```

Příkaz `for` můžeme použít například pro výpočet skalárního součinu dvou aritmetických vektorů

$$(a, b) = \sum_{i=1}^{n} a_i b_i$$

V následující ukázce vektory deklarujeme jako objekty třídy `Vec`, aktuální dimenzi vektorů zjišťuje metoda `dim()`.

```
Vec a, b;                              // a, b jsou objekty typu vektor
cin >> a >> b;                         // načtu a, b

double sks = 0;                        // před výpočtem sumace musím vynulovat
if (a.dim() == b.dim()) {              // dimenze obou vektoru se musí shodovat

  for (int i = 1; i <= a.dim(); i++)   // iterace pro všechny prvky vektoru
    sks += a[i] * b[i];                // výpočet skalárního součinu

}
```

2.6.4 Příkazy `break` a `continue`

Příkaz `break` můžeme používat v příkazu `switch` a v iteračních příkazech. Při provedení příkazu `break` je předáno řízení za iterační příkaz nebo příkaz `switch`. Příkazem `break` můžeme tedy předčasně ukončit vykonávání cyklu.

Příkaz `break` neumožňuje ukončit dva nebo více vnořených cyklů. V situacích, kdy potřebujeme ukončit provádění několika vnořených cyklů je oprávněné použití příkazu `goto`.

Příkaz `continue` může být použit pouze v iteračním příkazu. Jeho provedení způsobí předčasné ukončení probíhajícího cyklu a bezprostřední zahájení cyklu následujícího.

Použití příkazu `continue` v iteračních příkazech `while`, `do` a `for` je ekvivalentní příkazu `goto` *pokračuj* v následujících ilustračních příkladech.

```
while (podmínka) {          do {                    for (;podmínka;) {
 {                           {                       {
   // ...                      // ...                // ...
   goto pokračuj;              goto pokračuj;              goto pokračuj;
 }                           }                       }
pokračuj: ;                 pokračuj: ;             pokračuj: ;
}                           } while (podmínka);     }
```

Kapitola 3

Základní a odvozené typy

V této kapitole se seznámíme se základními a odvozenými typy jazyka C++. V kapitole 5 věnované třídám si řekneme, jak v jazyce C++ můžeme definovat vlastní uživatelské typy — třídy, které jsou základem objektového programování. Od základních typů a od tříd (uživatelem definovaných typů) můžeme odvodit následující typy: ukazatele, reference, pole, struktury a unie. Mezi odvozené typy dále řadíme výčtové typy.

V jazyce C++ má každý objekt a každý výraz jednoznačně daný typ. Říkáme například, že proměnná `x` je typu `int`, výraz `x+1.2` je typu `double` a podobně. Typ nese informace o tom, jakých hodnot může daná proměnná, resp. výraz, nabývat, a jaké operace můžeme s daným typem provádět. Jiné operace jsou definovány pro základní celočíselný typ `int` a jiné pro typ `double*` (ukazatel na typ `double`). Kromě jednoduchých základních typů můžeme v C++ pracovat i s velmi složitými odvozenými typy, jako je například kontejner `std::map`.

3.1 Základní typy

Základní typy (built-in/fundamental types) můžeme rozdělit do následujících skupin:

- celočíselné typy { `signed`, `unsigned` } { `char`, `short int`, `int`, `long int` }
- reálné typy { `float`, `double`, `long double` }
- boolovský typ `bool`
- znakový typ `wchar_t`
- prázdný typ `void`

O znakovém typu `wchar_t` jsme se zmínili v odstavci 2.4.2 věnovaném literálům na straně 39.

Typ `void` použijeme v definicích funkcí, které nevracejí hodnotu. Nelze definovat proměnné typu `void`, lze ale definovat ukazatele typu `void*`.

Klíčové slovo `signed` označuje *celočíselné typy (integral types)* se znaménkem, tj. typy, které mohou nabývat záporných i kladných hodnot. Klíčové slovo `unsigned` označuje celočíselné typy, které mohou nabývat pouze nezáporných hodnot.

V označení celočíselného typu je specifikace `signed`, resp. `unsigned`, nepovinná; implicitní hodnota je `signed`. Dále lze v označení celočíselného typu vynechat klíčové slovo `int` (pokud zbude alespoň jedno klíčové slovo).

Limitní hodnoty a další charakteristiky základních typů jsou implementačně závislé. Tyto charakteristiky jsou uvedeny v hlavičkách `<limits>`, `<climits>` a `<cfloat>` (viz str. 197).

Základní číselné typy můžeme volně kombinovat ve výrazech, kompilátor zajistí automaticky potřebné konverze. Při přiřazení mezi různými typy se také uplatní implicitní konverze, může ale dojít ke ztrátě informace (například při přiřazení hodnoty typu `double` do proměnné typu `float` nebo do proměnné některého z celočíselných typů).

Při psaní programů je třeba se vyvarovat implementačně závislých konstrukcí, jako je například

```
unsigned p = -1;   // přiřazení -1 do proměnné typu unsigned int
```

Operátor `sizeof`

Přesnost, resp. rozsah, základních typů závisí především na počtu bajtů (byte), které daný typ využívá. Pro zjištění počtu bajtů alokovaných pro daný typ nebo objekt slouží operátor `sizeof`, jehož syntax je

`sizeof` *unární_výraz*

nebo

`sizeof (` *označení typu* `)`

Pojem *bajt (byte)* se v jazyce C++ vyskytuje pouze v souvislosti s operátorem `sizeof`. Výraz `sizeof (char)` je vždy roven 1.

Pro základní číselné typy platí

```
1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
sizeof(I) ≡ sizeof(signed I) ≡ sizeof(unsigned I)
```

kde `I` je `char`, `short`, `int` nebo `long`.

Hodnota `sizeof(bool)` nemusí být rovna 1.

Operátor `sizeof` vrací konstantní hodnotu implementačně závislého typu `size_t` definovaného v hlavičce `<cstddef>`.

3.1.1 Proměnné základních typů

Pravidla pro tvorbu identifikátorů jsme uvedli v odstavci 2.3 na straně 36. Proměnné základních typů deklarujeme tak, že uvedeme specifikaci typu a seznam identifikátorů proměnných ukončený středníkem.

```
int    k;
float  delka_ab;
long   double objem;
double x, y, z;
```

Všechny uvedené deklarace proměnných jsou zároveň i definicemi. Deklarací zde nezavádíme pouze nové jméno (identifikátor proměnné) ale zároveň požadujeme i alokaci paměti. Identifikátor (jméno proměnné) reprezentuje vždy informace o paměti alokované pro danou proměnnou a typu proměnné, tj. jak má být alokovaná paměť interpretována. Identifikátor je platný od místa deklarace, platnost lokálního jména končí s blokem, ve kterém bylo deklarováno.

Každý výraz v C++ je buď *l-hodnota (l-value)* nebo r-hodnota (r-value). Označení je odvozeno od anglických slov *left* a *right* a označuje výrazy, které mohou být uvedeny *vlevo* nebo *vpravo* od operátoru přiřazení = (přiřazovacího výrazu). Jinak řečeno je l-hodnota výraz odvolávající se na paměť, jejíž obsah můžeme měnit a r-hodnota je výraz odvolávající se na paměť, jejíž obsah měnit nemůžeme. Pokud l-hodnota neodkazuje na funkci nebo pole, může být konvertována na r-hodnotu.

Všechny identifikátory z předchozího příkladu jsou l-hodnoty a mohou tedy vystupovat na obou stranách přiřazovacího výrazu.

```
k = 0;
x = 100.0;
y = z = x;        // y = ( z = x );
```

V ukázce je použit operátor přiřazení =, který je definován pro všechny základní typy a který přiřadí prvnímu operandu hodnotu druhého operandu. Volání operátoru přiřazení je pro základní typy výraz, jehož hodnota je rovna přiřazené hodnotě a můžeme ji tedy použít v dalším přiřazovacím výrazu (viz komentář).

Termín *výraz* se v C++ používá pro označení nějaké *akce*. Výraz ukončený středníkem je *příkaz (expression statement)*. Příkaz je nejmenší nezávislou jednotkou v C++. Fragment programu z naší ukázky tedy tvoří tři příkazy. Existují i jiné typy příkazů, o kterých si řekneme později.

Za identifikátorem může volitelně následovat rovnítko a výraz, kterým má být daná proměnná inicializována. Výraz může být například literál ale také aritmetický výraz a pod. Proměnné, které nejsou explicitně inicializovány mají nedefinovanou hodnotu (pokud nejde o *statické* proměnné).

```
char   z = 'A';
double a = 12.43;
double b = 100 - a*a;
```

Inicializaci proměnných bychom mohli zapsat i takto

```
char   z('A');
double a(12.43);
double b(100 - a*a);
```

Jde pouze o jiný zápis téhož.

Méně často se setkáváme s inicializací proměnných základních typů zapsanou takto

```
int k = int();
double q = double();
```

kde proměnné `k` a `q` jsou inicializovaný na nulu. Tento formát zápisu inicializace se používá téměř výhradně v šablonách (kap. 9) v případech, kdy potřebujeme inicializovat proměnné parametrických typů na implicitní hodnotu.

Deklarace proměnné je v C++ příkaz a proměnné můžeme deklarovat v programu až tam, kde je potřebujeme.

Pro proměnné všech základních typů jsou definovány operátory vstupu `>>` a výstupu `<<`. Operátor vstupu pro čtení ze standardního vstupního datového proudu `cin` a operátor výstupu pro zápis do standardního výstupního proudu jsme použili v další ukázce.

```
double a, b, c;                 // deklarace proměnných
std::cin >> a >> b >> c;        // načtu hodnoty ze std::cin
double v = a * b * c;           // deklarace další proměnné
std::cout << v << std::endl;    // výstup
```

Identifikátory proměnných základních typů a literály mohou být použity jako operandy binárních aritmetických operátorů

```
+   -   *   /   %
```

pro součet, rozdíl, součin a dělení. Operátor `%` počítá zbytek po dělení prvního celočíselného operandu druhým celočíselným operandem (pro záporné hodnoty je výsledek implementačně závislý). Příkladem použití celočíselného operátoru `%` může být následující funkce, která určuje, zda je daný rok přestupný:

```
inline bool prestupny(int rok)
{
  return rok%4 == 0 && rok%100 != 0 || rok%400 == 0;
}

rok 1700 neni prestupny
rok 1900 neni prestupny
rok 2000 je   prestupny
rok 2005 neni prestupny
rok 2008 je   prestupny
```

Pro přičtení 1 k hodnotě proměnné číselného typu je určen operátor inkrementace `++`. Je-li uveden ve výrazu jako *postfixový* (tj. za identifikátorem číselné proměnné), vyhodnotí se nejprve daný výraz a pak se teprve přičte 1. Je-li použit jako *prefixový* (tj. před identifikátorem), zvýší se nejprve hodnota proměnné a teprve pak se vypočte daný výraz. Analogicky pracuje operátor dekrementace `--`. Ve výrazu nelze použít pro danou proměnnou tyto operátory více než jednou, výsledná hodnota výrazu by byla implementačně závislá.

```
int i = 10, j = 10;
std::cout << i++ << std::endl;    // vytiskne 10 a pak zvýší hodnotu i o 1
std::cout << ++j << std::endl;    // zvýší hodnotu j o 1 a pak vytiskne 11
```

Navíc máme pro číselné typy skupinu binárních operátorů

operand1 `op=` *operand2* ;

které jsou ekvivalentní kombinaci operátoru *op* a operátoru přiřazení

operand1 `=` *operand1* `op` *operand2* ;

jako například

```
int a = 1;
a += 2;              // totéž jako: a = a + 2;
```

Proměnné základních typů můžeme použít jako operandy relačních operátorů *je menší, je větší, menší nebo rovno, větší nebo rovno, rovná se* a *nerovná se*

`<` `>` `<=` `>=` `==` `!=`

3.2 Výčtový typ

Výčtový typ je celočíselný odvozený typ s pojmenovanými konstantami. Výčtové konstanty mohou být použity všude tam, kde mohou být použity celočíselné konstanty. Syntax definice výčtového typu je

`enum` *identifikátor*$_{vol}$ `{` *seznam_výčtových_konstant* `}` ;

Implicitně má první výčtová konstanta hodnotu 0, následující výčtová konstanta má implicitně hodnotu o jedničku vyšší. Libovolná výčtová konstanta může být explicitně inicializována konstantním výrazem celočíselného typu nebo typu `char`. V daném výčtovém typu nemusí být všechny konstanty různé.

```
enum { A, B, C=0 };
enum { D, E, F=E+2 };
```

definuje konstanty `A`, `C`, a `D` jako 0, `B` a `E` jako 1 a `F` jako 3.

Deklarace

```
enum den { pondeli, utery, streda, ctvrtek, patek, sobota, nedele };
```

definuje výčtový typ `den`, který může nabývat hodnot `pondeli` až `nedele`. Protože `den` je samostatný typ, lze proměnným typu `den` přiřadit pouze hodnoty typu `den` a následující příkaz je chybný

```
den p = 1;           // !!! typu den nelze přiřadit hodnotu typu int !!!
```

Obráceně ale lze bez problémů přiřadit výraz typu `den` proměnné typu `int`

```
int i = pondeli;    // implicitni konverze na typ int
```

Pokud jsou všechny konstanty daného výčtového typu nezáporné, je obor tohoto výčtového typu interval $< 0, 2^k >$, kde 2^k je nejmenší exponent, který obsáhne všechny dané výčtové konstanty. Obsahuje-li výčtový typ záporné konstanty, je jeho obor obdobně interval $< -2^k, 2^k - 1 >$. Obor hodnot určuje, pro které celočíselné hodnoty je definována explicitní konverze.

```
enum E {a=0, b=1, c=2, d=4, e=8};      // textrminterval < 0, 8 >

E e1 = 2;          // !!! implicitní konverze z typu int neexistuje !!!
E e2 = E(2);       //     explicitní konverze
E e3 = E(3);       //     explicitní konverze, do e3 je uložena hodnota 3
E e4 = E(9);       // !!! výsledek operace není definován,  9 ∉ < 0,8 > !!!
```

3.3 Typ reference

Pomocí *reference (reference)* můžeme vytvořit alternativní jméno pro objekt. Syntaxe deklarace proměnné typu reference na `T` je

`T&` *identifikátor* `=` *objekt_typu_T* `;`

Příklad:

```
int  k  = 0;
int& rk = k;          // rk je alternativní jméno pro k
rk++;                 // zvětšíme o 1 hodnotu k
```

Pokud se nejedná o deklaraci `extern`, deklaraci datového členu v deklaraci třídy, deklaraci parametru funkce nebo deklaraci návratového typu funkce, musí deklarace reference vždy obsahovat inicializátor. Nelze deklarovat reference na typ reference, pole referencí ani ukazatel na referenci.

Můžeme definovat konstantní referenci. Ta může být inicializována i konstantou, dokonce i jiného typu, existuje-li konverze.

Typ reference má především uplatnění v deklaracích parametrů funkcí a funkcí vracejících typ reference (obvykle jde o metody, tj. členské funkce tříd).

Funkce s parametry typu reference mohou měnit hodnoty svých argumentů. Funkce `dalsi` v ukázce nastaví hodnotu argumentů na následující den (viz příklad výčtového typu z předchozího odstavce).

```
void dalsi(den& p)
{
  if (p == nedele) p = pondeli; else p = den(int(p)++);
}
// ...
den d = utery;
dalsi(d);           // po návratu z funkce d == streda
```

Další příklad je funkce s návratovou hodnotou typu reference na `int`, která může být uvedena vlevo od operátoru přiřazení.

```
int v[20];
// ...
int& g(int i) { return v[i]; }
// ...
g(3) = 7;      // přiřazení hodnoty 7 čtvrtému prvku pole v
```

3.4 Typ pole

Pro daný typ `T` deklaruje

`T` *identifikátor* `[` *n* `] ;`

pole (array) o *n* prvcích typu `T`. Typ `T` nemůže být typ reference, typ `void`, funkce nebo abstraktní třída (typ `T` ale může být `void*`, ukazatel na funkci i ukazatel na abstraktní třídu). Dimenze pole *n* musí být konstantní celočíselný výraz větší než nula, který lze vyhodnotit při překladu.

```
int    p[10];          // pole o 10 prvcích typu int
double x[100];         // pole o 100 prvcích typu double
char   text[1024];     // pole 1024 znaků
```

Nelze také deklarovat pole objektů typu `T`, pokud `T` je třída, resp. struktura, která nemá implicitní konstruktor `T()`. Například

```
struct T
{
    T(int) {}
};

int main()
{
 T pole[10];           // !!! chyba !!!
}
```

Deklarace pole ukazuje, jak přistupovat k jeho jednotlivým prvkům. V C++ indexujeme prvky pole od nuly.

```
p[0] = 6;              // přiřaď hodnotu 6 prvnímu prvku pole p
p[4] = p[3];           // přiřaď pátému prvku pole p čtvrtý prvek
```

Prvkem pole může být typ pole. Můžeme tedy deklarovat vícerozměrná pole. V ukázce deklarujeme pole o 50-ti prvcích, každý prvek je pole typu `double [2]`.

```
double yx[50][2];
//...
for (int k=0; k<50; k++)
   yx[k][0] = yx[k][1] = 0;
```

V jazyce C++ je

pole `[`*index*`]`

výraz, ve kterém `[]` je *operátor indexování (subscripting operator)*. Operátor `[]` je definován pro všechny základní typy. Nekontroluje hodnoty indexů (jejich rozsah).

V deklaraci pole je vhodné používat pojmenované konstanty a ne literály. Pojmenované konstanty použijeme pak při následných operacích s prvky pole, především v cyklech. Výsledný program je srozumitelnější a snáze se udržuje (při případné změně dimenze se úprava týká jediného místa). Program, ve kterém jsou v dimenzích polí a mezích cyklů literály *(„magická čísla“)* nesvědčí o dobrém stylu autora.

```
const int dim = 20;
double x[dim];
// ...
double suma_x = 0;
for (int i=0; i<dim; i++)
   suma_x += x[i];
```

V deklaraci pole můžeme inicializovat jeho prvky. Pokud v seznamu neuvedeme potřebný počet prvků, je zbývajícím prvkům přiřazena nula. Pokud uvedeme více prvků než je dimenze pole, je to chyba.

```
double p1[3] = { 2.4, 3.8, 6.2};        // OK
double p2[3] = { 2.4, 3.8 };            // OK, p2[2] == 0
double p3[3] = { 2.4, 3.8, 6.2, 4.2 };  // !!! chyba !!!
```

V deklaraci inicializovaného pole dimenzi uvádět nemusíme, určí se z počtu členů inicializačního seznamu. Dimenzi pole pak můžeme určit pomocí operátoru `sizeof`.

```
double pd[] = { 2.1, 3.6, 7.4, 0.2 };
const int dim_pd = sizeof pd / sizeof (double);
// nebo jinak:     sizeof pd / sizeof pd[0] ;
```

Pro pole je alokovaná souvislá oblast paměti potřebná pro uložení všech jeho prvků (pokud jde o vícerozměrné pole, jsou jeho prvky uloženy po řádcích). Operátor `sizeof` vrátí velikost vyhrazené paměti v bajtech, dimenze pole je velikost pole v bajtech dělená počtem bajtů na jeden prvek.

3.4.1 Pole typu `char` a C-řetězce

C-řetězce a funkce definované v hlavičce `<cstring>` převzal jazyk C++ z jazyka C, ve kterém byly hlavním nástrojem pro práci s texty. C-řetězec je souvislá posloupnost znaků ukončená nulovým bajtem. Běžně se jim říká *textové řetězce* nebo prostě jen *řetězce*. Protože ale jazyk C++ obsahuje objektový typ `std::string`, kterému také říkáme *řetězec* (nebo *standardní řetězec*), budeme používat označení C-řetězec tam, kde potřebujeme oba typy explicitně odlišit. Z kontextu je obvykle zřejmé, oč jde, a nedochází k nedorozumění.

Pole typu `char`, tj. pole znaků, můžeme inicializovat C-řetězcem. Poslední prvek pole bude v takovém případě znak `'\0'`.

```
char text1[] = "abc";               // sizeof text1 == 4
```

Nulový ukončující bajt se do pole nepřidává, je-li inicializováno po jednotlivých znacích

```
char text2[] = {'a', 'b', 'c'};     // sizeof text2 == 3
```

Na to je třeba pamatovat při práci se standardními C-řetězcovými funkcemi deklarovanými v hlavičce `<cstring>`, které vesměs předpokládají že, jejich argumenty jsou C-řetězce ukončené nulovým bajtem, tj. znakem `'\0'`.

Pole znaků sice můžeme C-řetězcem inicializovat ale nemůžeme do pole znaků řetězec přiřadit, řetězec nepatří mezi základní typy jazyka C++.

```
char t[6];
t = "chyba";     // !!! nelze !!!
```

Jak jsme se již zmínili, pro práci s C-řetězci poskytuje standardní C++ knihovna bohatou paletu funkcí, které byly převzaty z jazyka C, a které jsou popsány v hlavičce `<cstring>`, více si řekneme v kapitole 10.

Čtení textových řetězců

C++ operátor vstupu >> implicitně ignoruje bíle znaky na vstupu. Fragment programu

```
char text[80];
std::cin >> text;
```

načte ze vstupu jedno „slovo“ (posloupnost znaků ohraničená bílými znaky) a uloží jej do pole `text` jako C-řetězec ukončený znakem `'\0'`. Netestuje ale přetečení, pokud je vstupní slovo delší než je dimenze pole, přepíše se obsah paměti za hranicí pole.

Bezpečný způsob načítání textů poskytuje metoda `get`, jejímž prvním parametrem je pole znaků, druhý parametr je dimenze pole a třetí nepovinný parametr je znak ukončující vstup (implicitně znak `'\n'`). Jeden řádek textu načteme třeba takto

```
char radek[80];
std::cin.get(radek, 80, '\n');          // načte max. 79 znaků a přidá '\0'
char nl;
std::cin.get(nl);                       // metoda pro čtení jednoho znaku
if (nl != '\n') {
   std::cout << "prilis dlouhy radek!\n";
   std::cin.putback(nl);                // vrátím zpět poslední znak
   // ...  ošetření přetečení vstupního bufferu
}
```

Po načtení každého řádku je nutné ještě načíst znak ukončující řádek a otestovat, jestli na vstupu nebyl řádek přesahující kapacitu bufferu `text`.

Vstupní datové proudy v C++ poskytují metodu `putback`, která umožňuje vrátit zpět do vstupního proudu poslední načtený znak. Toho lze využívat pro různé testy nebo pro ošetření přetečení jako v předchozí ukázce.

Další užitečnou metodou standardních proudů je funkce `peek`, která nám umožňuje *nahlédnout* do vstupního proudu a zjistit jaká je hodnota následujícího znaku připraveného pro vstup.

```
if (vstup.peek() == '\n') { /* zpracování konce vstupního řádku */ }
```

Pro zápis jednoho znaku do výstupního datového proudu můžeme použít metodu `put`. Následující program kopíruje po znacích text ze standardního vstupního proudu `std::cin` a zapisuje jej na standardní výstupní proud `std::cout`.

```
#include <iostream>

int main()
{
  char c;
  while (std::cin.get(c))
    std::cout.put(c);       // totez jako std::cout << c;
}
```

Čtení do proměnných typu `string`

Pro běžné C++ aplikace pracující s texty je jednodušší a vhodnější používat místo polí typu `char` objektový typ `std::string`.

```
#include <iostream>
#include <string>

int main()
{
   std::string t;
   std::cin >> t;
   std::string s;
   s = "nacten retezec ";
   std::cout << s + t << std::endl;
}
```

Operátor `>>` v ukázce načte jedno *slovo* (tj. posloupnost znaků omezenou bílými znaky) do proměnné `t`. Pro načtení celého řádku do proměnné typu `string` slouží funkce `getline()`.

Funkce `getline()` se chová stejně jako `get(char*, int, char)`, pouze s tím rozdílem, že ze vstupního proudu vyjímá znak ukončující vstup (implicitně `'\n'`). Pro načítání proměnných typu `string` je určena (template) funkce `istream& getline(istream&, string&)`.

```
#include <iostream>
#include <string>

int main()
{
   std::string s;
   while ( std::getline(std::cin, s) )
      std::cout << s << '\n';
}
```

Pokud jsou ve vstupním textu *řádky* ukončovány jiným znakem než `'\n'`, můžeme jej funkci `getline` zadat jako třetí parametr:

```
std::istream getline(std::istream&, std::string&, char eol='\n');
```

3.5 Typ ukazatel

Každý objekt je umístěn v souvislé oblasti paměti, zabírá jistý počet bajtů a je jednoznačně určen svým typem a adresou (tj. adresou prvního bajtu vyhrazené oblasti). Mějme například deklaraci proměnné

```
double d_prom;
```

Použijeme-li v programu identifikátor proměnné `d_prom`, je jednoznačně dáno, že jde o jméno proměnné typu `double`, která se nachází na jisté adrese v paměti. Jméno v daném kontextu jednoznačně identifikuje proměnnou.

V C++ lze určit adresu pomocí prefixového unárního operátoru `&`, kterému říkáme *operátor získání adresy (address-of operator)*. Výsledkem výrazu `&T` je hodnota typu *ukazatel (pointer)* na typ `T`. V deklaraci zapisujeme typ ukazatel na `T` jako `T*`.

Ukazatele na různé typy (základní, odvozené nebo třídy) jsou různé typy. Ukazatel na proměnnou `d_prom` můžeme například deklarovat takto

```
double* pd = &d_prom;     // ukazatel na typ double
                          // inicializován adresou proměnné d_prom
```

Pro přístup k proměnným na které směřuje ukazatel na typ T slouží prefixový unární operátor `*`, *operátor dereference (indirection operator)*. Výraz `*p`, kde `p` je ukazatel na typ `T`, můžeme použít ve stejném kontextu jako jméno proměnné typu `T`.

```
double d_prom = 0;
// ...
double* pd = &d_prom;
// ...
*pd = 1;                  // zde totéž jako d_prom = 1;
```

Ukazatele jsou základním nástrojem pro tvorbu dynamických datových struktur. Jejich význam vynikne především v souvislosti s operátory pro správu paměti `new` a `delete` (o nich si řekneme později).

Není možné definovat ukazatel na referenci. Lze deklarovat ukazatel na typ ukazatel, ukazatel na funkci, ukazatel na pole a podobně (pointer nemůže ukazovat na bitové pole).

Elementární operace s ukazateli demonstruje následující ukázka. Zkuste určit, jaký bude výstup programu.

```
#include <iostream>

int main()
{
  double  a = -1, b = -2, c = -3;
  double* p[] = { &a, &b, &c };
  for (int k=0; k<3; k++)
  {
    double* t = p[k];
    *t = k;
    std::cout << a << '\t' << b << '\t' << c << std::endl;
  }
}
```

Ukazatele mohou vystupovat jako operandy relačních operátorů

```
==      !=
```

Výrazy, ve kterých ukazatele vystupují jako operandy jiných relačních operátorů, mohou být implementačně závisle a je třeba používat je obezřetně (správný výsledek je zaručen, pokud oba ukazatele směřují do jediného pole). Ukazatele můžeme porovnávat s hodnotou 0 (nulový ukazatel) — protože žádný objekt nemůže mít adresu 0, používá se ve smyslu *nedefinovaný ukazatel* v různých dynamických datových strukturách jako jsou spojové seznamy a podobně.

Výraz, ve kterém n je celočíselný výraz

ukazatel_na_T $\pm$ n

se vyhodnotí tak, že se k adrese na kterou směřuje *ukazatel_na_T* přičte, resp. odečte,

$n\times$`sizeof(T)`.

Jinak řečeno, ukazatel se posune o n objektů typu `T`. V tomto významu lze na ukazatele aplikovat operátory inkrementace `++` a dekrementace `--`.

Ukazatele nemůžeme sčítat, směřují-li dva ukazatele do jednoho pole, je jejich rozdíl roven právě rozdílu indexů příslušných prvků.

3.5.1 Ukazatele a pole

Mezi typem ukazatel a typem pole je úzká souvislost (někdy se dokonce mylně uvádí, že jde o ekvivalentní pojmy). V jazyce C++ existuje standardní konverze z typu *pole typu T* na typ *ukazatel na T*; výsledkem je ukazatel na první prvek pole. Jméno pole může být použito jako ukazatel na první prvek.

Operátor indexování `[]` je pro základní typy (tj. pokud nebyl operátor `[]` definován jako metoda nějaké třídy) interpretován tak, že výraz `T[n]` je identický s výrazem `*((T) + (n))`.

```
float  f[10];         // pole o 10 prvcích typu float
float* pf = &f[0];    // adresa prvního prvku
// ...
//         *pf     == f[0]
//         *(pf+1) == f[1]
// ...
//         *(pf+k) == f[k]
```

Výrazy `*(pf+`k`)` a `f[`k`]` jsou ekvivalentní a naopak můžeme aplikovat operátor `[]` na ukazatel, v našem příkladu `pf[`k`]`.

Operátory `new` a `delete`

Pro libovolný datový typ `T` je hodnota výrazu

```
new T
```

rovna ukazateli na objekt typu `T`, dynamicky vytvořenému operátorem `new` v paměti. Obdobně

```
new T [ n ]
```

alokuje *dynamické pole* o n prvcích typu `T`. Pokud je hodnota ukazatele, který vrátil operátor `new` rovna 0, nebylo možno paměť alokovat (implicitně způsobí selhání operátoru `new` *vyvolání výjimky* — k této otázce se ještě vrátíme, až budeme hovořit o funkcích).

Následující fragment programu načítá dimenzi, alokuje dynamické pole a ze standardního datového proudu `cin` načítá složky pole.

```
int x;
std::cin >> x;                  // dimenze pole
double* P = new double [x];     // dynamicky alokuji pole x prvků typu double
for (int k=0; k<x; k++)
   std::cin >> P[k];            // načtu k-tý prvek pole
```

Paměť alokovanou výrazem `p = new T` uvolníme operátorem `delete p`. Dynamicky alokované pole prvků typu `T` uvolníme výrazem `delele[] p`. Musíme tedy explicitně rozlišovat uvolnění paměti alokované pro jeden objekt a pro pole objektů (dimenze pole se neuvádí).

Chceme-li alokovat dynamicky vícerozměrné pole, může být proměnná pouze hodnota první meze pole. Například

```
new T [i][10]
```

Dvourozměrné dynamické pole (matici) bychom mohli realizovat jako pole ukazatelů na typ ukazatel na `double`.

```
#include <iostream>

int main()
{
   int m, n;
   std::cin >> m >> n;             // ctu dimenze matice
   double** A = new double* [m];   // alokuji matici
   for (int i=0; i<m; i++)
      A[i] = new double [n];
   // ...
   for (int i=0; i<m; i++)         // cteni matice
      for (int j=0; j<n; j++)
          std::cin >> A[i][j];
   // ...
   for (int i=0; i<m; i++)         // tisk matice
      {
         for (int j=0; j<n; j++)
            std::cout << A[i][j] << " ";
         std::cout << std::endl;
      }
   // ...
   for (int i=0; i<m; i++)         // uvolnim matici
      delete A[i];
   delete[] A;
}
```

Prostřednictvím ukazatelů můžeme pracovat s „proměnnými“, které nemají jméno, ale pouze alokovanou paměť a typ.

3.6 Struktury

Pole je *agregovaný typ*, jehož prvky jsou všechny stejného typu. *Struktura (structure)* je agregovaný typ, jehož prvky mohou být různého typu. Syntax deklarace struktury je

```
struct identifikátor { deklarace_členů } ;
```

Deklarace

```
struct bod {
   long    cislo;
   double  y;
   double  x;
};
```

definuje nový (uživatelský) typ `bod`, který má členy `cislo`, `y` a `x`. Jde o zavedení nového typu a ne o deklaraci proměnné. Deklarace struktury musí být ukončena středníkem.

Deklaraci proměnné typu `bod` a přístup k jednotlivým členům (složkám) demonstruje příklad. Pro přístup ke členům struktury slouží *operátor tečka (dot operator)*.

```
bod  A2;              // deklarace proměnné typu bod
// ...
A2.cislo = 17;        // členu cislo objektu A2 přiřaď 17
A2.y = 332.41;
A2.x = 567.12;
```

Proměnné typu struktura lze inicializovat podobně jako pole

```
bod p = { 247, 612.48, 272.06 };
```

Ke strukturám často přistupujeme přes ukazatele

```
bod* pb = &A2;        // ukazatel na bod A2
```

Pro přístup ke členům struktury bychom mohli přistupovat pomocí operátoru dereference (`*`) a operátoru tečka.

```
if ( (*pb).cislo == 0 )  ...  // je-li číslo bodu 0, pak ...
```

Pro přístup ke členům struktury přes ukazatele máme k dispozici operátor `->`, kterému říkáme *operátor nepřímého přístupu (member selection)*.

```
if ( pb->cislo == 0 ) {
   pb->y = 0;
   pb->x = 0;
}
```

Objekty strukturovaných typů lze přiřazovat (operátor =), předávat jako parametry funkcí a vracet jako návratovou hodnotu funkce. Nelze je použít jako argumenty operátorů `==`, `!=` a pod. Až budeme hovořit o třídách, řekneme si, jak lze definovat operátory pro nové uživatelské typy.

Počet bajtů `sizeof(T)`, kde `T` je struktura, se nemusí rovnat součtu velikostí jednotlivých složek. Je to dáno tím, že paměť se obvykle alokuje pouze na sudých adresách nebo na hranicích *slov*. Například

```
#include <iostream>

struct S {
  double d;
  char   c;
};

int main()
{
  std::cout << "sizeof double : " << sizeof (double) << std::endl;
  std::cout << "sizeof char   : " << sizeof (char)   << std::endl;
  std::cout << "sizeof S      : " << sizeof (S)      << std::endl;
}

sizeof double : 8
sizeof char   : 1
sizeof S      : 12
```

Nelze deklarovat objekt typu `struct T`, dokud nebyla struktura `T` definována. Lze ale použít ukazatel nebo referenci na jméno struktury, která bude definována později nebo v jiném souboru (v jiné jednotce překladu). To je potřeba například tehdy, kdy se na sebe odkazují různé struktury. Lze deklarovat *jméno* jako strukturovaný typ.

```
struct bod;             // bod bude definován později

struct delka {
   double d;            // měřená délka
   float* md;           // střední chyba d
   bod*   stanovisko;
   bod*   cil;
};

struct bod { long cislo; double y, x; };
```

Jméno struktury můžeme použít, pokud v dané chvíli není nutné znát její velikost (například v deklaraci typu `bod*`).

Dvě struktury představují různé typy, i když mají stejné členy.

```
struct A { int a; };
struct B { int a; };
A y;
B z;

y = z;      // !!! nelze !!!

int x;
y = x;      // !!! nelze !!!
```

3.6.1 Struktury a třídy

Třídy definujeme v C++ stejně jako struktury, namísto klíčového slova `struct` se ale v definici třídy použije klíčové slovo `class`. Formálně se C++ struktury a třídy liší pouze pravidly pro přístup k jejich

členům. Ve struktuře jsou implicitně všechny její členy přístupné (`public`), ve třídě jsou implicitně všechny členy privátní (`private`)

```
struct  A {                    class A {
                totéž jako     public:
   // ...                        // ...
};                             };


struct B {                     class B {
private:        totéž jako
   // ...                        // ...
};                             };
```

Strukturu obvykle použijeme v případech, kdy potřebujeme jednoduchý agregovaný typ pouze s datovými členy, třídu, pokud pro nový typ definujeme i členské funkce (metody) a potřebujeme zapouzdřit datové členy. Podrobněji se třídami budeme zabývat v kapitolách 5 a 6.

3.6.2 Bitové pole

Ve struktuře můžeme operovat s menšími objekty, než je `char`. Člen je definován jako *pole*, pokud za ním uvedeme počet bitů, které se pro něj mají vyhradit.

```
struct bit_pole {
   unsigned  a : 1;
   unsigned  b : 3;
   unsigned    : 6;   // nevyužito (nepojmenované pole)
   unsigned  c : 1;
};
```

Takto můžeme definovat typ, který například pojmenovává bity systémových registrů a podobně. Bitová pole obvykle vedou k implementačně závislým konstrukcím.

3.6.3 Unie

Datové členy struktury jsou v paměti ukládány postupně za sebou, v případě potřeby mohou být posunuty (na hranici slova), ale v každém případě začínají vždy na různých adresách.

Formálně deklarujeme unie stejně jako struktury, místo klíčového slova `struct` pouze použijeme klíčové slovo `union`. Rozdíl mezi nimi je ten, že všechny členy unie začínají v paměti na stejné adrese, jinak řečeno překrývají se. Velikost unie tedy určuje její největší člen, obdobně jako v případě struktur může být velikost unie zaokrouhlena na násobky velikosti *slova*.

```
#include <iostream>

union U {
  double d;
  char   t[9];
};
```

```
int main()
{
  std::cout << "sizeof double : " << sizeof (double)  << std::endl;
  std::cout << "sizeof char[9]: " << sizeof (char[9]) << std::endl;
  std::cout << "sizeof U      : " << sizeof (U)       << std::endl;
}

sizeof double : 8
sizeof char[9]: 9
sizeof U      : 12
```

Členem unie nemohou být třídy s konstruktorem, destruktorem nebo operací kopírování (kopírovací konstruktor nebo operátor přiřazení). Podobně unie nemůže být odvozena nebo být předkem jiné třídy. Unie může obsahovat nevirtuální členské funkce.

```
#include <string>

union ID {
  long int    n;
  std::string s;  // !!! třída s konstruktorem nemůže být členem unie !!!
};
```

Typ `union` lze obvykle vhodněji nahradit sadou odvozených tříd. Typ `union` představuje většinou potenciální zdroj problémů a implementačně závislých konstrukcí a nebudeme se jím proto zabývat.

3.7 Deklarace objektů

Deklaraci v jazyce C++ tvoří *posloupnost specifikátorů* a *seznam deklarátorů*, ve kterém oddělovačem je symbol čárka. Každý deklarátor zavádí jméno a volitelně může obsahovat inicializátor.

Posloupnost specifikátorů může obsahovat jednu paměťovou třídu, kvalifikátor `const` anebo `volatile` a typ objektu a to v libovolném pořadí. Objekt v C++ je přitom chápan nejen jako *instance objektového typu* ale jako libovolný datový typ.

Každý program se skládá z jedné nebo více jednotek překladu (viz též odstavec 2.1). Jména deklarovaná v dané jednotce překladu mohou mít externí linkování *(external linkage)* nebo interní linkování *(internal linkage)*.

Externí linkování mají entity, na jejichž jména se můžeme odkazovat i z jiných jednotek překladu

Lokální linkování mají entity, které jsou viditelné pouze v dané jednotce překladu.

3.7.1 Paměťové třídy

`auto`

označuje automatickou proměnnou, jejíž existence je omezena na blok ve kterém byla tato proměnná deklarována. Specifikátor `auto` je implicitní pro parametry funkcí a lokální proměnné. Nikde jinde nelze tento specifikátor použít. Všechny proměnné s interním linkováním mají paměťovou třídu `auto`, tato specifikace je proto redundantní a je v praxi používána jen zřídkakdy.

`extern`

označuje objekty s externím linkováním, které mohou být definovány v jiné jednotce překladu.

`mutable`

označuje datový člen, který může být měněn i v případě, že daná instance objektového typu je konstantní (viz odstavec 5.9).

`register`

označuje automatickou proměnnou a je pouhým doporučením pro kompilátor, aby tuto proměnnou uložil do systémového registru. Optimalizace alokace registrů je ale plně v režii kompilátoru a žádost o umístění proměnné do registru může překladač ignorovat. Jinak platí pro specifikátor `register` totéž, co pro paměťovou třídu `auto`.

`static`

globální proměnné a funkce přiděluje specifikátor `static` těmto entitám lokální linkování. Specifikátor `static` byl proto využíván pro deklaraci globálních proměnných a funkcí, které se v dané jednotce překladu jevily jako *lokální*. V C++ stejného efektu dosáhneme použitím *anonymního prostoru jmen (unnamed namespace)*.

Jiný význam má použití specifikátoru `static` v případě lokálních jmen v těle funkcí (odstavec 4.2.1), dále pak ve třídách v případe statických datových členů a statických členských funkcí (odstavec 5.7).

3.7.2 Konstantní objekty

Pro libovolný typ `T` můžeme deklarovat konstantní objekt daného typu.

`const T` *identifikátor* `=` *inicializátor* `;`

Konstantní objekty musí být vždy explicitně inicializovány. Jejich hodnotu nemůžeme v programu měnit. Identifikátor konstantního objektu je r-hodnota. Inicializaci objektových typů zajišťují konstruktory.

```
const double redukce = 0.9999;
```

Konstantním objektům budeme říkat stručně konstanty, případně pojmenované konstanty. V jazyce C++ nahrazují makra bez parametrů, o kterých jsme se zmínili na straně 32.

Kromě *proměnných* deklarujeme jako konstantní především parametry funkcí. Deklarací `const` informujeme kompilátor, že daný objekt nemá být měněn. Pokud se pokusíme konstantní objekt při ladění programu omylem změnit, upozorní nás na to kompilátor již při překladu.

3.7.3 Specifikace `volatile`

Specifikátor `volatile` používáme pokud potřebujeme informovat kompilátor, že daný objekt může měnit svoji hodnotu nezávisle na jazyce a nemůže proto podléhat podléhat optimalizaci. Jako příklad uvádí [2] deklaraci

```
extern const volatile long clock;
```

kde proměnná `clock` představuje v reálném čase pracující hodiny a následná čtení této proměnné dávají různé hodnoty. Předchozí příklad zároveň ukazuje, že specifikátory `const` a `volatile` mohou být uvedeny společně.

Anglické slovo *volatile* můžeme přeložit jako *nestálý* a hovoříme proto o *nestálých proměnných.* Obdobně jako v případě specifikace `const` lze deklarovat i členské funkce jako `volatile` a to v případech, kdy přistupují k *nestálým* datovým členům.

3.8 Lokální a globální jména

Každé jméno má v C++ programu určitou oblast platnosti. Jméno deklarované v bloku má platnost od místa deklarace do konce bloku a je v něm *lokální*. Jméno nemusí být viditelné v celé oblasti platnosti, může být zastíněno, pokud je znovu deklarováno v některém vnořeném bloku.

Zdrojový soubor C++ je z hlediska překladu samostatnou jednotkou. Jméno deklarované vně všech anonymních prostorů jmen, bloků, deklarací funkcí, definicí funkcí a tříd je globální (má globální prostor jmen). Oblast platnosti globálního jména je od místa deklarace do konce souboru a může být zastíněno obdobně jako lokální jméno.

Pokud v deklaraci jména globální proměnné uvedeme klíčové slovo `extern` a tato proměnná zde není inicializována, jde o pouhou deklaraci jména a ne definici proměnné. Jinak je deklarace jména globální proměnné zároveň definicí (v deklaraci globálního jména funkce klíčové slovo `extern` nemusíme uvádět).

Globální proměnné mohou být používány v různých souborech (jsou-li v nich deklarovány). Definice každé globální proměnné může být v C++ programu uvedena pouze jednou. Globální proměnné základních typů jsou implicitně inicializovány na hodnotu 0.

Mějme soubor *a.cpp*, který obsahuje

```
#include <iostream>

extern int k;    // deklarace globalni promenne k
int l;           // definice globalni promenne l

int main()
{
   std::cout << k << ' ' << l << "\n";   // k == 1,  l == 0
}
```

a soubor *b.cpp*

```
int k = 1;       // definice globalni promenne k
static int l;    // OK  ...  (int l; by bylo chybne: 2x definice l)
```

Pokud bychom v souboru *b.cpp* uvedli opět deklaraci „`int l;`“, skončilo by sestavení programu chybou *vícenásobná definice l.*

Globální proměnné a funkce můžeme deklarovat jako `static`, taková globální proměnná nebo funkce má pak platnost pouze v daném souboru. Statická globální proměnná „`l`“ proto nekoliduje s globální proměnnou „`l`“ deklarovanou v souboru *a.cpp*.

Deklarace statických globálních proměnných a funkcí má v C++ mnohem menší význam než tomu bylo v jazyce C (*modulární programování*). Globální proměnné, včetně statických globálních proměnných a funkcí, představují obecně nežádoucí programové konstrukce a není vhodné je používat.

Použití deklarace `static` pro zastínění globálních jmen, které jsou viditelné pouze v daném modulu, tj. jednotce překladu, je považováno za zastaralé (bylo převzato z jazyka C). V jazyce C++ je pro řešení obdobných situací určena deklarace *anonymního prostoru jmen (unnamed namespace)* o které si řekneme v následujícím odstavci 3.9.1.

3.9 Deklarace `namespace`

Při práci na větších projektech nepředstavují problém pouze globální proměnné ale především možné kolize jmen tříd a případně i jmen funkcí. O třídách budeme hovořit až v kapitole 5, prozatím nám postačí vědět, že třídy jsou nástrojem pro tvorbu uživatelských typů, a že jméno třídy používáme v deklaraci obdobně jako jména základních typů (například `int`). Jména tříd jsou stejně jako jména funkcí globální.

Řekněme, že v programu pro transformaci prostorových souřadnic do zobrazovací roviny potřebujeme pracovat s knihovnou tříd pro práci s globálním polohovým systémem (`gps.h`) a dále s knihovnou tříd kartografických zobrazení (`kartog.h`). Lze si představit, že obě knihovny budou obsahovat třídu `Bod`. Jednou je `Bod` chápán jako bod v prostoru a podruhé jako bod v zobrazovací rovině.

Zařazení obou hlavičkových souborů

```
#include "gps.h"
#include "kartog.h"
```

způsobí při překladu chybu *„vícenásobná definice třídy Bod“*. Třídy stejně jako funkce mohou být deklarovány vícekrát, v programu smí být ale pouze jediná definice třídy. Problém je v tom, že dochází ke kolizi dvou definic jména `Bod`, které ale daném kontextu má vždy jasný smysl a znamená něco zcela jiného.

Deklarace `namespace` (prostor jmen) poskytuje řešení podobných situací tím, že umožňuje pojmenovat části globálního prostoru jmen.

soubor gps.h

```
namespace GPS {
   class Bod {
      // deklarace třídy Bod v prostoru jmen GPS
   };
   // ...
}
```

soubor kartog.h

```
namespace Kartog {
   class Bod {
      // deklarace třídy Bod v prostoru jmen Kartog
   };
   // ...
}
```

V těle deklarace namespace uvádíme posloupnost deklarací, jméno *prostoru* se stává součástí *kvalifikovaných jmen* svých *členů*. V našem případě tak získáme dvě třídy s různými kvalifikovanými jmény GPS::Bod a Kartog::Bod. Symbol „::“ je *operátor rozlišení oboru (scope resolution operator)*, kterému se běžně říká *„čtyřtečka“*.

Deklarace prostoru jmen může být rozdělena na více částí a to i v různých souborech.

```
namespace XYZ {
int i = 10;
}

namespace XYZ {
int j = 11;
}

std::cout << XYZ::i << " " << XYZ:j;
```

V deklaraci namespace může být uvedena další deklarace namespace.

```
#include <iostream>

using namespace std;

namespace Vnejsi {
   int i;
   namespace Vnitrni {
      void f() { i++; } // Vnejsi::i
      int i;
      void g() { i++; } // Vnitrni::i
   }
}

int main()
{
   Vnejsi::Vnitrni::f();
   Vnejsi::Vnitrni::f();
   Vnejsi::Vnitrni::g();
   cout << Vnejsi::i << "  " << Vnejsi::Vnitrni::i << endl;
}
```

Opakované vypisování specifikace prostoru jmen jako například Vnejsi::Vnitrni z předchozí ukázky je poněkud těžkopádné. V deklaraci namespace můžeme deklarovat náhradní jméno prostoru (alias).

```
int main()
{
   namespace T = Vnejsi::Vnitrni;       // namespace alias
   T::f();
   T::f();
   T::g();
   cout << Vnejsi::i << "  " << T::i << endl;
}
```

3.9.1 Anonymní prostor jmen

Pokud v deklaraci `namespace` neuvedeme identifikátor, deklarujeme *anonymní prostor jmen (unnamed namespace)*.

