PROSTORY JMEN

Při používání rozsáhlých knihoven z různých zdrojů může snadno dojít ke konfliktu jmen, např. dvě globální proměnné, dvě funkce nebo dvě třídy se budou jmenovat stejně. Podobný problém se může vyskytnout i při týmovém vývoji velkých aplikací.

Prostor jmen (angl. *namespace*) připomíná po formální stránce strukturu nebo třídu. Jména, deklarovaná uvnitř prostoru jmen, lze používat i mimo něj, musí se ale specifikovat prostor jmen, do kterého patří. To znamená, že se musí kvalifikovat jménem prostoru, ve kterém byly deklarovány, nebo jej zpřístupnit pomocí deklarace či direktivy `using`.

Definice prostoru jmen

Syntaxe:

definice_prostoru_jmen:

namespace *identifikátor*$_{nep}$ **{** *posloupnost_deklarací* **}**

Identifikátor zde představuje jméno definovaného prostoru jmen. Pokud se vynechá, definuje se anonymní prostor jmen.

Posloupnost_deklarací se skládá z deklarací proměnných, funkcí, typů atd., které leží v definovaném prostoru jmen. Tyto deklarace podléhají stejným pravidlům jako deklarace globálních proměnných, funkcí, tříd atd. Uvnitř jednoho prostoru jmen lze definovat i další, *vnořený prostor jmen*. Prostor jmen, v němž je vnořený prostor jmen definován, se nazývá *nadřízený prostor jmen*.

Identifikátory, deklarované mimo uživatelem definovaný prostor jmen (pojmenovaný nebo anonymní), jsou součástí tzv. *globálního prostoru jmen* (angl. *global namespace*).

Složky deklarované v prostoru jmen, mohou být definovány uvnitř tohoto prostoru jmen nebo mimo něj. Pokud jsou definovány mimo něj, musí se kvalifikovat identifikátorem prostoru jmen a binárním rozlišovacím operátorem. Definice složky (tělo funkce apod.) je součástí prostoru jmen, i když je uvedena mimo prostor jmen. Definici složky mimo prostor jmen musí předcházet její deklarace v prostoru jmen. Např.:

```
namespace A {
   void f();
   // ...
   void f() { /* ... */ }
}
namespace B {
   void g();
}
void B::g() { /* ... */ } // musí se kvalifikovat
```

Prostor jmen se smí definovat pouze v oblasti platnosti prostoru jmen (globálního nebo uživatelem definovaného), tj. nesmí se objevit v těle funkce (metody), třídy nebo v seznamu parametrů prototypu funkce.

Pokud se identifikátor, deklarovaný v prostoru jmen, použije mimo něj, musí se kvalifikovat identifikátorem prostoru jmen pomocí binárního rozlišovacího operátoru.

Jména deklarovaná v prostoru jmen zastiňují jména z nadřízených prostorů jmen a globální jména deklarovaná v globálním prostoru jmen. Ke globálnímu identifikátoru se potom přistupuje pomocí unárního rozlišovacího operátoru a jména z nadřízených prostorů jmen se musí kvalifikovat identifikátorem jejich prostoru jmen.

Příklad

```
int a = 2, b = 3;

namespace ProstorJmen {
   int a = 4;
   void f();
}
void ProstorJmen::f()
{ std::cout << "funkce f: a*b = " << a*b << '\n'; }

int main()
{
   int a = 5;
   std::cout << "lokalni a = " << a << '\n';
   std::cout << "ProstorJmen::a = " << ProstorJmen::a << '\n';
   std::cout << "globalni a = " << ::a << '\n';
   ProstorJmen::f();