```
#include <iostream>
namespace {  // anonymni namespace
   int i;
   // ...
}

int main()
{
   std::cout << i << endl;
}
```

Jména deklarovaná v anonymním prostoru jmen používáme stejně jako globální jména. Jména zde deklarovaná mají platnost jen v dané jednotce překladu a nahrazují deklarace se specifikací `static`, kterou bychom měli používat pouze pro deklaraci statických lokálních proměnných ve funkcích a statických datových a funkčních členů tříd.

Deklaraci `static` můžeme totiž použít pouze pro restrikci viditelnosti obyčejných funkcí a globálních proměnných, ale ne pro třídy. Naproti tomu anonymní prostor jmen umožňuje omezení viditelnosti na danou jednotku překladu jak pro obyčejné funkce a globální proměnné tak pro třídy.

3.9.2 Deklarace a direktiva `using`

Pokud se v některé části programu opakovaně odvoláváme na určité jméno z daného prostoru jmen, můžeme použít deklaraci `using`. V následujícím zdrojovém textu můžeme používat jméno uvedené v deklaraci `using` bez explicitní specifikace identifikátoru prostoru jmen. Platnost direktivy je analogická platnosti proměnných.

```
int main()
{
   using Vnejsi::Vnitrni::f;            // deklarace using
   f();
   f();
   Vnejsi::Vnitrni::g();
   cout << Vnejsi::i << "  " << i << endl;
}
```

Takto může deklarovat více jmen. V jedné deklaraci `using` lze uvést vždy jen jedno jméno.

Kromě toho existuje direktiva `using`, která zviditelní v dané oblasti programu všechny identifikátory nacházející se v uvedeném prostoru jmen.

```
using namespace Vnejsi::Vnitrni;        // direktiva using

int main()
{
   f();
```

```
    f();
    g();
    cout << Vnejsi::i << "  " << i << endl;
}
```

Všechny konstanty, typy, šablony, třídy a objekty z C++ standardní knihovny (s výjimkou maker a operátoru `new` a operátoru `delete` a proměnné `errno`) jsou definovány v prostoru jmen `std` nebo v prostorech jmen vnořených v `std`.

Kapitola 4

Funkce

Funkce jsou základním pracovním nástrojem programátora v C++. Typickým způsobem, jak zajistit provedení nějaké netriviální akce je, že zavoláme funkci. Kromě standardních knihovních funkcí, jako jsou například matematické funkce deklarované v hlavičce `<cmath>` a pod., definujeme v C++ vlastní funkce. Funkce představuje jistý uzavřený algoritmus s přesně definovaným rozhraním, tj. jak zadáváme funkci vstupní data (argumenty) a jak funkce předává výsledky. Jako samostatné funkce přitom definujeme i takové akce, které se v daném programu provádějí pouze jedenkrát. Vyčlenění specifických akcí do samostatných funkcí vede k čitelnějšímu programu, který se snáze odladí a udržuje.

Deklarace funkce

typ_{vol} jméno (*seznam parametrů$_{vol}$*) *cv-kvalifikace$_{vol}$ specifikace výjimek$_{vol}$* ;

určuje

- typ funkce, tj. typ návratové hodnoty (existují i speciální členské funkce *konstruktory*, *destruktory* a konverzní operátory, které nemají typ)
- jméno funkce (identifikátor funkce)
- seznam parametrů (může být prázdný)
- v případě nestatických členských funkcí volitelnou specifikaci `const` anebo `volatile`
- volitelnou specifikaci výjimek, které může daná funkce vyvolat

Definice funkce

typ_{vol} jméno (*seznam parametrů$_{vol}$*) *cv-kvalifikace$_{vol}$ specifikace výjimek$_{vol}$ tělo funkce*

oproti deklaraci navíc obsahuje *tělo funkce*, tj. posloupnost deklarací a příkazů, které se provádějí při vyvolání funkce. Tělo funkce je buď blok (složený příkaz) nebo *try blok*. Definice funkce se v programu smí vyskytovat pouze jedinkrát, deklarace funkce se může vyskytovat opakovaně.

```
#include <iostream>
#include <cmath>

double odmocnina(double x);                    // deklarace funkce
```

```
int main()
{
  for (double i=0; i<10; i++)
    std::cout << odmocnina(i) << std::endl;   // volani funkce
}

double odmocnina(double x)                    // definice funkce
{
  if (x <= 0)
    return 0;
  double xm, N = x;
  do {
    xm = x;                                   // predchozi hodnota
    x = 0.5 * (x + N/x);                      // nova hodnota
  } while (std::abs(x - xm) > x*1e-6);
  return x;
}
```

Funkci voláme tak, že v programu napíšeme její jméno a do závorek uvedeme seznam argumentů (závorky musíme uvádět i tehdy, je-li seznam prázdný). V ukázce je uvedena definice funkce `odmocnina`. Její typ je `double`, jde o funkci která vrací hodnotu typu `double`. Jinak řečeno, volání této funkce je výraz, jehož hodnota je typu `double`. Algoritmus funkce `odmocnina` je posloupnost příkazů zapsaná ve složených závorkách (tělo funkce).

Rozlišujeme *parametry* a *argumenty* funkce. Parametry funkce definují typy a jména objektů, která používáme v definici v těle funkce a slouží pro popis rozhraní přes které funkce komunikuje s volajícím programem (prozatím se nebudeme zabývat možností komunikace přes globální proměnné). Seznam argumentů uvádíme při volání funkce (mohou to být jména objektů nebo různé výrazy) a specifikuje objekty se kterými volaná funkce pracuje.

Funkce `odmocnina` z naší ukázky má jeden parametr „`x`“ typu `double` a je volána v cyklu s argumentem „`i`“ typu `double`. Jména parametrů a argumentů nemají žádnou souvislost. Při volání funkce ale musí souhlasit jejich typ nebo musí existovat konverze z typu argumentu na typ parametru. Parametrům se někdy říká formální parametry a argumentům skutečné parametry, argumenty mohou být výrazy.

Pro předávání argumentů platí podobná pravidla jako pro inicializaci proměnných. Předávání argumentů můžeme chápat tak, že při vyvolání funkce se vytvoří lokální proměnné (parametry), které jsou inicializovány hodnotami argumentů a které přestanou existovat po ukončení činnosti funkce.

Parametr funkce může být typu *reference*. Parametr pak představuje náhradní jméno pro argument použitý při volání a funkce může argument modifikovat. Hovoříme o předávání argumentů referencí (funkce může argument modifikovat) nebo o předávání argumentů hodnotou (funkce pracuje s lokální kopií argumentu, případně změny se nepromítnou vně funkce).

Funkce `odmocnina` vrací vypočtenou hodnotu příkazem

`return` *výraz*`;`

Příkaz `return` ukončuje činnost funkce (vrací řízení volajícímu programu). Příkaz `return` může být ve funkci uveden na více místech, nemusí být pouze posledním příkazem. Funkce mohou být volány rekursivně, jak demonstruje funkce `faktorial`.

```
int faktorial(int n)
{
  if (n <= 1)
    return 1;
  else
    return n*faktorial(n-1);
}
```

Rekursivní volání funkce znamená, že funkce volá sama sebe a to přímo nebo nepřímo přes jiné funkce. Rekurze představuje v řadě případů mocný nástroj pro popis algoritmů jako je například průchod stromovou strukturou a pod. Naproti tomu je ale obecně rekurze výpočetně velmi náročná a pokud je to možné, je vhodné se jí vyvarovat. Předchozí příklad rekurzivního výpočtu je zde uveden pouze jako jednoduchá ukázka možností jazyka a ne jako příklad vhodného algoritmu pro výpočet faktoriálu.

Ve funkcích typu `void` (dále v konstruktorech a destruktorech) příkaz `return` neobsahuje *výraz* a nemusí být uveden. Funkce v takovém případě vrátí řízení při dosažení uzavírací složené závorky uzavírající tělo funkce (stejně jako by posledním příkazem byl `return`). O funkcích typu `void` říkáme, že nevracejí hodnotu.

```
void f()  // funkce s prazdnym seznamem parametru
{
  cout << "funkce typu void bez parametru \n";
}
```

Pokud funkce typu `void` obsahuje příkaz `return` s *výrazem*, musí být tento *výraz* typu `void`.

```
void g() { /* ... */ }

void f()
{
  // ...
  if (podmínka) return g();
  // ...
}
```

Rozlišujeme *deklaraci* a *definici* funkce. Funkce může být v programu definována jen jednou. Součástí definice funkce je tělo funkce. Deklarace funkce je ukončená středníkem, může se vyskytovat vícekrát a může být součástí definice jiné funkce (nelze ale definovat *lokální funkci*, tj. definice funkce nesmí obsahovat definici další funkce). Deklarace funkce informuje překladač, že existuje funkce, která má daný typ, jméno a seznam parametrů (prototyp funkce) — každý objekt musí být nejprve deklarován a pak jej teprve můžeme použít.

Protože funkce `odmocnina` z našeho příkladu byla definována až za funkcí `main`, museli jsme ji nejprve deklarovat. V deklaraci funkce nemusíme uvádět jména parametrů ale pouze jejich typ, pokud jména identifikátorů v deklaraci funkce uvádíme, pak pouze jako pomůcku pro lepší čitelnost zdrojového textu.

```
double odmocnina(double);          // deklarace funkce (prototyp funkce)

int main()
{
  double odmocnina(double);        // deklarace funkce (prototyp funkce)
  for (double i=0; i<10; i++)
```

```
    cout << odmocnina(i) << endl;
}

double odmocnina(double x)         // definice funkce
{
  // ... tělo funkce ...
}
```

Funkci stačí pochopitelně deklarovat jen jednou. Obvykle jsou deklarace funkcí zapsány v hlavičkovém souboru a explicitně je nevypisujeme.

4.1 Funkce `main()`

Každý C++ program musí obsahovat funkci `main`. Řízení programu začíná funkcí `main`, tato funkce nesmí být přetížena a nesmí být v programu volána. Jméno `main` není jinak v programu rezervováno (můžeme například definovat členskou funkci `main`).

Standardní definice funkce `main` je

```
int main() { /* ... */ }
```

nebo

```
int main(int argc, char* argv[]) { /* ... */ }
```

Ve druhém případě udává `argc` počet argumentů předaných programu při jeho vyvolání. První ukazatel `argv[0]` směřuje na řetězec obsahující jméno spuštěného programu nebo prázdný řetězec `""`. Další ukazatele v poli `argv` odkazují na jednotlivé argumenty předané programu.

```
#include <iostream>

int main(int argc, char* argv[])
{
  std::cout << "\nProgram " << argv[0];
  if (argc <= 1)
    std::cout << " byl volan bez argumentu\n";
  else
  {
    std::cout << " byl volan s temito argumenty:\n\n";
    for (int i=1; i<argc; i++)
      std::cout << i << ": " << argv[i] << std::endl;
  }
  std::cout << std::endl;
}
```

Předchozí program vypisuje seznam argumentů, se kterými byl volán (například pod operačním systémem GNU/Linux).

```
bash$
bash$ main_arg

Program main_arg byl volan bez argumentu

bash$
bash$
bash$ main_arg a ab abc abcd

Program main_arg byl volan s~temito argumenty:

1: a
2: ab
3: abc
4: abcd

bash$
```

4.2 Tělo funkce

Tělo funkce může být *složený příkaz*

```
void f() { /* tělo funkce tvoří blok  */ }
```

nebo *try blok*

```
void g() try { /* ... */ } catch (výjimka) { /* ... */ }
```

4.2.1 Tělo funkce jako složený příkaz

Tělo funkce obvykle ohraničuje dvojice složených závorek `{` a `}`, které říkáme *blok (block)* nebo *složený příkaz (compound statement)*. Složený příkaz tvořící tělo funkce obsahuje posloupnost příkazů a deklarací, může být i prázdný a neukončuje se středníkem.

Objekty deklarované v bloku jsou lokální, existují od místa deklarace a přestávají existovat po ukončení zpracování bloku (jejich paměťová třída je `auto`). Stejně tak objekty deklarované v těle funkce jsou lokální v dané funkci a přestávají existovat po návratu z funkce.

Proměnné mohou být deklarovány jako *static*, jejich hodnota pak zůstává zachována a lze ji použít při dalším volání.

```
void f()
{
  static int citac = 1;
  cout << citac++ << ". volani funkce f()" << endl;
}
```

Implicitně jsou statické proměnné inicializovány na 0, jejich paměťová třída je `static`. Statické proměnné deklarované v těle funkce jsou inicializovány pouze jednou a to při prvním průchod daným příkazem a *pamatují si* svoji hodnotu z předchozího volání. Toho lze využít například pro inicializaci interních datových struktur při prvním volání funkce. Pro řešení podobných situací jsou ale třídy téměř vždy vhodnějším kandidátem.

```
float Rand_N()
{
  static bool start = true;
  static float C1, C2, C3;
  if (start)                          // inicializace konstant proběhne pouze při prvním volání
     {
        C1 = sqrt(8.0/exp(1.0));
        C2 = 4*exp(0.25);
        C3 = 4*exp(-1.35);
        start = false;
     }

  // ... výpočet ...
}
```

Většina proměnných, se kterými jsme doposud pracovali, měly implicitní paměťovou třídu `auto`. Specifikace `auto` pro explicitní označení automatických proměnných může být použita pouze pro jména objektů deklarovaných v nějakém bloku a prakticky se nepoužívá.

Pokud nejsou automatické proměnné deklarovány s inicializátorem (nejsou explicitně inicializovány v deklaraci), není jejich hodnota definovaná.

Další paměťové třídy jsou `extern`, `mutable` a `register`. O paměťové třídě `extern` jsme hovořili v odstavci 3.8. Deklarace `mutable` se používá v případech, kdy potřebujeme umožnit modifikaci vybraných atributů i pro konstantní objekty, podrobněji viz kapitola 5 věnovaná třídám.

Paměťová třída `register` je pouhým nezávazným doporučením pro překladač, aby umístil automatickou proměnnou do registrů počítače, jinak má paměťová třída `register` stejnou sémantiku jako `auto`.

V následujícím příkladu deklarujeme ve funkci `f()` *vnořený* blok a v něm proměnnou `fp`, která již byla deklarována dříve jako lokální proměnná funkce.

```
void f()
{
  // ...
  double fp = 0.9999;
  // ...
  {             // zde začíná vnořený blok
    int fp;     // deklarace zastíní dříve deklarované jméno
    // ...
  }             // konec vnořeného bloku (neukončuje se středníkem)
  // ... lokální proměnná fp je zde opět viditelná
}
```

Deklarace proměnné `fp` ve vnořeném bloku pouze zastíní dříve deklarované jméno, které je po ukončení bloku opět viditelné. Pokud nejde o triviální situace jako zde, může být zastínění proměnné deklarované vně bloku zdrojem nepříjemných chyb, které se špatně identifikují, a je třeba na ně dávat pozor.

4.2.2 Tělo funkce jako `try` blok

Jazyk C++ umožňuje, aby tělem funkce byl *try blok*:

```
void f()
  try {
    }
  catch (výjimka1) {
    }
  catch (výjimka2) {
    }
  // ...
  catch (...) {
    }
```

Požijeme-li `try` blok jako tělo funkce `main`, můžeme poměrně jednoduše zajistit, že náš program bude reagovat i na neočekávané výjimky:

```
int main()
try {
   // ...
}
catch (výjimka1) { ... }
catch (výjimka2) { ... }
catch (...) {
  std::cerr << "zpracování neočekávané výjimky\n";
}
```

Nelze takto ale ošetřit výjimky vyvolané globálními instancemi objektových typů.

Pokud konstruktor inicializuje datové členy, které mohou vyvolat výjimky, lze je zachytit přímo v konstruktoru:

```
class P {
    Q q;
  public:
    P(int);
  // ....
};
```

```
P:P(int r)
  try : q(r) {    /* ... */
  }
  catch(výjimka vyvolaná při inicializaci členu q) {
    /* ... */  throw výjimka;
  }
```

4.3 `inline` funkce

Při volání funkce musí v programu vždy proběhnout standardní posloupnost operací. Program musí zaznamenat návratovou adresu, na kterou bude předáno řízení chodu programu po návratu z funkce, a musí funkci předat argumenty a řízení (potřebné údaje se ukládají do zásobníku, anglicky *stack*). Při předávání argumentů hodnotou je nutné argumenty překopírovat, při předávání argumentů referencí stačí předat pouze jejich adresu.

Volání funkce představuje vždy určitou režii a ne vždy ji můžeme ignorovat. Jazyk C++ umožňuje deklarovat funkce se specifikací `inline`, která je doporučením kompilátoru, aby příkazy z těla funkce přímo vložil na místo, odkud byla tato funkce ve zdrojovém textu volána *(vložená funkce)*.

```
// prevod radianu na grady / M_PI definovano v <cmath>
inline double grad(double rad) { return rad*200/M_PI; }
```

Ve většině případů nahrazují *vložené* funkce C++ makra, `inline` funkce jsou bezpečnější (kontrola argumentů `inline` funkcí proběhne při překladu) a odůvodněné použití maker je limitováno pouze na zcela specifické případy.

`inline` funkce můžeme uvádět v hlavičkových souborech.

Význam vložených funkcí vynikne až budeme hovořit o třídách, členské funkce tříd jsou nezřídka `inline` funkce.

4.3.1 Podmíněný výraz

Velmi často tvoří `inline` funkce pouze příkaz `return` a *podmíněný výraz (conditional expression)*. Podmíněný výraz obsahuje ternární operátor `?` (operátor vyžaduje tři argumenty; jde o jediný ternární operátor jazyka C++) a jeho syntax je

výraz_1 `?` *výraz_2* `:` *výraz_3*

Pokud je *výraz_1* pravdivý (různý od nuly), je hodnota podmíněného výrazu *výraz_2*, v opačném případě *výraz_3*.

```
inline double min(double a, double b) { return a < b ? a : b; }
```

4.4 Návratová hodnota a parametry typu reference

Návratová hodnota funkce může být typu reference, volání takové funkce pak může být uvedeno vlevo od přiřazovacího příkazu.

```
#include <iostream>

using namespace std;

int a = 10;   // globalni promenne a, b
int b = 20;

int& f(int n) { if (n == 1) return a; else return b; }

int main()
{
  f(1) = 1;   // volani funkce vlevo od operatoru =
  f(2) = 0;
  cout << a << " " << b << endl;
}
```

Funkce nesmí vracet referenci na lokální objekt (proměnnou), který po návratu z funkce neexistuje (tj. má paměťovou třídu `auto`). V ukázce je funkce vracející referenci na jednu ze dvou globálních proměnných. S funkcemi vracejícími typ reference se setkáme především jako s členskými funkcemi tříd (metodami).

Parametry typu reference umožňují funkci měnit hodnoty argumentů. Funkce z následující ukázky vymění hodnoty svých dvou argumentů.

```
#include <iostream>

void vymen(int& a, int& b)
{
  int pom = a;  a = b;  b = pom;
}

int main()
{
  int x = 1, y = 2;
  vymen(x, y);
  std::cout << x << " " << y << std::endl;
}
```

Uvedenou funkci bychom mohli napsat také s parametry typu ukazatel.

```
#include <iostream>

void vymen2(int* a, int* b)
{
  int pom = *a;  *a = *b;  *b = pom;
}

int main()
{
  int x = 1, y = 2;
  vymen2(&x, &y);
  std::cout << x << " " << y << std::endl;
}
```

Jako argumenty funkci `vymen2` musíme zadat adresy proměnných typu `int`. Pro určení adresy objektu slouží *operátor získání adresy (address of operator)* `&`.

Při volání funkce je vždy alokována paměť pro její formální parametry a každý formální parametr je inicializován odpovídající hodnotou skutečného parametru. Například volání funkce `f()`

```
void f(int& m, int n)
{
  m++;    // změní   argument použitý při volání funkce
  n++;    // nezmění argument použitý při volání funkce
}
....
int  a=1, b=2;
....
f(a, b);
```

si můžeme představit tak, jakoby před prvním příkazem (`m++;`) předcházelo vytvoření a inicializace lokálních proměnných `m` a `n`

```
void f(/* int& m, int n */)
{
  int& m =  a;    !!! pseudokód !!!
  int  n =  b;    !!! pseudokód !!!
```

```
  m++;    // změní   argument použitý při volání funkce
  n++;    // nezmění argument použitý při volání funkce
}
```

Parametr funkce typu reference použijeme také tam, kde nechceme, aby docházelo k vytváření lokální kopie argumentu (typicky u velkých objektů jako jsou například matice). Pokud funkce takový parametr předávaný referencí nemění, deklarujeme jej jako konstantní referenci. Pokusíme-li se (omylem) při psaní funkce takový parametr modifikovat, upozorní nás na chybu kompilátor již při překladu.

```
void sestaveni_normalnich_rovnic(const Matice& A, Matice& N) { ... }
```

Parametry typu konstantní reference jsou typické pro „vstupní parametry“, parametry typu reference pro „výstupní parametry“. Naučte se důsledně používat specifikaci `const` všude, kde má oprávnění (později si řekneme o konstantních členských funkcích tříd).

V jazyce C může funkce měnit hodnoty skutečných parametrů pouze prostřednictvím ukazatelů (jazyk C nemá typ reference). Otázka je, zda v obdobných případech v C++ používat parametry funkce typu ukazatel nebo reference. V některých specifických případech je použití reference jedinou možností, například pokud chceme napsat pro danou třídu `T` výstupní operátor `<<`. Tam kde existuje možnost volby, mnozí programátoři dávají přednost ukazatelům s odůvodněním, že v opačném případě ze zápisu volání funkce nelze určit, zda funkce modifikuje použité argumenty nebo ne.

4.5 Implicitní hodnoty parametrů

V definici funkce můžeme definovat implicitní hodnoty parametrů, které se použijí, pokud při volání funkce neuvedeme odpovídající argument. Implicitní hodnotu parametru zapíšeme v seznamu parametrů obdobně jako zapisujeme inicializaci proměnných.

```
#include <iostream>

void f(int a=1, int b=2, int c=3)
{
  std::cout << a << " " << b << " " << c << std::endl;
}

int main()
{
  f();            // implicitni hodnoty a=1, b=2, c=3
  f(6);           // implicitni hodnoty b=2, c=3
  f(6, 7);        // implicitni hodnota c=3
  f(6, 7, 8);     // vsechny argumenty zadany explicitne
}
```

V ukázce voláme čtyřikrát funkci `f()`, v prvních třech případech se uplatní implicitní hodnoty parametrů. Výstup programu je následující.

```
1 2 3
6 2 3
6 7 3
6 7 8
```

Pokud chceme definovat implicitní hodnotu pro některý parametr, musíme definovat implicitní hodnoty i pro všechny následující parametry funkce. Nelze tedy například napsat

```
void g(int a=1, int b, int c=3) { /* ... */ }     // !!! nelze !!!
```

4.6 Nepoužité parametry

Pokud s některým parametrem funkce nepracuje, nemusíme v seznamu parametrů uvádět jeho identifikátor. K takové situaci může dojít například v procesu ladění funkce, kdy nejsou ještě naprogramovány všechny její části. Pokud v seznamu parametrů uvedeme identifikátor, se kterým pak ve funkci nepracujeme, bude nás kompilátor varovat. Varovnou zprávu potlačíme ponecháme-li v seznamu parametrů pouze jméno typu.

```
void g(int, int b)
{
   cout << "prvni argument ignorovan, b = " << b << endl;
}
```

V prototypu funkce (v její deklaraci) můžeme identifikátory parametrů vynechat, jejich uvedení má význam pouze pro lepší čitelnost textu. V deklaraci funkce mají identifikátory *oblast platnosti prototyp* (platí pouze v daném prototypu).

4.7 Parametry typu pole

Pokud použijeme pole jako parametr funkce, je při volání předán ukazatel na jeho první prvek. Pole představují výjimku v předávání argumentů, pole nelze volat hodnotou. Každá změna prvků pole se projeví i vně funkce.

Jako skutečný argument můžeme uvádět libovolný výraz, jehož hodnota je typu ukazatel na daný typ prvku pole. Je-li parametrem funkce statické pole (s konstantní dimenzí), je situace celkem jednoduchá. V případech, kdy parametrem je pole, jehož dimenze se určuje až za běhu programu, nemusíme dimenzi v deklaraci parametru uvádět. Obvykle ale funkci musíme nějak předat informaci o skutečné dimenzi pole. U textových řetězců je to koncový znak `'\0'`, dimenzi předáváme implicitně. V ukázce funkce pro výpočet skalárního součinu předáváme dimenzi explicitně jako samostatný argument.

```
#include <iostream>

using namespace std;

double sks(double a[], double b[], int dim)
{
  double s = 0;
  for (int i=0; i<dim; i++)
    s += a[i]*b[i];
  return s;
}

int main()
```

```
{
  double x[] = {2.3, 6.8, 4.7, 0.2};
  double y[] = {0.6, 2.1, 7.0, 1.3};
  cout << sks(x, y, 4) << endl;
}
```

Vzhledem k úzké vazbě mezi poli a ukazateli se často můžeme setkat v podobných případech s funkcí, která má parametry deklarované jako typ ukazatel.

```
double sks(double* a, double* b, int dim);
```

Pole představují zdroj potenciálních problémů a je lépe nepracovat s nimi přímo, ale zapouzdřit je v bezpečnějších datových strukturách. Nástrojem pro to jsou třídy. Standardní C++ knihovna poskytuje širokou škálu *kontejnerů*, které pole mohou často nahradit. Pole a ukazatele jsou vhodné pro interní reprezentaci datových struktur, ale ne pro přímou manipulaci.

4.7.1 Vícerozměrná pole

Parametry vícerozměrných polí s pevnými dimenzemi nepředstavují žádný problém.

```
void f(double p[3][4]) { /* ... */ }
```

Snadno můžeme také předávat jako argumenty pole, u kterých je proměnná pouze první dimenze. Funkce `teziste` počítá těžiště seznamu souřadnic uložených v dvojrozměrném poli.

```
void teziste(const double yx[][2], int dim, double& y, double& x)
{
  y = x = 0;
  for (int i=0; i<dim; i++)
  {
    y += yx[i][0];
    x += yx[i][1];
  }
  y /= dim;
  x /= dim;
}
```

Specifikace `const` v deklaraci prvního parametru informuje překladač, že funkce `teziste` pole `yx` nemodifikuje.

Nelze psát takovéto deklarace

```
void f(double matice[][], int rad, ind slp);  // !!! chyba !!!
```

Jedna z možností, jak předat funkci matici (dvourozměrné pole, které má obě dimenze proměnné), je ta, že vytvoříme pole ukazatelů na jednotlivé řádky.

```
void tisk(double** m, int rad, int slp)
{
  using namespace std;

  for (int i=0; i<rad; i++)
```

```
  {
    double* r = m[i];
    for (int j=0; j<slp; j++)
      cout << r[j] << ' ';
   cout << endl;
  }
}
```

Pole v tomto pojetí představují konstrukce velmi nízké úrovně a jsou potenciálním zdrojem chyb. Pro implementaci objektů jako matice jsou ideálním nástrojem C++ třídy.

4.8 Přetížení funkcí

V C++ lze definovat více funkcí se stejným jménem, pokud je lze rozlišit podle seznamu parametrů. Hovoříme o *přetížení funkcí (overloaded functions)*. Typ návratové hodnoty se při rozlišování přetížených funkcí neuplatní.

Kompilátor vybírá při volání přetížené funkce vhodnou funkci na základě porovnání argumentů použitých při volání a parametrů z deklarace funkce. Zjednodušeně je postup tento:

- výběr podle přesné shody nebo po provedení nezbytných konverzí (jméno pole na ukazatel a pod.)
- numerické konverze (`char` na `int`, `short` na `int`, `float` na `double` a pod.)
- standardní konverze (`int` na `double`, *ukazatel na odvozenou třídu* na typ *ukazatel na základní třídu* a pod.)
- uživatelem definované konverze

Všechny funkce `vzdalenost` z následující ukázky počítají vzdálenost dvou bodů pro různé typy argumentů.

```
#include <cmath>

using namespace std;

struct yx  { double y; double x; };
struct bod { long cislo; double y; double x; };

double vzdalenost(double ya, double xa, double yb, double xb)
{
  double dy = ya - yb;
  double dx = xa - xb;
  return sqrt(dy*dy + dx*dx);
}

double vzdalenost(yx a, yx b)
{
  double dy = a.y - b.y;
  double dx = a.x - b.x;
  return sqrt(dy*dy + dx*dx);
```

```
}

double vzdalenost(bod a, bod b)
{
  return vzdalenost(a.y, a.x, b.y, b.x);
}
```

4.9 Ukazatel na funkci

Mějme funkci, která vrací typ `int` a má dva parametry typu `double`

```
int f(double a, double b) { /* ... tělo funkce ... */ }
```

Typ ukazatel na funkci, která má typ `int` a dva parametry typu `double`, deklarujeme takto

```
int (*pf)(double a, double b);   // pf je ukazatel na funkci
```

Jak zapsat deklaraci ukazatele na funkci si můžeme snadno zapamatovat takto. Napíšeme prototyp dané funkce ve kterém jméno funkce zaměníme deklarací ukazatele. Deklaraci jména ukazatele ale musíme uzavřít do závorek, protože bez nich je

```
int *pf(double a, double b);     // deklarace funkce typu int*
```

deklarací funkce, se jménem `pf`, která vrací ukazatele na `int` (funkce, která vrací `int*`).

Hodnotu ukazateli `pf` přiřadíme příkazem

```
pf = f;                          // nebo též:  pf = &f;
```

Operátor získání adresy můžeme v tomto případě vynechat, protože identifikátor funkce je výraz, jehož hodnota je adresa dané funkce.

Přes ukazatel voláme funkci následovně

```
int k = pf(1.2, 2.3);            // nebo též:  int k = (*pf)(1.2, 2.3);
```

Ukazatel na funkci je svázán s typem funkce a seznamem jejích parametrů. Ukazatele na jiné typy funkcí deklarujeme obdobně.

V C++ můžeme deklarovat například pole ukazatelů na funkce („pole funkcí“). Obvykle je pak pro lepší čitelnost vhodné zavést náhradní jméno pro typ ukazatel na funkci pomocí deklarace `typedef`.

```
typedef int (*TP)(double, double);
TP pole_funkci[10];
```

`TP` je zde náhradní jméno pro typ ukazatel na funkci. Deklarací `typedef` nevytváříme nový typ ale pouze náhradní jméno pro již existující typ.

Analogicky můžeme deklarovat ukazatel na typ pole

```
int (*PP)[10];                   // ukazatel na pole o 10 prvcích typu int
```

Ukazatel na funkci nám umožňuje předat funkci jako argument. Funkce `derivace` z následující ukázky počítá odhad derivace funkce jedné proměnné v daném bodě. Ukázka demonstruje předání funkce jako argumentu a ne numerický algoritmus.

```
#include <iostream>
#include <cmath>

double derivace(double (*F)(double), double x)
{
  double dx  = 0.001;
  double Fx  = (*F)(x);
  double dF1 = ((*F)(x + dx) - Fx) / dx;    dx /= 2;
  double dF2 = ((*F)(x + dx) - Fx) / dx;

  while (std::abs(dF2 - dF1) > 1e-6)
    {
      dF1 = dF2;
      dx /= 2;
      dF2 = ((*F)(x + dx) - Fx) / dx;
    }

  return dF2;
}

int main()
{
  float       (*pf)(float)       = &std::sin;
  double      (*pd)(double)      = &std::sin;
  long double (*pl)(long double) = &std::sin;

  std::cout << derivace(pd, 0.2) - std::cos(0.2) << std::endl;
}

-7.7606e-07
```

Předchozí příklad zároveň demonstruje, že lze získat i adresu přetížené funkce. Výběr příslušné přetížené funkce v takovém případě určuje typ cílové proměnné.

V C++ můžeme definovat uživatelskou funkci, která bude automaticky volána, pokud operátor `new` nedokáže alokovat požadovanou paměť. Toto řešení je považováno za zastaralé, korektní způsob, jak testovat podobné situace, je přes testování *výjimky (exception)*. Zde nám ale jde o ukázku předání funkce jako argumentu.

```
#include <iostream>
#include <new>           // set_new_handler
#include <cstdlib>       // exit

void neni_volna_pamet()
{
  std::cout << "nelze alokovat pozadovanou pamet\n";
  std::exit(0);         // predcasne ukoceni programu
}
```

```
int main()
{
  const long N = long(1e9);

  std::set_new_handler(&neni_volna_pamet);     // nastavim ovladac

  while (true)
    {
      char*p =  new char[N];
      std::cout << "alokovano " << double(N) << " bajtu\n";
    }
}

nelze alokovat pozadovanou pamet
```

4.10 Výpustka v seznamu parametrů funkce

Seznam parametrů funkce může obsahovat tzv. *výpustkou (ellipsis)*, která umožňuje psát funkce s proměnným počtem parametrů. Výpustka musí být v seznamu parametrů uvedena jako poslední a zapisuje se jako „. . .“ (trojice teček). Například

```
int printf(const char*, ...);
```

deklaruje funkci, která může být volána s proměnným počtem parametrů, první parametr je ale povinný a musí být typu `const char*`

```
printf("hello, world!");
printf("a=%d b=%6.1f", a, b);
```

Protože v zápisu výpustky je syntakticky správné `", ..."` synonymem pro `"..."`, mohli bychom funkci `printf` deklarovat i takto

```
int printf(const char* ...);
```

Mechanismus zpracování argumentů předávaných přes výpustku je definován ve standardní hlavičce `<cstdarg>`. Funkce s proměnným počtem parametrů definovaných prostřednictvím výpustky jsou jedním z temných dědictví jazyka C a v jazyce C++ bychom se jim měli důsledně vyhýbat. Jde o potenciálně nebezpečnou konstrukci, protože kompilátor nemá ve výpustce možnost kontrolovat typy použitých argumentů ani jejich počet, jak ukazuje následující příklad

```
#include <iostream>
#include <cstdarg>

void f(int k, ...)
{
  using namespace std;
  double    d;
  va_list  ap;                // seznam argumentu ve vypustce
  va_start(ap, k);            // incializace seznamu, kde k je
                              //    posledni parametr pred vypustkou
```

```
  for (int i=0; i<k; i++)
    {
      d = va_arg(ap, double);  // va_arg je makro, ktere vraci dalsi
      cout << d << " ";        //    argument ze zadaneho typu
    }

  va_end(ap);                  // uzavreni seznamu argumentu
  cout << endl;
}

int main()
{
  f(1, 1.1);
  f(2, 2.1, 2.2);
  f(3, 3.0);            // CHYBA: mene argumentu nez ocekavano
  f(3, 4.1, 4.2, 5);    // CHYBA: posledni argument je jineho typu
}
```

```
1.1
2.1 2.2
3 10.4479 -1.99936
4.1 4.2 3
```

4.11 Specifikace výjimek

V deklaraci funkce můžeme volitelně specifikovat i seznam výjimek, které funkce může vyvolat. Například funkce

```
void q() throw(e1, e2);
```

deklaruje, že nevyvolá jiné výjimky než `e1`, `e1` anebo jejich veřejné potomky.

Seznam výjimek v deklaraci funkce může být i prázdný, říkáme tím, že daná funkce žádnou výjimku nikdy nevyvolá. Pokud volitelnou specifikaci výjimek v deklaraci funkce neuvedeme, znamená to, že tato funkce může vyvolat libovolnou výjimku. Explicitní specifikaci výjimek v deklaraci funkce může překladač využít pro generování efektivnějšího kódu, v praxi se ale s touto technikou často nesetkáváme.

```
void f();                       // funkce může vyvolat libovolnou výjimku
void g() throw();               // funkce garantuje, že nevyvolá žádnou výjimku
```

Pokud funkce definovaná se seznamem výjimek vyvolá nespecifikovanou výjimku, je implicitně volána funkce `unexpected`, která zavolá funkci `terminate` a program skončí chybou. Uživatel může definovat vlastní ovladač `unexpected`, který musí buď zavolat funkci `terminate` nebo vyvolat výjimku.

Jestliže funkce deklaruje seznam výjimek, musí každá další její deklarace i definice specifikovat identickou sadu typů povolených výjimek.

```
void f() throw (std::exception);  // výjimka definovaná v hlavičce <exception>

void f() throw                    // CHYBA : prázdný seznam výjimek
{
   // ...
}
```

V kapitole 6 věnované dědičnosti se seznámíme s *virtuálními funkcemi*. Virtuální funkce mohou být v odvozené třídě deklarovány s jinou specifikací výjimek než v bázové třídě. Specifikace výjimek v odvozené třídě ale musí být restriktivnější než specifikace výjimek uvedená v bázové třídě.

```
class B {
public:
   virtual void f();
   virtual void g() throw(X, Y);
   virtual void h() throw(X);
};

class D : public B {
public:
   void f() throw(X);       //  OK     D::f() je restriktivnější než B::f()
   void g() throw(X);       //  OK     D::g() je restriktivnější než B::g()
   void h() throw(X, Y);    // CHYBA:  D::h() je méně restriktivní než B::h()
};
```

Obdobná pravidla týkající se restrikce sady výjimek platí pro ukazatele na funkce. Nelze přiřadit ukazatel na *volnější* funkci do ukazatele na funkci s restriktivnějšími omezeními. Například

```
void (*pf1)();
void (*pf2)() throw(int);

void f()
{
   pf1 = pf2;               //  OK     pf1 je méně restriktivní než pf2
   pf2 = pf1;               // CHYBA:  pf2 je restriktivnější
}
```

Specifikace seznamu výjimek není součástí typu dané funkce a nemůže být proto uváděna v deklaraci `typedef`

```
typedef void (*PF)() throw(std::exception);      // nelze!
```

Kapitola 5

Třídy

Základním nástrojem objektového programování v jazyce C++ je *třída (class)*. Stručně a zjednodušeně řečeno nám třídy umožňují definovat nové typy a množinu operací, které můžeme s proměnnými těchto typů provádět. Třídám se také někdy říká *objektové typy*, proměnným objektových typů říkáme *objekty*.

Jako třídu můžeme v C++ například definovat objektový typ `matice` s obvyklými operátory pro násobení matic, sčítání matic a pod.

```
matice A, B;
cin >> A >> B;
matice C = A * B;
```

Proměnné `A`, `B` a `C` jsou objekty, někdy se pro ně též používá termín *instance objektového typu* anebo jen stručně *instance*.

Součástí definice nového typu je popis dat (tj. jak je například implementována matice) a popis pravidel, jak lze s novým typem pracovat (funkce určené výhradně pro práci s daným typem). Objektový typ obvykle představuje konkrétní realizaci jistého abstraktního pojmu (zde například matice).

Definice C++ třídy má podobnou syntax jako struktura; liší se klíčovým slovem `class` místo `struct` a implicitními přístupovými právy ke svým členům. Kromě datových členů může obsahovat třída i členské funkce, resp. funkční členy. Datovým členům budeme také říkat *atributy (data members)*, funkčním členům *metody (member functions)*.

Přístup ke členům třídy je specifikován návěštími `public`, `protected` a `private`.

`public:` označuje *veřejné* členy třídy. Tyto atributy a metody mohou být použity libovolnou funkcí, přístup k nim není omezen.

`private:` je návěští označující *soukromé* členy. Soukromé členy mohou používat pouze metody dané třídy a její *přátelé (friends)*.

`protected:` označuje *chráněné* členy, o nich budeme hovořit v souvislosti s odvozenými třídami.

V C++ můžeme metody definovat ve struktuře stejně jako ve třídě. Ve třídě jsou ale implicitně všechny členy soukromé (private); ve struktuře jsou všechny členy implicitně veřejné (public).

5.1 Veřejné a soukromé členy

Omezení přístupu ke členům třídy si ukážeme na příkladu. Posloupnost čísel, ve které je každé číslo rovno součtu dvou předchozích, je známa jako Fibonacciho posloupnost (Leonardo Fibonacci, 1202)

$$F_0 = 0, \qquad F_1 = 1, \qquad F_{n+2} = F_{n+1} + F_n, \quad n \geq 0.$$

Abstraktní pojem *Fibonacciho posloupnost* budeme realizovat jako třídu `Fibonacci`. Atributy třídy `Fibonacci` jsou dvě proměnné typu `unsigned long`, které obsahují hodnotu běžného a následujícího členu posloupnosti. Metoda `dalsi()` vrátí hodnotu běžného členu a nastaví objekt na následující člen posloupnosti.

```
#include <iostream>

using namespace std;

class Fibonacci {
public:
  Fibonacci() { Fn = 0; Fn1 = 1; }              // konstruktor
  unsigned long dalsi() {
    unsigned long c = Fn; Fn = Fn1; Fn1 += c; return c;
  }
private:
  unsigned long Fn, Fn1;
};

int main()
{
  Fibonacci F;
  for (int i=0; i<15; i++)
    cout << F.dalsi() << ' ';
  cout << " ..." << endl;
}

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377  ...
```

Třída `Fibonacci` obsahuje čtyři členy. Dvě veřejné metody `Fibonacci()` a `dalsi()` a dva soukromé atributy `Fn` a `Fn1`.

V naší ukázce je v hlavním programu vytvořen jeden objekt `F` typu `Fibonacci`.

```
Fibonacci F;
```

Pro vytvoření tohoto objektu je automaticky volána speciální metoda `Fibonacci()`, která definuje, jak má být objekt `F` vytvořen. Metodám, které vytvářejí objekty, se říká *konstruktory*. Prozatím si o nich řekneme jen to, že mají stejné jméno jako třída a nemají typ. V našem případě přiřadil konstruktor počáteční hodnoty atributů `Fn = 0` a `Fn1 = 1`. Tato inicializace se provede při vytvoření každého objektu typu `Fibonacci`.

Druhá veřejná metoda je `dalsi()`. Jde o člen třídy `Fibonacci` a voláme ji proto obdobně jako se odvoláváme na členy struktur pomocí operátoru tečka. Obdobně voláme metody i pomocí operátoru nepřímého přístupu (member selection).

```
Fibonacci* pF = new Fibonacci;
// ....
cout << pF->dalsi() << ' ';
```

Metoda `dalsi()` nemá žádné parametry, pracuje výhradně s atributy daného objektu.

Objekt `F` v naší ukázce má jednoznačně definovaný stav, který může být změněn pouze voláním metody `dalsi()`. Pokud bychom se pokusili změnit hodnotu některého ze soukromých atributů, šlo by o chybu, kterou by odhalil při překladu kompilátor.

```
int main() {
// ....
F.Fn++;           // !!! nelze !!!
```

Metody vytvářejí *rozhraní (interface)* přes které komunikujeme s objekty. Atributy třídy obvykle bývají soukromé (nebo chráněné). Pokud je změněna implementace třídy a zůstane zachováno její rozhraní (prototypy jejích metod), neovlivní to programy, které se třídou pracují.

Máme-li například třídu `BodD2`, která interně pracuje s polárními souřadnicemi v rovině

```
class BodD2 {
  double r, a;     // tyto atributy jsou implicitně private
public:
// ....
  void xy(double& x, double& y) { x = r*cos(a); y = r*cos(a); }
// ....
};
```

a napíšeme novou verzi třídy, která má jako soukromé atributy kartézské souřadnice

```
class BodD2 {
  double x_, y_;
public:
// ....
  void xy(double& x, double& y) { x = x_; y = y_; }
// ....
};
```

pak se tato změna nijak nedotkne programů, které byly odladěny s původní verzí třídy (musíme je pouze znovu přeložit). Definice jasného rozhraní pro komunikaci s objekty dané třídy umožňuje mimo jiné rozdělit práci na velkých projektech mezi více programátorských týmů, které pracují nezávisle na různých částech projektu.

> Každý objekt je nezávislou entitou s vlastní životností, interním stavem a sadou nabízených služeb. Objekty by měly být živí, zodpovědní a inteligentní pomocníci.
>
> Objekty nepředstavují pouze jednoduchý a vhodný způsob, jak sdružit datové struktury a funkce dohromady. Základním smyslem objektu je, aby poskytoval služby.

5.2 Definice metod vně třídy

Metody zapsané přímo v definici třídy jsou implicitně chápány jako `inline` (tzv. *vložené funkce*) a v definici funkce obvykle zapisujeme jen triviální jednořádkové funkce. Metodu můžeme ale ve třídě pouze deklarovat, tj. zapíšeme ve třídě pouze její prototyp a její definici uvedeme jinde. V takovém případě musíme v definici funkce kvalifikovat její jméno pomocí operátoru rozlišení oboru. Součástí jména metody je i jméno příslušné třídy.

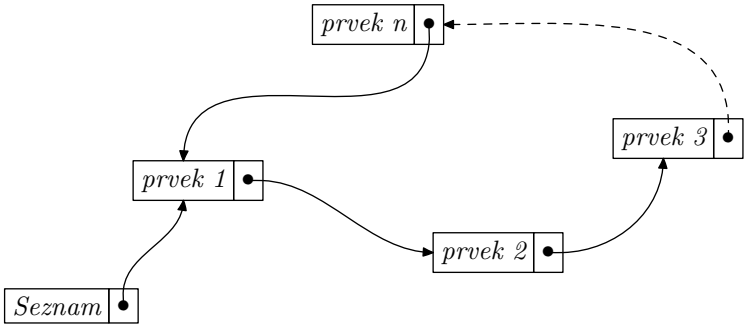
```
#ifndef X_h_
#define X_h_

class X {
  int a, b;
public:
  // ....
  int rozdil() { return a - b; }  // implicitně inline metoda
  int vypocet();                  // pouze prototyp funkce
  // ....
};

#endif

// ....  metoda výpočet bude obvykle zapsána v jiném souboru  ....

int X::vypocet() { /* .... */ }   // definice členské funkce mimo třídu
```

Deklarace třídy obsahující pouze vložené funkce se obvykle umísťuje do hlavičkového souboru. Technikou podmíněného překladu zajistíme, že v jednom souboru bude deklarace třídy zařazena jen jednou. Metody deklarované vně třídy se umístí do samostatného souboru, resp. souborů.

5.3 Ukazatel `this`

Metody (členské funkce) se odkazují na atributy objektu (datové členy) pouze jménem atributu. Metody jsou volány vždy pro určitý objekt a jméno atributu je jednoznačné.

```
class A {
  int n;
public:
  void nastav(int i) { n = i; }
  void prevod(const A&);
  // ....
};

int main()
{
  A a, b;
  a.nastav(1);    // nastaví a.n = 1
  b.nastav(2);    // nastaví b.n = 2
}
```



Obrázek 5.1: Schéma propojení prvků v jednosměrném cyklickém spojovém seznamu

Pokud ale má daná metoda parametr stejného objektového typu, musíme se na atributy parametru odvolávat pomocí operátoru *tečka*.

```
void A::prevod(const A& x) { n = x.n; }
```

V každém objektu je k dispozici ukazatel na tento objekt `this` (můžeme si představit, že v každé třídě `T` je implicitně deklarován atribut `T *const this;`). Hodnotu ukazatele `this` nemůžeme měnit. Ukazatel `this` není k dispozici ve statických metodách (viz dále).

Ukazatel `this` lze například použít pro odkaz na atribut pokud parametr metody zakryje viditelnost atributu stejného jména.

```
void A::nastav(int n) { this->n = n; }
```

Obdobně lze explicitně použít operátor rozlišení oboru

```
void A::nastav(int n) { A::n = n; }
```

Příklad

Typické použití ukazatele `this` je při vytváření různých spojových seznamů. Jednoduchý neprázdný cyklický spojový seznam je použit v následujícím příkladu (schéma propojení prvků znázorňuje obrázek 5.1). Program načítá ze standardního vstupu slova (posloupnost znaků ohraničená bílými znaky) a počítá jejich četnosti výskytu.

```
#include <iostream>
#include <string>

using namespace std;

class Prvek {
public:
  Prvek(const string& t) { s = t; pocet = 1; dalsi = this; }
  void pridej(const string&);
  void tisk();
private:
  string s;             // slovo
  int pocet;            // pocet vyskytu
```

```
  Prvek* dalsi;         // ukazatel na dalsi Prvek
};

void Prvek::pridej(const string& t)
{
  Prvek* q;
  Prvek* p = this;
  do {
    if (t == p->s) {    // slovo nalezeno
      p->pocet++;
      return;
    }
    q = p;              // predchozi prvek
    p = p->dalsi;       // bezny prvek
  }
  while (p != this);

  p = new Prvek(t);     // novy prvek
  p->dalsi = q->dalsi;  // pridam za predchozi
  q->dalsi = p;
}

void Prvek::tisk()
{
  int n = 1;
  Prvek* p = this;
  do {
    cout << p->pocet << " " << p->s;
    for (int i=p->s.length(); i<13; i++)
      cout << ' ';
    if (n++ == 5) { cout << endl; n = 1; }
    p = p->dalsi;
  }
  while (p != this);
  if (n != 1) cout << endl;
}

int main()
{
  string s;
  cin >> s;
  Prvek* seznam = new Prvek(s);

  while (cin >> s)
    seznam->pridej(s);

  seznam->tisk();
}
```

Pro vstupní text

```
A default constructor for a class X is a constructor of class X that
can be called without an argument
```

je výstup programu

```
1 A           1 default     2 constructor  1 for         2 a
2 class       2 X           1 is           1 of          1 that
1 can         1 be          1 called       1 without     1 an
1 argument
```

5.4 Konstruktory a destruktory

Existují speciální metody označované jako konstruktory, které jsou zodpovědné za vytváření objektů. Dále existují metody, kterým říkáme destruktory, které jsou volány při ukončení existence objektu. Jako motivace pro zavedení konstruktorů a destruktorů nám poslouží příklad třídy pro dynamicky alokované pole.

V odstavci 3.5.1 jsme hovořili o vztahu ukazatelů (pointerů) a polí. Řekli jsme si, že pokud proměnné typu ukazatel přiřadíme adresu dynamicky alokované paměti (pomocí operátoru `new`), můžeme dále s touto proměnnou pracovat jako s dynamicky alokovaným jednorozměrným polem.

```
// ....
int N = 20;
double p* = new double[N];
// ....
for (int i=0; i<N; i++)
  cin >> p[i];
// ....
delete[] p;
```

Při práci s takto definovaným dynamickým polem musí být vždy splněny následující předpoklady.

- Pole musí mít řádně přidělenu paměť pomocí operátoru `new`
- Použité indexy nesmí být mimo přípustný rozsah
- Přidělená paměť musí být uvolněna explicitním volání operátoru `delete`

Při porušení některého z uvedených předpokladů skončí program v lepším případě fatální chybou; v horším případě se zdánlivě nic nestane a my podlehneme falešnému dojmu, že program dělá to, co chceme. To hrozí zvláště u třetího předpokladu, kdy se za chodu programu *ztrácí paměť (memory leak).*

Pokusme se abstraktní pojem *dynamické pole* popsat bez aparátu, který nabízí C++ třídy. Dynamické pole v našem pojetí je jednoznačně definováno adresou alokované paměti a dimenzí. Dimenze různých polí přitom obecně nemají žádnou souvislost. Dynamické pole bychom proto nejlépe popsali jako strukturu o dvou složkách.

```
struct dPole {
  int N;
  double* p;
};

dPole vytvor_pole(int dim)
```

```
{
  dPole pom;
  pom.N = dim;
  pom.p = new double[dim];
  return pom;
}

void uvolni_pole(dPole& a)
{
  delete[] a.p;
  a.N = 0;
  a.p = 0;
}

void f()
{
  dPole x = vytvor_pole(100);
  dPole z = vytvor_pole(50);
  // ....
  uvolni_pole(x);
  uvolni_pole(z);
}
```

Struktura `dPole` definuje uživatelský typ (můžeme deklarovat proměnné typu `dPole`), přípustné operace jsme popsali pomocí sady funkcí jako jsou například `vytvor_pole` a `uvolni_pole`. To ale nikterak nezaručuje, že programátor bude tyto funkce správně používat.

Takto definovaný typ `dPole` není bezpečný v tom smyslu, že programátor má neomezený přístup ke složkám struktury `dPole`.

```
// ....
dPole w = vytvor_pole(500);
w.p--;            // chci indexovat pole od 1 do N
// ....
uvolni_pole(w);  // !!! kolaps !!!
```

Podobné konstrukce vedou navíc k implementačně závislým programům. Pokud se změní definice typu `dPole`, není zaručeno, že programy využívající přímo jednotlivé složky struktury, budou i nadále pracovat.

Při realizaci abstraktního pojmu, jakým je například *dynamické pole*, potřebujeme obvykle zakrýt jeho konkrétní reprezentaci a vytvořit jasně definované rozhraní (množinu funkcí), které slouží k výhradní komunikaci s objekty daného typu. Dynamické pole bychom mohli jako třídu napsat třeba takto.

```
class dPole {    // 1. verze
public:
  dPole(int dim) { N = dim; p = new double[N]; }  // konstruktor
 ~dPole() { delete[] p; }                         // destruktor
  double& prvek(int i) { return p[i]; }
  int dim() const { return N; }
private:
  int N;
  double* p;
};
```

V prvním návrhu třídy `dPole` jsme definovali konstruktor s jedním parametrem typu `int`, který vytváří dynamické pole o zadané dimenzi. Třída může mít obecně více konstruktorů, které se musí lišit seznamem svých parametrů. Dále jsme definovali destruktor, který uvolňuje alokovanou paměť při ukončení existence objektu. Jméno destruktoru je tvořeno *tildou* a jménem třídy, zde `~dPole()`. Destruktor nesmí mít parametry.

Metoda `prvek(int)` zpřístupňuje jednotlivé prvky dynamického pole. Tato členská funkce vrací referenci na typ `double` a její volání může proto být uvedeno i vlevo od operátoru přiřazení.

```
void f()
{
  dPole w(100);               // zde je volan konstruktor dPole(int)
  for (int i=0; i<w.dim(); i++)
    w.prvek(i) = i;
  // ....
}                             // destruktor pro w se vola automaticky
```

Pro jednoduchost v našem návrhu třídy `dPole` netestujeme platnost argumentu použitého při volání metody `prvek()` ani platnost zadané dimenze v konstruktoru. V tomto případě by bylo nejlepším řešením vyvolání výjimky, s mechanismem výjimek se ale seznámíme až v kapitole 8.

Metoda `dim()` vrací dimenzi daného objektu (dynamického pole). Tato metoda je definována jako *konstantní* (uvedením klíčového slova `const` za seznamem parametrů). Konstantní metody mohou být volány pro konstantní objekty, nemění stav objektu. Podrobněji se ke konstantním metodám vrátíme v souvislosti s deklarací atributů specifikovaných jako `mutable`. Metoda `dim()` je typickou ukázkou `inline` metody.

Konstruktory a destruktor třídy `T` mají tyto základní vlastnosti:

- konstruktor má stejné jméno jako třída a nemá typ: `T(`*seznam parametrů*`)`.
- třída může mít více konstruktorů, které se liší seznamem parametrů.
- konstruktor s prázdným seznamem parametrů se nazývá *implicitní konstruktor*.
- destruktor třídy `T` má jméno `~T()` a nesmí mít parametry.
- třída může mít pouze jeden destruktor.
- pokud má třída definován alespoň jeden konstruktor, jsou její instance vždy inicializovány. Buď má totiž třída definován implicitní konstruktor nebo musí být její instance vytvořena konstruktorem s explicitně zadanými parametry.
- pokud ve třídě není definován žádný konstruktor, použije se implicitní konstruktor, vytvořený kompilátorem. Není-li ve třídě definován destruktor, použije se při rušení objektů destruktor vytvořený kompilátorem.

O dalších vlastnostech konstruktorů a destruktorů si řekneme dále.

5.4.1 Předávání argumentů konstruktorům

Pokud je třeba předat při vytváření objektu konstruktoru argumenty, zapisujeme jejich seznam za identifikátor objektu v příkazu deklarace.

```
class Komplexni {
  double re, im;        // tyto cleny jsou implicitne private
public:
  Komplexni() { re = im = 0; }      // implicitni konstruktor
  Komplexni(double r) { re = r; im = 0; }
  Komplexni(double r, double i) { re = r; im = i; }
  // ....
};

Komplexni f()
{
  Komplexni alfa;
  Komplexni beta(1.0);
  Komplexni gama(0.0, -1.0);
  // ....
  double x = 2, y = 3;
  // ....
  return Komplexni(x+y, x-y);      // konstruktor ve vyrazu
}
```

Ve třídě `Komplexni` jsou definovány tři konstruktory, které jsou v předchozí funkci volány pro vytvoření objektů `alfa`, `beta` a `gama`.

Funkce `f()` vrací hodnotu typu `Komplexni`. V příkazu `return` je přímo volán konstruktor. V takovém případě vytvoří kompilátor lokální nepojmenovanou instanci daného typu, která je po předání volajícími programu automaticky zrušena.

Tři konstruktory z předchozí ukázky bychom mohli nahradit jediným konstruktorem se dvěma parametry s implicitní hodnotou.

```
class Komplexni {
  double re, im;
public:
  Komplexni(double r=0, double i=0) { re = r; im = i; }
  // ....
};
```

Konstruktor, který má pro všechny parametry definovanou implicitní hodnotu, je také *implicitní konstruktor* (konstruktor, který umožňuje vytvořit objekt bez explicitně zadaných argumentů).

Pokud je ve třídě `T` definován konstruktor s jedním parametrem typu `P`, definuje implicitní uživatelskou konverzi z typu `P` na typ `T`. V následujícím ukázce převede kompilátor hodnotu `1.0` typu `double` nejprve na typ `Komplexni` a pak ji teprve sečte s komplexní proměnnou `x` (předpokládejme, že jsme pro tento typ definovali operátor sčítání `+`).

```
Komplexni z, x(2.0, 5.2);
z = 1.0 + x;    // kompilátor nejprve převede 1.0 na Komplexni(1.0)
```

V uvedeném případě má implicitní konverze z typu `double` na typ `Komplexni` smysl. V řadě případů je ale implicitní konverze definovaná konstruktorem s jedním parametrem nežádoucí. Implicitní konverzi zabráníme, pokud deklaraci konstruktoru uvedeme klíčovým slovem `explicit`. Konverze prostřednictvím konstruktoru s jedním parametrem pak musíme v programu vypisovat explicitně.

```
class Komplexni {
  double re, im;
public:
  explicit Komplexni(double r=0, double i=0) { re = r; im = i; }
  // ....
};

void f()
{
  Komplexni z, x(2.0, 5.2);
  z = Komplexni(1.0) + x;      // explicitni konverze
  // ...
}
```

Konstruktor třídy `T`, s jedním parametrem typu *reference na* `T` (`T&`) je tzv. *kopírovací konstruktor (copy constructor)* a věnujeme mu samostatný odstavec 5.11.

5.4.2 Atributy objektových typů

Atributy třídy mohou být objektové typy. Jejich konstruktory se volají před konstruktorem dané třídy. Pokud potřebujeme předat argumenty konstruktorům objektových atributů, uvedeme je za dvojtečku za seznam parametrů konstruktoru dané třídy.

Při rušení se nejprve volá destruktor dané třídy a destruktory objektových atributů v obráceném pořadí než ve kterém byly uvedeny v definici konstruktoru (tj. v opačném pořadí než v jakém byly vytvořeny).

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

class X {
public:
  X(int i) { cout << "X ctor : " << i << endl; }
 ~X() { cout << "X dtor" << endl; }
};

class Y {
public:
  Y(int i, int j) { cout << "Y ctor : " << i << ' ' << j << endl; }
 ~Y() { cout << "Y dtor" << endl; }
};

class Z {
  X x_;
  Y y_;
public:
  Z(int n)
    : x_(10*n), y_(2*n, -n)  // predani argumentu konstruktorum X a Y
```

```
  { cout << "Z ctor : " << n << endl; }
  ~Z() { cout << "Z dtor" << endl; }
};

int main() { Z(1); }
```

```
X ctor : 10
Y ctor : 2 -1
Z ctor : 1
Z dtor
Y dtor
X dtor
```

Stejně můžeme inicializovat i atributy standardních typů. Často je pak tělo konstruktoru prázdné.

```
class Komplexni {
  double re, im;
public:
  explicit Komplexni(double r=0, double i=0) : re(r), im(i) {}
  // ....
};
```

Pokud konstruktor objektového datového členu argumenty nepotřebuje (atribut lze vytvořit pomocí implicitního konstruktoru), nemusíme jej uvádět. Pokud je to možné, je vhodné volat konstruktory objektových atributů předem a vyhnout se volání dalších metod pro nastavení jejich stavu v konstruktoru. Typickým příkladem je situace, kdy implicitní konstruktor alokuje paměť, která musí být realokována metodou volanou v konstruktoru dané třídy.

5.4.3 Lokální a statické objekty

Konstruktor lokálních objektů je volán vždy, když je řízení předáno na příkaz deklarace lokálního objektu. Při ukončení platnosti lokálního objektu je volán destruktor.

Statické objekty jsou vytvořeny pouze při prvním průchodu obdobně, jako jsou pouze jednou inicializovány statické lokální proměnné standardních typů. Destruktor statických objektů je volán při ukončení programu.

```
#include <iostream>
#include <string>

using namespace std;

class X {
  string s;
public:
  X(char* t) : s(t) { cout << "ctor : " << s << endl; }
 ~X() { cout << "dtor : " << s << endl; }
};

void f()
{
```

```
  X x("lokalni objekt x");
  static X s("staticky lokalni objekt s");
}

int main()
{
  static X m("staticky lokalni objekt m");
  for (int i=0; i<3; i++)
  {
    cout << "i = " << i << endl;
    f();
  }
  cout << "... konec funkce main() ..." << endl;
}
```

Ve výpisu výstupu předchozího programu je demonstrováno volání konstruktorů a destruktorů. Zkratka *ctor* označuje konstruktor a *dtor* destruktor.

```
ctor : staticky lokalni objekt m
i = 0
ctor : lokalni objekt x
ctor : staticky lokalni objekt s
dtor : lokalni objekt x
i = 1
ctor : lokalni objekt x
dtor : lokalni objekt x
i = 2
ctor : lokalni objekt x
dtor : lokalni objekt x
... konec funkce main() ...
dtor : staticky lokalni objekt s
dtor : staticky lokalni objekt m
```

Lokální objekty jsou rušeny při ukončení bloku v obráceném pořadí než byly vytvořeny. Statické objekty jsou rušeny při ukončení programu v obráceném pořadí než byly vytvořeny .

5.4.4 Globální a dynamicky vytvořené objekty

Objekty můžeme deklarovat jako globální proměnné. Konstruktory globálních objektů jsou volány před zahájením funkce `main()` a jsou rušeny po jejím ukončení. Destruktory globálních objektů jsou volány v obráceném pořadí než jejich konstruktory.

Objekty můžeme vytvářet pomocí operátoru `new` a rušit je pomocí operátoru `delete` analogicky jako proměnné standardních typů.

Pro objekty vytvořené operátorem `new` se ale volá destruktor pouze při explicitním použití operátoru `delete`.

```
#include <iostream>
#include <string>

using namespace std;
```

```
class X {
  string s;
public:
  X(char* t) : s(t) { cout << "ctor : " << s << endl; }
 ~X() { cout << "dtor : " << s << endl; }
};

int main()
{
  cout << "... zacatek funkce main() ..." << endl;
  X m1("lokalni objekt m1");
  static X m2("staticky lokalni objekt m2");
  X* px1 = new X("dynamicky vytvoreny  objekt d1");
  X* px2 = new X("dynamicky vytvoreny  objekt d2");
  delete px1;
  cout << "... konec funkce main() ..." << endl;
}

X g("globalni objekt g");
```

Destruktory se opět volají v obráceném pořadí než konstruktory. Pro druhý objekt vytvořený operátorem `new` není volán operátor `delete` a tudíž ani destruktor.

```
ctor : globalni objekt g
... zacatek funkce main() ...
ctor : lokalni objekt m1
ctor : staticky lokalni objekt m2
ctor : dynamicky vytvoreny  objekt d1
ctor : dynamicky vytvoreny  objekt d2
dtor : dynamicky vytvoreny  objekt d1
... konec funkce main() ...
dtor : lokalni objekt m1
dtor : staticky lokalni objekt m2
dtor : globalni objekt g
```

5.4.5 Pole objektů

Pole objektů třídy `T` můžeme deklarovat, pokud má třída `T` implicitní konstruktor, tj. konstruktor bez parametrů nebo konstruktor s implicitní hodnotou všech parametrů (nebo pokud třída nemá definovaný žádný konstruktor).

Destruktory pro jednotlivé prvky dynamicky alokovaného pole se volají při zrušení pole operátorem `delete[]`.

```
#include <iostream>
#include <string>

using namespace std;

class X {
  char c;
public:
```

```
  X() : c(' ') { cout << "c "; }
 ~X() { cout << "d" << c; }
  void nl() { c = '\n'; }
};

int main()
{
  X p[10];
  cout << endl;
  p[0].nl();          // volani metody nl() pro prvni objekt pole

  X* d = new X[20];   // dynamicke pole o 20 prvcich typu X
  cout << endl;
  d->nl();            // volani metody nl() pro prvni objekt pole

  delete[] d;         // zrusim dynamicky alokovane pole
  cout << " ... konec funkce main() ... \n";
}

c c c c c c c c c c
c c c c c c c c c c c c c c c c c c c c
d d d d d d d d d d d d d d d d d d d d
 ... konec funkce main() ...
d d d d d d d d d d
```

Implicitní konstruktor se volá postupně pro všechny prvky pole od prvního do posledního (od nejnižší adresy). Destruktory se volají pro prvky pole v obráceném pořadí. V naší ukázce proto pro odřádkování za posledním zrušeným prvkem pole voláme metodu `nl()` pro první prvek pole.

5.5 Přátelé třídy

Přístup k soukromým a chráněným členům třídy lze kromě metod dané třídy povolit i vybraným funkcím nebo třídám.

```
class A {
private:
  friend void f(A&);      // funkce void f(A&) je pritelem tridy A
  friend class F;         // trida F je pritelem tridy A
  int n;
  void s() { /* .... */ }
public:
  void v() { /* .... */ }
};

void f(A& a) {
  a.n = 0;               // OK - funkce void f(A&) je pritelem tridy A
}

class F {
public:
  void m() {
```

```
    A x;  x.n = 1;       // OK - trida F je pritelem tridy A
    // ....
  }
};
```

Funkci deklarujeme jako přítele tak, že ve třídě za klíčovým slovem `friend` uvedeme její prototyp. Za klíčovým slovem můžeme zapsat úplnou definici funkce, která v takovém případě bude implicitně `inline` (obdobně jako jsou implicitně `inline` metody definované ve třídě).

Třídu deklarujeme jako přítele tak, že ve třídě za klíčovým slovem `friend` uvedeme klíčové slovo `class` a jméno třídy. V takovém případě udělujeme právo přístupu k privátním členů všem metodám přátelské třídy.

Není významné, zda je deklarace `friend` uvedena v oblasti `public`, `protected` nebo `private`.

Příkladem použití deklarace `friend` mohou být třídy, které mají všechny členy soukromé a jejichž použití chceme omezit pouze na jistou skupinu funkcí a tříd. V příkladu uvedeném na straně 95 v odstavci 5.3 jsme pomocí třídy `Prvek` realizovali spojový kruhový seznam. Změníme definici třídy `Prvek` tak, že všechny její členy budou soukromé a třída bude přítelem jiné třídy `Seznam`.

```
class Prvek {        // vsechny cleny i konstruktor jsou soukrome
private:
  friend class Seznam;
  Prvek(const std::string& t) : s(t), pocet(1), dalsi(this) {}
  void pridej(const std::string&);
  void tisk();
  std::string s;        // slovo
  int pocet;            // pocet vyskytu
  Prvek* dalsi;         // ukazatel na dalsi Prvek
};
```

Původní metody `pridej()` a `tisk()` ponecháme beze změny a definujeme novou třídu `Seznam`, ve které zapouzdříme technické detaily implementace našeho seznamu

```
class Seznam {
public:
  Seznam() : seznam(0) {}
  ~Seznam();
  void pridej_slovo(const std::string&);
  void tisk_seznamu();
private:
  Prvek* seznam;
};

void Seznam::pridej_slovo(const std::string& s)
{
  if (!seznam) {
    seznam = new Prvek(s);
    return;
  }
  else
    seznam->pridej(s);
```

```
}

void Seznam::tisk_seznamu()
{
  if (seznam)
    seznam->tisk();
}

int main()
{
  Seznam S;

  std::string w;
  while (std::cin >> w)
    S.pridej_slovo(w);

  S.tisk_seznamu();
}
```

Třída `Seznam` je přítelem třídy `Prvek`, má proto přístup k jejím privátním členům (včetně konstruktoru) a může vytvářet objekty typu `Prvek`, jak demonstruje naše ukázka. Mimo třídu `Seznam` nelze vytvořit objekt typu `Prvek`.

Uživatel třídy `Seznam` nemá možnost využít informace o interní implementaci seznamu. Programy pracující s třídou `Seznam` jsou na její implementaci nezávislé.

Zbývá ještě doplnit chybějící destruktor třídy `Seznam`

```
Seznam::~Seznam()
{
  if (Prvek* p = seznam)
    {
      Prvek* q;
      do {
          q = p;
          p = p->dalsi;
          delete q;
      } while(p != seznam);
    }
}
```

5.6 Vnořené deklarace tříd

Součástí deklarace třídy může být deklarace jiné třídy *(nested class)*. Vnořená deklarace může být uvedena v části `public` třídy a pak je viditelná i mimo vnější třídu nebo v oblasti `private` (resp. `protected`) a pak se na její viditelnost vztahují stejná omezení jako na soukromé (resp. chráněné) členy třídy.

Definice *soukromé třídy* je často vnořena v privátní části své přátelské třídy. Například třídy `Seznam` a `Prvek` z předchozího odstavce bychom mohli definovat takto.

```
#include <string>

class Seznam {
public:
  Seznam() : seznam(0) {}
  ~Seznam();
  void pridej_slovo(const std::string&);
  void tisk_seznamu();
private:

  class Prvek {          // vnorena deklarace tridy
  private:
    friend class Seznam;
    Prvek(const std::string& t) : s(t), pocet(1), dalsi(this) {}
    void pridej(const std::string&);
    void tisk();
    std::string s;
    int pocet;
    Prvek* dalsi;
  };

  Prvek* seznam;
};
```

Vnořená deklarace třídy redukuje počet globálních jmen v programu. Třída `Prvek` je skryta před uživatelem a je přístupná pouze ve třídě `Seznam`.

Definice metod vně třídy `Prvek` musí mít úplnou kvalifikaci jména, jehož součástí je v tomto případě i jméno vnější třídy.

```
void Seznam::Prvek::tisk() { /* .... */ }
```

S výjimkou explicitního použití ukazatelů, referencí a jmen objektů může vnořená třída používat pouze jména typů, výčtové typy a statické členy třídy, ve které je deklarována. Deklarování vnořené třídy samo o sobě neznamená, že vnější třída obsahuje objekt typu vnořené třídy. Pro metody vnořené třídy platí stejná přístupová práva ke členům vnější třídy jako pro jiné funkce.

5.7 Statické členy třídy

Ve třídě můžeme deklarovat *statické členy (static members).* Statické atributy jsou sdíleny všemi objekty dané třídy v programu. Statické datové členy tedy nejsou součástí objektu dané třídy ale jsou to samostatné globální objekty.

Deklarace statických členů ve třídě není definice ale pouze deklarace. V programu proto musí být někde uvedena definice statického atributu. V definici můžeme statický atribut inicializovat a to i v případě, že je deklarován jako soukromý nebo chráněný.

Statické metody nemají ukazatel `this` a k nestatickým členům mohou přistupovat pouze přes operátor `.` (operátor tečka) nebo operátor `->` (operátor nepřímého přístupu). Statické metody nemohou být virtuální (s virtuálními metodami se seznámíme v odstavci 6.4).

Statické metody můžeme volat, aniž by byl vytvořen jediný objekt daného typu. V takovém případě ale musíme uvést jejich úplné jméno pomocí operátoru rozlišení oboru (jméno třídy, čtyřtečka a jméno metody).

Statické členy tříd jsou jedním z nástrojů, který redukuje potřebu globálních proměnných v C++ programech.

Pomocí statických proměnných mohou objekty daného typu navzájem komunikovat a sdílet informace. Lze například vytvořit spojový seznam všech vytvořených objektů. Třída S v následující ukázce udržuje čítač vytvořených objektů.

```
#include <iostream>

using namespace std;

class S {
public:
  S() { citac++; }
 ~S() { citac--; }
  static int pocet() { return citac; }
private:
  static int citac;    // deklarace atributu
};

int main()
{
  cout << "Pocet objektu typu S pri zahajeni programu: "
       << S::pocet() << endl;

  S* p = new S;
  {
    S a;               // lokalni promenna v bloku
    for (int i=0; i<5; i++) {
      new S;
      S m;
      cout << m.pocet() << ' ';
    }

    cout << "\nPocet objektu typu S pri ukonceni bloku   : "
       << p->pocet() << endl;
  }

  delete p;
  cout << "Pocet objektu typu S pri ukonceni programu: "
       << S::pocet() << endl;
}

int S::citac = 0;      // definice statickeho atributu

Pocet objektu typu S pri zahajeni programu: 0
4 5 6 7 8
Pocet objektu typu S pri ukonceni bloku   : 7
Pocet objektu typu S pri ukonceni programu: 5
```

Konstantním statickým atributům můžeme předepsat inicializační hodnotu přímo v definici třídy

```
class X {
  const static int X_k = 3;
};
```

přesto ale musíme v programu někde uvést definici této proměnné

```
const int X::X_k;
```

5.8 Ukazatele na členy třídy

Pro třídu `T` deklaruje `T::*` typ ukazatel na člen třídy `T`. Pro přístup k datovým členům můžeme použít standardní ukazatele nebo ukazatele na atributy.

Ukazatel na statickou metodu je stejný jako ukazatel na globální funkci se stejnou návratovou hodnotou a stejným seznamem parametrů. Ukazatel na nestatickou metodu je v C++ jiný typ než ukazatel na *globální funkci* (tj. na funkci, která není členem nějaké třídy).

Hodnotu ukazatele na člen získáme jako výraz

&*jméno_třídy*::*jméno_členu*

Pro přístup ke členům třídy přes ukazatel do třídy slouží operátory `.*` a `->*` s analogickým významem jako operátor `.` (tečka) a operátor nepřímého přístupu `->` .

Protože nestatické metody jsou vždy vázány na konkrétní objekt (nestatické metody mají k dispozici ukazatel `this`), musíme při volání metod přes ukazatele do třídy vždy specifikovat jméno objektu, jak demonstruje následující příklad.

```
#include <iostream>

using namespace std;

class M {
public:
  M(int i=1, int j=2) : dat1(i), dat2(j) {}
  int dat1, dat2;
  void f1() { cout << "f1() " << dat1 << " " << dat2 << endl; }
  void f2() { cout << "f2() " << dat1 << " " << dat2 << endl; }
};

int main()
{
   M obj1;
   M obj2(10, 20);

   /* pouziti standardniho ukazatele pro pristup k atributu */

   int* pi;               // ukazatel na typ int
   pi = &obj1.dat1;       // adresa atributu dat1 objektu obj1
```

```
    *pi = -1;              // zmena atributu obj1.dat1

    /* ukazatel na atribut */

    int M::* pMi;          // ukazatel na atribut typu int ve tride M
    pMi = &M::dat2;        // ukazatel na atribut dat2 tridy M
    obj2.*pMi = -2;        // zmena atributu obj2.dat2

    /* ukazatel na metodu  */

    void (M::*ptr)();      // ukazatel na metodu tridy M, typu void
                           // s prazdnym seznamem parametru

    ptr = &M::f1;          // ukazatel na metodu M::f1() ... & lze vynechat
    (obj1.*ptr)();         // volam metodu f1() pro objekt obj1

    M* p2 = &obj2;         // adresa objektu obj2
    ptr = &M::f2;          // ukazatel na metodu M::f2()
    (p2->*ptr)();          // volam metodu f2() pro objekt obj2
}

f1() -1 2
f2() 10 -2
```

Ve výrazech `(obj1.*ptr)();` a `(p2->*ptr)();` musíme vzhledem k prioritám operátorů použít závorky tak, jak je uvedeno.

Pro zvýšení čitelnosti můžeme pomocí deklarace `typedef` definovat pro typ ukazatel na člen náhradní jméno. Například

```
typedef void  (M::*ukazatel_na_metodu)();
ukazatel_na_metodu ukz = M::f1;
(obj2.*ukz)();    // volani metody f1 objektu obj2 pres ukazatel ukz
```

S ukazateli na atributy třídy se setkáme spíše výjimečně, místo ukazatelů na metody je často vhodnější použít virtuální metody.

5.9 Deklarace `mutable`

V C++ můžeme deklarovat konstantní objekty. Pro konstantní objekty a pro objekty, ke kterým přistupujeme přes konstantní ukazatele, můžeme volat pouze konstantní metody. Metodu definujeme jako konstantní uvedením klíčového slova `const` za seznam parametrů.

V některých případech ale potřebujeme, aby konstantní metody mohly měnit interní stav konstantních objektů, tj. některé jejich atributy. Příkladem může být konstantní objekt typu *seznam*, jehož obsah nesmí být v programu modifikován, který ale zároveň udržuje záznamy o nejfrekventovanějších dotazech a na základě nich optimalizuje přístup.

Atributy, které mohou být modifikovány i v konstantních objektech, deklarujeme ve třídě jako `mutable`.

Třída `X` v následující ukázce udržuje čítač přístupů pro nekonstantní i konstantní objekty.

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

class X {
  mutable int citac;
public:
  X() : citac(0) {}
  int test() const { return ++citac; }
};

int main()
{
  const X x;
  for (int i=0; i<5; i++)
    cout << x.test() << endl;
}
```

5.10 Přetěžování operátorů

V C++ můžeme přetížit většinu existujících operátorů, tj. definovat *operátorové funkce*, které budou volány při aplikaci operátoru na objekty daného typu.

Operátorové funkce jsou metody nebo globální funkce, které mají jméno skládající se z klíčového slova `operator` a operátoru. Přetížit lze unární nebo binární operátory. Výraz, ve kterém je použit přetížený operátor, se vyhodnotí jako volání operátorové funkce, jak ukazuje tabulka 5.1.

operátor	výraz	jako metoda	jako globální funkce
unární prefixový	`@a`	`(a).operator@ ()`	`operator@ (a)`
binární	`a@b`	`(a).operator@ (b)`	`operator@ (a, b)`
přiřazení	`a=b`	`(a).operator= (b)`	
indexování	`a[b]`	`(a).operator[](b)`	
nepřímý přístup	`a->`	`(a).operator-> ()`	
unární postfixový	`a@`	`(a).operator@ (0)`	`operator@ (a, 0)`

Tabulka 5.1: Přetížené operátory a volání operátorových funkcí v C++

Operátory, které lze přetížit, uvádí tabulka 5.2. Přetížit nelze operátory

```
.   .*   sizeof   ::   ?:
```

ani operátory preprocesoru `#` a `##`.

Přetížení operátorů je v první řadě cestou, jak zjednodušit zápis programu. Nelze definovat nové operátory, nelze měnit prioritu existujících operátorů ani jejich syntax. Operátory `operator=`, `operator[]` a `operator->` musí být nestatické metody.

Operátor musí být buď metoda nebo globální funkce s alespoň jedním objektovým parametrem (výjimkou jsou operátory `new` a `delete`). Nelze tedy přetížit operátory, které pracují pouze se standardními

new	delete		new[]		delete[]			
+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	++	--	,	->*	->
()	[]							

Tabulka 5.2: Operátory C++, které lze přetížit

typy. Jmenovitě nelze definovat operátory, které pracují pouze s ukazateli. Lze rozšířit význam existujících operátorů ale nelze měnit jejich smysl.

```
#include <iostream>

using namespace std;

class Racionalni {
  int p, q;
  int nsd();  // nejvetsi spolecny delitel

public:
  Racionalni(int, int);
  Racionalni(int c=0) : p(c), q(1) {}

  Racionalni operator+ (Racionalni b)
  {
    return Racionalni(p*b.q+b.p*q, q*b.q);
  }
  friend Racionalni operator+(int a, Racionalni b)
  {
    return b + a;
  }
  friend ostream& operator << (ostream& o, const Racionalni& r)
  {
    return o << r.p << '/' << r.q;
  }
};

int Racionalni::nsd()
{
  int m, n, r;

  if (p > 0)
    m = p;
  else
    m = -p;
  n = q;

  for (;;){
    r = m % n;
    if (r == 0) return n;
```

```
    m = n;
    n = r;
  }
}

Racionalni::Racionalni(int c, int j) : p(c), q(j)
{
  if (q < 0) {
    p = -p;
    q = -q;
  }
  else if (q == 0)
    throw "Deleni nulou!";

  int d = nsd();
  p /= d;
  q /= d;
}


int main()
{
  Racionalni a(1, 3);

  a = a + 1;         // a = a.operator+( Racionalni(1) );
  a = 1 + a;         // a = operator+(int, Racionalni);
  cout << "a = " << a << endl;
}

a = 7/3
```

Třída `Racionalni` z předchozí ukázky definuje několik operátorů. Prvním z nich je operátor `+` pro součet dvou racionálních čísel. Pro vyhodnocení prvního součtu z hlavního programu ale nemůže být volán přímo, celočíselná konstanta `1` typu `int` musí být nejprve převedena na typ `Racionalni`. Implicitní konverze je možná, protože ve třídě je definován konstruktor s jedním parametrem typu `int`. Kompilátor nedokáže přímo vyhodnotit výraz `a+1` (není definován operátor `+` pro operandy typu `Racionalni` a `int`) a hledá proto pro druhý operand vhodnou konverzi.

Konverze definovaná uživatelem se pro vyhodnocení výrazu použije pouze, je-li jednoznačná. Objekty vytvořené implicitním nebo explicitním voláním konstruktoru jsou dočasné a jsou zrušeny obvykle po provedení příkazu, který je vytvořil. Uživatelem definovaná konverze se uplatní pouze v jedné úrovni, tj. kompilátor nehledá posloupnost více uživatelských konverzí, které by umožnily vyhodnocení výrazu.

Ve druhém součtu `1+a` se implicitní konverze nemůže uplatnit (první argument operátoru `+` je typu `int`) a ve třídě `Racionalni` musí být pro vyhodnocení součtu `int` a `Racionalni` definován samostatný operátor. Protože první argument není typu `Racionalni`, nelze tento operátor definovat jako metodu, ale musíme použít globální funkci (obvykle je deklarována jako `friend`). Obdobně je jako globální `friend` funkce definován i operátor výstupu `<<`.

Pokud by operace součtu typů `Racionalni` a `int` byla pro využití dané třídy klíčová, mohli bychom příslušné operátory definovat explicitně a vyhnout se tak implicitním konverzím.

```
class Racionalni {
```

```
// ...
  Racionalni operator+(int b)
  {
    Racionalni t = *this; t.p += b*t.q; return t;
  }
  friend Racionalni operator+(int a, Racionalni b)
  {
    Racionalni t = b; t.p += a*t.q; return t;
 }
// ...
};
```

Podobně bychom pro třídu `Racionalni` definovali unární operátory inkrementace (a analogicky i dekrementace).

```
class Racionalni {
// ...
  Racionalni& operator++ ()
  {
    p += q; return *this;
  }
  Racionalni  operator++ (int)
  {
    Racionalni t = *this; p += q; return t;
  }
};
```

První metoda definuje prefixový inkrement, druhá metoda postfixový inkrement (liší se prázdným, tj. nepoužitým parametrem `int`). Prefixový operátor `++a` by měl vracet referenci na objekt `a`. Postfixový operátor `a++` by měl vracet `void` nebo kopii původního stavu objektu `a`.

Obecně by se operátory definované pro uživatelské typy měly chovat obdobně jako operátory standardních typů. Význam přetížených operátorů by měl být intuitivně zřejmý.

Definujeme-li ve třídě operátory `+` a `=`, neznamená to, že je tím zároveň definován operátor `+=` (totéž platí i pro ostatní operátory *op*`=`). Operátor `+=` bychom ale měli v takovém případě také definovat a to s významem stejným jako `a = a + inkrement`.

V dalším textu se zmíníme zvlášť o konverzních operátorech, operátorech indexování `[]`, volání funkce `()` a nepřímého přístupu `->`. Samostatný odstavec 5.11 věnujeme operátoru přiřazení a kopírovacímu konstruktoru.

5.10.1 Konverze

Jak jsme se již v této kapitole zmínili, definuje konstruktor třídy `T` s jedním parametrem typu `P` konverzi z typu `P` na typ `T`. Specifikací `explicit` můžeme zabránit implicitním konverzím ve výrazech.

Ve třídě `T` můžeme kromě toho definovat metodu `operator X()` zajišťující konverzi z typu `T` na typ `X`. Metoda `operator X()` nemá typ.

```
class T {
    // ...
```

```
public:
    operator int();
};
void f(T a)
{
    int i = int(a);
    i = (int)a;
    i = a;
}
```

Ve všech třech případech je přiřazovaná hodnota typu `T` konvertována na typ `int` metodou `T::operator int()`.

První dvě formy přitom lze použít i pro explicitní konverze standardních typů. Ve druhém případě hovoříme o *přetypování (cast)*. Následující ukázka demonstruje konverzi ukazatele na typ `double` na ukazatel na typ `void`, který je parametrem některých standardních funkcí (viz například metody pro čtení a zápis proudů `read` a `write`)

```
void* f(double d) {
   // ...
   double* dptr = &d;
   return (void*)dptr;   // konverze na typ ukazatel na void
}
```

Konverzi z naší třídy `Racionalni` na typ `int` definujeme jako metodu

```
Racionalni::operator int() const
{
  return p/q;
}
```

Konverzní metody jsou děděny a mohou být virtuální (viz další kapitola). Implicitně se uživatelské konverze uplatní pouze, jsou-li jednoznačné a pouze v jedné úrovni. Uživatelské konverzní funkce na rozdíl od konstruktorů s jedním parametrem nemohou být deklarovány se specifikací `explicit`.

5.10.2 Operátor indexování []

V odstavci 5.4 jsme na straně 98 uvedli třídu `dPole` realizující *dynamické pole.* Pro přístup k jednotlivým prvkům dynamického pole jsme definovali metodu `prvek`, která vracela referenci na `double` a mohla proto být uvedena i vlevo od přiřazovacího příkazu.

Pokud nahradíme zmíněnou metodu `prvek` operátorem indexování, nijak se nezmění mechanismus přístupu k jednotlivým prvkům, ale text programu pracující s jednotlivými prvky objektů typu `dPole` bude mnohem čitelnější a jasnější.

```
class dPole {    // 2. verze
public:
  dPole(int dim) { N = dim; p = new double[N]; }
 ~dPole() { delete[] p; }
  double& operator[](int i) { return p[i]; }     // operator indexovani
  int dim() const { return N; }
```

```
private:
  int N;
  double* p;
};

void f()
{
  dPole w(100);
  for (int i=0; i<w.dim(); i++)
    w[i] = i;                      // w.operator[](i)
  // ....
}
```

Touto cestou můžeme definovat i operátor indexování pro vícerozměrná pole. Uložíme-li například matici po řádcích a vrací-li `operator[](int)` vždy ukazatel na začátek i-tého řádku, pak následující standardní operátor `[]` vrátí referenci na prvek matice m_{ij}.

```
class Matice {
  double** m;
public:
  // ....
  double* operator[] (int i) { return m[i]; }
};

void f(Matice & A)
{
   int i, j;
   // ....
   A[i][j] = 1;   // ( A.operator[](i) )[j]
}
```

5.10.3 Operátor volání funkce `()`

Operátor volání funkce nám umožňuje vytvářet objekty, které se tváří jako funkce. Říká se jim proto *funkční objekty (function object)* nebo také *funktory (functor)*. Obvykle jde o instance tříd, které mají jen jednu metodu nebo ve kterých jedna z metod má dominantní postavení. S funkčními objekty se často setkáme při práci se standardními kontejnery, kde vystupují jako parametry šablon. Mimo jiné jsou zajímavé i tím, že mohou být definovány lokálně v těle funkcí.

Třídu `Fibonacci` ze strany 92 můžeme přepsat tak, že metodu `dalsi()` nahradíme operátorem volání funkce.

```
#include <iostream>

using namespace std;

class Fibonacci {
public:
  Fibonacci() { Fn = 0; Fn1 = 1; }
  unsigned long operator() () {
    unsigned long c = Fn; Fn = Fn1; Fn1 += c; return c;
```

```
  }
private:
  unsigned long Fn, Fn1;
};

int main()
{
  Fibonacci F;
  for (int i=0; i<15; i++)
    cout << F() << ' ';      // F.operator()()
  cout << " ..." << endl;
}

0 1 1 2 3 5 8 13 21 34 55 89 144 233 377  ...
```

První dvojice závorek `()` v definici operátoru volání funkce představuje jméno operátoru, který je součástí jména operátorové funkce. Druhá dvojice závorek je v našem případě prázdný seznam parametrů. Seznam parametrů nemusí být obecně prázdný a může obsahovat deklarace parametrů stejně jako u běžných funkcí.

5.10.4 Operátor nepřímého přístupu `->`

Operátor `->` může být přetížen jako unární operátor. Musí být definován jako nestatická metoda a musí vracet

- ukazatel na třídu,
- objekt nebo referenci na objekt třídy pro kterou byl definován `operator->`.

Výraz

```
x->m
```

je interpretován jako

```
(x.operator->())->m
```

pro objektový typ `x`. V anglické terminologii se v souvislosti s přetíženými operátory nepřímého přístupu hovoří o *smart pointers*. Umožňují totiž definovat třídy, které se chovají podobně jako standardní ukazatele a mimo to plní ještě další úkoly.

V následující ukázce ve třídě `SeznamBodu` definujeme operátor `->`, který nám zpřístupní informace o běžném bodu (bodu, který byl lokalizován metodou `hledej`). Operátor `->` třídy `SeznamBodu` v tomto případě vrací standardní ukazatel na `Bod`.

```
#include <iostream>
#include <string>

using namespace std;

struct Bod {
```

```
  const std::string cb;
  double y, x;
  Bod* dalsi;
  Bod(const std::string& c) : cb(c) {}
};

class SeznamBodu {
  Bod* szb;
  mutable Bod* bezny;
public:
  SeznamBodu() : szb(0) {}
  bool hledej(const std::string&) const;
  void uloz(const std::string&, double, double);
  Bod* operator->();
};

bool SeznamBodu::hledej(const std::string& s) const
{
  bezny = szb;
  while ( bezny )
    if (s == bezny->cb)
      return true;
    else
      bezny = bezny->dalsi;

  return false;
}

void SeznamBodu::uloz(const std::string& c, double y, double x)
{
  if (!hledej(c)) {
    bezny = new Bod(c);
    bezny->dalsi = szb;
    szb = bezny;
  }
  bezny->y = y;
  bezny->x = x;
}

Bod* SeznamBodu::operator->()
{
  if (bezny)
    return bezny;
  else
    throw "Nulovy ukazatel";
}

int main()
{
  SeznamBodu S;
  S.uloz("abc", 340.00, 178.40);
  S.uloz("xxx",   0.00,   0.00);
  S.uloz("g"  , 546.85, 569.33);
```

```
  S.uloz("xxx", 100.00, 200.00);
  S.uloz("zh3", 463.73, 536.77);

  if ( S.hledej("xxx") )
    cout << S->cb << " " << S->y << " " << S->x << endl;
}

xxx 100 200
```

Pokud bychom chtěli zpřístupnit souřadnice bodu pouze pasivně (tj. zviditelnit je, ale zabránit jejich přepsání), definovali bychom návratovou hodnotu operátoru `->` jako `const Bod*`.

5.11 Kopírovací konstruktor a operátor přiřazení

Kopírovací konstruktor (copy constructor) třídy `T` je konstruktor, který může být volán s jediným argumentem typu reference na `T` pro vytvoření nového objektu jako kopie jiného dříve vytvořeného objektu (argumentu).

Operátor přiřazení (assignment operator) ve třídě `T` zajišťuje přiřazení mezi dvěma objekty třídy `T`.

```
class T {
public:
  T()                       { /* implicitni konstruktor */ }
  T(const T& t)             { /* kopirovaci konstruktor */ }
  T& operator=(const T& t) { /* operator prirazeni */ return *this; }
 ~T()                       { /* destruktor */ }
// ....
};

void f()
{
  T a;            // implicitni konstruktor
  T b = a;        // kopirovaci konstruktor
  T c(a);         // kopirovaci konstruktor
  a = b;          // operator prirazeni
}
```

Vytvoření objektu jako kopie jiného objektu může být zapsáno dvěma způsoby, jak demonstruje ukázka. Symbol = zde neznamená přiřazení ale kopírování. Vytvoření kopie objektu a přiřazení objektu jsou dvě různé operace.

Destruktor, kopírovací konstruktor a operátor přiřazení mají zvlášť významné postavení mezi ostatními metodami třídy. Pokud je explicitně nedefinujeme, vytvoří kompilátor tyto metody implicitně. Obecně platí, že pokud ve třídě musíme definovat destruktor, musíme definovat i kopírovací konstruktor a operátor přiřazení.

Kompilátorem implicitně vytvořený destruktor, kopírovací konstruktor a operátor přiřazení zpracovávají postupně jednotlivé atributy třídy. Pokud jsou implicitně vytvořeny kompilátorem, všechny uvedené tři metody operují po jednotlivých atributech dané třídy. Implicitní operátor přiřazení přiřadí jednotlivé atributy, implicitně kopírovací konstruktor vytvoří kopie jednotlivých atributů a implicitní destruktor postupně volá destruktory jednotlivých objektových atributů.

```
#include <iostream>

using namespace std;

class C {
public:
  C() {}
  C(const C&)            { cout << "C copy ctor   "; }
  C& operator=(const C&) { cout << "C operator=   "; return *this; }
  ~C()                   { cout << "C dtor "; }
};

class T {
  C a;
  C b;
};

int main()
{
  {
    T x;
    cout <<   "T implicitni copy ctor : ";
    T y = x;
    cout << "\nT implicitni operator= : ";
    y = x;
    cout << "\nT implicitni dtor (2x) : ";
  }
  cout << endl;
}


T implicitni copy ctor : C copy ctor   C copy ctor
T implicitni operator= : C operator=   C operator=
T implicitni dtor (2x) : C dtor C dtor C dtor C dtor
```

Metody implicitně vytvořené kompilátorem jsou veřejné.

Ve třídě `T` z předchozí ukázky a v řadě dalších případů nemusíme destruktor, kopírovací konstruktor ani operátor přiřazení explicitně definovat. Tyto tři klíčové metody ale musíme definovat vždy, pokud třída obsahuje standardní ukazatele na externí data, která vlastní a spravuje daná třída.

```
#include <iostream>

using namespace std;

class X {
  double* d;
public:
  X(int n) : d(new double[n])
  {
    cout << "ctor X : " << (void*)d << endl;
  }
 ~X()
  {
```

```
    cout << "dtor X : " << (void*)d << endl; /* delete[] d; */
  }
};

int main()
{
  X x(1000);
  X y = x;    // implicitni kopirovaci konstruktor
              // destruktor volan 2x pro uvolneni teze oblasti pameti
}

ctor X : 0x8049f40
dtor X : 0x8049f40
dtor X : 0x8049f40
```

V programu jsme vytvořili objekt `x`. Pro vytvoření objektu `y` byl použit implicitní kopírovací konstruktor, který pouze překopíroval hodnotu ukazatele `x.d`. Pokud by destruktor v naší ukázce zavolal operátor `delete[]` a nevypsal pouze adresu alokované paměti, skončil by program chybou.

Stejný problém by vyvstal, pokud bychom pro objekt třídy `X` zavolali implicitní operátor přiřazení.

Třída `dPole` pro práci s dynamickým polem, jak jsme ji uvedli na straně 98 a 117, je proto zjevně chybná a musíme ji přepsat.

```
#include <iostream>

using namespace std;

class dPole {    // 3. verze
public:
  dPole(int dim) { N = dim; p = new double[N]; }
  dPole(const dPole&);               // kopirovaci konstruktor
  dPole& operator=(const dPole&);    // operator prirazeni
 ~dPole() { delete[] p; }
  double& operator[](int i) { return p[i]; }
  int dim() const { return N; }
private:
  int N;
  double* p;
};

dPole::dPole(const dPole& a)
{
  N = a.N;
  p = new double[N];
  for (int i=0; i<N; i++)
    p[i] = a.p[i];
}

dPole& dPole::operator=(const dPole& a)
{
  if (this != &a) {
    delete[] p;
```

```
    N = a.N;
    p = new double[N];
    for (int i=0; i<N; i++)
      p[i] = a.p[i];
  }
  return *this;
}

int main()
{
  dPole w(10);
  for (int i=0; i<w.dim(); i++)
    w[i] = i;

  dPole v(5);       // konstruktor
  dPole u = v;      // kopirovaci konstruktor
  u = w;            // operator prirazeni

  for (int i=0; i<u.dim(); i++)
    cout << u[i] << " ";
  cout << endl;
}

0 1 2 3 4 5 6 7 8 9
```

Úvodní test v operátoru přiřazení je nezbytný pro případ přiřazení objektu sama do sebe.

```
x = x;
```

Přímé přiřazení není příliš pravděpodobné ale může k němu dojít přes ukazatele. Operátor přiřazení musí být každopádně vždy na tuto situaci připraven. Alternativně bychom mohli operátor = napsat třeba takto.

```
dPole& dPole::operator=(const dPole& a)
{
  if (N != a.N) {
    delete[] p;
    N = a.N;
    p = new double[N];
  }
  for (int i=0; i<N; i++)
    p[i] = a.p[i];
  return *this;
}
```

Metody kopírovací konstruktor a operátor přiřazení mohou být deklarovány jako soukromé nebo chráněné. Jako soukromé jsou často deklarovány tehdy, pokud daná třída vyžaduje destruktor, ale operace kopírování a přiřazení nejsou pro její použití nezbytné. Metody stačí jako soukromé ve třídě deklarovat aniž bychom poskytli jejich definici, zabráníme tím jejich implicitnímu vytvoření kompilátorem.

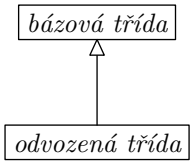
Veřejný operátor přiřazení by měl vracet referenci na danou třídu, aby jeho použití ve výrazech bylo obdobné přiřazení standardních typů. Soukromé nebo chráněné operátory přiřazení mohou případně vracet typ `void`.

Kapitola 6

Odvozené třídy

V této kapitole se seznámíme s pojmem *dědičnost (inheritance)*. Mechanismus dědičnosti představuje jeden ze základních nástrojů objektového programování a umožňuje nám vytvářet z existujících tříd *odvozené třídy (derived classes)*, které *dědí* vlastnosti svých *rodičovských tříd*. Nemusíme přitom mít přímo zdrojový text metod rodičovských tříd, ale pouze jejich deklarace v hlavičkovém souboru. Přeložené funkce mohou být uloženy například v knihovně a připojíme je až při sestavení našeho programu.

Odvození (dědičnost) se často znázorňuje graficky diagramem, ve kterém šipka směřuje od odvozené třídy k třídě bázové



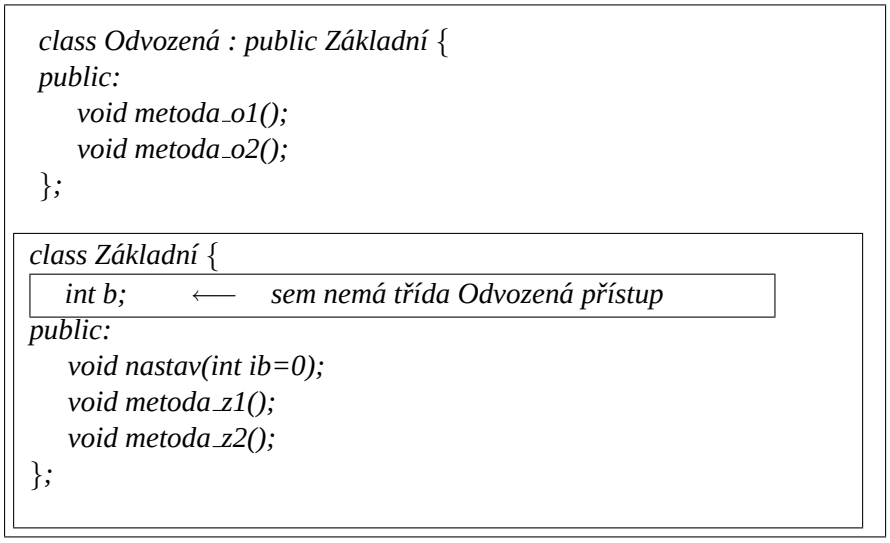
Namísto o odvozených třídách se také hovoří o *dceřiných třídách* nebo *potomcích*, rodičovským třídám se říká *bázové třídy (base classes)* a pod. Terminologie není zcela ustálená.

Analogie s pojmy z jiných oborů, které se často používají pro vysvětlení pojmu dědičnosti, mohou být právě tak instruktivní jako zavádějící. Dědičnost v C++ přitom není nic tajemného. Stručně řečeno jde o to, že odvozená třída získá vlastnosti bázové třídy (atributy a děděné metody) a k tomu ještě něco navíc — metody a atributy definované v odvozené třídě a další vlastnosti, o kterých si řekneme později. V následujícím příkladu je od rodičovské třídy `Zakladni` odvozena třída `Odvozena`.

```
class Zakladni {
  int b;
public:
  void nastav (int ib=0) { b = ib; }
  void metoda_z1() { cout << "metoda_z1()   b = " << b << endl; }
  void metoda_z2() { cout << "metoda_z2()   b = " << b << endl; }
};

class Odvozena : public Zakladni {
public:
  void metoda_o1() { cout << "metoda_o1() " << endl; }
```

```
  void metoda_o2() { cout << "metoda_o2() vola zdedene metody" << endl;
                     metoda_z1();
                     metoda_z2();
                   }
};
```

Schématicky si můžeme celou situaci znázornit takto



V našem příkladu zdědila třída `Odvozena` všechny členy třídy `Zakladni`. Třída `Zakladni` je podtřídou (podmnožinou) třídy `Odvozena`, která ale nemá přístup k jejím soukromým členům.

```
int main()
{
  Zakladni a;          // definice objektu bazove tridy
  a.nastav(1);         // volani metody bazove tridy
  a.metoda_z1();

  Odvozena x;          // definice objektu odvozene tridy
  x.nastav(2);         // volani metody bazove tridy
  x.metoda_z1();       //    pro instanci odvozene tridy
  x.metoda_z2();
  x.metoda_o1();
  x.metoda_o2();
}
```

Bázová třída může být opět odvozena z jiné třídy. Například tak, jako v naší další ukázce.

```
#include <iostream>

class X { public: void x() {std::cout << "fce X::x()\n";} };
class Y : public X { public: void y() {std::cout << "fce Y::y()\n";} };
class Z : public Y { public: void z() {std::cout << "fce Z::z()\n";} };

int main() {  Z q;  q.x(); q.y(); q.z();  }
```

Třída `Z` zde zdědila metody třídy `Y` a jejím prostřednictvím i metody třídy `X`.

Implicitní konverze typu ukazatel na odvozenou třídu

V C++ existuje implicitní konverze z typu ukazatel na odvozenou třídu na typ ukazatel na bázovou třídu (je-li přístupná a je-li jednoznačné, o kterou bázovou třídu se jedná). Totéž platí i pro typ reference. Odvozené třídy navíc mohou předefinovat *virtuální metody* bázové třídy. Hovoříme pak o *polymorfismu*, který je třetím základním stavebním kamenem objektového programování. O tom ale až později.

V následující ukázce je ukazatel na odvozenou třídu `Savec` při druhém volání funkce `f()` implicitně konvertován na ukazatel na bázovou třídu `Zivocich`.

```
class Zivocich {
public:
  void metoda_ziv1() { /* ... */ }
};

class Savec : public Zivocich {
  // ...
};

void f(Zivocich* pz)
{
  pz->metoda_ziv1();
  // ...
}

void g(Zivocich& rz)
{
  rz.metoda_ziv1();
  // ...
}

int main()
{
  Zivocich kapr;
  Savec    slon;

  f(&kapr);   // ukazatel na tridu Zivocich
  f(&slon);   // implicitni konverze

  g(kapr);    // reference na tridu Zivocich
  g(slon);    // implicitni konverze
}
```

Pravidlo pro implicitní konverzi ukazatelů na bázový typ si můžete snadno zapamatovat podle našeho příkladu. Každý *savec* je *živočich* (ukazatel `Savec*` lze použít všude tam, kde má být ukazatel `Zivocich*`) ale ne každý *živočich* je *savec* (nelze použít ukazatel `Zivocich*` tam, kde má být ukazatel `Savec*`). Zcela analogicky totéž platí pro odvozený typ reference, jak demonstruje funkce `g()` z předchozí ukázky.

Obvykle je možné i přiřadit instanci potomka do instance rodičovské třídy, například

```
kapr = slon;                    // přiřazení instance odvozené třídy do instance bázové třídy
```

obvykle jde ale o programátorskou chybu a je třeba na podobné omyly dávat bedlivý pozor, protože jazyk taková přiřazení připouští (obráceně ale ne). Do proměnné typu bázové třídy se překopírují pouze datové členy zděděné z bázové třídy a ostatní atributy odvozené třídy jsou ignorovány. V anglické terminologii se používá termín *slicing*.

6.1 Přístupová práva ke členům bázové třídy

Omezíme se nejprve na případ *veřejného dědění*, kdy bázová třída je specifikována jako `public`. Odvozená třída nemá přístup k privátním členům bázové třídy. Odvozená třída má přístup k veřejným členům bázové třídy a ke členům deklarovaným jako `protected`. Specifikace přístupu `protected` označuje v C++ *chráněné členy.* Zde je rekapitulace tří možných specifikací přístupu ke členům C++ třídy.

`public:`

veřejné členy může používat kterákoliv funkce nebo inicializátor.

`private:`

privátní členy mohou používat pouze členské funkce, *členské inicializátory (member initializer)* a přátelé třídy (funkce a třídy deklarované zde jako `friend`). Specifikace `private` je ve třídách implicitní.

`protected:`

chráněné členy mohou používat pouze členské funkce, členské inicializátory a přátelé třídy a členské funkce tříd odvozených z této třídy a jejich přátelé.

Uvedené tři základní možnosti pro případ veřejné bázové třídy demonstruje následující příklad:

```
class bazova_trida {
private:    void privatni_metoda() { /* ... */ }
protected:  void chranena_metoda() { /* ... */ }
public:     void verejna_metoda()  { /* ... */ }
};

class odvozena_trida : public bazova_trida {
public:
  void f() {
    // privatni_metoda();  /* nelze */
    chranena_metoda();     /*  OK   */
    verejna_metoda();      /*  OK   */
  }
};

class dalsi_odvozena : public odvozena_trida {
public:
  void p() {
    chranena_metoda();     /*  OK  */
  }
};

int main()
```

```
{
  odvozena_trida T;
  // T.privatni_metoda();  /* nelze */
  // T.chranena_metoda();  /* nelze */
  T.verejna_metoda();      /*  OK   */
  dalsi_odvozena U;
  U.p();                   /*  OK   */
}
```

Vyzkoušejte si, jak bude reagovat váš kompilátor, pokud v uvedeném příkladu odstraníte komentáře u nepřípustných případů.

6.1.1 Specifikace přístupu pro bázovou třídu

Specifikaci přístupových práv `public`, `private` a `protected` uvádíme také v deklaraci odvozené třídy před jménem bázové třídy. Pokud tuto specifikaci neuvedeme, platí implicitní hodnota `private`. Tímto způsobem můžeme ještě více omezit přístup ke členům bázové třídy ale ne naopak. Žádnou deklarací nemůžeme například zpřístupnit privátní členy bázové třídy.

Mějme třídu `D` odvozenou od bázové třídy `B`. Odvozená třída `D` může být od třídy `B` odvozena třemi různými způsoby (prozatím nehovoříme o deklaraci `virtual`)

`class D : public B { /* ... */ };`

Veřejné členy třídy `B` se stávají veřejnými členy třídy `D`, chráněné členy třídy `B` se stávají chráněnými členy třídy `D`. K privátním členům třídy `B` nemá třída `D` přístup. Třída `B` je veřejně přístupným předkem třídy `D`.

```
protected  ⟶  protected
public     ⟶  public
```

`class D : private B { /* ... */ };`

Veřejné a chráněné členy třídy `B` se stávají soukromými členy třídy `D`. K privátním členům třídy `B` nemá třída `D` přístup. Třída `B` je privátním předkem třídy `D`.

```
protected  ⟶  private
public     ⟶  private
```

`class D : protected B { /* ... */ };`

Veřejné a chráněné členy třídy `B` se stávají chráněnými členy třídy `D`. K privátním členům třídy `B` nemá třída `D` přístup. Třída `B` je chráněným předkem třídy `D`.

```
protected  ⟶  protected
public     ⟶  protected
```

Napoprvé to zní komplikovaně? Snad vám pro začátek trochu pomůže následující příklad, především ale sami experimentujte.

```
#include <iostream>

using namespace std;

class B {
private:   void a() { cout << "    B::a()\n"; }
protected: void b() { cout << "    B::b()\n"; }
public:    void c() { cout << "    B::c()\n"; }
};


class D1 : private B {
public:
  void f() { cout << "  D1::f()\n";
    // a();         // nelze !
    b();  c();      // OK
  }
};
class D2 : protected B {
public:
  void f() { cout << "  D2::f()\n";
    // a();       // nelze !
    b();  c();    // OK
  }
};
class D3 : public B {
public:
  void f() { cout << "  D3::f()\n";
    // a();       // nelze !
    b();  c();    // OK
  }
};

class E1 : D2 {
public:
  void g() { cout << "E1::g()\n";
    f();  b();  c();    // OK
  }
};

int main()
{
  D1 d1;   D2 d2;   D3 d3;

  // d1.a();  d1.b();  d1.c();   // nelze !
  d1.f();                       // OK

  // d2.a();  d2.b();  d2.c();   // nelze !
  d2.f();                       // OK

  // d3.a();   d3.b();          // nelze !
  d3.f();   d3.c();             // OK
```

```
  E1 e1;
  e1.g();                         // OK
}
```

6.2 Konstruktory a destruktory

Konstruktory ani destruktor se v C++ nedědí (nedědí se ani operátor přiřazení, viz tabulka na straně 143). Obvykle proto musíme tyto metody v odvozené třídě explicitně deklarovat a zajistit předání argumentů konstruktorům bázových tříd. Argumenty konstruktorům bázových tříd předáváme obdobně jako argumenty konstruktorů datových členů.

```
#include <iostream>

using namespace std;

class E {
  int e;
public:
  E(int a) { e = a; }
  int ee() { return e; }
};

class F : public E {
  int f;
public:
  F(int n=0) : E(2*n) { f = 1; }
};

int main()
{
  F p(3);
  cout << p.ee() << endl;     // vytiskne 6
}
```

Třída `E` v naší ukázce nemá *implicitní konstruktor (default constructor)*, tj. konstruktor, který lze volat bez argumentů. Bez explicitního předání argumentů konstruktoru třídy `E` bychom nemohli vytvářet objekty typu `F`. Konstruktor třídy `F` proto musí zajistit předání argumentu konstruktoru třídy `E`.

Pokud má bázová třída implicitní konstruktor a pokud nepotřebujeme předat argumenty jinému jejímu konstruktoru, nemusíme v konstruktorech odvozené třídy konstruktory bázové třídy volat.

Při vytváření objektů odvozených tříd jsou nejprve volány konstruktory pro vytvoření členů bázové třídy a pak konstruktory vytvářející členy odvozené třídy. Destruktory jsou při zrušení objektu volány v obráceném pořadí.

```
#include <iostream>

using namespace std;

class C1 {
public:
  C1() { cout << "konstruktor C1\n"; }
 ~C1() { cout << "destruktor  C1\n"; }
```

```
};

class C2 {
public:
  C2() { cout << "konstruktor C2\n"; }
 ~C2() { cout << "destruktor  C2\n"; }
};

class CB {
 C1 c1;
public:
  CB() { cout << "konstruktor CB\n"; }
 ~CB() { cout << "destruktor  CB\n"; }
};

class CD : public CB {
 C2 c2;
public:
  CD() { cout << "konstruktor CD\n"; }
 ~CD() { cout << "destruktor  CD\n"; }
};

int main() { CD x; }
```

Zde je výstup předchozího programu.

```
konstruktor C1
konstruktor CB
konstruktor C2
konstruktor CD
destruktor  CD
destruktor  C2
destruktor  CB
destruktor  C1
```

6.3 Vícenásobná dědičnost

V C++ může být třída přímo odvozena od více tříd. Za dvojtečkou v deklaraci odvozené třídy pak píšeme seznam bázových tříd. Pro specifikaci přístupu platí stejná pravidla jako pro případ s jedinou bázovou třídou.

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };

class ABC : public A, private B, protected C { /* ... */ };
```

Vícenásobná dědičnost obvykle nepřináší žádné problémy. Pokud vede dědičnost k nejednoznačnostem, tj. pokud jsou v odvozené funkci členy se stejným jménem, hovoříme o *konfliktu jmen (ambiguity)*.

```
class A { public: void f()      { /* ... */ } };
class B { public: int  f(int n) { return n; } };
```

```
class AB : public A, public B { /* ... */  };

class AB2 : public A, public B {
// ...
public:
  void f() { A::f(); }
  int  f(int n) { return B::f(n); }
};

int main()
{
  AB p;
  // p.f();     /* !!! nejednoznacne !!! */
  p.A::f();     // OK
  p.B::f(0);    // OK

  AB2 x;        // bez problemu
  x.f();
  x.f(0);
}
```

Pokud v programu pro objekty třídy `AB` nikde metodu `f()` nevoláme, nic se neděje. V opačném případě můžeme nápravy dosáhnout například pomocí operátoru rozlišení oboru. Všimněte si, že v daném kontextu kompilátor nerozlišuje metody podle různých typů argumentů (kompilátor nejprve kontroluje konflikt jmen a seznam argumentů až potom).

Mnohem elegantnější je ale řešení naznačené ve třídě `AB2`. Metody `f()` zastíní stejná jména použitá v bázových třídách a kompilátor může jednoznačně rozhodnout, kterou funkci má volat podle argumentů použitých při volání.

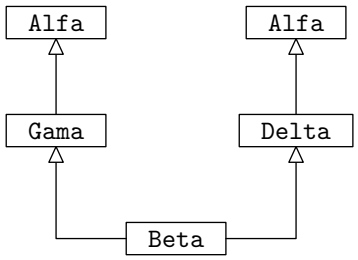
Jazyk C++ nepřipouští přímé vícenásobné odvození od jedné bázové třídy. Nelze tedy například psát

```
class B        { /* ... */ };
class A : B, B { /* ... */ };  // !!! nelze !!!
```

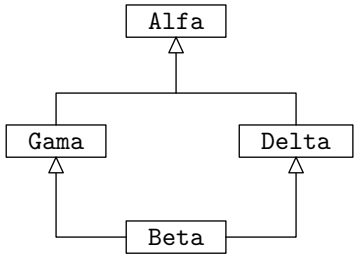
Jedna třída ale může být vícekrát předkem odvozené třídy nepřímo.

```
#include <iostream>

using namespace std;

class Alfa {
public:
  Alfa() { cout << "konstruktor Alfa()" << endl; }
  virtual ~Alfa() { cout << "destruktor  Alfa()" << endl; }
};
class Gama  : Alfa { /* ... */ };
class Delta : Alfa { /* ... */ };
class Beta  : Gama, Delta { /* ... */ };
int main() { Beta B; }
```

Členy třídy `Alfa` budou ve třídě `Beta` uvedeny dvakrát, jak ukazuje následující diagram. Můžete si ověřit, že konstruktor a destruktor `Alfa` je při vytvoření, resp. zrušení, objektu typu `Beta` opravdu volán dvakrát.



Zda jsou podobné případy přípustné či nikoliv, závisí na daném problému a ne na programovacím jazyku. Každopádně jsou situace, kdy musíme zajistit, že předek `Alfa` je ve třídě `Beta` obsažen jednou, jak ukazuje další diagram.



V C++ toho dosáhneme tak, že třídu `Alfa` coby přímého předka odvozených tříd budeme deklarovat jako *virtuální (virtual)*. Přeložíte-li upravenou verzi předchozí ukázky, můžete se opět přesvědčit, že konstruktor a destruktor třídy `Alfa` budou volány pouze jednou.

```
#include <iostream>

class Alfa {
public:
  Alfa() { std::cout << "konstruktor Alfa() \n"; }
  virtual ~Alfa() { std::cout << "destruktor  Alfa() \n"; }
};
class Gama  : virtual public Alfa { /* ... */ };
class Delta : virtual public Alfa { /* ... */ };
class Beta  : Gama, Delta  { /* ... */ };
int main() { Beta B; }
```

6.4 Virtuální metody a polymorfismus

Jazyk C++ umožňuje deklarovat metody bázové třídy jako *virtuální (virtual function)*. Odvozená třída pak může takovou metodu předefinovat. Virtuální metody nám tedy umožňují definovat hierarchii odvozených tříd ve které pak se všemi jejich instancemi pracujeme jednotným způsobem.

Důležité je, že při *virtuálním dědění* obvykle pracujeme s instancemi prostřednictvím ukazatelů na bázovou třídu, ale díky mechanismu, kterém říkáme *polymorfismus*, jsou vždy volány správné virtuální funkce. Totéž platí i pro reference na bázovou třídu.

Mějme například bázovou třídu `B`, která definuje virtuální funkci `p()` a odvozenou třídu `D`, která tuto virtuální funkci nově definuje:

```
class B {
public:
  virtual ~B() {}     // virtualni destruktor
  virtual void p()
  {
    std::cout << "volana virtualni funkce B::p() \n";
  }
};

class D : public B {
public:
  void p()    // zde jiz nemusime klicove slovo virtual uvadet
  {
    std::cout << "volana virtualni funkce D::p() *********\n";
  }
};
```

Pro funkci `f()` s parametrem typu *reference na bázovou třídu B*, která volá pro zadaný parametr `x` virtuální funkci `p()`

```
void f(B& x)  {  x.p();  }
```

vypíše demonstrační program

```
int main()
{
  B x;
  D y;
  f(x);
  f(y);
}
```

následující výstup

```
volana virtualni funkce B::p()
volana virtualni funkce D::p() *********
```

Funkce `f()` z našeho příkladu nezná přesný typ svého zadaného argumentu, ale díky polymorfismu volá vždy správnou virtuální funkci.

Virtuální funkce podporují *dynamickou vazbu* a objektové programování. Třída, která deklaruje nebo dědí virtuální funkci se nazývá polymorfní třída.

Specifikace `virtual` je v odvozené třídě v definici virtuální funkce redundantní.

Jméno, typ a seznam parametrů virtuální funkce se musí v odvozené a bázové třídě shodovat. Pokud je ale návratovou hodnotou virtuální funkce ukazatel nebo reference na bázovou třídu, může být v odvozené třídě změněn na její typ.

Metoda třídy může být v C++ definovaná jako *čistě virtuální (pure virtual function)*. V deklaraci čistě virtuální metody používáme notaci `= 0;` (místo těla funkce). Například

```
class A {
public:
  virtual void f() = 0;
  // ...
};
```

Touto deklarací říkáme, že ve třídě `A` není a nemá být metoda `f()` definována. Důsledkem je, že nemůžeme deklarovat objekty typu `A`. Třída `A` může být použita pouze jako bázová třída pro definici jiných odvozených tříd. Třída `A` definuje jednotné rozhraní pro používání všech svých odvozených potomků.

Každá třída obsahující alespoň jednu čistě virtuální metodu je *abstraktní třída (abstract class)*. Není-li v odvozené třídě čistě virtuální metoda definována, zůstává v odvozené třídě čistě virtuální a odvozená třída je také abstraktní.

Nelze vytvářet instance abstraktních tříd. Můžeme ale používat ukazatele a reference na abstraktní třídu. Pokud do proměnné typu *ukazatel na abstraktní třídu* uložíme ukazatel na instanci libovolného odvozeného typu a pokud přes tento ukazatel na abstraktní třídu voláme virtuální metodu, zajistí výběr správné členské funkce mechanismus polymorfismu. Totéž platí analogicky i pro reference na abstraktní třídu.

```
class A {                   // třída A je abstraktní, protože obsahuje čistě virtuální funkci
public:
  virtual void f() = 0;     // deklarace čistě virtuální funkce
  // ...
};

class B : public A {        // třída B zůstává abstraktní
  // ...                    //       protože nedefinuje virtuální funkci f()
};

class C : public B {        // polymorfní třída C, definuje virtuální funkci f()
public:
   void f() { /* ... */ } // definice virtuálni funkce
};

void q()
{
   A  a;                    // !!! NELZE vytvořit instanci abstraktního typu !!!
   B  b;                    // !!! NELZE vytvořit instanci abstraktního typu !!!
   C  c;                    // instance polymorfní třídy C

   A* pa = &c;              // do ukazatele na abstraktní třídu uložím ukazatel na c
   pa->f();                 // volám virtuální funkci c->f()

   A& ra = c;               // referenci na abstraktní třídu inicializuji instancí c
   ra.f();                  // volám virtuální funkci c.f()
}
```

Konstruktory nemohou být virtuální. Virtuální ale může být destruktor. Předpokládáme-li, že daná třída bude použita jako bázová, deklarujeme destruktor vždy jako virtuální. Právě tak deklarujeme destruktor jako virtuální, pokud daná třída deklaruje nějakou virtuální funkci.

Příklad

Zapouzdření, dědičnost a polymorfismus tvoří základ objektového programování. Polymorfismus nám umožňuje psát funkce, které budou správně operovat i na typech, které nemusí být v dané chvíli vůbec definovány. Vezměme si jako příklad abstraktní třídu `Mereni` (pro jednoduchá geometrická měření jako jsou délky a úhly), pro kterou budeme chtít napsat funkci, která vypíše *seznam měření*

```
void tisk_seznamu(std::ostream&, const std::list<Mereni*>&);
```

Ve třídě `Mereni` můžeme definovat čistě virtuální funkce pro čtení a zápis objektů typu *měření*

```
class Mereni {
protected:
  double hodnota;
public:
  virtual ~Mereni() {}

  virtual void cti (std::istream&) = 0;
  virtual void tisk(std::ostream&) = 0;
  // ....
};
```

Nyní můžeme napsat funkci pro tisk seznamu měření, který je realizován jako standardní kontejner typu `list`

```
void tisk_seznamu(std::ostream& out, const std::list<Mereni*>& seznam)
{
  for (std::list<Mereni*>::const_iterator
         i=seznam.begin(), e=seznam.end(); i!=e; ++i)
    {
      (*i)->tisk(out);
    }
}
```

Funkce `tisk_seznamu()` má pouze informaci o bázové třídě `Mereni` a spoléhá na polymorfismus při volání virtuální funkce `tisk()`. Pro jednoduchost se omezíme pouze na dva typy měření a to na *délky* a *úhly*

```
class Delka : public Mereni {
  std::string stanovisko, cil;
public:

  ~Delka()
  {
    std::cout << "destruktor: "; tisk(std::cout);
  }

  void cti (std::istream& inp)
  {
    inp >> stanovisko >> cil >> hodnota;
  }
  void tisk(std::ostream& out)
  {
    out << "delka " << stanovisko
        << " " << cil << " " << hodnota << "\n";
  }
};

class Uhel : public Mereni {
  std::string stanovisko, vlevo, vpravo;
```

```
public:

  ~Uhel()
  {
    std::cout << "destruktor: "; tisk(std::cout);
  }

  void cti (std::istream& inp)
  {
    inp >> stanovisko >> vlevo >> vpravo  >> hodnota;
  }
  void tisk(std::ostream& out)
  {
    out << "uhel  " << stanovisko
        << " " << vlevo << " " << vpravo << " " << hodnota << "\n";
  }
};
```

Doplníme ještě demonstrační program, který načte a vypíše seznam měření. Výstup programu zároveň dokládá, že pro polymorfní třídy jsou náležitě volány i příslušné virtuální destruktory:

```
int main()
{
  std::list<Mereni*> seznam;
  std::string        t;
  Mereni*            m;

  while (std::cin >> t)
    {
      if      (t == "delka") m = new Delka;
      else if (t == "uhel" ) m = new Uhel;
      else ;  // ...

      m->cti(std::cin);     // volani virtualni funkce
      seznam.push_back(m);
    }

  tisk_seznamu(std::cout, seznam);

  std::cout << "\n-----------------------------------------\n";
  for (std::list<Mereni*>::const_iterator
         i=seznam.begin(), e=seznam.end(); i!=e; ++i)
    {
      delete *i;
    }
}
```

```
delka A B 872.152
delka A C 1541.81
uhel  A W Q 33.9815
uhel  A W Z 24.6645
delka C D 796.176
```

```
-----------------------------------------
destruktor: delka A B 872.152
destruktor: delka A C 1541.81
destruktor: uhel  A W Q 33.9815
destruktor: uhel  A W Z 24.6645
destruktor: delka C D 796.176
```

6.4.1 Příklad abstraktní třídy

Jako příklad nám poslouží třída `BaseOLS`, která definuje rozhraní pro metody úlohy, označované v geodézii jako *vyrovnání měření zprostředkujících* (v naší ukázce je uvedena zkrácená verze, která neobsahuje všechny nezbytné funkce). Jde o řešení soustavy lineárních rovnic

$$\mathsf{A}\mathsf{x} - \mathsf{l} = \mathsf{v}, \tag{6.1}$$

kde A je první matice plánu (koeficienty rovnic oprav), l vektor redukovaných měření a v vektor oprav měřených veličin (residuí). Při vyrovnání nekorelovaných různorodých měření musíme uvažovat i diagonální matici vah P. Hledáme řešení, které vyhovuje základní podmínce metody nejmenších čtverců

$$\mathsf{v}^{\mathsf{T}}\mathsf{P}\mathsf{v} = \min. \tag{6.2}$$

Atributy třídy `BaseOLS` jsou konstantní ukazatele na matici A a vektory absolutních členů l a vektor vah P (`const Mat* pA; const Vec* pb; const Vec* pW;`). Uvedené objekty se při řešení nemodifikují. Třída `BaseOLS` má dále datové členy obsahující vektor neznámých, vektor residuí, vektor odmocnin z vah měřených veličin a stav řešení (`Vec x; Vec r; Vec sqrt_w; bool is_solved;`).

Třída `BaseOLS` má implicitní konstruktor `BaseOLS()` a dále konstruktory

```
BaseOLS(const Mat& A, const Vec& b);
BaseOLS(const Mat& A, const Vec& b, const Vec& w);
```

určené pro případ vyrovnání stejnorodých, resp. různorodých měření (vektor vah měření je třetím argumentem). Přechod objektu k jiné soustavě zajišťují metody `reset()` se stejnými argumenty

```
void reset(const Mat& A, const Vec& b);
void reset(const Mat& A, const Vec& b, const Vec& w);
```

Destruktor je definován jako `virtual ~BaseOLS() {}`.

Vektor neznámých získáme prostřednictvím přetížené metody `solve()`, obdobně vektor oprav voláním funkce `residuals()`.

Kovarianční matici vyrovnaných neznámých vrací virtuální metoda `cov_xx(Mat&)`. Tuto metodu můžeme, ale nemusíme v odvozené třídě definovat. Zbývající metody jsou již všechny čistě virtuální. Kromě metod pro výpočet kovariancí vyrovnaných veličin je to především metoda `solve_me()`, která řeší soustavu (6.1).

```
#ifndef BaseOLS_h_
#define BaseOLS_h_

#include "matvec.h"
```

```
class BaseOLS {

public:
  BaseOLS() {}
  BaseOLS(const Mat& A, const Vec& b)
    : pA(&A), pb(&b), pw(0),  is_solved(false) {}
  BaseOLS(const Mat& A, const Vec& b, const Vec& w)
    : pA(&A), pb(&b), pw(&w), is_solved(false) {}
  virtual ~BaseOLS() {}

  void reset(const Mat& A, const Vec& b) {
    pA = &A;
    pb = &b;
    pw = 0;
    is_solved = false;
  }
  void reset(const Mat& A, const Vec& b, const Vec& w) {
    pA = &A;
    pb = &b;
    pw = &w;
    is_solved = false;
  }

  Vec& solve(Vec& x)       { solve_me(); return x = BaseOLS::x; }
  const Vec& solve()       { solve_me(); return x; }
  Vec& residuals(Vec& res) { solve_me(); return res = r; }
  const Vec& residuals()   { solve_me(); return r; }

  Float trwr();   // trans(r)*w*r

  virtual void cov_xx(Mat&);
  virtual Float cov_xx(int, int) = 0;   // Ax = b;  cov (xi, xj)
  virtual Float cov_bb(int, int) = 0;   //          cov (bi, bj)
  virtual Float cov_bx(int, int) = 0;   //          cov (bi, xj)


protected:

  virtual void solve_me() = 0;

  const Mat* pA;
  const Vec* pb;
  const Vec* pw;
  Vec x;
  Vec r;
  Vec sqrt_w;
  bool is_solved;

};

#endif
```

Soustava (6.1) se běžně řeší přechodem na *normální rovnice*

$$\mathsf{N}x = \mathsf{n}, \qquad \text{kde} \quad \mathsf{N} = \mathsf{A}^T\mathsf{P}\mathsf{A}, \quad \mathsf{n} = \mathsf{A}^T\mathsf{P}l. \tag{6.3}$$

a následným Choleskyho rozkladem matice soustavy normálních rovnic.

Předpokládejme, že máme třídu `CholD`, která realizuje tento rozklad. Můžeme pak například odvodit třídu

```
class OLS_Chol : public BaseOLS, private CholD { /* ... */ };
```

Ve třídě `OLS_Chol` musíme předefinovat všechny virtuální metody s využitím metod třídy pro výpočet Choleskyho rozkladu. Výpočet vektoru neznámých pro případ stejnorodých měření pak můžeme zapsat třeba takto

```
int main()
{
   Mat A;   Vec b;   cin >> A >> b;
   OLS_Chol rov_opr(A, b);
   cout << rov_opr.solve();
}
```

Pro většinu praktických úloh naznačený postup zcela vyhovuje. Nicméně někdy řešíme špatně podmíněné soustavy, jejichž numerický výpočet přes normální rovnice selže. Pro jejich řešení musíme použít jinou numerickou metodu. Vhodným nástrojem je numericky stabilní algoritmus *singulárního rozkladu* matice

$$\mathsf{A} = \mathsf{U}\mathsf{W}\mathsf{V}^T, \tag{6.4}$$

kde matice U a V jsou ortogonální a W je diagonální matice.

Odvodíme tedy jinou třídu pro řešení soustavy (6.1). Bázovou třídou bude `SVD`, realizující rozklad (6.4). Čistě virtuální metody třídy `BaseOLS` jsou předefinovány v následující třídě `OLS` s využitím metod třídy `SVD`

```
#ifndef OLSsvd_h_
#define OLSsvd_h_

#include "baseols.h"

class OLS : public BaseOLS, private SVD {

public:
  OLS() {}
  OLS(const Mat& A, const Vec& b) : BaseOLS(A, b) {}
  OLS(const Mat& A, const Vec& b, const Vec& w): BaseOLS(A, b, w) {}

  void reset(const Mat& A, const Vec& b) { BaseOLS::reset(A, b); }
  void reset(const Mat& A, const Vec& b, const Vec& w)
                                        { BaseOLS::reset(A, b, w); }

  void  cov_xx(Mat& C) { BaseOLS::cov_xx(C); }
  Float cov_xx(int i, int j) { return is_solved ? SVD::cov_xx(i, j) :
                                    (solve_me(), SVD::cov_xx(i, j)); }
  Float cov_bb(int i, int j)
```

```
    { solve_me(); return SVD::cov_bb(i, j) / (sqrt_w[i] * sqrt_w[j]); }
  Float cov_bx(int i, int j)
    { solve_me(); return SVD::cov_bx(i, j) / sqrt_w[i]; }

protected:
  void solve_me();

};

#endif
```

Bázová třída `SVD` byla deklarovaná jako `private`, aby uživatelé třídy `OLS` měli přístup k jejím metodám pouze prostřednictvím metod abstraktní třídy `BaseOLS`. Protože třída `SVD` má metodu `reset()`, obsahuje třída `OLS` vlastní metody `reset()`, které zastíní metody rodičovské třídy stejného jména a zamezí tak konfliktu jmen.

Protože jsme třídu `OLS` odvodili od stejné abstraktní třídy jako třídu `OLS_Chol`, budou mít stejné rozhraní. Pokud budeme potřebovat v našem programu nahradit v úloze vyrovnání měření zprostředkujících numerickou metodu řešící soustavu (6.1) jinou metodou (v našem případě jde o přechod od normálních rovnic a Choleskyho rozkladu k algoritmu singulárního rozkladu), stačí pouze zaměnit jméno třídy `OLS_Chol` za `OLS`.

6.5 Přehled metod, přátel a speciálních metod

Odvozené třídy nedědí metody, které vytvářejí objekty bázové třídy (konstruktory a operátor přiřazení), destruktory a nedědí ani přístupová práva, která má bázová třída případně přidělena deklarací `friend`. V definici odvozené třídy musíme proto konstruktory a destruktory explicitně popsat (musíme například zadat argumenty konstruktorů bázových tříd). Přátelé se v C++ také nedědí. Tabulka 6.1 shrnuje uvedené vlastnosti konstruktorů, destruktorů, konverzních metod, operátorových funkcí, ostatních metod a přátel.

6.6 Dynamické informace o polymorfnich typech

Při psaní algoritmů, které volají virtuální funkce se spoléháme na polymorfismus a skutečný typ instancí polymorfních tříd obvykle v programu explicitně nevystupuje. Ve zcela výjimečných případech ale potřebujeme za běhu programu určit, jaký je skutečný typ instance pro daný ukazatel nebo referenci na bázovou třídu. Informace o polymorfních typech určované za běhu programu a jejich přetypování se souhrnně označují zkratkou RTTI (z anglického *Run Time Type Information*). Pro určení typu instance polymorfní třídy slouží operátor `typeid`, pro dynamické přetypování operátor `dynamic_cast`.

6.6.1 Operátor `dynamic_cast` a přetypování ukazatelů

Prvním parametrem operátoru `dynamic_cast` je specifikace ukazatele nebo reference na polymorfní typ a uvádíme ji v lomených závorkách `<>`.

Jestliže za běhu programu je při volání operátoru `dynamic_cast<T*>(p)` parametr `p` ukazatel na objekt, jehož jednoznačným a veřejným předkem je třída `T` (nebo `p` ukazuje přímo na typ T), přetypuje operátor `dynamic_cast` výraz `p` na ukazatel typu `T*`; jinak je výsledkem dynamického přetypování 0.

	lze dědit	může být virtuální	může mít typ	metoda nebo přítel	implicitně generována
implicitní konstruktor	ne	ne	ne	metoda	ano†
kopírovací konstruktor	ne	ne	ne	metoda	ano
jiný konstruktor	ne	ne	ne	metoda	ne
destruktor	ne	ano	ne	metoda	ano
= (přiřazení)	ne	ano	ano	metoda	ano
`()`	ano	ano	ano	metoda	ne
`[]`	ano	ano	ano	metoda	ne
`->`	ano	ano	ano	metoda	ne
konverze	ano	ano	ne	metoda	ne
`new`	ano	ne	`void*`	statická metoda	ne
`delete`	ano	ne	`void`	statická metoda	ne
op=	ano	ano	ano	obojí	ne
jiný operátor	ano	ano	ano	obojí	ne
jiná metoda	ano	ano	ano	metoda	ne
`friend`	ne	ne	ano	`friend`	ne

Tabulka 6.1: † Implicitní konstruktor je generován, pokud nebyl deklarován jiný konstruktor, s výjimkou kopírovacího konstruktoru.

Mějme například tři polymorfní třídy

```
class B {
public:
  virtual ~B(){}
  // ...
};

class C : public B { /* ... */ };
class D : public B { /* ... */ };
```

a dvě jejich instance a jim odpovídající ukazatele na bázovou třídu `B`

```
C c;       B* pc = &c;
D d;       B* pd = &d;
```

Dynamické přetypování můžeme použít například v podmínce příkazu `if (...)`

```
if (D* dd = dynamic_cast<D*>(pd)) {
       // po přetypování dd ukazuje na instanci D
   }

if (D* cc = dynamic_cast<D*>(pc) {
       // tato podmínka nebude splněna, třída C není předkem D a proto cc == 0
   }
```

6.6.2 Operátor `dynamic_cast` a přetypování referencí

Operátor `dynamic_cast` umožňuje přetypování referencí. Pro reference neexistuje obdoba nulového ukazatele a při pokusu o neplatné dynamické přetypování je vyvolána standardní výjimka `bad_cast`. Například pro instance `c` a `d` z předchozího odstavce

```
void f1(B& b)
{
   D& d = dynamic_cast<D&>(b);
   // ...
}

f(c);      // vyvolá výjimku std::bad_cast
f(d);      // v pořádku, B je jednoznačným veřejným předkem D
```

6.6.3 Operátor `typeid`

Operátor `typeid` je dalším nástrojem pro získání informací o typu daného objektu za chodu programu. Parametrem operátoru `typeid` může být jméno typu nebo výraz.

```
if (typeid(c) == typeid(D)) { /* tato podmínka nebude splněna */ }
if (typeid(d) == typeid(D)) { /* tato podmínka  bude  splněna */ }
```

Výraz `typeid` vrací referenci na konstantní statický objekt typu `type_info` deklarovaný v hlavičce `<typeinfo>` a mimo jiné nám poskytuje jméno daného typu jako konstantní C-řetězec

```
class DalsiTrida : public D { /* ... */ };

void f2()
{
  cout << typeid(c).name()          << endl;
  cout << typeid(d).name()          << endl;
  cout << typeid(DalsiTrida).name() << endl;
}

1C
1D
10DalsiTrida
```

Formát C-řetězce reprezentujícího jméno třídy je implementačně závislý. Například kompilátor `g++` verze 3.4.1 nepředává kanonické jméno třídy ale kompaktní řetězec, ve kterém úvodní číslice udávají délku následujícího jména třídy (tato vlastnost ale může být v příštích verzích změněna).

Kapitola 7

Datové proudy

Ve většině našich ukázek jsme doposud vstupní data četli ze standardního vstupního proudu `cin` a výstup jsme zapisovali do výstupního proudu `cout`. Jde o objekty definované ve standardní C++ knihovně. Protože vstupní a výstupní operace nejsou přímo součástí jazyka, ale jsou definovány ve standardní knihovně, musíme objekty `cin` a `cout` v programu deklarovat zařazením hlavičky `<iostream>`. V této kapitole se seznámíme s metodami pro řízení standardních datových proudů a se třídami pro čtení a zápis souborů a paměťových proudů.

Pro vstupní operace je určena třída `istream`, pro výstupní operace třída `ostream`. Třída `iostream` je odvozena od obou uvedených tříd a umožňuje vstup i výstup. V jazyce C++ jsou předdefinovány v hlavičce `<iostream>` tyto standardní objekty

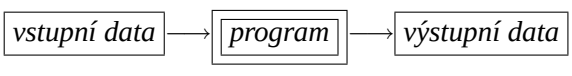
- objekt `cin` třídy `istream` je spojen se standardním vstupním zařízením,
- objekt `cout` třídy `ostream` je spojen se standardním výstupním zařízením.

Standardní vstupní zařízení je u osobního počítače implicitně klávesnice, standardní výstupní zařízení obrazovka. Lze je ale přesměrovat při volání programu z/do souboru, jak ukazuje následující příklad

```
bash$ program < vstup.txt  > vystup.txt
```

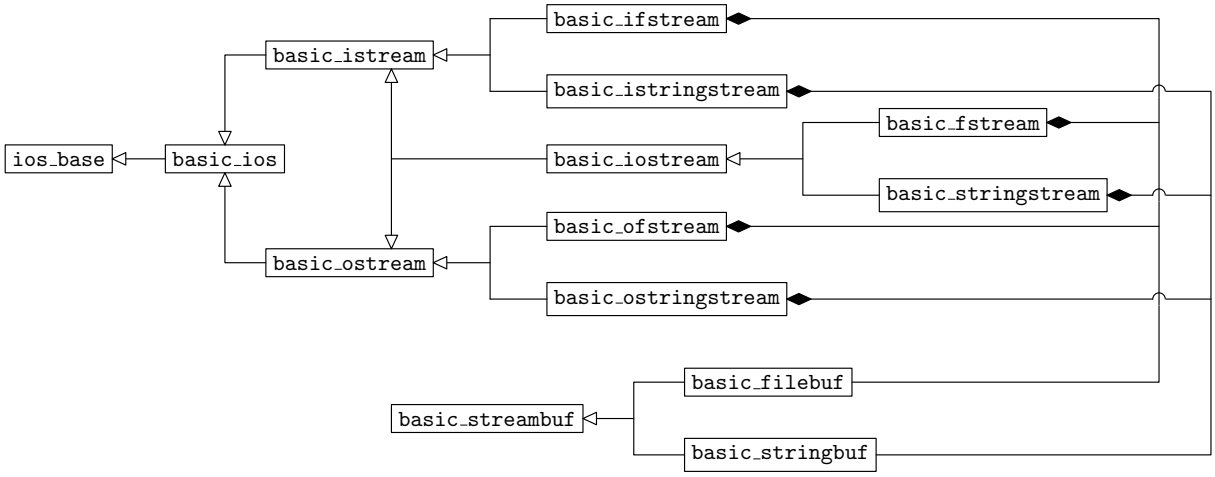
Kromě toho jsou implicitně definovány objekty `cerr` a `clog` pro výstup na zařízení určené pro výpis chybových zpráv (`cerr` bez vyrovnávací paměti, `clog` s vyrovnávací pamětí; u osobního počítače jde opět implicitně o obrazovku).

Dávkovým programům, které používají pro čtení a zápis pouze standardní zařízení vstupu a standardní zařízení výstupu, se říká *filtry*



Filtry lze řetězit, tj. výstup z jednoho programu přímo posílame jako vstup do následujícího programu typu filtr:

```
bash$ program1 < vstup.txt | program2 | program3 > vystup.txt
```



Obrázek 7.1: Hierarchie základních standardních proudů

Filtry jsou velmi mocným nástrojem operačních systémů GNU/Linux a všech variant systému Unix, protože jsou navrženy podle jednoduché zásady *vykonávej jedinou operaci, ale dělej ji dobře.* Jednoduché programy typu filtr jsou také ideálním prostředkem, jak se učit programování.

Pro práci se soubory v C++ programu vytváříme objekty tříd

- `ifstream` (potomek třídy `istream`) pro čtení souboru,
- `ofstream` (potomek třídy `ostream`) pro zápis do souboru,
- `fstream` (potomek třídy `iostream`), umožňuje současné čtení i zápis souboru.

Abychom je mohli použít, musíme v programu začlenit hlavičku `<fstream>`.

Obdobně jako se soubory pracujeme i s paměťovými proudy, které čtou, resp. zapisují, data v řetězcích typu `std::string`

- `istringstream` pro čtení z řetězce,
- `ostringstream` pro zápis do řetězce a
- `stringstream` pro současné čtení i zápis z/do řetězce typu `string`.

Paměťové proudy typu `string` jsou definovány v hlavičce `<sstream>`. Kromě paměťových proudů typu `string` poskytuje C++ i starší paměťové proudu realizované na C-řetězcích typu `char*` (viz odstavec 7.6). Pokud k tomu není závažný důvod (potřebujeme například využít starší zdrojové kódy a pod.), je lépe pracovat s paměťovými proudy typu `string`.

Schéma hierarchie základních tříd pro práci se vstupními a výstupními proudy je znázorněno na obrázku 7.1 (symbol ◁ označuje dědičnost, vyplněný kosodélník agregaci). Jednou z podstatných výhod C++ standardních proudů je, že s nimi pracujeme konzistentně bez ohledu na to, zda čteme data ze souboru nebo třeba ze standardního řetězce `string`. Pro uživatelské třídy můžeme definovat operátory vstupu a výstupu, které se budou chovat analogicky jako operátory pro vstup a výstup základních typů.

S výjimkou třídy `ios_base` jsou všechny třídy znázorněné na obrázku 7.1 šablony, které jsou v běžné praxi patrně nejčastěji používány pro znakový typ `char` a standardní knihovna proto pro ně definuje následující synonyma

```
typedef basic_string<char> string;
typedef basic_ifstream<char> istream;
typedef basic_ofstream<char> ostream;
typedef basic_fstream<char>  iostream;
typedef basic_ifstream<char> ifstream;
typedef basic_ofstream<char> ofstream;
typedef basic_fstream<char>  fstream;
typedef basic_istringstream<char> istringstream;
typedef basic_ostringstream<char> ostringstream;
typedef basic_stringstream<char>  stringstream;
```

7.1 Výstup

Pro všechny základní typy `T` je ve třídě `ostream` definován operátor výstupu

```
ostream& ostream::operator<<(const T&);
```

Například příkaz

```
cout << "ABC ... \n";
```

je voláním operátorové funkce pro vložení literálu typu `const char*` do objektu `cout`.

Protože `operator<<` vrací jako výsledek referenci na `ostream`, můžeme jej aplikovat v jednom příkazu vícekrát. Vícenásobné použití operátoru `<<`

```
cout << "x = " << x;
```

by bylo možno zapsat jako

```
(cout.operator<<("x = ")).operator<<(x);
```

Pro typ `char*`, resp. typ `const char*`, netiskne `operator<<` hodnotu ukazatele (adresu), ale příslušný C-řetězec ukončený nulovým bajtem `'\0'`. Pokud bychom chtěli vytisknout adresu typu `char*`, museli bychom použít explicitní konverzi například na typ `void*`

```
#include <iostream>

int main()
{
   const char* s = "XYZ ...";
   std::cout << "adresa promenne 's' je " << (void*)s << std::endl;
}

adresa promenne 's' je 0x8048a04
```

Konvence pro tisk hodnot ukazatelů se pro různé systémy liší.

V předchozí ukázce je `endl` tzv. manipulátor (bez parametrů), který způsobí přechod na nový řádek. Manipulátor `endl` bychom mohli nahradit znakem `'\n'` (nový řádek) nebo řetězcem `"\n"`. O manipulátorech s parametry budeme hovořit v této kapitole později v odstavci 7.3 věnovaném formátování.

7.1.1 Výstup uživatelských typů

Mějme třídu `datum`, pro kterou chceme napsat funkci `operator<<`. Binární operátor výstupu nemůže být metodou třídy `datum`, protože jeho první operand (operand vlevo od `<<`) je typu `ostream` a musíme jej proto deklarovat jako `friend` funkci se dvěma parametry.

```
#include <iostream>
#include <iomanip>

class datum {
  int den, mesic, rok;
  friend std::ostream& operator<<(std::ostream&, const datum&);
public:
  datum(int d, int m, int r) : den(d), mesic(m), rok(r) {}
  // ...
};

std::ostream& operator<<(std::ostream& out, const datum& d)
{
  const char vz = out.fill();   // ulozim puvodni vyplnovaci znak
  out.fill('0');                // 0 je nyni novy vyplnovaci znak

  out << d.rok << '-'
      << std::setw(2) << d.mesic << '-' << std::setw(2) << d.den;

  out.fill(vz);                 // obnovim puvodni vyplnovaci znak
  return out;
}

int main()
{
  datum dnes(16, 7, 2004);
  std::cout << dnes << std::endl;
}
```

Program vypíše datum jako rok, měsíc a den ve formátu ISO 8601

```
2004-07-16
```

Předchozí příklad představuje základní schéma pro zápis operátoru výstupu `<<` pro libovolný uživatelský typ `T`.

Operátorovou funkci `operator<<` pro zápis objektu typu `datum` bychom nemuseli deklarovat jako `friend` funkci, pokud by v dané třídě byly metody předávající hodnoty atributů `den`, `mesic` a `rok`. Operátor výstupu (a vstupu) ale obvykle píše autor dané třídy a zpřístupnění privátních členů deklarací `friend` je typické.

Situace je jednoduchá, pokud víme, že od třídy `datum` nebudou odvozeny další dceřiné třídy. Pokud má být třída `datum` použita jako bázová, musíme operátor výstupu `<<` napsat jinak. Operátorová funkce `operator<<` není metodou třídy `datum` a nemůže proto být deklarována jako virtuální; obecně `friend` funkce nemůže být virtuální. Standardní řešení je, že v bázové třídě napíšeme pro tento účel speciální virtuální chráněnou metodu, která je volána `friend` funkcí `operator<<`.

```
#include <iostream>
#include <iomanip>

class datum {
protected:
  int den, mesic, rok;
  virtual std::ostream& tisk(std::ostream& out) const
  {
    const char vz = out.fill();   // ulozim puvodni vyplnovaci znak
    out.fill('0');                // 0 je nyni novy vyplnovaci znak

    out << rok << '-'
        << std::setw(2) << mesic << '-' << std::setw(2) << den;

    out.fill(vz);                 // obnovim puvodni vyplnovaci znak

    return out;
  }
  friend std::ostream& operator<<(std::ostream& ostr, const datum& d)
  {
    return d.tisk(ostr);
  }
public:
  datum(int d, int m, int r) : den(d), mesic(m), rok(r) {}
  // ...
};


// czdatum --- obvykly cesky zapis data den/mesic/dvojcisli roku
class czdatum : public datum {
protected:
  std::ostream& tisk(std::ostream& out) const
  {
    const char vz = out.fill();
    out.fill('0');

    out << std::setw(2) << den   << '-'
        << std::setw(2) << mesic << '-' << std::setw(2) << rok%100;

    out.fill(vz);

    return out;
  }
public:
  czdatum(int d, int m, int r) : datum(d, m, r) {}
};

int main()
{
  datum   dnes (16, 7, 2004);
  czdatum zitra(17, 7, 2004);
  std::cout << dnes << "   " << zitra << std::endl;
}
```

Výstup programu

```
2004-07-16   17-07-04
```

demonstruje, že pro odvozenou třídu `czdatum` se změní požadovaným způsobem chování operátoru výstupu `<<`.

Pro naznačenou techniku psaní virtuálních `friend` funkcí je zavedeno anglické označení *virtual friend function idiom*. V našem příkladu pro vyhodnocení výrazu `cout << zitra` použije kompilátor `friend` funkci `ostream& operator<<(ostream&, const datum&)`, které jako druhý argument předá referenci na veřejně odvozenou třídu `czdatum`. V operátoru `<<` se pak díky polymorfismu volá správná virtuální metoda `czdatum::tisk`.

7.2 Vstup

Pro všechny základní typy `T` je ve třídě `istream` definován operátor vstupu

```
istream& istream::operator>>(T&);
istream& istream::operator>>(char*);      // řetězec
```

Implicitně C++ operátor `>>` ignoruje bílé znaky na vstupu. Pro čtení posloupnosti celých čísel ukončených hodnotou `-1` bychom mohli použít cyklus

```
int n = 0;
while (n != -1) {           // špatně zapsaná podmínka
  cin >> n;
  // ...
{}
```

Na podobné konstrukce je třeba dávat bedlivý pozor, protože program se dostane do nekonečného cyklu, pokud se na vstupu vyskytne znak neslučitelný s danou konverzí, v tomto případě s celočíselným literálem.

Korektní zápis je

```
int n;
while (cin >> n) {
  if (n == -1) break;
  // ...
}
```

V podmínce cyklu `while` je přímo použit výraz pro načtení hodnoty. Výraz je vyhodnocen jako pravdivý (`true`), pokud operace vstupu proběhla úspěšně — použití příkazů vstupu v podmínce je pro C++ typické. Předchozí cyklus je tedy ukončen při dosažení konce souboru, při výskytu neplatného znaku, nebo po načtení celočíselné hodnoty `-1`.

Hodnota výrazu `cin >> n` je typu `istream&`. Pokud jej uvedeme přímo v podmínce, je použita implicitní konverze na hodnotu, která zde může být testována. Totéž platí i pro typ `ostream&`. Dále můžeme obdobně pro testování stavu proudů použít operátor `!` (operátor negace).

```
if (!cin) {
  // ... poslední operace se nepodařila ...
}
```

Bílé znaky jsou implicitně ignorovány i při čtení jednotlivých znaků (`char`). Pro čtení znaků a řetězců můžeme použít metody `get` třídy `istream`

```
istream& get(char& c);                      // znak
istream& get(char* p, int n, char='\n');    // C-řetězec
```

které bíle znaky interpretují stejně jako znaky ostatní.

Následující program kopíruje po znacích standardní vstup na standardní výstup.

```
#include <iostream>

int main()
{
  char c;
  while (std::cin.get(c))
    std::cout.put(c);       // totéž jako cout << z
}
```

Volání metody `get` jsme opět použili přímo v podmínce cyklu `while`; metoda `put` zapisuje jeden znak do výstupního proudu (zde `cout`).

Použijeme-li jako druhý argument operátoru vstupu `>>` výraz typu `char*`, načte operátor řetězec (omezený na vstupu bílými znaky) a uloží jej do paměti na adresu, na kterou ukazuje pointer, a připojí ukončující znak `'\0'`.

```
char slovo[20];
cin >> slovo;        // zde může dojít k přetečení
```

Pro načítání textů je proto bezpečnější požít metodu `get`, jejíž druhý argument udává kapacitu pole a třetí nepovinný argument oddělovač záznamů. Oddělovač záznamů se do výstupního pole neukládá a musíme jej načíst samostatně. Pokud načteme jiný znak, znamená to, že na vstupu byl nalezen záznam delší než je kapacita zadaného pole znaků. Metodou `putback` můžeme v tomto a v podobných případech poslední znak vrátit zpět do vstupního proudu. Následující program kopíruje standardní vstup na standardní výstup po řádcích.

```
#include <iostream>

int main()
{
  using namespace std;

  const int max_rad = 80;
  char rad[max_rad];
  char c;

  while (cin.get(rad, max_rad))
  {
    cout << rad;
```

```
    if (cin.get(c) && c != '\n')
    {
      cin.putback(c);
      // ... prilis dlouhy radek ...
    }
    else
      cout << endl;
  }
}
```

7.2.1 Vstup uživatelských typů

Pro danou třídu můžeme napsat operátorovou funkci `operator>>` obdobně jako operátor výstupu `<<`. Pro třídu `datum` budeme požadovat, aby den, měsíc a rok byly na vstupu odděleny bílými znaky a/nebo jedním znakem '-'.

```
istream& operator>>(istream& istr, datum& dat)
{
  int d, m, r;
  char c;

  istr >> r >> c;
  if ( c != '-')  istr.putback(c);
  istr >> m >> c;
  if ( c != '/') istr.putback(c);
  istr >> d;

  dat = datum(d, m, r);
  return istr;
}
```

Funkce `operator>>` z naší ukázky nepracuje přímo s privátními členy třídy `datum` a nemusí být proto deklarována jako `friend` (pro vytvoření nového objekty typu `datum` volá explicitně veřejný konstruktor). Naše ukázka neřeší chyby jako záporné datum nebo hodnota mimo přípustný rozsah; podobné situace by funkce mohla například signalizovat pomocí metody `setstate`, o které si řekneme později.

7.3 Formátování a řízení datových proudů

Třídy `ostream` a `istream` mají společného předka — třídu `ios_base`, která udržuje informace o formátovacích příznacích, základu číselné soustavy, stavu proudu, způsobu otevření proudu a další informace. K atributům třídy `ios_base` přistupujeme buď prostřednictvím jejích metod nebo pomocí tzv. manipulátorů.

Pro popis možných chybových stavů, formátovacích příznaků a dalších charakteristik používá třída `ios_base` tzv. *příznaky (flags)*, jejich přehled uvádí tabulka 7.1, kde synonymum *streamsize* označuje celočíselný typ se znaménkem (typicky `signed long`) používaný pro počet bytů přenesených při vstupní/výstupní operaci.

typ	příznak	popis	
`fmtflags`	`boolalpha`	symbolické vyjádření hodnot `true` a `false`	7.3.5
	`dec`	výstup celých čísel v desítkové soustavě	
	`fixed`	dddd.dd	
	`hex`	výstup celých čísel v šestnáctkové soustavě	
	`internal`	znaménko vlevo, hodnota vpravo	
	`left`	zarovnání výstupu do leva	
	`oct`	výstup celých čísel v osmičkově soustavě	
	`right`	zarovnání výstupu do prava	
	`scientific`	d.ddddd Edd	
	`showbase`	zobrazovat základ soustavy	
	`showpoint`	zobrazuje se desetinná tečka	
	`showpos`	explicitně se uvádí znaménko '+'	
	`skipws`	na vstupu jsou přeskočeny bílé znaky	
	`unitbuf`	zápis bufferu po každé operaci výstupu	
	`uppercase`	'E', 'X' namísto 'e', 'x'	
	`adjustfield`	`left | right | internal`	
	`basefield`	`dec | oct | hex`	
	`floatfield`	`scientific | fixed`	
`iostate`	`badbit`	proud nelze použít	7.3.4
	`eofbit`	konec souboru	
	`failbit`	operace selhala, obvykle chybná konverze	
	`goodbit`	žádná indikace výjimečného stavu	
`openmode`	`app`	výstup na konec souboru (append)	7.4
	`ate`	otevře a nastaví na konec souboru	
	`binary`	otevře soubor jako binární (implicitně text)	7.4.2
	`in`	otevře soubor pro čtení	
	`out`	otevře soubor pro zápis	
	`trunc`	otevření a výmaz souboru, pokud existuje	
`seekdir`	`beg`	druhý parametr funkce `seekg()`	7.4.1
	`cur`		
	`end`		

Tabulka 7.1: Příznaky třídy `ios_base`

Příznaky jsou definovány jako veřejné konstanty, číselné hodnoty jednotlivých příznaků jsou mocniny čísla 2, takže je možné vyjádřit jejich kombinace pomocí bitové operace *or* (`|`) a předávat je jako jediný argument metodám `flags()` a `setf()`. Technika vytváření bitových součtů (*or*) je ale zastaralá a je lépe používat pro řízení proudů manipulátory.

7.3.1 Šířka výstupního a vstupního pole

Operátor výstupu `<<` je pro základní typy `T` definován tak, že implicitně na výstup zapisuje minimální počet znaků potřebných pro vyjádření dané hodnoty typu `T` a po sobě následující údaje neodděluje bílými znaky. Například funkce

`fmtflags flags() const;`	7.3.5
`fmtflags flags(fmtflags` *fmtfl*`);`	
`fmtflags setf(fmtflags` *fmtfl*`);`	
`fmtflags setf(fmtflags` *fmtfl*`, fmtflags` *mask*`);`	
`void unsetf(fmtflags` *mask*`);`	
`size_t precision() const;`	7.3.3
streamsize `precision(`*streamsize přesnost*`);`	
streamsize `width(); const`	7.3.1
streamsize `width(`*streamsize šířka)*`;`	

Tabulka 7.2: Metody třídy `ios_base` pro řízení příznaků a formátování

```
void f1(std::ostream& ostr)
{
  for (double n, m = 1, x = 2; x <= 20; n = m, m = x, x = m + n)
    ostr << x << ' ' << 1/x << std::endl;
}
```

vypíše takovýto text:

```
2 0.5
3 0.333333
5 0.2
8 0.125
13 0.0769231
```

Pokud chceme, aby program vypsal tabulku o dvou sloupcích pevné šířky, musíme naši funkce upravit.

```
void f2(std::ostream& ostr)
{
  for (double n, m = 1, x = 2; x <= 20; n = m, m = x, x = m + n)
  {
    ostr.width(3);
    ostr << x << ' ';
    ostr.width(10);
    ostr << 1/x << std::endl;
  }
}
```

Výstup bude zarovnán do dvou sloupců o třech a deseti znacích (plus jedna mezera zapsaná příkazem `ostr << x << ' ';` oddělující oba sloupce).

```
  2       0.5
  3  0.333333
  5       0.2
  8     0.125
 13 0.0769231
```

Ve funkci `f2` voláme metodu `width(int)` nastavující šířku pole, která má být použita pro výstup následujícího údaje. Pokud daný údaj nelze zapsat do pole o nastavené šířce, je použito tolik znaků, kolik je potřeba. Při výstupu se tedy neztrácejí znaky. Šířka výstupního pole platí pouze pro bezprostředně

následující výstup a pak je přenastavena na implicitní hodnotu 0. Nulová šířka výstupního pole znamená, že má být použito pouze tolik znaků, kolik je pro vyjádření dané hodnoty potřeba.

Metoda `width()` bez argumentů vrací aktuální nastavenou šířku výstupního pole.

```
int sirka_vystupniho_pole = ostr.width();
```

Šířku výstupního pole pro následující údaj můžeme také zadat pomocí manipulátoru `setw(int)`. Jde o manipulátor s parametrem a abychom jej mohli použít, musíme do programu začlenit standardní hlavičku `<iomanip>`. Manipulátory bez parametrů jsou deklarovány v hlavičce `<iostream>`, příkladem je manipulátor `endl` pro přechod na nový řádek. S použitím manipulátoru `setw(int)` bychom napsali naši funkci následovně.

```
void f3(std::ostream& ostr)
{
  using namespace std;

  for (double n, m = 1, x = 2; x <= 20; n = m, m = x, x = m + n)
    ostr << setw(3) << x << ' ' << setw(10) << 1/x << endl;
}
```

```
  2        0.5
  3   0.333333
  5        0.2
  8      0.125
 13  0.0769231
```

Výstup by byl stejný jako v předchozím případě.

Manipulátor `setw` můžeme použít také pro zabránění přetečení při čtení textových řetězců.

```
#include <iostream>
#include <iomanip>

using namespace std;

int main()
{
  const int max_ret = 9;
  char ret[max_ret];

  while (cin >> setw(max_ret) >> ret)
    cout << ret << endl;
}
```

Řetězce delší než 8 znaků jsou rozděleny na více částí.

7.3.2 Vyplňovací znak

V předchozích ukázkách byl pro vyplnění výstupu číselných údajů na zadanou šířku výstupního pole použit implicitní znak mezera. Hodnotu aktuálního vyplňovacího znaku můžeme zjistit metodou `char fill()`. Vyplňovací znak můžeme změnit metodou `char fill(char)` nebo manipulátorem `setfill(char)`. Obě možnosti demonstruje následující ukázka. Nastavení vyplňovacího znaku zůstává v platnosti až do další explicitní změny.

```
void f4(std::ostream& ostr)
{
  using namespace std;

  const char puvodni_znak = ostr.fill();
  for (double n, m = 1, x = 2; x <= 20; n = m, m = x, x = m + n)
    ostr << setfill('*') << setw(3) << x << ' '
         << setfill('#') << setw(10) << 1/x << endl;
  ostr.fill(puvodni_znak);
}
```

```
**2 #######0.5
**3 ##0.333333
**5 #######0.2
**8 #####0.125
*13 #0.0769231
```

Vyplňovací znak můžeme změnit například pro potřebu ladění formátování výstupu (znaky '#' se snáze počítají než mezery). Vyplňování jiným znakem než mezerou se běžně používá v bankovnictví při tisku složenek a pod.

7.3.3 Přesnost výstupu reálných čísel

Implicitně se reálná čísla na výstupu vypisují na 6 platných cifer (úvodní nuly se nepočítají). Tento atribut třídy `ios_base` můžeme změnit manipulátorem `setprecision(int)` nebo metodou `precision(int)`. Aktuální nastavení atributu zjišťuje metoda `precision()`. Nastavení přesnosti zobrazení reálných hodnot na výstupu zůstává v platnosti do následující explicitní změny.

```
void f5(std::ostream& ostr)
{
  using namespace std;

  const int presnost = ostr.precision();
  for (double n, m = 1, x = 2; x <= 20; n = m, m = x, x = m + n)
    ostr << setw(3) << x << ' '
         << setprecision(10) << setw(10) << 1/x << endl;
  ostr.precision(presnost);
}
```

```
  2        0.5
  3 0.3333333333
  5        0.2
  8      0.125
 13 0.07692307692
```

7.3.4 Stav proudu

Třída `ios_base` definuje metody pro testování a řízení stavu vstupních a výstupních proudů. Některé metody třídy `ios_base` jsme uvedli v předchozích odstavcích, o dalších se zmíníme zde. Bázová třída

`ios_base` používá pro popis příznaků *(flags)* výčtové typy. Pro jejich nastavení lze kromě metod použít i manipulátory.

Metoda `ios_base::rdstate() const` vrací stav všech příznaků daného proudu. Návratová hodnota umožňuje testovat libovolnou kombinaci příznaků `goodbit`, `eofbit`, `failbit` a `badbit`. Jednotlivé příznaky jsou uloženy jako bity atributu `iostate` a lze na ně aplikovat operace jako bitový součet (disjunkce po bitech `|`), bitový součin (konjunkce po bitech `&`) a pod.

```
int s = cin.rdstate();  // vraci nastaveni stavovych bitu

if (s & ios_base::goodbit) {
   // zadna indikace vyjimecneho stavu
} else if (s & ios_base::badbit)  {
   // proud nelze pouzit
} else if (s & ios_base::failbit) {
   // operace selhala, obvykle chybna konverze na vstupu
} else if (s & ios_base::eofbit)  {
   // konec souboru
}
```

Jednotlivé hodnoty bitů v atributu `iostate` lze testovat pomocí metod `good()`, `bad()`, `fail()` a `eof()`. Návratová hodnota uvedených metod je `int`. Například:

```
if (cin.fail()) { /* chybná vstupní konverze */ }
```

Metoda `void ios_base::setstate(iostate` *`stav`*`)` nastavuje příznaky stavu proudu; metoda „přidává“ *stav* k již nastaveným příznakům.

Metoda `void ios_base::clear(iostate` *`stav`*`)` nastavuje jednotlivé příznaky na hodnotu argumentu *`stav`*. Je-li metoda volána bez argumentu, je proud nastaven na hodnotu ‘good’ (příznaky chyb jsou vynulovány).

Připomeňme, že objekt typu `istream`, resp. `ostream`, můžeme jednoduše testovat přímo v podmínce. Například

```
if (cin)  { /* pokud je vše v pořádku ... */ }
```

nebo

```
if (!cin) { /* pokud došlo k chybě ... */ }
```

7.3.5 Formátovací příznaky třídy `std::ios_base`

Metoda `ios_base::fmtflags ios_base::flags() const` vrací nastavení všech formátovacích příznaků proudu. Formátovací příznaky můžeme kombinovat pomocí bitových operací, pro jejich globální nastavení pro daný proud je určena metoda `ios_base::fmtflags ios_base::flags (ios_base::fmtflags FMT)`.

```
std::ios_base::fmtflags puvodni_nastaveni_priznaku = cout.flags();
// zmena formatovacich priznaku ...
cout.flags(puvodni_nastaveni_priznaku);
```

V následujícím přehledu uvádíme význam, který mají jednotlivé příznaky, pokud jsou nastaveny.

`dec, oct, hex`

Základ číselné soustavy použité při převodu celých čísel mezi interním vyjádřením a vnější znakovou reprezentací.

Základ soustavy můžeme změnit také pomocí manipulátoru `setbase(int)` — argumentem mohou být hodnoty 8, 10 nebo 16; změna je trvalá — nebo některým z manipulátorů `dec`, `oct` nebo `hex`; změna se uplatní pouze v následující operaci vstupu resp. výstupu.

Na vstupu, pokud nejsou tyto příznaky nastaveny, jsou číselné konstanty interpretovány podle prefixu. Čísla v desítkové soustavě nemají žádný prefix, oktalové konstanty začínají prefixem '`0`' a šestnáctkové prefixem '`0x`' nebo '`0X`'.

```
void f6()
{
  using namespace std;

  ios_base::fmtflags puvodni_nastaveni_priznaku = cout.flags();

  for (int k = 1; k <=3; k++, cout << endl) {
    for (int n = 0; n <= 15; n++)
    {
      switch (k) {
        case 1: cout << dec << setw(4); break;
        case 2: cout << oct << setw(4); break;
        case 3: cout << setbase(16)     // permanentni zmena
                     << setw(4);
      };
      cout << n;
    }
  }
  cout.flags(puvodni_nastaveni_priznaku);
}
```

```
   0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
   0   1   2   3   4   5   6   7  10  11  12  13  14  15  16  17
   0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f
```

`fixed`

Potlačí výstup reálných čísel v tzv. *vědecké notaci* (v semilogaritmickém tvaru s normalizovanou mantisou a dekadickým exponentem). Reálná čísla jsou zobrazována na výstupu s desetinnou tečkou, počet cifer udává atribut `ios_base::precision`.

`scientific`

Pro výstup reálných čísel je použit semilogaritmický tvar s normalizovanou mantisou a dekadickým exponentem (tzv. *vědecká notace*).

`setf(ios_base::fmtflags FLAG)`

`setf(ios_base::fmtflags FLAG, ios_base::fmtflags MASK)`

`unsetf(ios_base::fmtflags FLAG)`

Metoda `setf()` nastavuje příznaky zadané jako argument `FLAG`. Metoda `setf(FMT, MASK)` nejprve vymaže příznaky nastavené v poli `MASK` a potom nastaví příznaky specifikované v argumentu `FMT`. Druhý parametr `MASK` může být

`ios_base::basefield` příznaky číselné soustavy

`ios_base::adjustfield` zarovnání výstupu

`ios_base::floatfield` příznaky řízení výstupu reálných čísel

Metoda `unsetf()` ruší nastavení příznaku, který je předán jako argument.

Všechny uvedené metody vracejí předchozí nastavení všech příznaků.

```
void f7()
{
  using namespace std;

  for (double n = 5; n <= 11; n += 3, cout << endl) {
    cout.setf(ios_base::fixed);
    cout << 1/n << ' ';
    cout.unsetf(ios_base::fixed);

    cout.setf(ios_base::scientific);
    cout << 1/n << ' ';
    cout.unsetf(ios_base::scientific);
  }
}
```

```
0.200000 2.000000e-01
0.125000 1.250000e-01
0.090909 9.090909e-02
```

`showbase`

Zobrazí na výstupu prefix indikující základ použité číselné soustavy. Pro dekadická čísla není žádný prefix, pro osmičkovou soustavu je prefix '0' a pro šestnáctkovou soustavu '0x',

```
void f8()
{
  using namespace std;

  for (int base = 0; base <= 1; base++) {
    if (base == 0)
      cout.setf(ios_base::showbase);
    else
      cout.unsetf(ios_base::showbase);
    for (int n = 5; n <= 11; n += 3)
      cout << setw(4) << dec << n << setw(4) << oct << n
           << setw(4) << hex << n;
    cout << endl;
  }
}
```

```
   5  05 0x5   8 010 0x8  11 013 0xb
   5   5   5   8  10   8  11  13   b
```

`left, right, internal`

Určuje, zda výstup bude zarovnán do leva, do prava nebo na střed (tj. bude-li výstupní pole doplněno znaky `ios_base::fill` před údajem, za údajem nebo mezi znaménkem, resp. označením číselné soustavy, a hodnotou).

```
void f9()
{
  using namespace std;

  cout.fill('#');
  for (double n = 5; n <= 11; n += 3, cout << endl) {
    cout.setf(ios_base::left);
    cout << setw(14) << 1/n << ' ';
    cout.unsetf(ios_base::left);

    cout.setf(ios_base::right);
    cout << setw(14) << 1/n << ' ';
    cout.unsetf(ios_base::right);
  }
}
```

```
0.2########### ###########0.2
0.125######### #########0.125
0.0909091##### #####0.0909091
```

`skipws` Při čtení jsou bílé znaky na vstupu přeskočeny (implicitní hodnota).

`showpos` Znaménko '+' je explicitně zobrazováno na výstupu (implicitně se nezobrazuje).

`showpoint` Desetinná tečka a koncové nuly se zobrazují při vyplnění výstupního pole.

```
void f10()
{
  using namespace std;

  ios_base::fmtflags f = cout.flags();
  for (double n = 5; n <= 11; n += 3, cout << endl) {
    cout.setf(ios_base::showpos);
    cout << setw(14) << 1/n << ' ';
    cout.flags(f);

    cout.setf(ios_base::showpoint);
    cout << setw(14) << 1/n << ' ';
    cout.flags(f);

    cout.setf(ios_base::showpoint | ios_base::fixed);
    cout << setw(14) << 1/n << ' ';
    cout.flags(f);
  }
}
```

```
          +0.2       0.200000       0.200000
         +0.125       0.125000       0.125000
      +0.0909091      0.0909091       0.090909
```

`uppercase`

Pro nenumerické prvky jsou na výstupu použita velká písmena; například ‘`0X5D`’ namísto ‘`0x5d`’ nebo ‘`5.83E-02`’ namísto ‘`5.83e-02`’.

`boolalpha` Hodnoty typy `bool` vypisuje jako `true` a `false`.

7.3.6 Manipulátory

Manipulátory nám umožňují snadno měnit charakteristiky vstupních/výstupních proudů přímo ve výrazech tvořených operátory vstupu `>>` a výstupu `<<`. Kromě manipulátorů bez parametrů (jako například `endl`) jsou i manipulátory s parametry (např. `setw(9)`). Standardní manipulátory s parametry vyžadují hlavičku `<iomanip>`. S většinou standardních manipulátorů jsme se již setkali v předchozích odstavcích, zde je jejich přehled.

`ws` Přeskočí bílé znaky na vstupu.

`flush` Vyprázdní vyrovnávací paměť (buffer), tj. zapíše její obsah do výstupního souboru. Tuto operaci zajišťuje i metoda `flush()`. Například `cout << flush` je totéž jako `cout.flush()`.

`endl` Zapíše znak *konce řádku* `'\n'` a vyprázdní výstupní buffer.

`ends` Zapíše na výstup znak `'\0'` (znak ukončující řetězce).

`setprecision(int)` Nastaví novou hodnotu atributu `ios_base::precision`, který udává požadovanou přesnost zobrazení číselných údajů na výstupu.

`setw(int)` Nastaví požadovanou šířku vstupního/výstupního pole pro další operaci vstupu anebo výstupu.

`setbase(int)` Nastaví základ číselné soustavy. Přípustné hodnoty argumentů jsou 10, 8 a 16.

`dec` Následující vstupní/výstupní operace bude v desítkové soustavě.

`oct` Následující vstupní/výstupní operace bude v osmičkové soustavě.

`hex` Následující vstupní/výstupní operace bude v šestnáctkové soustavě.

`setfill(char)` Nastavuje vyplňovací znak, stejně jako metoda `ios_base::fill(char)`.

7.4 Soubory

Pro čtení souboru je určena třída `ifstream`, pro zápis souboru třída `ofstream` a pro současný zápis i čtení souboru slouží třída `fstream`. Deklarace těchto tříd jsou uvedeny v hlavičce `<fstream>`.

Se soubory pracujeme podobně jako se standardními proudy. Pro přístup k souboru ale musíme nejprve deklarovat objekt příslušného typu. Následující ukázka čte posloupnost čísel ze souboru `vstup.dat`, nezáporná čísla zapisuje do souboru `vystup1.dat` a záporná čísla do souboru `vystup2.dat`.

```
#include <fstream>

int main()
{
  using namespace std;

  ifstream vstup("vstup.dat");
  ofstream nezaporna("vystup1.dat");
  ofstream zaporna("vystup2.dat");

  double n;
  while (vstup >> n)
    if (n >= 0)
      nezaporna  << n << endl;
    else
      zaporna    << n << endl;
}
```

Pro vytvoření objektu `vstup` (typu `ifstream`) je volán konstruktor s jedním parametrem typu `const char*`, jako argument je mu předáno jméno vstupního souboru. Objekt `vstup` pak používáme obdobně jako standardní vstupní proud `cin`; objekty `nezaporna` a `zaporna` (typu `ofstream`) používáme pro zápis údajů analogicky jako standardní výstupní proud `cout`.

Můžeme také vytvořit vstupní/výstupní proud aniž bychom konstruktoru zadali jméno souboru. S takto vytvořeným objektem nemůžeme pracovat, dokud jej nespojíme se souborem pomocí metody `open()`, která má stejné parametry jako konstruktor. Proud můžeme uzavřít, a odpojit od souboru, metodou `close()`.

```
ifstream vstup;
// ...
vstup.open("vstup.dat");
// ...
vstup.close();
```

Tento postup je v C++ programech méně používán. Většinou je výhodnější předat jméno souboru přímo konstruktoru. Uzavření souboru zajistí automaticky destruktor.

Vstupní a výstupní operace jsou v programu vždy klíčové a po otevření souboru bychom vždy měli ověřit, že operace proběhla úspěšně. Nejjednodušší test je ten, že identifikátor proudu použijeme přímo v podmínce.

```
ifstream vstup("vstup.dat");
if (!vstup) {
  // ...
}
```

Stav proudu bychom měli testovat v programu všude tam, kde může dojít k situacím, jako je konec souboru, chybná konverze na vstupu a pod. Pro testování použijeme metody `eof()`, `fail()` a další, o kterých jsme hovořili v předchozím textu.

```
if (vstup.eof() || vystup1.bad() ) {
  // ...
}
```

Při zadávání jména souboru musíme dodržovat konvence našeho operačního systému. Následující fragment programu, ve kterém chceme zadat kromě jména souboru i adresář, demonstruje velmi častou chybu.

```
ifstream ukazka("tmp\test.txt");  // !!! chyba !!!
```

Specifikace jména souboru `test.txt` v adresáři `tmp` je uvedena podle konvencí DOS, Windows, OS/2. Program ale soubor nemůže nikdy otevřít, protože řídící sekvence `\t` označuje tabelátor.

Pro oddělení adresáře a jména souboru používejte znak lomítko (`'/'`). Lomítkem se oddělují adresáře v Unixu a tato konvence je v C++ programech ve jménech souborů akceptována i v systémech, které pro tento účel používají obrácené lomítko (DOS, Windows, OS/2). Obyčejným lomítkem oddělujeme i adresáře ve specifikaci souboru v direktivě `#include`.

Konstruktory tříd `ifstream` a `ofstream` mají druhý nepovinný parametr, který umožňuje specifikovat podrobněji režim otevření souboru.

```
char  jmeno[80];
// ...
ofstream vystupni_proud(jmeno, ios_base::app);
if (vystupni_proud.bad()) {
  // ...
}
```

Soubor můžeme otevřít pro vstup i pro výstup.

```
fstream sez("seznam.dat", ios_base::in | ios_base::out);
```

Na vstupní/výstupní proud se můžeme dívat jako na pole n znaků. Aktuální pozici vstupního proudu zjišťuje metoda `tellg()` a pro výstupní proud metoda `tellp()`. Pokud proud používáme pro vstup i výstup současně, musíme obě metody důsledně rozlišovat.

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

int main()
{
  using namespace std;

  fstream vv("iotell1.dat",
             ios_base::in | ios_base::out | ios_base::trunc);

  for (char z = 'a'; z <= 'c'; z++) {
    cout << "tellp() = " << vv.tellp() << ' ' << z << endl;
    vv << z;
  }
  vv << endl;
  cout << endl;

  vv.seekg(0);
  char p;
  long g = vv.tellg();
  while (vv >> p) {
```

```
    cout << "tellg() = " << g << ' ' << p << endl;
    g = vv.tellg();
  }

}
```

```
tellp() = 0 a
tellp() = 1 b
tellp() = 2 c

tellg() = 0 a
tellg() = 1 b
tellg() = 2 c
```

Pokud v předchozí ukázce chceme, aby cyklus `while` proběhl pouze pro znaky zapsané v cyklu `for`, musíme explicitně zadat volbu `ios_base::trunc` (pokud soubor daného jména již existuje, je při otevření zkrácen na délku 0). Příkaz `vv.seekg(0)` nastavuje pro operaci čtení soubor na začátek.

7.4.1 Funkce `seekg()` a `seekp()`

Pro nastavení vstupního proudu na danou pozici slouží metoda `basic_istream::seekg()`, pro operaci zápisu analogicky slouží metoda `basic_ostream::seekp()`. Jejich jména si snadno zapamatujeme podle anglických slov *get* a *put*. Obě metody `seekg()` a `seekp()` udávají pozici v proudu absolutně od začátku souboru, pokud je voláme s jedním argumentem. Při volání se dvěma argumenty udává první argument relativní pozici vzhledem ke druhému argumentu. Druhý argument metod `seekg()` a `seekp()` může nabývat těchto hodnot:

`ios_base::beg` první argument udává pozici od začátku proudu a může nabývat pouze nezáporných hodnot (implicitní hodnota).

`ios_base::cur` první argument udává relativní pozici od běžné pozice; tj. $\pm n$ bajtů vpřed, resp. zpět, od stávající pozice.

`ios_base::end` první argument udává relativní pozici od konce proudu.

```
#include <fstream>

int main()
{
  using namespace std;

  fstream T("iotell2.dat",
            ios_base::in | ios_base::out | ios_base::trunc);

  T << "0123456789";

  T.seekp( 4, ios_base::beg);  T << "**";    // 0123**6789
  T.seekp(-5, ios_base::cur);  T << "##";    // 0##3**6789
  T.seekp(-3, ios_base::end);  T << "==";    // 0##3**6==9

  T.seekp( 0, ios_base::end);  T << endl;
}
```

```
0##3**6==9
```

7.4.2 Binární přístup

Se soubory a proudy jsme doposud pracovali v implicitním textovém režimu. Při čtení, resp. zápisu, numerických údajů (například typu `double`) docházelo vždy k implicitní konverzi mezi interní binární reprezentací a jejich textovým vyjádřením (posloupností znaků). Alternativou textového režimu je binární přístup. Příznak binárního přístupu je `ios_base::binary`, pro binární čtení slouží metoda `read()` a pro binární zápis `write()`. Binární přenos jednotlivých bajtů také zajišťují metody `put(char)` a `get(char)`, o kterých jsme hovořili dříve.

Při binárním přístupu se přenáší posloupnost bajtů mezi proudem a pamětí bez jakýchkoliv konverzí. Parametry metod `read()` a `write()` jsou:

```
read(T* buff, int n);

write(const T* buff, int n);
```

kde `T*` je typ `char*`, `unsigned char*` nebo `void*`. První parametr udává adresu posloupnosti přenášených bajtů, druhý parametr jejich počet. Pokud je při čtení na vstupu méně bajtů než udává druhý argument, dojde k chybě. Při přenosu se jednotlivé bajty nijak neinterpretují, rozhodující je pouze jejich počet (například `sizeof(double)` pro hodnotu typu `double`).

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

using namespace std;

void b1()
{
  fstream T("test.txt", ios_base::out | ios_base::trunc);
  fstream B("test.bin", ios_base::out | ios_base::trunc
                                      | ios_base::binary);

  double d = 1 / 3.0;
  T << d << endl;
  B.write( (const char*)&d, sizeof(d));
}

void b2()
{
  fstream T("test.txt", ios_base::in /* | ios_base::beg */);
  fstream B("test.bin", ios_base::in /* | ios_base::beg */
                                       | ios_base::binary);

  double dT, dB, d = 1 / 3.0;
  T >> dT;
  B.read( (char*)&dB, sizeof(dB));

  cout << dT - d << "  " << dB - d << endl;
```

```
}

int main()
{
   b1();  b2();
}

-3.33333e-07  0
```

V předchozí ukázce zapisujeme a následně čteme hodnotu `1/3` typu `double` v textovém a binárním režimu. V textovém režimu se zapisuje implicitně pouze šest platných cifer (GNU C++) a při následném čtení získáme proto již pouze zaokrouhlenou hodnotu. Při binárním zápisu a čtení ke ztrátě přesnosti nedochází. Binárně zapsaná data obvykle zabírají menší prostor na disku a přístup k nim je rychlejší.

Připomeňme, že ve výrazech použitých jako první argument metody `write()` a `read()` jsme použili *přetypování (cast)*. Adresu typu ukazatel na `double` (výraz `&dB`) jsme explicitně převedli na typ ukazatel na `char` (výraz `(char*)&dB`).

Binárně můžeme zapisovat, resp. číst, celé objekty (nebo struktury). Mají-li všechny záznamy v souboru stejnou délku, můžeme snadno vypočítat pozici n-tého záznamu.

```
class Bod {
  long cislo;
  double y, x;
  // ....
public:
  // ....
};

void uloz(fstream& f, const Bod& b)
{
  // ...
  f.write( (const char*)&b, sizeof(Bod) );
}
```

Pokud objekt (struktura) obsahuje přímo nebo nepřímo ukazatele na dynamicky alokovanou paměť, je situace složitější a nelze mechanicky celý objekt binárně zapsat, resp. načíst. Pokud by například v naší ukázce číslo bodu bylo typu `string`, bylo by nutné uložit atribut `cislo` explicitně po znacích.

7.5 Paměťové proudy typu `string`

Paměťové proudy realizující vstupní anebo výstupní operace ve standardních řetězcích typu `string` jsou definovány v hlavičce `<sstream>`. Můžeme s nimi realizovat stejné operace jako se soubory (proudy typu *file*):

`istringstream` je třída, jejíž konstruktor je inicializován standardními řetězci typu `string` (nebo i C-řetězci), a která je určena pro čtení údajů.

`ostringstream` vytváří objekty, do kterých můžeme zapisovat stejně jako například do souboru.

`stringstream` je paměťový proud, který umožňuje současný zápis i čtení.

Obsah paměťových proudů typu `stringstream` můžeme získat pomocí konstantní metody `str()`, která vrací hodnotu `string`. Jednou z hlavních výhod těchto paměťových proudů je, že automaticky zajišťují dynamickou alokaci paměti, a proto například při zápisu do proudu typu `ostringstream` nemůže dojít k přepsání alokované paměti (jako tomu může být v případě starších paměťových proudů typu `strstream`, o kterých se zmíníme v odstavci 7.6).

Příklad zápisu do paměťového proudu demonstruje následující funkce, která zajišťuje jednoduchou konverzi číselných typů na standardní řetězec:

```
#include <iostream>
#include <sstream>

template <typename T>
std::string t2string(const T& t)
{
  std::ostringstream ostr;
  ostr << t;
  return ostr.str();
}

int main()
{
  std::cout << t2string(-197)     << std::endl;
  std::cout << t2string(12.2e6)   << std::endl;
}

-197
1.22e+07
```

Čtení dat z paměťového proudu si můžeme ukázat na jednoduché funkci `f()`, která čte vstupní proud po řádcích (volání funkce `getline`), v jednotlivých řádcích odstraní oddělovače *čárka* a jednotlivé řádky pak následně čte po slovech ze vstupního paměťového proudu `istr`. Výstupní proud zapisuje funkce `f()` po slovech, tak, aby délka výstupních řádků nepřesáhla zadanou maximální délku, prázdné řádky opisuje funkce beze změny. Pro řízení výstupu volá funkce `f()` statickou metodu `text()` lokální struktury `zapis`, v praxi by ale takové řešení spíše vhodněji nahradila třída a její spolupracující metody.

```
#include <sstream>

void f(std::istream& in, std::ostream& out)
{
  /**/ struct zapis {
  /**/   static void text(std::ostream& out, std::string& t)
  /**/   {
  /**/     if (t.empty()) return;
  /**/
  /**/     out << t << std::endl;
  /**/     t.clear();
  /**/   }
  /**/ };
```

```
  const unsigned int max_rad = 60;
  std::string radek, slovo, vystup;

  while(getline(in, radek))
    {
      // ... predzpracovani radku pred ctenim jednotlivych slov
      for (int i=0; i<radek.length(); i++)
        if (radek[i] == ',')
          radek[i] = ' ';

      std::istringstream istr(radek);
      if (istr >> slovo)
        {
          do {
            if (vystup.length() + slovo.length() > max_rad)
              zapis::text(out, vystup);

            vystup += slovo;
            vystup += " ";
          } while (istr >> slovo);
        }
      else
        {
          zapis::text(out, vystup);
          out << std::endl;
        }
    }

  zapis::text(out, vystup);
}
```

Pro demonstrační program, který využívá objekt `data` typu `stringstream` pro zápis i čtení v paměti,

```
int main()
{
  std::stringstream data;

  data << "1,    2,    3,\n\n";
  for (int i=4; i<=40; i++) data << i << ",\n";

  f(data, std::cout);
}
```

je výstup následující

```
1 2 3

4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

7.6 Paměťové proudy typu C-řetězec

Hlavička <strstream> definuje třídy istrstream, ostrstream a strstream, které umožňují jak čtení tak i zápis přímo v paměti, resp. v C-řetězcích. Protože C-řetězce neposkytují dynamickou správu paměti, je většinou lépe použít místo knihovny <strstream> paměťové proudy definované v hlavičce <sstream>, které využívají standardní řetězce typu string.

C-řetězcové proudy z knihovny <strstream> jsou realizovány třemi třídami:

istrstream slouží pro čtení z C-řetězce (obdobně jako třída ifstream umožňuje čtení ze souboru).

ostrstream je třída určená pro zápis do C-řetězce (obdobně jako třída ofstream pro zápis do souboru).

strstream je třída určená pro zápis i čtení údajů do/z C-řetězce (analogie třídy fstream pro soubory).

První parametr konstruktorů řetězcových tříd je typu char*, jako argument uvádíme adresu řetězce. Druhý parametr udává počet bajtů, které jsou k dispozici. Třetí parametr slouží pro nastavení příznaků proudu. Ukončující nulový bajt '\0' je interpretován jako konec souboru. V řetězcových proudech nejsou definovány metody open() a close().

```
#include <strstream>
#include <iostream>

int main()
{
  using namespace std;
  const int MAXR = 80;

  char ret_1[MAXR];
  strstream cppr(ret_1, MAXR, ios_base::in | ios_base::out);
  cppr << 1/3.0 << "  " << 0.25 << ends;

  double a, b;
  cppr >> a >> b;
  cout << a << " " << b << endl;

  char ret_2[MAXR] = "# 123 2345 34567 ...";
  cppr.seekp(0);
  cppr.write(ret_2, MAXR);
  cppr.seekg(2);
  cppr >> a >> b;
  cout << a << " " << b << endl;
}
```

```
0.333333 0.25
123 2345
```

V ukázce jsme vytvořili objekt `cppr` typu `strstream`, který jsme spojili s řetězcem `ret_1`. Do prázdného proudu jsme následujícím příkazem zapsali dvě čísla a manipulátorem `ends` ukončující nulový bajt `'\0'`. Z proudu jsme dále obě čísla přečetli a vypsali na standardní výstup. Obsah proudu jsme binárně přepsali obsahem jiného řetězce (`write()`), nastavili explicitně pozici pro čtení a přečetli dva číselné údaje.

Kapitola 8

Výjimky

Výjimky (exceptions) představují nástroj, který je v C++ určen především pro ošetření chybových stavů. Program může vyvolat výjimku (vyhlásit výjimečný stav) příkazem

`throw` *r-hodnota*$_{vol}$ `;`

Volitelný výraz *r-hodnota* má za cíl charakterizovat výjimečný stav a je předáván jako argument kódu pro ošetření výjimky, zcela analogicky jako je předáván argument funkci. Může to být například textový C-řetězec nebo celé číslo ale častěji spíše instance objektového typu.

Typické je vyvolání výjimky v situacích, kdy v daném kontextu nelze řešit nastalou situaci jinak. Představme si metodu pro výpočet součinu dvou matic, která musí reagovat na chybné argumenty, pro které součin neexistuje (nesouhlasí příslušné dimenze) — vyvolání výjimky je ideálním řešením. Pokud v programu není vyvolaná výjimka ošetřena, program předčasně skončí chybou. Akce určená pro ošetření výjimky může být definována nezávisle mimo danou metodu.

Pokud v některé části programu musíme předpokládat možný výskyt chybových stavů, které mohou způsobit vyvolání výjimky, zapíšeme kritický kód do *bloku příkazu* `try` *(try block)*, který je ukončen jedním nebo více *obsluhovači výjimek (exception handler)*. Obsluhovač (handler) začíná klíčovým slovem `catch`, následuje specifikace výjimky v kulatých závorkách a blok, ve kterém je uveden vlastní kód obsluhovače pro ošetření specifikované výjimky.

```
class Q { /* ... */ };

try {
  // ... oblast kódu, který může vyvolat výjimku
  if (podmínka 1) throw "výjimka jako C-řetězec";
  if (podmínka 2) throw int();                  // výjimka typu int
  if (podmínka 3) throw Q();                    // výjimka typu Q
}
catch (const char* s) {
  // ... kód obsluhovače výjimky typu const char*
}
catch (int n) {
  // ... kód obsluhovače výjimky typu int
}
catch (const Q& q) {
```

```
  // ... kód obsluhovače výjimky typu Q
}
```

Blok `try` a pro něj definované obsluhovače výjimek tvoří jednu nedělitelnou programovou konstrukci a obsluhovače výjimek musí následovat bezprostředně za blokem `try`. Pořadí obsluhovačů je přitom významné. Pokud dojde v bloku `try` k vyvolání výjimky `E`, testují se postupně všechny obsluhovače v tom pořadí, v jakém jsou uvedeny. Použit je první obsluhovač jehož parametr `T` vyhovuje výjimce `E` a zbývající jsou pak ignorovány. Je-li dosažen obsluhovač s výpustkou (. . .), je vybrán pochopitelně vždy.

Pro zpracování výjimky typu `E` je vybrán první obsluhovač typu `T`, pro který platí:

- obsluhovač je typu *cv* `T` nebo *cv* `T&` a typy `T` a `E` jsou shodné (*cv* zde označuje volitelnou kvalifikaci `const` anebo `volatile`), nebo
- obsluhovač je typu *cv* `T` nebo *cv* `T&` a `T` je jednoznačným veřejným předkem `E`, nebo
- obsluhovač je typu *cv* `T*` *cv2* a `E` je ukazatel na typ, který může být převeden na typ obsluhovače pomocí standardní konverzí ukazatele (vyjma konverzí na privátní, chráněnou nebo nejednoznačnou třídu) anebo konverze kvalifikace.

Pokud je obsluhovač typu *pole prvků typu T* nebo *funkce vracející T*, je převeden na typ *ukazatel na T*, resp. *ukazatel na funkci vracející T*.

Konverze kvalifikace ukazatelů přibližně řečeno říká, že při automatické konverzi ukazatelů lze podle potřeby přidávat kvalifikace `const` anebo `volatile`. Protože to ale neplatí obráceně, skončí následující program chybou (výjimka nebude zachycena):

```
#include <iostream>

int main()
  try
    {
      throw "ahoj";      // vyvolana vyjimka typu const char*
    }
    catch (char* v)
    {
      std::cout << "zachycena vyjimka typu char* : " << v << "\n";
    }
```

Příklad výběru obsluhovače pro jednoduchou nevirtuální dědičnost ukazuje následující program:

```
#include <iostream>

struct B {};
struct D : public B {};
struct E : public D {};

int main()
{
  for (int i=0; i<3; i++)
    try
      {
```

```
            std::cout << i << " ";
            try
              {
                switch (i)
                  {
                  case 0: std::cout << "throw B()"; throw B();
                  case 1: std::cout << "throw D()"; throw D();
                  case 2: std::cout << "throw E()"; throw E();
                  }
              }
            catch (const D&)            // zachyti vyjimky D a E
              {
                std::cout << " ==> ";
                throw;                  // znovu vyvola puvodni vyjimku
              }
            catch (...)                 // zachyti vyjimku B
              {
                std::cout << " --> ";
                throw;                  // znovu vyvola puvodni vyjimku
              }
          }
        catch (const E&)
          {
            std::cout << "catch E\n";
          }
        catch (const D&)
          {
            std::cout << "catch D\n";
          }
        catch (const B&)
          {
            std::cout << "catch B\n";
          }
    }

    0 throw B() --> catch B
    1 throw D() ==> catch D
    2 throw E() ==> catch E
```

Jak jsme si řekli, představuje blok `try` a jeho obsluhovače nedělitelnou programovou konstrukci a pokud proto některý z obsluhovačů vyvolá výjimku, může být zachycena pouze v nadřazeném příkazu `try-catch`.

Výjimka (exception) nemusí být vyvolána přímo v `try` bloku, ale kdekoliv na libovolné úrovni volání funkcí, konstruktorů nebo operátorů. V ukázce slouží třída `Sleduj` pro trasování volaných funkcí a volání destruktorů.

```
#include <iostream>
#include <string>

using namespace std;

class Sleduj {
```

```
  string s;
public:
  Sleduj(string s_) : s(s_) { cout << "konstruktor: " << s << endl; }
 ~Sleduj()                  { cout << "destruktor : " << s << endl; }
};

void g()
{
  Sleduj G("objekt G ve funkci g()");

  cout << "vyvolana vyjimka string(\"<VYJIMKA>\") ve funkci g()\n\n";
  throw string("<VYJIMKA>");

  Sleduj nenastane("!!! tento objekt nebude vytvoren !!!");
}

void f()
{
  Sleduj F("objekt F ve funkci f()");
  g();
}

int main()
{
  try
    {
      Sleduj T("objekt T v bloku try");
      f();
      Sleduj nenastane("!!! tento objekt nebude vytvoren !!!");
    }
  catch (string x)
    {
      cout << "\nobsluhovaci byl predan string: " << x << endl;
    }

  cout << "\nzpracovani programu pokracuje ... \n";
}
```

Při vyvolání výjimky kompilátor zajistí, že jsou zavolány destruktory pro všechny vytvořené lokální objekty (i pro objekty vytvořené v bloku `try`) a řízení je předáno obsluhovači (handleru) specifikovaného typu, pokud ovšem existuje, viz výstup předchozího programu.

```
konstruktor: objekt T v bloku try
konstruktor: objekt F ve funkci f()
konstruktor: objekt G ve funkci g()
vyvolana vyjimka string("<VYJIMKA>") ve funkci g()

destruktor : objekt G ve funkci g()
destruktor : objekt F ve funkci f()
destruktor : objekt T v bloku try

obsluhovaci byl predan string: <VYJIMKA>
```

```
zpracovani programu pokracuje ...
```

Specifikace výjimky v obsluhovači se řídí obdobnými pravidly, jaká platí pro deklaraci parametrů funkce a předávání argumentů. Obsluhovač výjimky (handler) poněkud připomíná funkci nebo příkaz `switch`. Obsluhovač ale nemusíme ukončit explicitně příkazem `break`, řízení programu pokračuje za příkazem `try` a ne následujícím obsluhovačem.

Vyvolání výjimky způsobí přerušení normálního chodu programu a předání řízení příslušnému obsluhovači výjimky (handleru). Jde přitom výhradně o synchronní přerušení a nelze je použít pro zpracování asynchronních událostí jako je například stisknutí klávesy nebo tlačítka myši.

Na mechanismus zpracování výjimek bychom se docela dobře mohli dívat jako na další řídící konstrukci jazyka C++ a používat ji obdobně jako příkaz `return` pro návrat z volané funkce. Přestože to jazyk umožňuje, tento přístup se nedoporučuje. Pokud používáme výjimky důsledně pouze pro ošetření chybových stavů, můžeme jasně oddělit oblasti kódu určené pro zpracování standardních (správných) situací od výjimečných (chybných) stavů a mimo jiné tak výrazně zlepšit čitelnost programu.

Podívejme se, jaké máme možnosti ošetření chyb zjištěných za chodu programu, pokud bychom nepoužívali C++ mechanismus výjimek:

- Ukončit program: to je implicitní reakce na výjimku, která není zachycena žádným obsluhovačem (handlerem).
- Vrátit hodnotu reprezentující „chybu“: to nejde vždy, například všechny hodnoty typu `double` jsou platné pokud vyjadřují úhel v obloukové míře (v radiánech).
- Vrátit legitimní hodnotu a ponechat program v nelegitimním stavu: pro signalizaci může být použita globální proměnná. Tak například reaguje mnoho standardních C funkcí, které nastavují globální proměnnou `errno`; většina programů ale `errno` netestuje.
- Zadat funkci, která bude volána pro zpracování chyby: v řadě případů jde o uspokojivé řešení. Bez použití výjimek má ale taková funkce k dispozici pouze předchozí tři uvedené možnosti.

Implicitní reakcí na nezachycenou výjimku je ukončení programu, tento drakonický přístup je ve svých důsledcích mnohem bezpečnější než ponechání programu v neplatném stavu, který nemusíme vůbec zaregistrovat.

Třída `ObecnyPrumer` z následující ukázky reaguje na dva výjimečné stavy: a) uživatel požaduje výpočet průměru, aniž zadal vstupní hodnoty (dělení nulou); b) záporná nebo nulová váha údaje.

```
#include <iostream>
#include <cmath>
#include <cstdlib>

using namespace std;

class ObecnyPrumer {
  double s;
  double s2;
  double sp;
  int    N;
public:
```

```
  ObecnyPrumer() : s(0), s2(0), sp(0), N(0) {}
  double prumer() const;
  double m0() const { return N <= 1 ? 0 : sqrt((s2 - s*s/sp)/(N-1)); }
  void pridej(double x, double p=1);
  int  pocet() const { return N; }
};

double ObecnyPrumer::prumer() const
{
  if (sp == 0) throw "suma vah == 0 (nulovy pocet clenu)";
  return s/sp;
}

void ObecnyPrumer::pridej(double x, double p)
{
  if (p <= 0) throw "chybna vaha (vaha mensi nebo rovna nule)";
  sp += p; s += p*x; s2 += p*x*x; N++;
}

int main()
{
  cout << "Vypocet obecneho prumeru\n\n";

  double prvek, vaha;
  ObecnyPrumer P;
  try {                               // v bloku muze dojit k chybe ...
    while (cin >> prvek >> vaha)
      P.pridej(prvek, vaha);          // ... chybna vaha

    double vp = P.prumer();           // ... deleni nulou
    cout << "prumer = " << vp;
  }
  catch (const char* s) {             // zde jsou osetreny chyby
    cout << "vypocet selhal: " << s << endl;
    exit(1);
  }

  cout << "   empiricka jednotkova stredni chyba = "
       << P.m0() << endl;             // zde jiz nemuze nastat chyba
}
```

Tak jako ve funkci, která nepoužívá některý ze svých parametrů, stačí v deklaraci uvést pouze specifikaci typu bez jména parametru, můžeme ve specifikaci výjimky uvádět pouze typ. Zvláštní specifikaci výjimky představuje *výpustka (ellipsis)*, která znamená „vše“. Obsluhovač s výpustkou zachytí všechny výjimky bez rozdílu. Výpustku zapisujeme jako tři tečky (. . .).

Někdy může obsluhovač (handler) ošetřit pouze část kódu. V takovém případě lze po částečné nápravě opětovně vyhlásit danou výjimku příkazem

```
throw ;
```

bez argumentu a ponechat dokončení jinému obsluhovači. V další ukázce třída `TA` uvolňuje dynamicky alokovanou paměť a všechny výjimky předává k ošetření nadřazenému obsluhovači.

```
#include <iostream>

using namespace std;

class TB {
  int N;
public:
  TB(int n) : N(n) {}
  void m_B();
};

void TB::m_B()
{
  switch (N) {
  case 0:  break;
  case 1:  throw "<retezec>";  // throw const char*
  case 2:  throw 123;          // throw int
  default: throw 1.0;          // throw double
  }
}

class TA {
  TB x;
public:
  TA(int n=0) : x(n) {}
  void m_A();
};

void TA::m_A()
{
  double* dm;     cout << "m_A() alokuje pamet";
  try {
    dm = new double[100];
    x.m_B();
    // ... dalsi vypocet ...
  }
  catch (...) {                // zachytim vsechny vyjimky
    delete[] dm;
    cout << " / vyjimka : uvolnena pamet / ";
    throw;                     // znovu vyvolam vyjimku
  }
  // ... dalsi vypocet ...
  delete[] dm;    cout << " / standardni ukonceni\n";
}

int main()
{
  int n;
  while (cin >> n) {
    try {
      TA a(n);
      a.m_A();
      // ... dalsi zpracovani, pokud nedojde k vyjimce ...
```

```
    }
    catch (const char*) {
      cout << "try blok : const char*\n";
    }
    catch (int) {
      cout << "try blok : int  \n";
    }
  }

}
```

Pro vstupní hodnoty `0, 1, 2, 3` skončí program chybou (nezachycená výjimka `double`); výstup programu je

```
m_A() alokuje pamet / standardni ukonceni
m_A() alokuje pamet / vyjimka : uvolnena pamet / try blok : const char*
m_A() alokuje pamet / vyjimka : uvolnena pamet / try blok : int
m_A() alokuje pamet / vyjimka : uvolnena pamet /
```

8.1 Vyvolání výjimky konstruktorem

Konstruktor nemá návratový kód, který by bylo možné předat volajícímu programu, a vyvolání výjimky v situaci, kdy konstruktor z jakékoliv příčiny nemůže vytvořit požadovaný objekt, je nejlepším řešením. Jako příklad nám poslouží konstruktor třídy `Matice`.

Když jsme hovořili o operátoru `new`, řekli jsme si, že operátor může vracet 0 (nulový ukazatel), pokud nedokáže požadovaný objekt vytvořit. Toto řešení je anachronismem, podobně jako možnost zadat prostřednictvím `set_new_handler()` vlastní funkci, která bude volána při neúspěšném pokusu o dynamické vytvoření objektu, a ukončí činnost programu voláním funkce `exit()`. Standardní reakcí na neúspěšné volání operátoru `new` by mělo být vyvolání výjimky (`bad_alloc`).

Mějme třídu

```
class Matice {
  int rad, slp;
  double** m;
public:
  Matice(int r, int s);
 ~Matice();
  // ... dalsi metody ...
};
```

Matici budeme realizovat jako pole ukazatelů na jednotlivé řádky. Destruktor třídy `Matice` musí nejprve uvolnit paměť alokovanou pro jednotlivé řádky a teprve pak vektor ukazatelů.

```
Matice::~Matice()               // destruktor
{
  for (int i=0; i<rad; i++)
    delete[] m[i];              // uvolni i-ty radek
  delete[] m;                   // uvolni vektor ukazatelu
}
```

Destruktor bude volán pouze pro řádně vytvořené objekty, nemusíme v něm předpokládat výjimečné stavy.

Konstruktor musí počítat s možností vyčerpání dynamické paměti během alokace prostoru pro jednotlivé řádky a vyžaduje proto nepatrně více pozornosti.

```
Matice::Matice(int r, int s)     // konstruktor
{
  m = new double* [r];           // pokud vyvola vyjimku, nemusim se
                                 //        ji na tomto miste zabyvat
  slp = s;                       // nastavim pocet sloupcu matice

  // pri alokaci pameti pro radky musim predpokladat moznost vyjimky
  try {
    for (rad=0; rad<r; rad++)    // pocet alokovanych radku
      m[rad] = new double[slp];  // alokovan jeden radek
  }
  catch (...) {
    for (int i=0; i<rad; i++)
      delete[] m[i];             // uvolnim i-ty radek
    delete[] m;                  // uvolnim vektor ukazatelu
    throw;                       // vyvolam znovu vyjimku a poslu ji dal
  }
}
```

Následující fragment programu používá třídu `Matice` pro řešení přeurčené soustavy metodou nejmenších čtverců. Výjimka `bad_alloc` je testována na jediném místě v hlavním programu a ošetřuje možné vyčerpání paměti v hlavním programu stejně jako ve funkci `mnc` — ošetření chyby není vázáno na místo jejího výskytu.

```
void mnc(const Matice& A, const Matice& P, const Matice& l,
         Matice& x, Matice& Qxx)
{
  Matice N = A.transp()*P*A;         // matice normalnich rovnic
  Matice n = A.transp()*P*l;         // absolutni cleny
  Qxx = N.inv();                     // kovariance == inv(N)
  x   = Qxx * n;                     // reseni soustavy
}

int main()
{
  // ...
  try {
    Matice A, P, l;       // matice planu, matice vah, redukovana mereni
    cin >> A >> P >> l;   // ***  zde muze byt vyvolana vyjimka  ***
    Matice x, Qxx;        // nezname a jejich kovarianci matice
    mnc(A, P, l, x, Qxx); // ***  zde muze byt vyvolana vyjimka  ***
    cout << x << Qxx;
  }
  catch (bad_alloc) {
    cout << "***  program skoncil chybou - nedostatecna pamet  ***\n";
    // ...
  }
}
```

Pokud program zachytí výjimku `bad_alloc`, jsou zavolány destruktory pro všechny lokální matice vytvořené po vstupu do bloku `try`, veškerá paměť alokovaná pro matice je uvolněna a program může dále pokračovat.

Příklad

Následující příklad demonstruje ošetření výjimky v konstruktoru:

```
#include<iostream>

class BadClass {
public:

  BadClass(int d):dim(d), pole(0)
  {
    pole = new double[dim];
    std::cout << "konstruktor tridy BadClass" << std::endl;
    throw int();
  }

  ~BadClass()
  {
    delete[] pole;
    std::cout << "destruktor tridy BadClass" << std::endl;
  }

private:

  int     dim;
  double* pole;
};

class P {
public:

  P(int i)
    try : q(i)
      {
        std::cout << "konstruktor tridy P" << std::endl;
      }
      catch(int)
      {
        std::cout << "zachycena vyjimka" << std::endl;
        q.~BadClass();
        // throw;         !!!! NENI POTREBA !!!
      }

  ~P() { std::cout << "desktruktor tridy P" << std::endl; }

private:

  BadClass q;
```

```
};

int main()
  try
    {
      P p(10);
    }
    catch (int)
    {
      std::cout << "nepodarilo se vytvorit objekt p" << std::endl;
    }

konstruktor tridy BadClass
zachycena vyjimka
destruktor tridy BadClass
nepodarilo se vytvorit objekt p
```

8.1.1 Konstruktory s objektovými datovými členy

Naskýtá se otázka, jak řešit situace, kdy daná třída obsahuje objektové datové členy, jejichž konstruktory mohou vyvolat výjimku. Odpověď je prostá. Každý objekt se musí postarat o vlastní likvidaci.

```
#include <iostream>

using namespace std;

class A {
public:
  A() { cout << "ctor A \n"; }
 ~A() { cout << "dtor A \n"; }
};

class B {
public:
  B() { cout << "ctor B - vyvola vyjimku int(1)\n";  throw int(1);}
 ~B() { cout << "dtor B \n"; }
};

class C {
  A a;
  B b;
public:
  C() : a(), b() { cout << "ctor C \n"; }
 ~C() { cout << "dtor C \n"; }
};

int main()
{
  try {
    C c;
  }
```

```
  catch (int x) {
    cout << "hlavni program zachytil vyjimku : int = " << x << endl;;
  }
}
```

Výstup programu dokládá posloupnost probíhajících akcí.

```
ctor A
ctor B - vyvola vyjimku int(1)
dtor A
hlavni program zachytil vyjimku : int = 1
```

Před spuštěním konstruktoru `C()` je volán konstruktor `A()` pro vytvoření objektu "`a`", který proběhne bez chyby. Jako druhý je volán konstruktor `B()` pro vytvoření objektu "`b`", který vyvolá výjimku. Tělo konstruktoru `C()` se neprovede, je zavolán destruktor pro zrušení objektu "`a`" a výjimku `int(1)` zachytí hlavní program.

Situace bude analogická, pokud třída `C` bude odvozena od tříd `A` a `B` a konstruktor `B()` vyvolá výjimku.

```
class C : A, B {
public:
  C() : A(), B() { /* ... */ }
 ~C()            { /* ... */ }
  // ...
};
```

Na závěr se ještě musíme zmínit o možnosti vyvolání výjimky v destruktoru. Situace je zde mnohem jednodušší než u konstruktorů. Rušíme objekt, který již nikdo dále nesmí používat a vyvolání výjimky by proto nemělo způsobit žádné problémy … pokud ovšem destruktor není volán v reakci na ošetření jiné dřívější výjimky. Za takové situace by nebylo jasné, jak má systém reagovat a v jazyce C++ je proto vyvolání výjimky v takovém případě zakázáno. Pokud k ní ale přesto dojde, je volána funkce `terminate` a program je ukončen.

Chceme-li mít jistotu, že náš program nebude v reakci na nepovolené vyvolání výjimky v destruktoru ukončen, můžeme kritický kód destruktoru uzavřít do bloku `try-catch` a pokusit se o nápravu.

8.2 Seznam výjimek v deklaraci funkce

Vyvolání nebo zachycení výjimky ovlivňuje vztah dané funkce k jiným funkcím. V deklaraci funkce můžeme jako sufix uvést seznam výjimek, které daná funkce může přímo nebo nepřímo vyvolat.

```
void f() throw (X, Y)  { /* ... */ }
```

Ukázka deklaruje funkci, ve které může být vyvolána výjimka `X` nebo `Y`.

Seznam výjimek v deklaraci funkce může být prázdný. Pro takovou funkci nepředpokládáme vyvolání žádné výjimky.

```
void g() throw ()  { /* ... */ }
```

Pokud je v deklaraci funkce specifikován seznam výjimek a daná funkce vyvolá výjimku, která není uvedena v seznamu, je volána funkce `unexpected()` (česky *„neočekávaná()“*). Implicitně volá funkce `unexpected()` funkci `terminate()`, která implicitně volá funkci `abort()`.

V následující ukázce je při vyvolání výjimky `int(1)` volána funkce `unexpected()`, která ukončí program. Obsluhovač (handler) s výpustkou `catch (...)` tuto výjimku nezachytí, protože není uvedena v seznamu výjimek v deklaraci funkce `h(int)`.

```
#include <iostream>
#include <cstdlib>

using namespace std;

class E1 { /* ... */ };
class E2 { /* ... */ };
class E3 { /* ... */ };

void h(int e) throw(E1, E2, E3)
{
  switch (e) {
  case 1 : throw E1();
  case 2 : throw E2();
  case 3 : throw E3();
  default: throw int(1);
  }
}

int main ()
{
  for (int i=1; i<=4; i++)
    try {
      h(i);
    }
    catch (E1)  { cout << "E1\n"; }
    catch (E2)  { cout << "E2\n"; }
    catch (E3)  { cout << "E3\n"; }
    catch (...) { cout << "...\n"; }  // !!! nenastane !!!
}
```

Program je po spuštění ukončen voláním `abort()` přes funkci `unexpected()`, viz ukázka z prostředí operačního systému GNU/Linux:

```
bash$ excep_fce
E1
E2
E3
Aborted
bash$
```

Implicitní reakci programu na vyvolání výjimky neuvedené v seznamu výjimek v deklaraci funkce můžeme změnit a nastavit vlastní funkci, která bude volána v takovém případě. Změníme-li v našem příkladu řádky

```
#include <exception>

// ...
void unexpected_xxx() { cout << "unexpected()\n"; exit(1); }

int main ()
{
  std::set_unexpected(unexpected_xxx);
  // ...
}
```

bude výstup programu:

```
bash$ excep_fce
E1
E2
E3
unexpected()
bash$
```

Funkce `set_unexpected()` vrací ukazatel na předchozí nastavenou funkci (analogicky můžeme změnit implicitní funkci `terminate()` prostřednictvím funkce `set_terminate()`). V případě potřeby můžeme obnovit původní nastavení.

8.3 Dědičnost a výjimky

Výjimečné situace, které musí být řádně ošetřeny při práci s maticemi, mohou například být záporné dimenze předané konstruktoru, index prvku mimo přípustný rozsah nebo nepřípustné argumenty operátoru (součet dvou matic o různých dimenzích a pod.). Pokud dojde k výjimce vyvolané některou ze zmíněných chyb, bude v následující ukázce zachycena.

```
try {
  Matice A, P, l;
  cin >> A >> P >> l;
  // ... maticove vypocty ...
}
catch (Mat_dimenze p) {
  // ...
}
catch (Mat_index n) {
  // ...
}
catch (Mat_operator o) {
  // ...
}
```

Ne vždy má smysl explicitně definovat obsluhovač (handler) pro každou možnou výjimku, tak jako v uvedeném příkladu práce s maticemi. Někdy to ani není možné. Představte si, že dostanete novou verzi třídy `Matice`, ve které bude definována nová třída pro ošetření výjimky, která se ve staré verzi nevyskytovala.

Náš příklad bychom pochopitelně mohli přepsat tak, že bychom v příkazu `try` doplnili jako poslední obsluhovač s výpustkou, který by zachytil všechny explicitně nespecifikované výjimky

```
catch (...)  { /* ... */ }
```

Ani toto řešení není vždy přijatelné. Uvedené výjimky `dimenze`, `index` a `operator` mohou být ošetřeny v úseku programu zodpovědném za maticové výpočty; případné jiné výjimky, které s nimi nemají žádnou souvislost by měly (mohly) být ošetřeny jinde. Navíc může v daném kontextu být zcela legitimní požadavek ošetřit explicitně pouze jeden typ chyby a všechny zbývající maticové výjimky řešit společně.

```
try {
  Matice A, P, l;
  cin >> A >> P >> l;
  // ... maticove vypocty ...
}
catch (Mat_dimenze p) {
  // ...
}
catch ( zde chci zachytit všechny ostatní maticové výjimky ) {
  // ...
}
```

Chybnou dimenzi můžeme například načíst ze vstupu a musíme podobné situace řádně ošetřit. Výskyt indexu mimo platný rozsah nebo nepřípustné operace spíše ukazují na chybu v programu, při jejich výskytu se toho obvykle mnoho dělat nedá a lze je ošetřit společně.

Při vyvolání a zachycení výjimky se nepředává typ, ale vždy objekt daného typu. V obsluhovači můžeme deklarovat typ výjimky jako referenci nebo ukazatel a při zpracování zachycené výjimky můžeme proto v plné míře využívat možností, které nám skýtá polymorfismus.

Spolu s třídou `Matice` můžeme definovat rodinu tříd, které budeme používat pro předávání informací při vyvolání výjimek. Nejprve napíšeme bázovou třídu `Mat_chyba`.

```
class Mat_chyba {
public:
  friend ostream& operator<< (ostream& ostr, const Mat_chyba& ch)
    { ch.tisk(ostr);  return ostr; }
protected:
  virtual void tisk(ostream& ostr) const
    { ostr << "Matice: zjistena chyba"; }
};
```

Metodu `tisk()` bychom mohli definovat jako čistě virtuální, třída `Mat_chyba` by pak byla abstraktní.

Od třídy `Mat_chyba` odvodíme třídy `Mat_dimenze`, `Mat_index` a `Mat_operator` a předefinujeme v nich virtuální metodu `tisk()`.

```
class Mat_dimenze : public Mat_chyba {
public:
  Mat_dimenze(int r, int s) : rad(r), slp(s) {}
private:
  int rad, slp;
protected:
```

```
  void tisk(ostream& ostr) const
    { ostr << "Matice: chybna dimenze " << rad << " x " << slp;}
};

class Mat_index : public Mat_chyba {
public:
  Mat_index(int i) : ind(i) {}
private:
  int ind;
protected:
  void tisk(ostream& ostr) const
    { ostr << "Matice: chybny index " << ind; }
};

class Mat_operator : public Mat_chyba {
public:
  Mat_operator(string o) : op(o) {}
private:
  string op;
protected:
  void tisk(ostream& ostr) const
    { ostr << "Matice: chybna operace " << op; }
};
```

Náš příklad můžeme nyní přepsat následovně s využitím mechanismu volání virtuálních funkcí.

```
void f()
{
  try {
    Matice A, P, l;
    cin >> A >> P >> l;
    // ... maticove vypocty ...
  }
  catch (Mat_dimenze p) {
    cout << p << endl;
    // ... osetreni chybne dimenze ...
  }
  catch (Mat_chyba& ch) {
    cout << ch << endl;     // chybny index, chybna operace, ...
    // ... spolecne osetreni ostatnich maticovych chyb ...
  }
}
```

Při vyvolání výjimky typu `Mat_dimenze` ji zachytí obsluhovač určený explicitně pro její zpracování. Při vyvolání výjimky typu, jehož předkem je třída `Mat_chyba`, ji zachytí obsluhovač ve kterém jsme uvedli příkaz `catch (Mat_chyba& ch)`. Typ výjimky je specifikován jako reference a v těle obsluhovače budou volány vždy příslušné virtuální funkce. Případné jiné výjimky nebudou zachyceny a musí byt ošetřeny jinde.

Kapitola 9

Šablony

Pro anglický termín *template* se používá český překlad *šablona*. Deklarace `template` umožňuje deklarovat celou rodinu tříd nebo funkcí, které se liší pouze deklaracemi typů datových členů, argumentů a návratových hodnot funkcí. V deklaraci `template` vystupují jako parametry formální jména těchto typů. Hovoří se proto také někdy o *parametrizovaných* nebo *generických typech*.

Deklarace `template` patří mezi novější rysy jazyka C++. Častěji se s ní setkáme jako s nástrojem pro deklarace parametrizovaných tříd, ale může být použita právě tak pro deklarace skupiny globálních funkcí. Typickým příkladem efektivního využití C++ šablon jsou třídy, označované jako kontejnery *(container)*. Jsou to třídy sloužící k ukládání jiných objektů reprezentující různé datové struktury, jako jsou seznamy, fronty, slovníky a pod. Řada užitečných datových struktur (kontejnerů) je ve standardní knihovně C++ deklarována jako šablony. O některých z nich se stručně zmíníme v kapitole 10.

9.1 `template` funkce

Funkce `mocnina()` v následující ukázce počítá n-tou mocninu svého prvního argumentu, její druhý argument `N` je přirozené číslo.

```
template <typename T>
  T mocnina(const T& A, unsigned int N)
  {
    T tmp = A;
    for (unsigned int i=1; i<N; i++)
      tmp = tmp * A;
    return tmp;
  }
```

Za klíčovým slovem `template` následuje v ostrých závorkách seznam *argumentů šablony*, které v deklaraci funkce používáme obdobně jako standardní nebo uživatelské typy. Kompilátor vygeneruje potřebnou funkci, jestliže v programu narazí na volání funkce, jejíž argumenty odpovídají šabloně, a která nebyla deklarována jinde.

Jde přitom o deklaraci a ne definici funkce. Jazyk C++ nedefinuje, jakým způsobem kompilátor zajistí, že v programu bude daná `template` funkce vygenerována jen jednou.

V následujícím příkladu voláme funkci `mocnina` dvakrát. Poprvé pro výpočet mocniny hodnoty typu `double`, podruhé pro výpočet mocniny matice. S výpočtem mocnin čtvercové matice se určitě běžně nesetkáváme, náš příklad má především demonstrovat, že použití `template` funkcí se nemusí omezovat jen na základní datové typy.

```
#include <iostream>

#include "matvec.h"
#include "mocnina.h"

int main()
{
   double d = 2.0;
   std::cout << mocnina(d, 3) << std::endl;

   Mat A(2, 2);
   A[1][1] = 1.1; A[1][2] = 0.9; A[2][1] = -0.4; A[2][2] = 1.7;
   std::cout << mocnina(A, 5) << std::endl;
}
```

Pokud bychom se pokusili volat funkci `mocnina` s prvním argumentem typu vektor `Vec`, skončil by překlad chybou. Pro typ `Vec` není definován operátor násobení (`*`) a kompilátor by nemohl takovou funkci vygenerovat.

Klíčové slovo `typename` uvozuje formální jméno typu pro danou šablonu. Hovoříme o *typovém argumentu* šablony (v případě `template` tříd máme k dispozici navíc i *hodnotové argumenty*).

Jiným příkladem užití šablony je funkce `Diff`, která počítá odhad diferenciálu zadané funkce jedné proměnné (v našem příkladu odhad diferenciálu funkce sinus v bodě 0.3).

```
template <typename T>
   inline T Diff(T (*f)(T), T x, T dx)  { return f(x+dx) - f(x); }

#include <iostream>
#include <cmath>

int main()
{
  for (double d=0.1; d > 0.0001; d/=2)
    std::cout << Diff(sin, 0.3, d) << " "
              << Diff(sin, 0.3, d)/d - cos(0.3) << std::endl;
}
```

Aby bylo možné určit parametry šablony z volání funkce, musí všechny parametry šablony být i parametry deklarované funkce. Nelze tedy například psát

```
template <class A, class B> B f(A a) { return a; }

void g() { double d = f(12.3); }   // !!! NELZE, B neni argumentem f() !!!
```

protože ze zápisu volání funkce `f()` nelze určit typ návratové hodnoty `B`. Je sice možné při volání funkce zadat parametry šablony explicitně, ale jde o řešení omezené na velmi specifické případy

```
void g() { double d = f<int, double>(12.3); }
```

9.1.1 Přetížení `template` funkcí

`Template` funkce může být přetížena jinými funkcemi se stejným jménem nebo jinými `template` funkcemi stejného jména, pokud je lze rozlišit podle argumentů. Deklarace funkce definované šablonou (zápis prototypu funkce ukončený středníkem) je chápána jako explicitní požadavek na vygenerování příslušné funkce.

```
#include <iostream>

template <typename T>
  inline T AbsDif(T a, T b) { return  a<b ? b-a : a-b; }

int main()
{
  int ia=1, ib=2;
  std::cout << AbsDif(ia, ib) << std::endl;

  double da=1.1;
  double AbsDif(double, double);        // deklarace funkce
  std::cout << AbsDif(da, ib) << std::endl;
}
```

V předchozím příkladu deklarace funkce způsobila její vygenerování podle šablony. Funkce `double AbsDif(double, double)` je volána v následujícím příkazu, kde je hodnota proměnné `ib` překladačem implicitně konvertována na typ `double`. Podle dané šablony kompilátor nemůže vytvořit funkci `AbsDif` se dvěma různými argumenty a bez uvedeného explicitního požadavku na vygenerování funkce by překlad skončil chybou.

Předchozí případ jsme ale mohli vyřešit také explicitním uvedením typu

```
  std::cout << AbsDif<double>(da, ib) << std::endl;
```

a podobně bychom postupovali i v případech, kdy ze zadaných parametrů nedokáže překladač určit typ šablony.

```
#include <iostream>

/* Machine epsilon: the difference between 1 and the least value
 * greater than 1 that is representable. */

template <typename T> T Epsilon()
{
  T one=1, two=2, eps1=0, eps2=1, t, s;

  do
    {
      t = (eps1+eps2)/two;
      s = one + t;
      if (s != one)
        eps2 = t;
      else
        eps1 = t;
      }
```

```
  while ((eps2 - eps1)/eps1 > 0.01);

  return eps1;
}

int main()
{
  std::cout << "float       " << Epsilon<float>()       << std::endl
            << "double      " << Epsilon<double>()      << std::endl
            << "long double " << Epsilon<long double>() << std::endl;
}
```

```
float       5.96046e-08
double      1.11022e-16
long double 5.42101e-20
```

9.2 `template` třídy

S využitím deklarace `template` můžeme napsat třídu `Pole`, aniž bychom předem specifikovali typ jeho prvků.

```
#include <iostream>

template <typename T>
class Pole
{
  T* p;
  int N;
public:
  Pole(int dim) { p = new T[dim]; N = dim; }
 ~Pole() { delete[] p; }
  T& operator[] (int n) { return p[n]; }
};

int main()
{
  Pole<int>    iPole(10);
  Pole<double> dPole(100);
  // ...
}
```

Za klíčovým slovem `template` následuje v ostrých závorkách seznam *parametrů šablony*, které v deklaraci šablony používáme stejně jako standardní nebo uživatelské typy.

V uvedeném příkladu překladač podle šablony `Pole` vygeneruje dvě samostatné třídy pro pole prvků typu `int` a typu `double`. Protože toto generování různých *instancí šablony (template instantiation)* probíhá při překladu, je výsledný kód stejně efektivní jako kdybychom manuálně vytvořili dvě samostatné třídy s různými jmény a editorem v nich pouze zaměnili parametr šablony `T` za `int`, resp. `double`.

Jediná deklarace třídy nám tedy umožňuje vytvářet pole různých typů, které jsou v definici šablony `Pole` označovány typovým parametrem `T`. V netriviálních případech, jako je tento, je odladění šablony obvykle

obtížnější než odladění třídy s pevně danými typy a je možné doporučit obrácený postup, tj. že nejprve odladíme třídu s konkrétními typy a teprve pak ji přepíšeme jako šablonu.

Argumentem šablony může být také jiná template třída (šablona objektového typu)

```
template<typename K, typename H, template<typename T> class P=std::vector>
class Seznam {
     P<K> klic;
     P<H> hodnota;
     // ...
};
```

Třetí parametr šablony `Seznam` je kontejner, který používáme pro ukládání objektů typů `T` a `H` a má definovánu implicitní hodnotu (pokud nestanovíme jinak, je `Seznam` překladačem vygenerován s využitím standardní šablony `vector`).

V deklaraci `template` třídy anebo `template` funkce mohou být argumentem šablony nejen označení typů, ale i konstantní výrazy — *hodnotové argumenty*. Hodnotovým argumentem šablony nemohou být objektové typy nebo pole, ale pouze skalární typy (například ukazatele, velmi často typ `int`). Podmínkou je, že se musí jednat o konstantní výraz, který lze vyhodnotit při překladu. Podobně jako u argumentů funkcí může mít argument šablony předepsánu implicitní hodnotu.

```
#include <iostream>

template <typename Typ, int Dim>
class sPole {                    // staticke pole
  Typ p[Dim];
public:
  const int dim() const { return Dim; }
  Typ& operator[] (int n) { return p[n]; }
};

int main()
{
  sPole< sPole<float, 4>, 3>  A, B;
  for (int i=0; i<A.dim(); i++)
    for (int j=0; j<A[0].dim(); j++)
      A[i][j] = 10*(i+1) + j;
  B = A;
  for (int i=0; i<A.dim(); i++) {
    for (int j=0; j<A[0].dim(); j++)
      std::cout << B[i][j] << ' ';
    std::cout << std::endl;
  }
}
```

V ukázce jsme deklarovali dvě statická pole o třech prvcích, přičemž prvkem pole bylo statické pole o čtyřech prvcích — jinak řečeno, deklarovali jsem statickou matici 3×4 typu `float`. Obvykle si v podobných případech vytvoříme příkazem `typedef` synonymum pro datový typ. V našem případě například

```
typedef sPole< sPole<float, 4>, 3>  Mat3x4;
```

```
Mat3x4  A, B;
// ...
```

Zvláště u složitých deklarací, které jsou těžko čitelné, je to obvyklý postup. Připomeňme, že deklarací `typedef` se nevytváří nový typ, ale pouze náhradní jméno. Jméno třídy `string` ze standardní knihovny C++je například synonymem pro typ

```
typedef basic_string<char> string;
```

9.3 Specializace

Specializace je nástrojem umožňujícím popsat, jak mají být generovány instance dané šablony v případě určitých specifických argumentů. Jako první musíme vždy uvést primární nespecializovanou šablonu a teprve pak její specializace. Argumenty šablony, pro které má být daná specializace používána, se zapisují do lomených závorek <> za jménem šablony.

9.3.1 Explicitní specializace

Deklarace explicitní specializace *(explicit specialization)* začíná `template<>` a může být použita pro

- `template` funkci
- `template` třídu
- členskou funkci `template` třídy
- statický datový člen `template` třídy
- členskou třídu `template` třídy
- členskou `template` třídu `template` třídy
- členskou `template` funkci `template` třídy

Mějme například šablonu

```
template<class T>  T Min(T a, T b} { return a < b ? a : b; }
```

která vrací menší ze dvou zadaných argumentů. Pokud bychom ale chtěli tuto šablonu použít pro výběr menšího ze dvou řetězců typu `const char*`, dávala by chybné výsledky (porovnávala by pouze adresy obou C-řetězců). Můžeme ale snadno doplnit specializaci pro tento případ

```
template<> const char* Min<const char*>(const char* a, const char* b)
     {
         return std::strcmp(a,b)<0 ? a : b;
     }
```

Protože typ argumentů této šablony lze určit ze seznamu parametrů, můžeme za jménem šablony `<const char*>` nahradit prázdnými lomenými závorkami <> nebo je dokonce úplně vypustit a psát

```
template<> const char* Min(const char* a, const char* b)
     {
         return std::strcmp(a,b)<0 ? a : b;
     }
```

9.3.2 Částečná specializace

Částečná specializace *(partial specialization)* může být použita pouze pro `template` třídy. Její použití demonstruje následující příklad převzatý z [2]:

```
template<class T1, class T2, int I> class A             { }; // #1
template<class T, int I> class A<T, T*, I>              { }; // #2
template<class T1, class T2, int I> class A<T1*, T2, I> { }; // #3
template<class T> class A<int, T*, 5>                   { }; // #4
template<class T1, class T2, int I> class A<T1, T2*, I> { }; // #5
```

Jako první je uvedena primární (nespecializovaná) šablona. Podle skutečných argumentů vybírá překladač vhodnou specializaci.

```
A<int, int, 1> a1;    // šablona #1
A<int, int*, 1> a2;   // šablona #2, T je int, I je 1
A<int, char*, 5> a3;  // šablona #4, T je char
A<int, char*, 1> a4;  // šablona #5, T1 je int, T2 je char, I je 1
A<int*, int*, 2> a5;  // nejednoznačné: vyhovuje #3 i #5
```

9.4 Klíčové slovo `typename`

Klíčové slovo `typename` v deklaraci formálního jména parametru šablony může být nahrazeno klíčovým slovem `class`, podle šablony ale může kompilátor generovat funkce i pro základní typy jako je například typ `double`. Možnost použití `class` místo `typename` v deklaraci parametrů šablony je pouze z důvodu historické kompatibility a v daném kontextu je spíše matoucí.

Klíčové slovo `typename` musíme v definici šablony použít tam, kde překladač nedokáže rozpoznat, že dané jméno je typ. To nastane například v případě, že parametrem šablony je objektový typ, ve kterém je definován další vnořený typ.

```
class Vektor {
public:

  class Vyjimka  { /* ... */ };
  class Iterator { /* ... */ };

  int dimenze()    const { /* ... */ }
  Iterator begin() const { /* ... */ }
  Iterator end()   const { /* ... */ }
  // ...
};

template <class T> double SkalarniSoucin(const T& a, const T& b)
{
  if (a.dimenze() != b.dimenze())
    throw
      typename T::Vyjimka();          // bez typename by zde byla chyba

  double s=0;
```

```
  typename T::Iterator i = a.begin();  // bez typename by zde byla chyba
  // ...
  return s;
}
```

Například v definici šablony `SkalarniSoucin` by kompilátor nemohl rozpoznat, že `T::Vyjimka` a `T::Iterator` jsou jména vnořených typů a bez použití klíčového slova `typename` by ohlásil chybu.

Klíčové slovo `typename` nezavádí automaticky nové jméno (synonymum) typu jako `typedef`. V definicích šablon je ale jejich spojení typické, jak ukazuje příklad definice standardního kontejneru `queue` (česky *fronta*):

```
namespace std {
  template <typename T, typename Container = deque<T> >
  class queue {
  public:
    typedef typename Container::value_type           value_type;
    typedef typename Container::size_type            size_type;
    typedef typename Container                       container_type;
  protected:
    Container c;
  public:
    explicit queue(const Container& = Container());

    bool      empty() const             { return c.empty(); }
    size_type size()  const             { return c.size(); }
    value_type&       front()           { return c.front(); }
    const value_type& front() const     { return c.front(); }
    value_type&       back()            { return c.back(); }
    const value_type& back() const      { return c.back(); }
    void push(const value_type& x)      { c.push_back(x); }
    void pop()                          { c.pop_front(); }
  };

  template <typename T, typename Container>
    bool operator==(const queue<T, Container>& x,
                    const queue<T, Container>& y);
  template <typename T, typename Container>
    bool operator< (const queue<T, Container>& x,
                    const queue<T, Container>& y);
  template <typename T, typename Container>
    bool operator!=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
  template <typename T, typename Container>
    bool operator> (const queue<T, Container>& x,
                    const queue<T, Container>& y);
  template <typename T, typename Container>
    bool operator>=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
  template <typename T, typename Container>
    bool operator<=(const queue<T, Container>& x,
                    const queue<T, Container>& y);
}
```

Obecně je nutno klíčové slovo `typename` použít vždy, když je dané jméno závislé na parametru šablony a je typem. Například

```
template <typename C> class X
{
  void f() {
    typename C::T * z;  // deklarace ukazatele z;  C::T označuje typ
  }
  void g() {
    C::T * z;           // součin C::T*z;  C::T je zde statický člen
  }
};
```

Bez klíčového slova `typename` je zápis `C::T` považován za statický člen. V důsledku toho je výraz

```
C::T * z
```

vyhodnocen jako součin statického datového členu `C::T` a proměnné `z`.

9.5 Organizace zdrojových kódů šablon

V principu máme dvě možnosti jak nakládat s definicemi šablon

- direktivou `#include` vložit definice šablon před jejich použitím v dané jednotce překladu (viz 2.1.1 str. 35)
- v každé jednotce překladu vložit před jejich použitím pouze deklarace šablon a přeložit jejich definice odděleně.

V prvním případě přistupujeme k šablonám stejně jako k vloženým funkcím (`inline` funkcím) a řešení problému redundantních definic šablon ponecháváme na kompilátoru.

V druhém případě je nutné deklarovat definice šablon s klíčovým slovem `export` aby byly dostupné i z jiných jednotek překladu.

```
export template <class T> Pole::operator[](int n)
{
   return p[n];
}
```

Běžně dostupné překladače C++ ale klíčové slovo `export` zatím nepodporují.

Pokud náš překladač umožňuje potlačení implicitního generování definic šablon (například pro překladač `g++` je to parametr `-fno-implicit-templates`), můžeme dosáhnout podobného efektu tak, že v nějakém souboru deklarujeme všechny potřebné šablony explicitně

```
export template <class double> Pole;
```

Vazby mezi šablonami jsou ale často složité a výběr všech potřebných deklarací může být pracný. Překladač g++ pod operačním systémem GNU/Linux podporuje automatické ošetření redundantních template definic, na systémech, kde g++ tuto vlastnost neposkytuje, je k dispozici parametr překladu `-frepo` a který lze použít i pod OS GNU/Linux.

Přestože klíčové slovo `export` v deklaraci šablon nemůžeme prozatím používat, je vhodné oddělit deklarace a definice šablon do různých souborů s tím, že soubor definic zařadíme na konec hlavičkového souboru deklarací direktivou `#include`. Takto budeme mít připraven přechod na oddělený překlad šablon, až budou překladače C++ deklaraci `export` podporovat.

9.5.1 Statické datové členy šablon

Všechny statické datové členy šablon (deklarované jako `static`) musí být definovány v některé jednotce překladu. Pokud to jde, je lépe se statickým datovým členům v šablonách vyhnout, protože komplikují jejich praktické použití.

```
#include <iostream>

template <typename T>
struct A {
  static T s;

  A(T t) { s = t; }
};

int main()
{
  A<char> a('x');
  A<int>  i(100);

  std::cout << a.s        << " " << i.s       << std::endl;
  std::cout << A<char>::s << " " << A<int>::s << std::endl;
}

template<char> char  A<char>::s;     // definice statickych atributu
template<int>  int   A<int> ::s;     //
```

Kapitola 10

Standardní knihovna

Prvky standardní C++ knihovny jsou deklarovány nebo definovány ve 33 *hlavičkách (headers)*. Jejich přehled uvádí tabulka 10.1.

<algorithm>	<iomanip>	<list>	<queue>	<streambuf>
<bitset>	<ios>	<locale>	<set>	<string>
<complex>	<iosfwd>	<map>	<sstream>	<typeinfo>
<deque>	<iostream>	<memory>	<stack>	<utility>
<exception>	<istream>	<new>	<stdexcept>	<valarray>
<fstream>	<iterator>	<numeric>	<strstream>	<vector>
<functional>	<limits>	<ostream>		

Tabulka 10.1: Hlavičky standardní knihovny jazyka C++

Pro přístup ke standardním funkcím jazyka C je dále k dispozici 18 hlaviček uvedených v tabulce 10.2.

<cassert>	<ciso646>	<csetjmp>	<cstdio>	<ctime>
<cctype>	<climits>	<csignal>	<cstdlib>	<cwchar>
<cerrno>	<clocale>	<cstdarg>	<cstring>	<cwctype>
<cfloat>	<cmath>	<cstddef>		

Tabulka 10.2: Hlavičky pro využití standardní knihovny jazyka C

Všechna jména jsou ve standardní knihovně deklarována v prostoru jmen `std`, ale z důvodu zpětné kompatibility jsou k dispozici i alternativní hlavičky s příponou `.h`, které definují příslušná jména v globálním prostoru jmen. Standardní hlavičky, jejichž jména začínají písmenem *c*, jsou ekvivalenty odpovídajících hlavičkových souborů jazyka C, tedy ke hlavičce `<cxxx>` definující jména v `std` existuje vždy ekvivalent `<xxx.h>`.

Standardní C++ knihovnu tvoří následující části:

Obecná podpora jazyka

typy	`<cstddef>`
implementační vlastnosti	`<climits>`
	`<cfloat>`
zahájení a ukončení	`<cstdlib>`
dynamická správa paměti	`<new>`
identifikace typu	`<typeinfo>`
správa výjimek	`<exception>`
další runtime podpora	`<cstdarg>`
	`<csetjmp>`
	`<ctime>`
	`<csignal>`
	`<cstdlib>`

Diagnostika

třídy výjimek	`<stdexcept>`
makro *assert*	`<cassert>`
chybové kódy proměnné `errno`	`<cerrno>`

Obecné nástroje

obecné nástroje	`<utility>`
funkční objekty	`<functional>`
paměť	`<memory>`
datum a čas	`<ctime>`

Řetězce

vlastnosti znaků *(character traits)*	`<string>`
řetězcové třídy	`<string>`
práce s řetězci ukončenými znakem *"null"*	`<cctype>`
	`<cwctype>`
	`<cstring>`
	`<cwchar>`
	`<cstdlib>`

Lokalizace

lokalizace (podpora národních nastavení)	`<clocale>`

Kontejnery

Sekvenční kontejnery	`<deque>`
	`<list>`
	`<queue>`
	`<stack>`
	`<vector>`
Asociativní kontejnery	`<map>`
	`<set>`
Kontejner *bitset*	`<bitset>`

Iterátory

Primitiva iterátorů	`<iterator>`
Předdefinované iterátory	
Proudové iterátory	

Algoritmy

Nemodifikující sekvenční operace	`<algorithm>`
Modifikující sekvenční operace	
Operace související s tříděním	

Seminumerické operace

Komplexní čísla	`<complex>`
Numerická pole	`<valarray>`
Zobecněné numerické operace	`<numeric>`
C knihovna	`<cmath>`
	`<cstdlib>`

Vstupy a výstupy

Dopředné deklarace *(forward declarations)*	`<iosfwd>`
Standardní iostream objekty	`<iostream>`
Bázové třídy proudů	`<ios>`
Proudové buffery	`<streambuf>`
Formátování a manipulátory	`<istream>`
	`<ostream>`
	`<iomanip>`
Řetězcové proudy	`<sstream>`
	`<cstdlib>`
Souborové proudy	`<fstream>`
	`<cstdio>`
	`<cwchar>`

10.1 Informace o číselných typech

Hlavičkové soubory `<limits>`, `<climits>` a `<cfloats>` poskytují implementačně závislé informace o základních typech. V jazyce C++ poskytuje jednotný přístup k těmto charakteristikám šablona `numeric_limits` z hlavičky `<limits>`.

```
namespace std {
  template<class T> class numeric_limits;
  enum float_round_style;

  class numeric_limits<bool>;

  class numeric_limits<char>;
  class numeric_limits<signed char>;
  class numeric_limits<unsigned char>;
  class numeric_limits<wchar_t>;
```

```
  class numeric_limits<short>;
  class numeric_limits<int>;
  class numeric_limits<long>;
  class numeric_limits<unsigned short>;
  class numeric_limits<unsigned int>;
  class numeric_limits<unsigned long>;

  class numeric_limits<float>;
  class numeric_limits<double>;
  class numeric_limits<long double>;
}
```

Definice šablony `numeric_limits` je následující

```
namespace std {
  template<class T> class numeric_limits {
  public:
    static const bool is_specialized = false;
    static T min() throw();
    static T max() throw();

    static const int  digits = 0;
    static const int  digits10 = 0;
    static const bool is_signed = false;
    static const bool is_integer = false;
    static const bool is_exact = false;
    static const int  radix = 0;
    static T epsilon() throw();
    static T round_error() throw();

    static const int  min_exponent = 0;
    static const int  min_exponent10 = 0;
    static const int  max_exponent = 0;
    static const int  max_exponent10 = 0;

    static const bool has_infinity = false;
    static const bool has_quiet_NaN = false;
    static const bool has_signaling_NaN = false;
    static const bool has_denorm = false;
    static const bool has_denorm_loss = false;
    static T infinity() throw();
    static T quiet_NaN() throw();
    static T signaling_NaN() throw();
    static T denorm_min() throw();

    static const bool is_iec559 = false;
    static const bool is_bounded = false;
    static const bool is_modulo = false;

    static const bool traps = false;
    static const bool tinyness_before = false;
    static const float_round_style round_style = round_toward_zero;
```

```
  };
}
```

S využitím šablony `numeric_limits` bychom mohli přepsat funkci pro výpočet odmocniny, kterou jsme použili jako triviální příklad v kapitole 1 na straně 20. Template funkce `odmocnina` využívá standardní šablonu `numeric_limits` pro stanovení tolerance pro ukončení iterace při výpočtu pro daný numerický typ (v toleranci `tol` faktor 10 zamezuje možnému zacyklení vlivem *numerického šumu*).

```
#include <iomanip>
#include <iostream>
#include <cmath>

template <typename T> T odmocnina(T n)
{
  const T tol =  n*10*std::numeric_limits<T>::epsilon();
  T x = n/2, q = 0.5;

  while (std::abs(x*x - n) > tol)  x = q * (x + n/x);

  return x;
}

void t(long double x)
{
  std::cout << std::setw(x < 1 ? 14 : 6) << x << " ";
}

int main()
{
  for (long i=1; i<1e+6; i+=153013)
    {
      long double k = i;
      double      d = i;
      float       f = i;

      long double z;
      z = odmocnina(k);  t(i);  t((z*z - k)/k);
      z = odmocnina(d);         t((z*z - k)/k);
      z = odmocnina(f);         t((z*z - k)/k);  std::cout << '\n';
    }
}
```

```
     1              0    2.22045e-15              0
153014              0   -1.16555e-16    2.14371e-08
306027    9.28732e-20    1.48597e-16    3.95655e-08
459040              0   -1.59618e-16   -1.46046e-08
612053   -9.28734e-20     4.7644e-17    -7.0078e-08
765066    7.42987e-20    3.37242e-16    6.60593e-08
918079              0   -4.83561e-17    2.06312e-07
```

10.2 Standardní matematické funkce

Standardní matematické funkce jsou definovány v hlavičkovém souboru `<cmath>` a částečně i v hlavičce `<cstdlib>`.

```
double abs(double);        // absolutní hodnota
double fabs(double);       // totéž jako abs() (pro kompatibilitu s jazykem C)

double ceil(double d);     // nejmenší celé číslo, které není menší než d
double floor(double d);    // největší cele číslo, které není větší než d

double sqrt(double d);     // odmocnina z nezáporného čísla d

double pow(double d, double e);     // d umocněno na e
double pow(double d, int e);

double sin(double);        // sinus
double cos(double);        // kosinus
double tan(double);        // tangens

double asin(double);       // arkussinus
double acos(double);       // arkuskosinus
double atan(double);       // arkustangens
double atan2(double x, double y);  // arkustangens x/y (definováno i pro y ≡ 0)

double sinh(double);       // hyperbolický sinus
double cosh(double);       // hyperbolický kosinus
double tanh(double);       // hyperbolický tangens

double exp(double);        // exponent se základem e
double log(double);        // přirozený logaritmus se základem e
double log10(double);      // dekadický logaritmus

double modf(double d, double* p);  // desetinná část d, celá část v *p
double frexp(double d, int* p);    // hledá x ∈ <0.5, 1) a y,
                                   // tak aby d = x*pow(2,y)
double fmod(double d, double m);   // zbytek po dělení; stejné znaménko jako d
double ldexp(double d, int i);     // d*pow(2, i)
```

Výše uvedené funkce jsou také k dispozici pro argumenty typu `float` a `long double`.

```
#include <iostream>
#include <cmath>

int main()
{
  using std::cout;
  float d=213.4, p, x;
  int   i;

  x = std::modf(d, &p);
  cout << "modf  (" << d << ", float*) : " << x << "  " << p << "\n";
```

```
  x = std::frexp(d, &i);
  cout << "frexp (" << d << ", int*)   : " << x << "  " << i << "\n";

  p=31.7; x = std::fmod(d, p);
  cout << "fmod  (" << d << ", " << p << ")   : " << x <<  "\n";

  i = -3; x = std::ldexp(d, i);
  cout << "ldexp (" << d << ", " << i << ")     : " << x <<  "\n";
}
```

```
modf  (213.4, float*) : 0.399994  213
frexp (213.4, int*)   : 0.833594  8
fmod  (213.4, 31.7)   : 23.2
ldexp (213.4, -3)     : 26.675
```

Z historických důvodů jsou následující matematické funkce definovány v hlavičce `<cstdlib>` a ne v hlavičkovém souboru `<cmath>`.

```
int abs(int);              // celočíselná absolutní hodnota
long abs(long);
long labs(long);
```

Pokud dojde k chybě při volání některé z uvedených funkcí, je nastavena hodnota globální proměnné `errno` na hodnotu `EDOM` *(domain error)* nebo na `ERANGE` *(range error)*. Proměnná `errno` a konstanty `EDOM` a `ERANGE` jsou definovány v hlavičce `<cerrno>`.

```
#include <iostream>
#include <cerrno>
#include <cmath>

int main()
{
  errno = 0;
  std::asin(1.1);
  if (errno == EDOM)   std::cout << "error domain\n";
  if (errno == ERANGE) std::cout << "error range\n";

  errno = 0;
  std::log(0.0);
  if (errno == EDOM)   std::cout << "error domain\n";
  if (errno == ERANGE) std::cout << "error range\n";
}
```

```
error domain
error range
```

10.3 Řetězce

Ze standardní C++ knihovny pro práci s řetězci se v tomto odstavci pouze stručně zmíníme o šabloně `basic_string` a jejích metodách. Nejčastěji se setkáme se dvěma jejími synonymy zavedenými ve standardní knihovně jako

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

10.3.1 basic_string

Ve stručném výčtu metod a funkcí zkracujeme explicitní typ šablony na `basic_string`. Následující operátory a funkce jsou vesměs deklarovány jako šablony s parametry

```
template<typename charT, typename traits, typename Allocator>
```

deklarace šablony

```
namespace std {
  template<typename charT, typename traits = char_traits<charT>,
           typename Allocator = allocator<charT> >
  class basic_string {
  public:
  // types:
    typedef          traits                       traits_type;
    typedef typename traits::char_type            value_type;
    typedef          Allocator                    allocator_type;
    typedef typename Allocator::size_type         size_type;
    typedef typename Allocator::difference_type   difference_type;
    typedef typename Allocator::reference         reference;
    typedef typename Allocator::const_reference   const_reference;
    typedef typename Allocator::pointer           pointer;
    typedef typename Allocator::const_pointer     const_pointer;
    typedef implementation defined                iterator;
    typedef implementation defined                const_iterator;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    static const size_type npos = -1;
        ....
  };
}
```

konstruktory, destruktor a operátory =

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<typename InputIterator>
  basic_string(InputIterator begin, InputIterator end,
               const Allocator& a = Allocator());
~basic_string();
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
```

iterátory

```
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;
```

kapacita

```
size_type size() const;
size_type length() const;
size_type max_size() const;
void resize(size_type n, charT c);
void resize(size_type n);
size_type capacity() const;
void reserve(size_type res_arg = 0);
void clear();
bool empty() const;
```

přístup k prvkům

```
const_reference operator[](size_type pos) const;
reference       operator[](size_type pos);
const_reference at(size_type n) const;
reference       at(size_type n);
```

modifikátory

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
basic_string& append(const charT* s, size_type n);
basic_string& append(const charT* s);
basic_string& append(size_type n, charT c);
template<typename InputIterator>
   basic_string& append(InputIterator first, InputIterator last);
void push_back(charT c);

basic_string& assign(const basic_string&);
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
```

```
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c);
template<typename InputIterator>
   basic_string& assign(InputIterator first, InputIterator last);

basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator      insert(iterator p, charT c = charT());
void          insert(iterator p, size_type n, charT c);
template<typename InputIterator>
    void      insert(iterator p, InputIterator first, InputIterator last);

basic_string& erase(size_type pos = 0, size_type n = npos);
iterator      erase(iterator position);
iterator      erase(iterator first, iterator last);

basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(iterator i1, iterator i2, const basic_string& str);
basic_string& replace(iterator i1, iterator i2, const charT* s, size_type n);
basic_string& replace(iterator i1, iterator i2, const charT* s);
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
template<typename InputIterator>
  basic_string& replace(iterator i1, iterator i2,
                        InputIterator j1, InputIterator j2);

size_type copy(charT* s, size_type n, size_type pos = 0) const;
void swap(basic_string<charT,traits,Allocator>&);
```

operace s řetězci

```
const charT* c_str() const; // explicit
const charT* data() const;
allocator_type get_allocator() const;

size_type  find (const basic_string& str, size_type pos = 0) const;
size_type  find (const charT* s, size_type pos, size_type n) const;
size_type  find (const charT* s, size_type pos = 0) const;
size_type  find (charT c, size_type pos = 0) const;
size_type  rfind(const basic_string& str, size_type pos = npos) const;
size_type  rfind(const charT* s, size_type pos, size_type n) const;
size_type  rfind(const charT* s, size_type pos = npos) const;
size_type  rfind(charT c, size_type pos = npos)
```

```
size_type  find_first_of(const basic_string& str, size_type pos = 0) const;
size_type  find_first_of(const charT* s, size_type pos, size_type n) const;
size_type  find_first_of(const charT* s, size_type pos = 0) const;
size_type  find_first_of(charT c, size_type pos = 0) const;
size_type  find_last_of (const basic_string& str, size_type pos = npos) const;
size_type  find_last_of (const charT* s, size_type pos, size_type n) const;
size_type  find_last_of (const charT* s, size_type pos = npos) const;
size_type  find_last_of (charT c, size_type pos = npos) const;

size_type  find_first_not_of(const basic_string& str,  size_type pos = 0) const;
size_type  find_first_not_of(const charT* s, size_type pos,  size_type n) const;
size_type  find_first_not_of(const charT* s, size_type pos = 0) const;
size_type  find_first_not_of(charT c, size_type pos = 0) const;
size_type  find_last_not_of (const basic_string& str, size_type pos = npos) const;
size_type  find_last_not_of (const charT* s, size_type pos, size_type n) const;
size_type  find_last_not_of (const charT* s, size_type pos = npos) const;
size_type  find_last_not_of (charT c, size_type pos = npos) const;

basic_string substr(size_type pos = 0, size_type n = npos) const;
int        compare(const basic_string& str) const;
int        compare(size_type pos1, size_type n1, const basic_string& str) const;
int        compare(size_type pos1, size_type n1, const basic_string& str,
                   size_type pos2, size_type n2) const;
int        compare(const charT* s) const;
int        compare(size_type pos1, size_type n1, const charT* s,
                   size_type n2 = npos) const;
```

operator

```
basic_string  operator+   (const basic_string& lhs, const basic_string& rhs);
basic_string  operator+   (const charT*        lhs, const basic_string& rhs);
basic_string  operator+   (charT               lhs, const basic_string& rhs);
basic_string  operator+   (const basic_string& lhs, const charT*        rhs);
basic_string  operator+   (const basic_string& lhs, charT               rhs);

bool          operator==  (const basic_string& lhs, const basic_string& rhs);
bool          operator==  (const charT*        lhs, const basic_string& rhs);
bool          operator==  (const basic_string& lhs, const charT*        rhs);
bool          operator!=  (const basic_string& lhs, const basic_string& rhs);
bool          operator!=  (const charT*        lhs, const basic_string& rhs);
bool          operator!=  (const basic_string& lhs, const charT*        rhs);

bool          operator<   (const basic_string& lhs, const basic_string& rhs);
bool          operator<   (const basic_string& lhs, const charT*        rhs);
bool          operator<   (const charT*        lhs, const basic_string& rhs);
bool          operator>   (const basic_string& lhs, const basic_string& rhs);
bool          operator>   (const basic_string& lhs, const charT*        rhs);
bool          operator>   (const charT*        lhs, const basic_string& rhs);

bool          operator<=  (const basic_string& lhs, const basic_string& rhs);
bool          operator<=  (const basic_string& lhs, const charT*        rhs);
```

```
bool          operator<=  (const charT*        lhs, const basic_string& rhs);
bool          operator>=  (const basic_string& lhs, const basic_string& rhs);
bool          operator>=  (const basic_string& lhs, const charT*        rhs);
bool          operator>=  (const charT*        lhs, const basic_string& rhs);
```

swap

```
template<typename charT, typename traits, typename Allocator>
void swap(basic_string& lhs, basic_string& rhs);
```

iostream

```
basic_istream<charT,traits>&
operator>>(basic_istream<charT,traits>& is,   basic_string& str);

basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,  const basic_string& str);

basic_istream<charT,traits>&
getline(basic_istream<charT,traits>& is,      basic_string& str, charT delim);

basic_istream<charT,traits>&
getline(basic_istream<charT,traits>& is,      basic_string& str)
```

10.4 Iterátory

Iterátory jsou zobecněním pojmu ukazatel a umožňují C++ programům pracovat s datovými strukturami (kontejnery) jednotným způsobem. S iterátory pracujeme podobně jako s ukazateli a každá funkce, která pracuje s iterátory, může pracovat i s obyčejnými ukazateli. Pro každý iterátor `i` je definována operace dereferencování `*i`, výsledná hodnota je typu příslušné třídy, výčtového nebo základního typu. Pro všechny iterátory, pro které je definován výraz `(*i).m`, lze použít také zápis `i->m` se stejným významem.

Jazyk C++ definuje pět kategorií iterátorů, v následujícím přehledu uvádíme anglická jména, která jsou dále používána v popisu parametrů generických algoritmů:

Random Access Iterator označuje iterátor pro náhodný přístup,

Bidirectional Iterator obousměrný iterátory,

Forward Iterator dopředný iterátor,

Output Iterator výstupní iterátor a

Input Iterator vstupní iterátor.

Dopředné iterátory (`Forward`) vyhovují všem požadavkům kladeným jak na vstupní tak na výstupní iterátory a mohou je proto vždy zastoupit. Podobně obousměrné iterátory (`Bidirectional`) splňují všechny požadavky kladené na dopředné iterátory (`Forward`) a stejné tak iterátory pro náhodný přístup

(Random Access) mohou být použity všude tam, kde jsou vyžadovány obousměrné iterátory (Bidirectional).

V následujících odstavcích označují a a b hodnoty typu X, n označuje hodnotu typu Distance (*difference type*), u, tmp a m označují identifikátory a r označuje hodnotu X& a t označuje hodnotu typu T.

10.4.1 Vstupní iterátory

Třída nebo základní typ X je vstupním iterátorem, pokud má definovány následující operace a splňuje uvedené požadavky:

`X u(a);`	kopírovací konstruktor. Předpokládá se, že destruktor je přítomen a je přístupný.
`u = a;`	operátor přiřazení.
`a == b`	výsledek relačního operátoru musí být převoditelný na typ bool a operátor musí splňovat vztah ekvivalence.
`a != b;`	výsledek musí být převoditelný na typ bool a musí být roven `!(a == b)`.
`*a`	pro hodnotu a se předpokládá, že je dereferencovatelná a výsledek musí být převoditelný na typ T. Pro hodnoty a a b ze stejného oboru musi platit, že pro `a == b` platí i `*a == *b`.
`a->m`	je ekvivaletní výrazu `(*a).m`, pokud je tento definován
`++r`	výsledný typ musí být převoditelný na `const X&`.
`(void)r++`	ekvivaletní `(void)++r`
`*r++`	totéž jako { `T tmp= *r; ++r; return tmp;` }.

Pro vstupní iterátory rovnost `a == b` neimplikuje `++a == ++b`. Důsledkem je, že algoritmy pracující se vstupními iterátory musí být omezeny na jediný průchod sledem. Algoritmy pro vstupní iterátory se nikdy nesmí pokoušet o průchod přes jeden iterátor dvakrát. Navíc pro typ T nemusí být garantována operace přiřazení. Vstupní iterátory mohou zprostředkovat vstup dat ze vstupních proudů prostřednictvím iterátoru istream_iterator.

10.4.2 Výstupní iterátory

`X(a)`	`a = t` je ekvivalentní `X(a) = t`. Předpokládá se destruktor.
`X u(a);`	u je kopií a.
`X u = a;`	totéž jako v předchozím případě.
`*a = t`	přiřazení hodnoty t na pozici, na kterou ukazuje iterátor a.
`++r`	musí platit `&r == &++r`.

`r++`	výsledek musí být převoditelný na `const X&` a musí platit { `X tmp = r; ++r; return tmp;` }.
`*r++ = t`	přiřazení přes iterátor

Algoritmy mohou výstupní iterátor použít pouze v jednom průchodu. Relace rovnosti a nerovnosti nemusí být definovány. Algoritmy pracující s výstupními iterátory mohou být použity s výstupními proudy prostřednictvím iterátoru typy `ostream_iterator`.

10.4.3 Dopředné iterátory

`X u;`	hodnota `u` může být nedefinovaná (*singular value*). Předpokládá se destruktor.
`X()`	hodnota může být nedefinovaná.
`X(a)`	musí platit `a == X(a)`.
`X u(a);`	musí platit `u == a`.
`X u = a;`	totéž jako v předchozím případě.
`a == b`	výsledek musí být převoditelný na typ `bool`
`a != b`	výsledek musí být převoditelný na typ `bool` a musí být roven `!(a == b)`.
`r = a`	operace přiřazení (`r == a`).
`*a`	výraz `a` musí být dereferenovatelný, `a == b` implikuje `*a == *b` a pokud `X` není konstatní, je výraz `*a = t` platný.
`a->m`	pokud je výraz `(*a).m` platný.
`++r`	před vyhodnocením `r` musí být dereferencovatelné, po provedení musí být `r` dereferencovatelné nebo ukazovat za poslední prvek kontejneru. Pro dereferencovatelné `r` relace `r == s` implikuje `++r == ++s`. Dále `&r == &++r`.
`r++`	výsledek musí být převoditelný na typ `const X&` { `X tmp = r; ++r; return tmp;` }
`*r++`	výsledek je typu `T&`

Jestliže `a == b` pak jsou oba iterátory dereferencovatelné nebo ani jeden z nich není dereferencovatelný. Jestliže jsou `a` i `b` dereferencovatelné, pak platí `a == b` tehdy a jen tehdy když `*a` a `*b` je jeden a tentýž objekt.

10.4.4 Obousměrné iterátory

`--r` musí existovat takové `s`, pro které před vyhodnocením výrazu platí `r == ++s`. Dále `--(++r) == r`. Pro dereferencovatelné `s` rovnost `--r == --s` implikuje `r == s`. Musí platit `&r == &--r`.

`r--` výsledek musí b ýt převoditelný na typ `X&`.

{ `X tmp = r; --r; return tmp;` }

`*r--` výsledek je převoditelný na typ T.

Kromě toho splňují obousměrné iterátory všechny požadavky kladené na dopředné iterátory.

10.4.5 Iterátory pro náhodný přístup

`r += n` hodnota výrazu je typu `X&`, Sémantika výrazu je

```
{  Distance m = n;
   if (m >= 0)
      while (m--)
         ++r;
   else
      while (m++)
         --r;
   return r;  }
```

`a + n` typ výrazu je `X` a platí `a + n == n + a`.

{ `X tmp = a; return tmp += n;` }

`n + a` viz předchozí bod

`r -= n` totéž jako { `return r += -n;` }

`a - n` { `X tmp = a; return tmp -= n;` }

`b - a` vzdálenost mezi dvěma prvky, výsledek je typu `Distance`. Musí existovat takové `n`, že `a + n == b`. Dále platí `b == a + (b - a)`.

`a[n]` výsledek výrazu m musí být převoditelný na typ `T`, sémantika výrazu je `*(a + n)`.

`a < b` výsledek musí být převoditelný na typ `bool`, sémantika výrazu je `b - a > 0`.

`a > b` výsledek musí být převoditelný na typ `bool`

`a >= b` výsledek musí být převoditelný na typ `bool`, `!(a < b)`

`a <= b` výsledek musí být převoditelný na typ `bool`, `!(a > b)`

Kromě toho splňují iterátory pro náhodný přístup všechny požadavky kladené na obousměrné iterátory.

10.5 Kontejnery

Součástí C++ standardní knihovny *(C++ Standard library)* je celá řada *kontejnerů (Containers library)*, které jsou napsány jako `template` třídy. Standardní C++ knihovna kromě nich obsahuje `template` funkce určené pro práci s kontejnery *(Algorithms library)*. Protože pracují s generickými typy, hovoří se někdy o *generických algoritmech*. Pro komunikaci s kontejnery používají generické funkce zvláštní objekty, kterým se říká iterátory *(iterator)*.

Standardní kontejnery a jejich používání je koncipováno jednotně, odpovídající metody pro různé typy se volají stejně. Standardní C++ knihovna má do značné míry základ v knihovně STL *(Standard Template Library)*. K jejímu masovému rozšíření přispělo, že firma Hewlett-Packard koncem roku 1994 poskytla na Internetu k volnému použití svoji implementaci STL. Knihovna STL byla zařazena do návrhu ANSI/ISO standardu C++ a dnes je nedílnou součástí C++ standardní knihovny.

Standardní C++ knihovna obsahuje sekvenční a asociativní kontejnery, které zajišťují operace se seznamy jiných objektů. V následujícím výčtu jsou označovány jménem hlavičky

sekvenční kontejnery:

`<vector>` tento kontejner by měl být použit jako implicitní, tj. pokud není explicitní důvod pro použití jiného typu sekvenčního kontejneru

`<list>` bychom měli použít, pokud jsou často vkládány a rušeny prvky uprostřed seznamu

`<deque>` je datová struktura vhodná pro případy, kdy dochází k častému vkládání a rušení prvků na začátku nebo na konci seznamu

`<queue>` realizuje *frontu*, do které přidáváme prvky na konci a na začátku fronty je odebíráme. Kromě toho existuje i prioritní fronta `priority_queue`, ve které jsou prvky řazeny podle velikosti.

`<stack>` je *zásobník*, ve kterém přidáváme a vybíráme prvky pouze na jednom konci seznamu. Poslední uložený prvek vybíráme jako první.

asociativní kontejnery:

`<map>` je asociativní kontejner, který zajišťuje rychlý přístup k libovolnému prvku přes jednoznačný klíč. Kontejnerům typu `map` se někdy také říká slovníky *(dictionary)*. Podobným typem je `multimap`, který umožňuje ukládat prvky seznamu s duplicitními klíči (několik různých prvků může mít stejný klíč).

`<set>` datový typ množina, obdobně `multiset` je *množina*, ve které se prvky mohou vyskytovat vícekrát.

`<bitset>` definuje `template` třídu a několik souvisejících funkcí pro reprezentaci a práci s bitovými posloupnostmi pevné délky

10.5.1 Kontejner `vector`

Pro kontejner `vector` máme k dispozici následující konstruktory (`T` je typ prvků, které chceme pomocí kontejneru `vector` ukládat).

`vector<T> v1;` implicitní konstruktor, který vytvoří prázdný seznam

`vector<T> v2(n, hodnota);` vytvoří seznam o n členech, které budou všechny kopií výrazu `hodnota` (musí být typu `T` nebo pro něj musí existovat implicitní konverze na typ `T`).

`vector<T> v3(n);` obdobně jako v předchozím případě, vytvoří se ale n kopií implicitní hodnoty typu `T`. Jde-li o objektový typ, musí mít implicitní konstruktor.

```
#include <iostream>
#include <vector>

class Foo {
  int n;
public:
  Foo(int N=0) { n = N; }
  friend std::ostream& operator <<(std::ostream&, const Foo&);
};

std::ostream& operator<<(std::ostream& ostr, const Foo& h)
{
  return ostr << h.n;
}

int main()
{
  std::vector<Foo> VecFoo;
  Foo a(1), b(3), c(5), d(7), e(11);
  VecFoo.push_back(a);    // ulozim prvek 'a' na konec seznamu
  VecFoo.push_back(b);    //              'b'
  VecFoo.push_back(c);    //              'c'
  VecFoo.push_back(d);    //              'd'
  VecFoo.push_back(e);    //              'e'

  // pristup k prvkum pres iterator (stejne pro list, deque, ...)
  for (std::vector<Foo>::const_iterator i=VecFoo.begin(); i != VecFoo.end(); ++i)
    {
      std::cout << (*i) << std::endl;
    }

  // zruseni posledniho prvku seznamu
  VecFoo.pop_back();

  // reference na prvni a posledni prvek seznamu
  std::cout << VecFoo.front() << " " << VecFoo.back() << std::endl;

  // vector umoznuje indexovani  (nelze pouzit pro list nebo deque)
```

```
  for (unsigned int i=0; i < VecFoo.size(); i++)
    std::cout << VecFoo[i] << std::endl;
}
```

V naší předchozí ukázce jsme vytvořili prázdný kontejner `VecFoo` určený pro ukládání objektů typu `Foo`. Dále jsme deklarovali objekty `a` až `e` typu `Foo` a uložili je do připraveného kontejneru. Objekty jsme uložili do kontejneru metodou `push_back()` (česky *ulož na konec*).

V následujícím cyklu jsme postupně prošli všechny prvky kontejneru a vytiskli je na standardní výstupní proud `std::cout`. Řídící proměnná cyklu byla typu iterátor, přesněji řečeno iterátor typu

```
std::vector<Foo>::iterator
```

Iterátory jsou ve standardní C++ knihovně objekty, které mají pro kontejnery analogický význam jako ukazatele v C++. Iterátor ukazující na první prvek kontejneru vrací metoda `begin()`, ukazatel za poslední prvek vrací metoda `end()`. Z naší ukázky je zřejmé, že pro iterátor je definován relační operátor nerovnosti `!=` a prefixový inkrement. Analogicky můžeme procházet všechny standardní C++ kontejnery (změní se pouze deklarace typu kontejneru před operátorem rozlišení oboru `::`).

Metodou `pop_back()` jsme zrušili v kontejneru poslední prvek. Referenci na poslední a první prvek kontejneru vrací metody `front()` a `back()` použité v naší ukázce v příkazu zápisu na `std::cout`. Všechny zmíněné metody bychom zcela analogicky použili i pro sekvenční kontejnery `list` a `deque`.

Poslední cyklus, ve kterém indexujeme prvky kontejneru pomocí operátoru `[]`, je ale již specifický pro typ `vector`. Metoda `size()` vrací počet prvků uložených v kontejneru.

Metody kontejneru `vector`

Operace vkládání a rušení prvků na konci vektoru jsou prakticky stejně efektivní jako odpovídající operace s poli, právě tak jako náhodný přístup k prvkům vektoru. Operace vkládání a rušení jinde než na konci jsou ale výrazně nákladnější, protože vyžadují kopírování všech následujících prvků (srovnatelně nákladné by ale bylo, pokud bychom totéž chtěli provádět s poli). Standardní kontejner `vector` je téměř vždy vhodnější volbou než dynamická pole.

Vložení nebo zrušení prvku obecně zneplatní dřívější iterátory.

```
template <typename T, typename Allocator = allocator<T> >
class vector {
public:
  typedef typename Allocator::reference             reference;
  typedef typename Allocator::const_reference       const_reference;
  typedef implementation defined                    iterator;
  typedef implementation defined                    const_iterator;
  typedef implementation defined                    size_type;
  typedef implementation defined                    difference_type;
  typedef T                                         value_type;
  typedef Allocator                                 allocator_type;
  typedef typename Allocator::pointer               pointer;
  typedef typename Allocator::const_pointer         const_pointer
  typedef std::reverse_iterator<iterator>           reverse_iterator;
  typedef std::reverse_iterator<const_iterator>     const_reverse_iterator;
```

```
  explicit vector(const Allocator& = Allocator());
  explicit vector(size_type n, const T& value = T(),
      const Allocator& = Allocator());
  template <typename InputIterator>
    vector(InputIterator first, InputIterator last,
      const Allocator& = Allocator());
  vector(const vector<T,Allocator>& x);
 ~vector();
  vector<T,Allocator>& operator=(const vector<T,Allocator>& x);
  template <typename InputIterator>
    void assign(InputIterator first, InputIterator last);
  template <typename Size, typename U> void assign(Size n, const U& u = U());
  allocator_type get_allocator() const;
  ...
};
```

konstruktory, destruktor a operátor přiřazení kontejneru `vector`

```
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
```

iterátor ukazující na první prvek, resp. za poslední prvek, kontejneru

```
reverse_iterator       begin();
const_reverse_iterator begin() const;
reverse_iterator       end();
const_reverse_iterator end() const;
```

iterátory pro zpětný průchod kontejnerem — vrací iterátor na poslední prvek kontejneru (tj. první prvek obráceného sledu), resp. iterátor za poslední prvek sledu při průchodu v obráceném pořadí (reverse iterator)

```
size_type size() const;
```

počet prvků uložených v kontejneru

```
size_type max_size() const;
```

vrací maximální počet prvků, které mohou být v kontejneru uloženy

```
void resize(size_type sz, T c = T());
```

realokace paměti kontejneru (zneplatní všechny iterátory)

```
size_type capacity() const;
```

vrací počet prvků, které lze v kontejneru uložit bez realokace paměti

```
bool empty() const;
```

vrací hodnotu `true`, pokud je kontejner prázdný

```
void reserve(size_type n);
```

informuje `vector` o plánované změně velikosti, tak aby mohl realokovat vhodným způsobem paměť

```
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
reference       at(size_type n);
const_reference at(size_type n) const;
```

vrací referenci na n-tý prvek kontejneru typu `vector`

```
reference       front();
const_reference front() const;
reference       back();
cosnt_reference back() const;
```

reference na první, resp. poslední, prvek

```
void push_back(const T& x);
```

vloží prvek `x` na konec kontejneru typu `vector`

```
void pop_back();
```

zruší poslední prvek kontejneru. Výsledek není definován, je-li `vector` prázdný.

```
iterator insert(iterator position, const T& x);
```

vloží prvek `x` na pozici, na kterou ukazuje iterátor `position`, a posune dříve uložené prvky podle potřeby. Návratová hodnota iterátoru ukazuje na pozici vloženého prvku

```
void insert(iterator position, size_type n, const T& x)
```

vloží n kopií prvku `x`

```
template <typename InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);
```

vloží prvky ze sledu `[first, last)` na pozici zadanou iterátorem `position`

```
iterator erase(iterator position);
```

zruší prvek na který ukazuje iterátor `position`

```
void erase(iterator first, iterator last)
```

zruší všechny prvky ze sledu `[first, last)`

```
void swap(vector<T, Allocator>& x)
```

vymění prvky běžného vektoru a vektoru `x`.

```
void clear();
```

zruší všechny prvky kontejneru

```
bool operator==(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
bool operator< (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
bool operator!=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
bool operator> (const vector<T,Allocator>& x, const vector<T,Allocator>& y);
bool operator>=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
bool operator<=(const vector<T,Allocator>& x, const vector<T,Allocator>& y);
```

relační operátory pro instance typu `vector`

```
void swap(const vector<T,Allocator>& x, const vector<T,Allocator>& y)
```

zamění obsah kontejnerů `x` a `y`

10.5.2 Kontejner `deque`

Sekvenční kontejner `deque` je velmi podobný kontejneru `vector`. Hlavní rozdíl mezi nimi je ten, že `deque` umožňuje efektivní vkládání a rušení prvků nejen na konci ale i na začátku posloupnosti. Ostatní operace jsou stejně rychlé nebo o konstantní faktor pomalejší než u typu `vector`.

Přidání nebo zrušení prvku zneplatní ostatní dřívější iterátory.

Metody kontejneru `deque`

```
template <typename T, typename Allocator = allocator<T> >
class deque {
public:
  typedef typename Allocator::reference             reference;
  typedef typename Allocator::const_reference       const_reference;
  typedef implementation defined                    iterator;
  typedef implementation defined                    const_iterator;
  typedef implementation defined                    size_type;
  typedef implementation defined                    difference_type;
  typedef T                                         value_type;
  typedef Allocator                                 allocator_type;
  typedef typename Allocator::pointer               pointer;
  typedef typename Allocator::const_pointer         const_pointer;
  typedef std::reverse_iterator<iterator>           reverse_iterator;
  typedef std::reverse_iterator<const_iterator>     const_reverse_iterator;

  explicit deque(const Allocator& = Allocator());
  explicit deque(size_type n, const T& value = T(),
      const Allocator& = Allocator());
  template <typename InputIterator>
    deque(InputIterator first, InputIterator last,
          const Allocator& = Allocator());
  deque(const deque<T,Allocator>& x);
 ~deque();
  deque<T,Allocator>& operator=(const deque<T,Allocator>& x);
  template <typename InputIterator>
    void assign(InputIterator first, InputIterator last);
  template <typename Size, typename T>
    void assign(Size n, const T& t = T());
  allocator_type get_allocator() const;
  ...
};
```

konstruktory a destruktor kontejneru `deque`

```
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;
```

iterátor ukazující na první prvek, resp. za poslední prvek, kontejneru

```
reverse_iterator       begin();
```

```
const_reverse_iterator begin() const;
reverse_iterator       end();
const_reverse_iterator end() const;
```

iterátory pro zpětný průchod kontejnerem — vrací iterátor na poslední prvek kontejneru (tj. první prvek obráceného sledu), resp. iterátor za poslední prvek sledu při průchodu v obráceném pořadí (reverse iterator)

```
size_type size() const;
```

počet prvků uložených v kontejneru

```
size_type max_size() const;
```

vrací maximální počet prvků, které mohou být v kontejneru uloženy

```
void resize(size_type sz, T c = T());
```

realokace paměti kontejneru (zneplatní všechny iterátory)

```
bool empty() const;
```

vrací hodnotu `true`, pokud je kontejner prázdný

```
reference       operator[](size_type n);
const_reference operator[](size_type n) const;
reference       at(size_type n);
const_reference at(size_type n) const;
```

vrací referenci na n-tý prvek kontejneru typu `deque`

```
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;
```

reference na první, resp. poslední, prvek

```
void push_front(const T& x);
void push_back(const T& x);
```

vloží prvek `x` na začátek, resp. na konec, kontejneru typu `deque`

```
void pop_front();
void pop_back();
```

zruší první, resp. poslední, prvek kontejneru. Výsledek není definován, je-li `deque` prázdný.

```
iterator insert(iterator position, const T& x);
```

vloží prvek `x` na pozici, na kterou ukazuje iterátor `position`, a posune dříve uložené prvky podle potřeby. Návratová hodnota iterátoru ukazuje na pozici vloženého prvku

```
void insert(iterator position, size_type n, constT& x);
```

vloží n kopií prvku `x`

```
template <typename InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);
```

vloží prvky ze sledu `[first, last)` na pozici zadanou iterátorem `position`

```
iterator erase(iterator position);
```

zruší prvek na který ukazuje iterátor `position`

```
void erase(iterator first, iterator last);
```

zruší všechny prvky z intervalu `[first, last)`

```
void swap(deque<T, Allocator>& x);
```

vymění obsah běžného kontejneru `deque<T>` za kontejner `x`.

```
void clear();
```

zruší všechny prvky kontejneru

```
bool operator==(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
bool operator< (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
bool operator!=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
bool operator> (const deque<T,Allocator>& x, const deque<T,Allocator>& y);
bool operator>=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
bool operator<=(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
```

relační operátory pro instance typu `deque`

```
void swap(const deque<T,Allocator>& x, const deque<T,Allocator>& y);
```

zamění obsah kontejnerů `x` a `y`

10.5.3 Kontejner `list`

Sekvenční kontejner `list` je obousměrný spojový seznam. Pro iterátor ukazující na určitý prvek v kontejneru můžeme tedy vždy určit následující anebo předchozí prvek. Prvky můžeme vkládat nebo rušit stejně efektivně na kterékoli pozici seznamu. Na rozdíl od sekvenčního kontejneru `vector` ale neexistuje pro `list` náhodný přístup k prvkům (operace indexování). Sekvenční průchod kontejnerem `list` je podstatně pomalejší než průchod kontejnerem typu `vector`. Operace vkládání a rušení prvků jsou ale řádu $\mathcal{O}(1)$ a tedy efektivnější než u kontejneru `vector` (zvláště, jsou-li prováděny hromadně).

Zrušení nebo přidání prvku nezneplatní ostatní iterátory.

Metodu kontejneru `list`

```
template <typename T, typename Allocator = allocator<T> >
class list {
public:
  typedef typename Allocator::reference            reference;
  typedef typename Allocator::const_reference      const_reference;
  typedef implementation defined                   iterator;
  typedef implementation defined                   const_iterator;
  typedef implementation defined                   size_type;
  typedef implementation defined                   difference_type;
  typedef T                                        value_type;
  typedef Allocator                                allocator_type;
  typedef typename Allocator::pointer              pointer;
  typedef typename Allocator::const_pointer        const_pointer;
  typedef std::reverse_iterator<iterator>          reverse_iterator;
```

```
  typedef std::reverse_iterator<const_iterator>     const_reverse_iterator;

  explicit list(const Allocator& = Allocator());
  explicit list(size_type n, const T& value = T(),
                const Allocator& = Allocator());
  template <typename InputIterator>
    list(InputIterator first, InputIterator last,
         const Allocator& = Allocator());
  list(const list<T,Allocator>& x);
 ~list();
  list<T,Allocator>& operator=(const list<T,Allocator>& x);
  template <typename InputIterator>
    void assign(InputIterator first, InputIterator last);
  template <typename Size, typename T>
    void assign(Size n, const T& t = T());
  allocator_type get_allocator() const;
  ...
};
```

konstruktory, kopírování a rušení kontejneru `list`

```
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;
```

mezní iterátory kontejneru `list`

```
bool      empty() const;
size_type size() const;
size_type max_size() const;
void      resize(size_type as, T c = T());
```

informace o kapacitě a rozšíření kontejneru `list`

```
reference       front();
const_reference front() const;
reference       back();
const_reference back() const;
```

přístup k prvnímu, resp. poslednímu, prvku kontejneru `list`

```
void push_front(const T& x);
void pop_front();
void push_back(const T& x);
void pop_back();
```

vložení, resp. zrušení, prvku na začátku, resp. na konci, kontejneru `list`

```
iterator insert(iterator position, const T& x);
void     insert(iterator position, size_type n, const T& x)
template <typename InputIterator>
void insert(iterator position, InputIterator first, InputIterator last);
```

vkládání prvků do kontejneru `list`

```
iterator erase(iterator position);
iterator erase(iterator position, iterator last);
void     swap(list<T, Allocator>&);
void     clear();
```

rušení a výměna prvků. Funkce `erase` vrací iterátor na prvek, který se před jejím voláním nacházel za prvkek `position`, resp. za prvkem `last`.

```
void splice(iterator position, list<T, allocator>& x);
```

pokud je `position` platný iterátor, vloží obsah kontejneru `x` před prvek `position`, po provedeni operace je kontejner `x` prázdný. Kontejner `x` nesmí být totožný s instancí, pro kterou voláme funkci `splice`.

```
void splice(iterator position, list<T, allocator>& x, iterator i);
```

pokud je `position` platný iterátor a `i` je platný iterátor seznamu `x`, vyjme ze seznamu `x` prvek na který ukazuje iterator `i` a vloží jej před prvek `position`. Kontejner `x` může být totožný s instancí pro kterou voláme funkci `splice`.

```
void splice(iterator position, list<T, allocator>& x, iterator first, iterator last);
```

pokud je `position` platný iterátor a pokud jsou `[first, last)` platný interval kontejneru `x`, vyjme prvky `[first, last)` z kontejneru `x` a vloží je před prvek `position`. Kontejner `x` může být totožný s instancí pro kterou voláme funkci `splice`.

```
void remove(const T& value);
template <typename Predicate> void remove_if(Predicate pred);
```

odstraní ze seznamu prvky, které jsou rovny `value`, resp. prvky, které vyhovují zadané podmínce.

```
void unique();
template <typename BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

odstraní ze seznamu po sobě jdoucí duplicitní prvky (tj. skupinu duplicitních prvků nahradí jejich jediným reprezentantem), resp. odstraní duplicitní prvky podle zadané podmínky.

```
void merge(list<T, Allocator>& x);
template <typename Compare> void merge(List<T, Allocator>& x, Compare comp);
```

spojí dva setříděné seznamy

```
void sort();
template <typename Compare> void sort(Compare comp);
```

setřídí prvky kontejneru `list`

```
void reverse();
```

obrátí pořadí prvků v seznamu `list`

```
template <typename T, typename Allocator>
void swap(list<T, Allocator>& x, list<T, Allocator>& y);
```

vymění obsah kontejnerů `x` a `y`

Pro kontejner typu `list` jsou definovány relační operátory `==`, `<` , `=!`, `>` , `>=` a `<=`.

10.5.4 Adaptor `stack`

Adaptory jsou komponenty standardní knihovny, které modifikují rozhraní jiných komponent (kontejnerů, iterátorů a funkcí). Příkladem takového adaptoru je kontejner *stack (zásobník)*.

Zásobník je jednoduchá datová struktura, která umožňuje pouze přidávat a odebírat prvky na konci seznamu. Poslední prvek, který jsme do zásobníku vložili, je vždy prvním prvkem, který ze zásobníku odebíráme.

Pro tento účel bychom mohli použít libovolný sekvenční kontejner. Zavedením restriktivního rozhraní adaptoru `stack` ale říkáme, jaké operace jsou přípustné, aniž bychom specifikovali jak budou implementovány. `Stack` je definován jako šablona, která obsahuje kontejner, jehož typ je jejím parametrem. Zásobník je do té míry jednoduchým kontejnerem, že pro jeho popis postačí uvést C++ kód (viz [2]).

```
template <typename T, typename Container = deque<T> >
class stack {
public:
  typedef typename Container::value_type   value_type;
  typedef typename Container::size_type    size_type;
  typedef          Container               container_type;
protected:
  Container c;

public:
  explicit stack(const Container& = Container());

  bool      empty() const         { return c.empty(); }
  size_type size()  const         { return c.size();  }
  value_type&       top()         { return c.back();  }
  const value_type& top()         { return c.back();  }
  void push(const value_type& x)  { c.push_back(x);   }
  void popo()                     { c.pop_back();     }
};
```

Pro adaptor `stack` jsou tak jako pro ostatní standardní kontejnery definovány všechny relační operátory.

Zásobník můžeme vytvořit s využitím kteréhokoliv standardního sekvenčního kontejneru, tj. kontejneru `vector`, `deque` nebo `list`. Implicitně se použije kontejner `deque`.

Ukázku použití kontejneru `stack` uvádíme v příkladu věnovaném kontejnerům `set` a `multiset` v odstavci 10.5.9.

10.5.5 Adaptor `queue`

Kontejner `queue` realizuje datovou strukturu, které česky říkáme *fronta*. Do fronty vkládáme prvky na jejím konci a odebíráme je na začátku fronty.

Fronta je stejně jako standardní zásobník adaptor a definuje pouze rozhraní pro použitý kontejner. Pro popis opět plně postačí zdrojový kód šablony `queue`.

```
template <typename T, typename Container = deque<T> >
class queue {
```

```
public:
  typedef typename Container::value_type   value_type;
  typedef typename Container::size_type    size_type;
  typedef          Container               container_type;
protected:
  Container c;

public:
  explicit queue(const Container& = Container());

  bool      empty() const         { return c.empty(); }
  size_type size()  const         { return c.size();  }
  value_type&       front()       { return c.front(); }
  const value_type& front() const { return c.front(); }
  value_type&       back()        { return c.back();  }
  const value_type& back()        { return c.back();  }
  void push(const value_type& x)  { c.push_back(x);   }
  void pop()                      { c.pop_front();    }
};
```

Frontu (queue) můžeme vytvořit s využitím standardních sekvenčních kontejnerů deque nebo list. Implicitně se použije kontejner deque. Kontejner vector nemůžeme použít, protože nemá operace pro přidávání, resp. rušení, prvku na začátku posloupnosti.

10.5.6 Adaptor priority_queue

Prioritní fronta (priority queue) na rozdíl od adaptoru queue z fronty vybírá vždy maximální prvek:

```
#include <iostream>
#include <queue>

int main()
{
  std::priority_queue<int>  pq;

  for (int i=0; i<5; i++) { pq.push(i); pq.push(2*i); pq.push(i+100); }

  while (!pq.empty())
    {
      std::cout << pq.top() << " ";
      pq.pop();
    }

  std::cout << "\n";
}
```

```
104 103 102 101 100 8 6 4 4 3 2 2 1 0 0
```

Použití metod adaptoru priority_queue je pět zřejmé ze zdrojového kódu. Třetím volitelným parametrem šablony je *funkční objekt* Compare, který porovnává dva objekty typu T (o funkčních objektech jsme hovořili v odst. 5.10.3).

```
template <typename T, typename Container = vector<T>,
          typename Compare = less<typename Container::value_type> >
class priority_queue {
public:
  typedef typename Container::value_type   value_type;
  typedef typename Container::size_type    size_type
  typedef          Container               container_type;
protected:
  Container c;
  Compare comp;

public:
  explicit priority_queue(const Compare& x = Compare(),
                          const Container& = Container());
  template<typename InputIterator>
    priority_queue(InputIterator first, InputIterator last,
                   const Compare& x = Compare(),
                   const Container& = Container());

  bool empty() const            { return c.empty(); }
  size_type size() const        { return c.size();  }
  const value_type& top() const { return c.front(); }
  void push(const value_type& x);
  void pop();

};
```

10.5.7 Kontejner `map`

Kontejner `map` je typ asociativního kontejneru, který podporuje jednoznačné klíče (nemohou se vyskytnout dva prvky se stejným klíčem), a který poskytuje rychlý přístup k hodnotám jiného typu `T` založený na těchto klíčích. Kontejner `map` podporuje obousměrné iterátory.

Kontejnery typu `map` jsou někdy také označovány jako slovníky nebo asociativní pole. Tento typ kontejneru tvoří dvojice *(pair)* klíč a hodnota T. Prvky kontejneru jsou setříděny podle klíčů. Klíče přitom můžeme používat obdobně jako indexy jednorozměrných polí, jak ukazuje následující příklad triviálního seznamu bodů.

```
#include <iostream>
#include <map>

class bod {
  double _y, _x;
public:
  bod() {}
  bod(double y, double x) : _y(y), _x(x) {}
  double& y() { return _y; }
  double& x() { return _x; }
  // ...
};
```

```
typedef std::map<long, bod> SezBod;

int main()
{
  SezBod S;

  S[8456] = bod(5437.45, 7345.33);
  S[5366] = bod(5492.45, 7980.33);
  S[7438] = bod(4935.92, 7903.21);
  S[3734] = bod(5046.01, 6994.29);

  const int S_poc = S.size();
  long* S_cb = new long[S_poc];
  int n = 0;

  for (SezBod::iterator i=S.begin(); (i != S.end()); ++i) {
    S_cb[n++] = (*i).first;
    std::cout << (*i).first << "   "
              << (*i).second.y() << "  " << (*i).second.x() << std::endl;
  }

  std::cout << std::endl;
  for (int i=0; i<S_poc; i++) {
    long cb = S_cb[i];
    std::cout << cb << "  "
              << S[cb].y() << "  " << S[cb].x() << std::endl;
  }
}
```

Do kontejneru jsme vložili čtyři body, klíčem byly hodnoty typu `long`. Zápis na první pohled připomíná ukládání prvků do pole s celočíselnými indexy, odtud pochází označení asociativní pole, které se pro tento typ kontejneru někdy používá. Klíčem v kontejneru map ale může být libovolný typ, pro který je definovaná relace *je menší*.

S hodnotou klíče je v kontejneru `map` asociována hodnota typu `T`, v tomto příkladu `bod`. Jednotlivé prvky mají proto dvě části `first` a `second` (česky *první* a *druhá*). Pomocí nich musíme při práci s iterátory specifikovat, se kterou složkou chceme pracovat, jak ukazuje cyklus pro tisk bodů v naší ukázce. Druhý cyklus tiskne stejné hodnoty, v tomto případě ale k bodům přistupujeme přímo přes klíče (specifikace `second` se neuvádí).

Jiný příklad užití kontejneru `map` uvádí následující program, který počítá četnosti slov ve vstupním textu. Klíčem je zde typ `string`, ukládané hodnoty jsou četnosti nalezených slov.

```
#include <iostream>
#include <string>
#include <map>

int main ()
{
   typedef std::map <std::string, int> Map;
   Map M;

   std::string s;
```

```
        while (std::cin >> s)  ++M[s];

        for (Map::iterator i=M.begin(); i != M.end(); ++i)
           std::cout << (*i).first << " " << (*i).second << std::endl;
    }
```

Metody kontejneru `map`

Kontejner `map` poskytuje rychlý přístup k uloženým prvkům podle klíčů, pro které je definována relace uspořádání, příkladem mohou být klíče typu `std::string`. Otázka pochopitelně je, *co je to rychlý přístup?* Typická implementace kontejneru `map` využívá techniky vyvážených stromů a pro běžné aplikace je přístup vyžadující $\mathcal{O}(\log N)$ porovnání pro vyhledání prvku vyhovující.

Sekvenční průchod kontejnerem `map` je o něco pomalejší v porovnání se sekvenčním kontejnerem `list`. Iterátory kontejneru `map` neukazují přímo na prvky ale na standardní typ `pair`, jehož první složka (`first`) je klíč a druhá složka (`second`) je vlastní hodnota uloženého prvku.

Zrušení nebo vložení prvku neovlivní ostatní iterátory.

```
template <typename Key, typename T, typename Compare = less<Key>,
          typename Allocator = allocator<T> >
class map {
public:
  typedef Key                                       key_type;
  typedef T                                         mapped_type;
  typedef pair<const Key, T>                        value_type;
  typedef Compare                                   key_compare;
  typedef Allocator                                 allocator_type;
  typedef typename Allocator::reference             reference;
  typedef typename Allocator::const_reference       const_reference;
  typedef implementation defined                    iterator;
  typedef implementation defined                    const_iterator;
  typedef implementation defined                    size_type;
  typedef implementation defined                    difference_type;
  typedef Allocator::pointer                        pointer;
  typedef Allocator::const_pointer                  const_pointer;
  typedef std::reverse_iterator<iterator>           reverse_iterator;
  typedef std::reverse_iterator<const_iterator>     const_reverse_iterator;

  class value_compare
    : public binary_function<value_type,value_type,bool> {
  friend class map;
  protected:
    Compare comp;
    value_compare(Compare c) : comp(c)
  public:
    bool operator()(const value_type& x, const value_type& y) const {
      return comp(x.first, y.first);
    }
  };

  explicit map(const Compare& comp = Compare(), const Allocator& = Allocator());
```

```
  template <typename InputIterator>
    map(InputIterator first, InputIterator last,
        const Compare& comp = Compare(), const Allocator& = Allocator());
  map(const map<Key,T,Compare,Allocator>& x);
 ~map();
  map<Key,T,Compare,Allocator>&
    operator=(const map<Key,T,Compare,Allocator>& x);
  ...
};
```

konstruktory, kopírování, destruktor a typy kontejneru `map`

```
iterator               begin();
const_iterator         begin() const;
iterator               end();
const_iterator         end() const;
reverse_iterator       rbegin();
const_reverse_iterator rbegin() const;
reverse_iterator       rend();
const_reverse_iterator rend() const;
```

iterátory kontejneru `map`

```
bool empty() const;
```

vrací hodnotu `true`, pokud je kontejner prázdný, jinak `false`

```
size_type size() const;
```

vrací počet prvků uložených v kontejneru.

```
size_type max_size() const;
```

vrací maximální možnou velikost kontejneru.

```
reference operator[](const key_type& x);
```

vrací referenci na typ `T` hodnoty asociované s klíčem `x`. Pokud kontejner typu `map` neobsahuje prvek s klíčem `x`, je vytvořena dvojice (x, T()) a vložena do kontejneru. Konkrétně v případě standardních typů to znamená, že hodnota prvku `x` je implicitně vynulována.

Operátor indexování kontejneru `map` má jiný význam než operátor indexování v sekvenčních kontejnerech `vector` a `deque`.

Konstatntní verze operátoru indexování `([])` pro kontejner `map` není definována.

```
pair<iterator, bool> insert(const value_type& x);
```

vloží prvek `x` do kontejneru, pokud již nebyl dříve uložen jiný prvek se stejným klíčem. Vrací dvojici, kde první člen je iterátor ukazující na vložený prvek nebo na dříve uložený prvek, druhý člen má hodnotu `true` pokud byl vložen do kontejneru prvek `x`.

```
iterator insert(iterator position, const value_type& x);
```

vloží hodnotu `x` do kontejneru, pokud žádný prvek nemá stejný klíč. Pro kontejner `map` je `value_type&` dvojice `pair<const key, T>`. Iterátor `position` je nápověda, kde má začít hledání případného duplicitního klíče. Čas pro uložení `x` bude mininimalizován, pokud prvek `x` je uložen bezprostředně za pozicí, na kterou ukazuje iterátor `position`. Vrací iterátor ukazující na vložený prvek nebo na dříve uložený prvek.

```
template <typename InputIterator>
void insert(InputIterator first, InputIterator last);
```

kopíruje prvky ze sledu `[first, last)` do kontejneru `map`

```
void erase(iterator position);
```

zruší prvek kontejneru, na který ukazuje iterátor.

```
size_type erase(const key_type& x);
```

zruší prvek se zadaným klíčem a vrátí počet zrušených prvků, tj. 1, pokud v kontejneru byl uložen prvek s klíčem `x`, 0 v opačném případě.

```
void erase(iterator first, iterator last);
```

zruší prvky z intervalu `[first, last)` v kontejneru `map`

```
void swap(map<Key,T,Compare,Allocator>&);
```

vymění obsah běžného kontejneru `map` a kontejneru `x`.

```
void clear();
```

zruší všechny prvky kontejneru

```
key_compare key_comp() const;
```

vrací objekt `comp` kontejneru `map`

```
value_compare value_comp() const;
```

vrací objekt `value_compare` kontejneru `map`

```
iterator       find(const key_type& x);
const_iterator find(const key_type& x);
```

hledá v kontejneru prvek s klíčem `x`. Vrací hodnotu iterátoru na nalezený prvek nebo hodnotu iterátoru `end()`, pokud prvek s klíčem `x` nebyl nalezen.

```
size_type count(const key_type& x) const;
```

vrací počet prvků s klíčem rovným `x`, tj. v případě kontejneru `map` 1 nebo 0. Pro kontejner `multimap` může být ale počet uložených prvků vyšší než 1.

```
iterator       lower_bound(const key_type& x);
const_iterator lower_bound(const key_type& x);
```

vrací iterátor na první prvek, jehož klíč není menší než `x`. Pokud takový prvek neexistuje, vrací `end()`.

```
iterator       upper_bound(const key_type& x);
const_iterator upper_bound(const key_type& x);
```

vrací iterátor na první prvek, jehož klíč je větší než `x`. Pokud takový prvek neexistuje, vrací `end()`.

```
pair<iterator, iterator>             equal_range(const key_type& x);
pair<const_iterator, const_iterator> equal_range(const key_type& x) const;
```

vrací dvojici `(lower_bound(x), upper_bound(x))`

Pro kontejner `map` jsou dále definovány všechny relační operátory a funkce `swap`.

10.5.8 Kontejner `multimap`

Asociativní kontejner `multimap` je podobný kontejneru `map`, připouští ale vícenásobný výskyt prvků se stejným klíčem. Nemůže proto definovat operátor indexování `[]`, všechny ostatní metody jsou stejné jako u kontejneru `map`.

Následující ukázka naznačuje, jak by bylo možné použít kontejner `multimap` například pro tvorbu jednoduchého rejstříku. Zjednodušená vstupní data zde představují dvojice *heslo* a *číslo stránky:*

```
#include <iostream>
#include <sstream>
#include <map>

int main()
{
  std::istringstream data("string  56  const   66  const   82  const 111 "
                          "queue  197  abs    202  atan   202  queue 204 "
                          "abs    203  string 147  string 151  queue 213 ");

  typedef std::multimap<std::string, int> Rejstrik;
  Rejstrik     rejstrik;
  std::string  heslo;
  int          strana;

  while (data >> heslo >> strana)
    {
      rejstrik.insert(Rejstrik::value_type(heslo, strana));
    }

  for (Rejstrik::const_iterator i=rejstrik.begin(), e=rejstrik.end(); i !=e ; )
    {
      std::cout << (heslo = i->first) << " " << i->second;

      while (++i != e && heslo == i->first) std::cout << " " << i->second;

      std::cout << "\n";
    }
}

abs 202 203
atan 202
const 66 82 111
queue 197 204 213
string 56 147 151
```

10.5.9 Kontejnery `set` a `multiset`

Asociativní kontejner `set` *(množina)* pracuje podobně jako kontejner `map`, ukládá ale pouze jedinečné klíče (a ne dvojice *klíč-hodnota*). Kontejner `multiset` umožňuje ukládání duplicitních klíčů a je obdobou kontejneru `multimap`. Kontejnery `set` a `multiset` poskytují stejnou sadu funkcí jako `multimap`, některé metody jsou ale k dispozici pouze v konstantní verzi, jak uvádí tabulka 10.4.

Použití kontejnerů `set`, `multiset` a `stack` si ukážeme na příkladu jednoduché implementace algoritmu *značkování vrcholů grafu.*[1] Zde jde pouze o ukázku použití kontejnerů, algoritmy pro práci s grafy v C++ najdete v knihovně [4].

Naším úkolem je rozhodnout, zda je zadaný neorientovaný graf souvislý nebo ne (například v oboru geodézie umožňuje tento algoritmus ověřit souvislost geodetické sítě). Vrcholy grafu jsou popsány přirozenými čísly, hrany dvojicemi čísel (spojnice dvou vrcholů). Protože v naší ukázce předpokládáme neorientovaný graf, pořadí v jakém jsou uvedena čísla vrcholů dané hrany není významné.

Program nejprve načte všechny `hrany` a do prázdného seznamu vrcholů vloží jeden z vrcholů první uložené hrany. Vlastní algoritmus spočívá v opakovaném průchodu seznamem hran a postupném budování souvislé množiny vrcholů. Pokud je nalezena hrana, jejíž alespoň jeden vrchol je v množině `vrcholy`, přidá funkcí `insert` oba vrcholy do souvislé komponenty a uloží iterátor hrany do zásobníku.

Pokud je po průchodu množinou hran `zasobnik` prázdný, nebyla nalezena žádná hrana rozšiřující množinu `vrcholy`, znamená to, že graf není souvislý a algoritmus končí. V opačném případě jsou zrušeny všechny hrany registrované v zásobníku.

Pokud jsou z kontejneru `hrany` typu `multiset` úspěšně odstraněny všechny prvky, je graf souvislý a zpracování končí.

```
#include <iostream>
#include <sstream>
#include <stack>
#include <set>

typedef std::set<int>                       Vrcholy;
typedef std::multiset<std::pair<int,int> >  Hrany;
typedef std::stack<Hrany::iterator>         Zasobnik;

void nacti(Hrany& h)      // izolovana komponenta: 3-15-18
{
  std::istringstream data("12  5   13 16   16  4    7  8   14  6 "
                          "15 18   15  3    3 18   18  3    5 12 "
                          "14  6   11  6    8  5   12 10   13  7 "
                          "14 11    2 17    9  4   14  9    7 10 "
                          " 7  5    2 10   17  1    9 16    1 12 ");
  int a, b;
  while (data >> a >> b) h.insert(Hrany::value_type(a,b));
}

int main()
{
  Hrany     hrany;
  Zasobnik  zasobnik;
  Vrcholy   vrcholy;

  nacti(hrany);
  vrcholy.insert( hrany.begin()->first );

  while (!hrany.empty())
```

[1] Jiří Demel, Grafy a jejich aplikace, Academia 2002

```
        {
          for (Hrany::iterator i=hrany.begin(), e=hrany.end(); i!=e; ++i)
            {
              int a = i->first;
              int b = i->second;

              if (vrcholy.find(a) != vrcholy.end() ||
                  vrcholy.find(b) != vrcholy.end() )
                {
                  vrcholy.insert(a);
                  vrcholy.insert(b);
                  zasobnik.push(i);
                }
            }

          if (zasobnik.empty())
            {
              std::cout << "graf neni souvisly\n";
              return 0;
            }

          while(!zasobnik.empty())
            {
              hrany.erase(zasobnik.top());
              zasobnik.pop();
            }
        }

      std::cout << "graf je souvisly\n";
    }

    graf neni souvisly
```

10.6 Algoritmy

10.6.1 Operace nemodifikující posloupnost

for_each

```
template<typename InputIterator, typename Function>  Function
for_each(InputIterator first, InputIterator last, Function f);
```

Aplikuje funkci `f` na všechny prvky zadané posloupnosti.

find

```
template <typename InputIterator, typename T>  InputIterator
find(InputIterator first, InputIterator last, costn T& value);

template<typename InputIterator, typename Predicate> InputIterator
find_if(InputIterator first, InputIterator last, Predicate pred);
```

kontejnery vector, deque, list, stack, queue, prioritu_queue:			v	d	l	s	q	p
iterator	begin(),	end()	●	●	●			
const_iterator	begin() const,	end() const	●	●	●			
reverse_iterator	rbegin(),	rend()	●	●	●			
const_reverse_iterator	rbegin() const,	rend() const	●	●	●			
bool	empty() const		●	●	●	●	●	●
size_type	size() const		●	●	●	●	●	●
size_type	max_size() const		●	●	●			
void	resize(size_type, T = T())		●	●	●			
size_type	capacity() const		●	●				
void	reserve(size_type)		●	●				
reference	operator[](size_type)		●	●				
const_reference	operator[](size_type) const		●	●				
reference	at(size_type)		●	●				
const_reference	at(size_type) const		●	●				
reference	front()		●	●	●		●	
const_reference	front() const		●	●	●		●	
reference	back()		●	●	●		●	
const_reference	back() const		●	●	●		●	
reference	top()					●		●
const_reference	top() const					●		●
void	push(const value_type&)					●	●	●
void	pop()					●	●	●
void	push_front(const T&)			●	●			
void	pop_front()			●	●			
void	push_back(const T&)		●	●	●			
void	pop_back()		●	●	●			
iterator	insert(iterator, const T&)		●	●	●			
void	insert(iterator, size_type, const T&)		●	●	●			
void	insert(InputIterator, InputIterator)		●	●	●			
iterator	erase(iterator)		●	●	●			
iterator	erase(iterator, iterator)		●	●	●			
void	swap(Container<T,Allocator>&)		●	●	○			
void	clear()		●	●	●			
void	splice(iterator,...)				○			
void	remove(const T& value)				●			
void	remove_if(Predicate pred)				●			
void	unique()				●			
void	merge(list<T>&, Compare comp)				●			
void	sort(Compare comp)				●			
void	reverse()				●			

Tabulka 10.3: Přehled sekvenčních kontejnerů a jejich metod

			map	multimap	set	multiset
iterator	begin(),	end()	•	•	•	•
const_iterator	begin() const,	end() const	•	•	•	•
reverse_iterator	rbegin(),	rend()	•	•	•	•
const_reverse_iterator	rbegin() const,	rend() const	•	•	•	•
bool	empty() const		•	•	•	•
size_type	size() const		•	•	•	•
size_type	max_size() const		•	•	•	•
T&	operator[](const key_type&)		•			
pair<iterator,bool>	insert(const value_type&)		•	•	•	•
iterator	insert(iterator, const value_type&)		•	•	•	•
void	insert(InputIterator, InputIterator)		•	•	•	•
void	erase(iterator)		•	•	•	•
size_type	erase(const key_type&)		•	•	•	•
void	erase(iterator, iterator)		•	•	•	•
void	swap(Container<T,Allocator>&)		•	•	•	•
void	clear()		•	•	•	•
key_compare	key_comp() const		•	•	•	•
value_compare	value_comp() const		•	•	•	•
iterator	find(const key_type&)		•	•		
const_iterator	find(const key_type&) const		•	•	•	•
size_type	count(const key_type&) const		•	•	•	•
iterator	lower_bound(const key_type&)		•	•		
const_iterator	lower_bound(const key_type&) const		•	•	•	•
iterator	upper_bound(const key_type&)		•	•		
const_iterator	upper_bound(const key_type&) const		•	•	•	•
pair<iterator, iterator>	equal_range(const key_type&)		•	•		
pair<iterator, iterator>	equal_range(const key_type&) const		•	•	•	•

Tabulka 10.4: Přehled asociativních kontejnerů a jejich metod

Vrací první iterátor `i` ze zadané posloupnosti, pro který platí `*i == value`, resp. `pred(*i) != false`.

find_end

```
template<typename ForwardIterator1, typename ForwardIterator2> ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
         ForwardIterator2 first2, ForwardIterator2 last2);

template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryPredicate>  ForwardIterator1
find_end(ForwardIterator1 first1, ForwardIterator1 last1,
```

```
        ForwardIterator2 first2, ForwardIterator2 last2,
        BinaryPredicate pred);
```

V zadaném sledu hledá sekvenci stejných hodnot.

find_first

```
template<typename InputIterator, typename ForwardIterator> InputIterator
find_first_of(InputIterator   first1, InputIterator   last1,
              ForwardIterator first2, ForwardIterator last2)

template<typename InputIterator, typename ForwardIterator> InputIterator
find_first_of(InputIterator   first1, InputIterator   last1,
              ForwardIterator first2, ForwardIterator last2,
              BinaryPredicate comp)
```

Hledá prvek, který se shoduje s některým prvkem ze zadané množiny.

adjacent_first

```
template<typename ForwardIterator>  ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last);

template<typename ForwardIterator, typename BinaryPredicate>  ForwardIterator
adjacent_find(ForwardIterator first, ForwardIterator last,
              BinaryPredicate pred);
```

Najde iterátor `i`, pro který platí `*i == *(i+1)`, resp. `pred((*i), *(i+1)) != false`.

count

```
template<typename InputIterator, typename T>
        iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& value);

template<typename InputIterator, typename Predicate>
        iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Počet prvků ze zadaného sledu, pro které platí `*i == value`, resp. `pred(*i) != false`.

mismatch

```
template<typename InputIterator1, typename InputIterator2>
        pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2);

template<typename InputIterator1, typename InputIterator2, typename BinaryPredicate>
        pair<InputIterator1, InputIterator2>
mismatch(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, BinaryPredicate pred);
```

Porovnává dva sledy a vrací iterátory prvních dvou různých prvků, resp. prvků, pro které je hodnota binární podmínky `pred() == false`

equal

```
template<typename InputIterator1, typename InputIterator2>  bool
equal(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2);

template<typename InputIterator1, typename InputIterator2,
         typename BinaryPredicate>  bool
equal(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, BinaryPredicate pred);
```

Vrací `true` pokud oba sledy obsahují stejné prvky, resp. pokud je binární podmínka `pred` platná postupně pro všechny dvojice prvků.

search

```
template<typename ForwardIterator1, typename ForwardIterator2>
         ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2);

template<typename ForwardIterator1, typename ForwardIterator2,
         typename BinaryPredicate>  ForwardIterator1
search(ForwardIterator1 first1, ForwardIterator1 last1,
       ForwardIterator2 first2, ForwardIterator2 last2,
       BinaryPredicate pred);
```

Hledá souvislý sled stejných prvků v zadané posloupnosti, resp. souvislý sled prvků vyhovujících dané podmínce.

```
template<typename ForwardIterator, typename Size, typename T>
         ForwardIterator
search_n(ForwardIterator first, ForwardIterator last,
         Size count, const T& value);

template<typename ForwardIterator, typename Size, typename T,
         typename BinaryPredicate>   ForwardIterator1
search_n(ForwardIterator first, ForwardIterator last,
         Size count, const T& value, BinaryPredicate pred);
```

Hledá souvislý sled stejných prvků v posloupnosti `[first, last-count)`, resp. souvislý sled vyhovujících dané podmínce.

10.6.2 Operace modifikující posloupnost

copy

```
template<typename InputIterator, typename OutputIterator>
         OutputIterator
copy(InputIterator first, InputIterator last, OutputIterator result);
```

Kopíruje prvky ze zadané posloupnosti do sledu [result, result + (last - first)).

```
template<typename BidirectionalIterator1, typename BidirectionalIterator2>
        BidirectionalIterator2
copy_backward(BidirectionalIterator1 first, BidirectionalIterator1 last,
              BidirectionalIterator2 result);
```

Kopíruje v obráceném pořadí prvky ze zadaného *sledu (range)* [first, last) do sledu [result - (last - first), result).

swap

```
template<typename T> void
swap(T& x, T& y);
```

Vymění obsah x a y.

```
template<typename ForwardIterator1, typename ForwardIterator2>
         ForwardIterator2
swap_ranges(ForwardIterator1 first1, ForwardIterator1 last1,
            ForwardIterator2 first2);
```

Vymění obsah dvou posloupností. Pro všechna nezáporná n pro která platí n < last1-first1 provede swap(*(first1 + n), *(first2 + n)).

```
template<typename ForwardIterator1, typename ForwardIterator2>  void
iter_swap(ForwardIterator1 x, ForwardIterator2 y);
```

Vymění obsah dvou prvků, na které ukazují iterátory x a y.

transform

```
template<typename InputIterator, typename OutputIterator, typename UnaryOperation>
         OutputIterator
transform(InputIterator first, InputIterator last,
          OutputIterator result, UnaryOperation op);

template<typename InputIterator1, typename InputIterator2, typename OutputIterator,
         typename BinaryOperation>  OutputIterator
transform(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, OutputIterator result,
          BinaryOperation binary_op);
```

Pro všechny iterátory i z rozsahu [result, result + (last1 - first1)) přiřadí novou hodnotu op(*(first1 + (i - result))), resp. binary_op(*(first1 + (i - result), *(first2 + (i - result)))).

replace

```
template<typename ForwardIterator, typename T> void
replace(ForwardIterator first, ForwardIterator last,
        const T& old_value, const T& new_value);
```

```
template<typename ForwardIterator, typename Predicate, typename T>  void
replace_if(ForwardIterator first, ForwardIterator last,
           Predicate pred, const T& new_value);
```

Nahradí v intervalu `[first, last)` novou hodnotou `new_value` ty prvky, jejichž původní hodnota je rovna `old_value`, resp. pokud platí podmínka `pred(*i) != false`.

```
template<typename InputIterator, typename OutputIterator, typename T>
         OutputIterator
replace_copy(InputIterator first, InputIterator last,
             OutputIterator result,
             const T& old_value, const T& new_value);

template<typename Iterator, typename OutputIterator, typename Predicate, typename T>
         OutputIterator
replace_copy_if(Iterator first, Iterator last,
                OutputIterator result,
                Predicate pred, const T& new_value);
```

Pro všechny iterátory `i` ze sledu `[result, result + (last - first))` přiřadí `new_value` nebo `*(first + (i - result))`, pokud platí `*( first + (i - result)) == old_value`, resp. `pred(*(first + (i - result)) != false)`.

fill

```
template<typename ForwardIterator, typename T>  void
fill(ForwardIterator first, ForwardIterator last, const T& value);

template<typename OutputIterator, typename Size, typename T>  void
fill_n(OutputIterator first, Size n, const T& value);
```

Přiřadí hodnotu `value` všem prvkům ze sledu `[first, last)`, resp. ze sledu `[first, first + n)`.

generate

```
template<typename ForwardIterator, typename Generator>  void
generate(ForwardIterator first, ForwardIterator last, Generator gen);

template<typename OutputIterator, typename Size, typename Generator>  void
generate_n(OutputIterator first, Size n, Generator gen);
```

Volá funkční objekt `gen` a přiřadí jeho hodnotu všem prvkům posloupnosti `[first, last)`, resp. `[first, first + n)`.

remove

```
template<typename ForwardIterator, typename T>  ForwardIterator
remove(ForwardIterator first, ForwardIterator last,  const T& value);

template<typename ForwardIterator, typename Predicate>  ForwardIterator
remove_if(ForwardIterator first, ForwardIterator last,  Predicate pred);
```

Odstraní všechny prvky ze sledu `[first, last)`, pro které platí `*i == value`, resp. je splněna podmínka `pred(*i) != false`.

```
template<typename InputIterator, typename OutputIterator, typename T>
         OutputIterator
remove_copy(InputIterator first, InputIterator last,
            OutputIterator result, const T& value);

template<typename InputIterator, typename OutputIterator, typename Predicate>
         OutputIterator
remove_copy_if(InputIterator first, InputIterator last,
               OutputIterator result, Predicate pred);
```

Všechny prvky ze sledu `[first, last)`, pro které neplatí `*i == value`, resp. neplatí podmínka `pred(*i) != false`, kopíruje do sledu, na který ukazuje iterátor `result`.

unique

```
template<typename ForwardIterator>  ForwardIterator
unique(ForwardIterator first, ForwardIterator last);

template<typename ForwardIterator, typename BinaryPredicate>  ForwardIterator
unique(ForwardIterator first, ForwardIterator last, BinaryPredicate pred);
```

Z každé souvislé skupiny shodných prvků ponechá pouze první prvek, resp. odstraní prvky, které jsou identické podle podmínky `pred(*i, *(i - 1) != false`.

```
template<typename InputIterator, typename OutputIterator>  OutputIterator
unique_copy(InputIterator first, InputIterator last, OutputIterator result);

template<typename InputIterator, typename OutputIterator,
         typename BinaryPredicate> OutputIterator
unique_copy(InputIterator first, InputIterator last,
            OutputIterator result, BinaryPredicate pred);
```

Kopíruje první prvek z každé souvislé skupiny shodných prvků, resp. prvků, pro které platí podmínka `pred(*i, *(i - 1) != false`.

reverse

```
template<typename BidirectionalIterator>  void
reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Změní pořadí prvků v zadané neprázdné posloupnosti.

```
template<typename BidirectionalIterator, typename OutputIterator>
         OutputIterator
reverse_copy(BidirectionalIterator first, BidirectionalIterator last,
             OutputIterator result);
```

V obráceném pořadí nakopíruje zadaný sled `[first, last)` do sledu `[result, result + (last - first))`.

rotate

```
template<typename ForwardIterator>  void
rotate(ForwardIterator first, ForwardIterator middle, ForwardIterator last);
```

Rotace směrem vlevo. Pro každé nezáporné číslo `n < (last - first)` umístí prvek z pozice `from + i` na pozici `first + (i + (last - middle)) % (last - first)`.

```
template<typename ForwardIterator, typename OutputIterator>
         OutputIterator
rotate_copy(ForwardIterator first, ForwardIterator middle,
            ForwardIterator last, OutputIterator result);
```

Kopíruje prvky s rotací stejnou jako v předchozím případě do výstupní posloupnosti `[result, result + (last - first))`.

random_shuffle

```
template<typename RandomAccessIterator>  void
random_shuffle(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator,
         typename RandomNumberGenerator>   void
random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
               RandomNumberGenerator& rand);
```

Podle rovnoměrného rozdělení *promíchá* náhodně prvky sledu `[first, last)`. Generátor náhodných čísel může být zadán jako parametr funkce.

partition

```
template<typename BidirectionalIterator, typename Predicate>
         BidirectionalIterator
partition(BidirectionalIterator first, BidirectionalIterator last,
          Predicate pred);
```

Uspořádá posloupnost `[first, last)` tak, že prvky pro které je splněna podmínka `pred` umístí před ostatní prvky.

```
template<typename BidirectionalIterator, typename Predicate>
         BidirectionalIterator
stable_partition(BidirectionalIterator first, BidirectionalIterator last,
                 Predicate   pred);
```

Podobně jako v předchozím případě uspořádá posloupnost `[first, last)` tak, že prvky pro které je splněna podmínka `pred`, umístí před ostatní prvky. Relativní uspořádání prvků je přitom zachováno.

10.6.3 Třídění a příbuzné operace

sort

```
template<typename RandomAccessIterator>  void
```

```
sort(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
sort(RandomAccessIterator first, RandomAccessIterator last,
     Compare comp);
```

Setřídí sled `[first, last)`. Průměrný počet porovnání je přitom $N \log N$, kde N == (last - first).

stable_sort

```
template<typename RandomAccessIterator>  void
stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
stable_sort(RandomAccessIterator first, RandomAccessIterator last,
            Compare comp);
```

Stabilní setřídění sledu `[first, last)`, které zachovává relativní pozice prvků se stejným klíčem. Provádí maximálně $N(\log N)^2$ porovnání; je-li k dispozici dostatečná paměť, je počet porovnání pouze $N \log N$.

partial_sort

```
template<typename RandomAccessIterator>  void
partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
             RandomAccessIterator last);
template<typename RandomAccessIterator, typename Compare>  void
partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
             RandomAccessIterator last, Compare comp);
```

Do sledu `[first, middle)` umístí prvky, které by na tomto místě byly při setřídění celého sledu `[first, last)`. Pořadí prvků ve zbytku posloupnosti `[mildle, last)` je nespecifikováno.

partial_sort_copy

```
template<typename InputIterator, typename RandomAccessIterator>
         RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last);

template<typename InputIterator, typename RandomAccessIterator,
         typename Compare>  RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

Kopíruje setříděnou podmnožinu prvků ze sledu `[first, last)` do sledu `[result_first, result_last)`.

nth_element

```
template<typename RandomAccessIterator>  void
nth_element(RandomAccessIterator first, RandomAccessIterator nth,
            RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
nth_element(RandomAccessIterator first, RandomAccessIterator nth,
            RandomAccessIterator last, Compare comp);
```

Setřídí částečně posloupnost `[first, last)` tak. že n-tý prvek (anglicky n-th element) je na svém místě, tj. na stejném místě, na kterém by byl po setřídění celé posloupnosti.

lower_bound

```
template<typename ForwardIterator, typename T>  ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
            const T& value);

template<typename ForwardIterator, typename T, typename Compare>
         ForwardIterator
lower_bound(ForwardIterator first, ForwardIterator last,
            const T& value, Compare comp);
```

Hledá první pozici, na kterou může být vložena hodnota `value`, aniž by došlo k porušení uspořádání sledu.

upper_bound

```
template<typename ForwardIterator, typename T>  ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
            const T& value);

template<typename ForwardIterator, typename T, typename Compare>
         ForwardIterator
upper_bound(ForwardIterator first, ForwardIterator last,
            const T& value, Compare comp)
```

Hledá nejzazší pozici, na kterou lze vložit hodnotu `value`, aniž by došlo k porušení uspořádání sledu.

equal_range

```
template<typename ForwardIterator, typename T>
         pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value);

template<typename ForwardIterator, typename T, typename Compare>
         pair<ForwardIterator, ForwardIterator>
equal_range(ForwardIterator first, ForwardIterator last, const T& value,
            Compare comp);
```

Hledá maximální sled `[i, j)`, do kterého lze vložit hodnotu `value`, aniž by došlo k porušení uspořádání posloupnosti.

binary_search

```
template<typename ForwardIterator, typename T>  bool
binary_search(ForwardIterator first, ForwardIterator last,
              const T& value);

template<typename ForwardIterator, typename T, typename Compare>  bool
binary_search(ForwardIterator first, ForwardIterator last,
              const T& value, Compare comp);
```

Vrací hodnotu `true`, pokud existuje iterátor `i`, pro který platí `!(*i < value) && !(value < *i)` nebo `comp(*i, value) == false && comp(value, *i) == false`.

merge

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>  OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename Compare>  OutputIterator
merge(InputIterator1 first1, InputIterator1 last1,
      InputIterator2 first2, InputIterator2 last2,
      OutputIterator result, Compare comp);
```

Spojí dvě setříděné posloupnosti `[first1, last1)` a `[first2, last2)` do jediné setříděné posloupnosti `[result, result + (last1 - first1) + (last2 - first2))`.

```
template<typename BidirectionalIterator>  void
inplace_merge(BidirectionalIterator first,
              BidirectionalIterator middle,
              BidirectionalIterator last);

template<typename BidirectionalIterator, typename Compare>  void
inplace_merge(BidirectionalIterator first,
              BidirectionalIterator middle,
              BidirectionalIterator last, Compare comp);
```

Spojí dvě setříděné posloupnosti `[first, middle)` a `[middle, last)` do jediné setříděné posloupnost `[first, last)`.

includes

```
template<typename InputIterator1, typename InputIterator2>  bool
includes(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, InputIterator2 last2);
```

```
template<typename InputIterator1, typename InputIterator2,
         typename Compare>  bool
includes(InputIterator1 first1, InputIterator1 last1,
         InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Výsledek je `true`, pokud každý prvek sledu `[first2, last2)` je obsažen ve sledu `[first1, last1)`. Oba sledy musí být předem setříděny.

set_union

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>  OutputIterator
set_union(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename Compare>
         OutputIterator
set_union(InputIterator1 first1, InputIterator1 last1,
          InputIterator2 first2, InputIterator2 last2,
          OutputIterator result, Compare comp)
```

Vytváří setříděné spojení prvků z obou setříděných sledů, tj. vytváří setříděnou množinu prvků, které patří do prvního anebo druhého sledu.

set_intersection

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>  OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename Compare>  OutputIterator
set_intersection(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result, Compare comp);
```

Vytváří setříděný průnik prvků obou setříděných sledů, tj. vytváří setříděnou množinu prvků, které se nacházejí v obou sledech.

set_difference

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator>  OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result);
```

```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename Compare>  OutputIterator
set_difference(InputIterator1 first1, InputIterator1 last1,
               InputIterator2 first2, InputIterator2 last2,
               OutputIterator result, Compare comp);
```

Kopíruje prvky ze setříděného sledu `[first1, last1)`, které se nevyskytují v setříděném sledu `[first2, last2)`, do sledu, který začíná na pozici `result`.

set_symmetric_difference

```
template<typename InputIterator1, typename InputIterator2,
        typename OutputIterator>  OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result);

template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename Compare>  OutputIterator
set_symmetric_difference(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, InputIterator2 last2,
                         OutputIterator result, Compare comp);
```

Kopíruje prvky ze sledu `[first1, last1)`, které nejsou obsaženy ve sledu `[first2, last2)` a prvky ze sledu `[first2, last2)`, které nejsou obsaženy ve sledu `[first1, last1)` do sledu, který začíná na pozici `result`. Oba sledy musí být předem setříděny a výsledná posloupnost je setříděna.

make_heap

```
template<typename RandomAccessIterator>  void
make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
make_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Vytváří ze zadaného sledu `[first, last)` *haldu (heap)*. Halda je takové uspořádání prvků ve sledu `[a, b)`, pro které platí:

1. v daném sledu není žádný prvek větší než `*a`
2. prvek `*a` může být odstraněn funkcí `pop_heap()` anebo nový prvek může být přidán pomocí operace `push_heap()`. Časová náročnost těchto operací je přitom $\mathcal{O}(\log N)$.

Halda představuje užitečnou a efektivní implementaci prioritní fronty. Funkce `make_heap()` převádí zadaný sled iterátorů pro náhodný přístup na haldu.

push_heap

```
template<typename RandomAccessIterator>  void
```

```
push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
push_heap(RandomAccessIterator first, RandomAccessIterator last,
          Compare comp);
```

Sled `[first, last-1)` musí být platná *halda (heap)*. Umístí prvek z pozice `last-1` do výslední haldy `[first, last)`.

pop_heap

```
template<typename RandomAccessIterator>  void
pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Zamění prvky haldy na pozicích `first` a `last-1` a ze sledu `[first, last-1)` vytvoří haldu.

sort_heap

```
template<typename RandomAccessIterator>  void
sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>  void
sort_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Setřídí prvky v haldě `[first, last)`.

min, max

```
template<typename T>
const T& min(const T& x, const T& y);

template<typename T, typename Compare>
const T& min(const T& x, const T& y, Compare comp);

template<typename T>
const T& max(const T& x, const T& y);

template<typename T, typename Compare>
const T& max(const T& x, const T& y, Compare comp);
```

Menší, resp. větší, z hodnot `x` a `y`.

```
template<typename ForwardIterator>  ForwardIterator
min_element(ForwardIterator first, ForwardIterator last);

template<typename ForwardIterator, typename Compare>  ForwardIterator
min_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

První iterátor `i` ze sledu `[first, last)`, který ukazuje na nejmenší prvek, tj. `!(*j < *i)`, resp. `comp(*j, *i) == false`, kde `j` je libovolný iterátor z daného sledu.

```
template<typename ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<typename ForwardIterator, typename Compare>  ForwardIterator
max_element(ForwardIterator first, ForwardIterator last, Compare comp);
```

První iterátor `i` ze sledu `[first, last)`, který ukazuje na největší prvek, tj. `!(*i < *j)`, resp. `comp(*i, *j) == false`, kde `j` je libovolný iterátor z daného sledu.

lexicographic_compare

```
template<typename InputIterator1, typename InputIterator2>  bool
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2);

template<typename InputIterator1, typename InputIterator2, typename Compare>  bool
lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        Compare comp);
```

Vrací `true`, pokud jsou prvky ze sledu `[first1, last1)` lexikograficky menší než prvky ze sledu `[first2, last2)`.

next_permutation

```
template<typename BidirectionalIterator>  bool
next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<typename BidirectionalIterator, typename Compare>  bool
next_permutation(BidirectionalIterator first, BidirectionalIterator last,
                 Compare comp);
```

Transformuje posloupnost definovanou sledem `[first, last)` na *následující (next)* permutaci.

```
#include <iostream>
#include <algorithm>

int main()
{
  int q[] = {0, 1, 2};
  int N   = sizeof q/sizeof(int);

  for (int k=0;; k++, std::cout << " ")
    {
      bool b = std::next_permutation(q, q+N);
      for (int i=0; i<N; i++) std::cout << q[i];
      if (!b) break;
    }
  std::cout << "\n";
}

021 102 120 201 210 012
```

Následující permutace je nalezena na základě předpokladu, že množina všech permutací je lexikograficky setříděna vzhledem k operátoru <, resp. k podmínce `comp`. Pokud je výsledná permutace setříděná, vrací funkce hodnotu `true`, v opačném případě `false`.

prev_permutation

```
template<typename BidirectionalIterator>  bool
prev_permutation(BidirectionalIterator first, BidirectionalIterator last);

template<typename BidirectionalIterator, typename Compare>  bool
prev_permutation(BidirectionalIterator first, BidirectionalIterator last,
                 Compare comp)
```

Počítá *předchozí (previous)* permutaci sledu `[first, last)`.

10.6.4 Zobecněné numerické operace

Následující algoritmy jsou definovány v hlavičce `<numeric>` a jsou určeny pro výpočet sumace, skalárního součinu, částečných součtů a diferencí.

```
template <typename InputIterator, typename T>
T accumulate(InputIterator first, InputIterator last, T init);
template <typename InputIterator, typename T, typename BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);

template <typename InputIterator1, typename InputIterator2, typename T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);
template <typename InputIterator1, typename InputIterator2, typename T,
          typename BinaryOperation1, typename BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);

template <typename InputIterator, typename OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result);
template <typename InputIterator, typename OutputIterator, typename BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryOperation binary_op);

template <typename InputIterator, typename OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result);
template <typename InputIterator, typename OutputIterator, typename BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                                   OutputIterator result,
                                   BinaryOperation binary_op);
```

10.6.5 Algoritmy knihovny C

Standardní hlavička `<cstdlib>` obsahuje deklarace dvou funkcí `bsearch` a `qsort` pro binární vyhledávání a algoritmus třídění *quick sort*.

```
extern "C"   void *bsearch(const *void key, const *void base,
                           size_t nmemb, size_t size,
                           int (*compar)(const void *, const void *));
extern "C++" void *bsearch(const *void key, const *void base,
                           size_t nmemb, size_t size,
                           int (*compar)(const void *, const void *));

extern "C"   void qsort(void* base, size_t nmemb, size_t size,
                        int (*compar)(const void*, const void*));
extern "C++" void qsort(void* base, size_t nmemb, size_t size,
                        int (*compar)(const void*, const void*));
```

Dodatek A

Generování náhodných čísel

Generování náhodných čísel je obvykle úzce spojeno s měřením času, protože systémový čas se používá pro inicializaci generátoru. Podívejme se tedy stručně nejprve na C++ funkce pro měření času z knihovny `<ctime>`.

A.1 Datum a čas

C++ knihovna `<ctime>` obsahuje stejné funkce, makra a struktury jako standarní C hlavičkový soubor `<time.h>`

typ	*jméno*
makro	`NULL`
typy	`size_t clock_t time_t`
struktura	`tm`
funkce	`asctime clock difftime localtime strftime ctime gmtime mktime time`

Jejich použití demonstruje následujicí ukázka

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

int main()
{
  time_t zacatek, konec;

  zacatek = time(0);
  cout << "zahajeni testu:   " << ctime(&zacatek) << endl;

  clock_t cpu_1 = clock();

  for (double z, i=1; i<=10000; i++)
```

```
    for (double j=1; j<=10000; j++)
      {
        z = sin(1/(i+j));
      }

  clock_t cpu_2 = clock();

  konec = time(0);
  cout << "ukonceni testu:   " << ctime(&konec)   << endl;

  double cas = difftime(konec, zacatek);
  cout << "cas ve  vterinach: " << cas << endl;
  cout << "cas CPU          : "
       << double(cpu_2 - cpu_1)/CLOCKS_PER_SEC << endl;
}
```

Předchozí ukázka počítá 10^8-krát hodnotu goniometrické funkce sinus a měří skutečný čas běhu programu a pro porovnání také čas procesoru (čas CPU). Výsledek může vypadat například takto

```
zahajeni testu:   Sat Aug 14 12:08:33 2004

ukonceni testu:   Sat Aug 14 12:08:43 2004

cas ve  vterinach: 10
cas CPU          : 9.83
```

A.2 Rovnoměrné rozdělení

Pro generování rovnoměrného rozdělení slouží funkce `rand()` a `srand()`, která jsou deklarovány v hlavičkovém souboru `<cstdlib>`. Funkce `rand()` vrací pseudonáhodné číslo z intervalu 0 až `RAND_MAX`. Funkce `srand()` slouží pro inicializaci generátoru a umožňuje v případě potřeby opakování generované posloupnosti. Pokud chceme, aby se náš program choval *náhodně* a produkoval pokaždé jiné výsledky, můžeme pro inicializaci generátoru použít hodnotu systémového času, který získáme voláním funkce `time()`.

```
#include <iostream>
#include <iomanip>
#include <ctime>
#include <cstdlib>

using namespace std;

int main()
{
  cout << "RAND_MAX = " << RAND_MAX << "\n\n";

  for (long seed=0; seed<436427274; seed+=144634572)
    {
      cout << setw(10) << seed << " :";
      srand(seed);
      for (int n=0; n<5; n++) cout << setw(11) << rand();
```

```
        cout << endl;
      }


    cout << "\ninicializace generatoru podle systemoveho casu\n\n";
    srand((unsigned)time(0));     // inicializace podle casu
    for (int n=0; n<6; n++) cout << setw(11) << rand();
    cout << endl;
  }

  RAND_MAX = 2147483647

           0 : 1804289383  846930886 1681692777 1714636915 1957747793
   144634572 : 1440726074 1639022228 2060147763 1436770712 1885714978
   289269144 : 1862992643  119865510 2035512727  791691917  625253601
   433903716 : 1209863182 1822157199  940565993  144941674 1512122861

  inicializace generatoru podle systemoveho casu

   1082856188   94938174  890764167 1189959127   27475313  525198377
```

Pokud často potřebujeme generovat náhodná čísla s rovnoměrným rozdělením z intervalu `<0, 1)`, můžeme transformaci z intervalu `<0, RAND_MAX>` popsat pomocnou inline funkcí, například

```
inline float Rand_U()
{
   return rand()/(RAND_MAX + 1.0f);
}
```

Z podstaty věci (konečný počet stavů generátoru) plyne, že každá softwarově generovaná náhodná posloupnost musí být nutně periodická. Každý generátor má jistou periodu (u kvalitního generátoru velmi dlouhou), před kterou může předcházet úvodní neperiodická řada s konečným počtem členů. Je tedy přesnější mluvit o pseudonáhodné posloupnosti produkované daným generátorem. Máme-li statisticky kvalitní generátor rovnoměrného rozdělení, můžeme s jeho pomocí poměrně snadno generovat pseudonáhodné veličiny s jiným rozdělením pravděpodobnosti.

Hrací kostku (tj. funkci, která vrací náhodná čísla 1 až 6) můžeme naprogramovat třeba takto

```
int Kostka()
{
  return 1 + int(6.0*rand()/(RAND_MAX+1.0));
}
```

Protože dolní bity náhodných čísel generovaných funkcí `rand()` často vykazují špatné statistické vlastnosti, nikdy bychom neměli používat konstrukce jako `rand()%6`, především pokud chceme, aby náš program byl dobře přenositelný.

A.3 Normální rozdělení

Normální normované rozdělení $N(0,1)$ můžeme snadno generovat například s využitím následující funkce `Rand_N()` z knihovny projektu GNU Gama

http://www.gnu.org/software/gama/

Najdete zde také alternativní generátor rovnoměrného rozdělení `Rand_N()` (viz Knuth, Algorithm A, *Additive number generator*, The Art of Computer Programming).

```
/*
   Ratio method for normal deviates

   Algorithm R from The Art of Computer Programming by D.E.Knuth,
   Addison-Wesley Publishing Company, 2nd ed., 1981, vol. 2,
   ISBN 0-201-03822-6, pp. 125-127.
*/

float Rand_N()
{
   static bool start = true;
   static float C1, C2, C3;
   if (start)
      {
         C1 = sqrt(8.0/exp(1.0));
         C2 = 4*exp(0.25);
         C3 = 4*exp(-1.35);
         start = false;
      }

   float U, V, X, X2;

   for (;;)
      {
         do
            U = Rand_U();
         while (!U);
         V = Rand_U();
         X = C1*( V - 0.5)/U;
         X2 = X*X;
         if (X2 <= 5.0 - C2*U)  return X;
         if (X2 >= C3/U + 1.4)  continue;
         if (X2 <= -4.0*log(U)) return X;
      }
}
```

A.4 Příklad simulace

Narozeninový paradox říka, že pokud se v místnosti sejde 23 či více lidí, šance, že alespoň dva mají narozeniny ve stejném dni, je značná (Knuth: The Art of Computer Programming, Vol. 3). Pokud bychom si toto tvrzení chtěli ověřit numericky, můžeme použít následující program.

```
#include <iostream>
#include <iomanip>
#include <cstdlib>
```

```
#include <ctime>

using namespace std;

bool pokus(int N)
{
  int rok[365] = {0};

  for (int i=0; i<N; i++)
    rok[int(365.0*rand()/(RAND_MAX + 1.0))]++;

  for (int i=0; i<365; i++)
    if (rok[i] > 1)
      return true;

  return false;
}

float pn(int M, int N)
{
  int test=0;
  for (int i=0; i<M; i++)
    if (pokus(N))
      test++;

  return float(test)/float(M);
}

int main()
{
  const int pocet[] = {10, 22, 23, 30, 50 };
  const int M = 100000;
  srand(time(0));

  cout.setf(ios_base::fixed, ios_base::floatfield);
  cout.precision(4);

  for (int i=0; i<sizeof(pocet)/sizeof(int); i++)
  {
    const int N = pocet[i];
    cout << "Pocet osob: " << N << "   pravdepodobnost: ";
    for (int p=0; p<4; p++)
      cout << setw(7) << pn(M, N) << " ";
    cout << endl;
  }
}
```

Výsledky naší simulace pak mohou vypadat například takto:

```
Pocet osob: 10   pravdepodobnost:  0.1176  0.1163  0.1164  0.1155
Pocet osob: 22   pravdepodobnost:  0.4756  0.4736  0.4751  0.4770
Pocet osob: 23   pravdepodobnost:  0.5067  0.5084  0.5066  0.5057
Pocet osob: 30   pravdepodobnost:  0.7041  0.7048  0.7051  0.7054
Pocet osob: 50   pravdepodobnost:  0.9704  0.9703  0.9702  0.9700
```

Rejstřík anglických termínů

Rejstřík

Literatura

[1] Bjarne Stroustrup: The C++ programming Language, 3rd ed., Addison–Wesley, Reading, Mass., June 1997, 910 p.

`http://www.research.att.com/~bs/3rd.html`

[2] The C++ standard, BS ISO/IEC 14882:2003 (Second Edition), John Wiley & Sons, Ltd., West Sussex, England, 2003, 782 p.

[3] `news://comp.lang.c++.moderated`

[4] C++ Boost source libraries

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

[5] Bruce Eckel: Thinking in C++, Volume 1 and 2.

`http://www.mindview.net/`

[6] David Vandevoorde – Nicolai M. Josuttis: C++ Templates, The Complete Guide, Addison–Wesley, 2003, 528 p.

[7] Brian W. Kernighan – Dennis M. Ritchie: The C Programming Language, Prentice-Hall Inc. New Jersey 1978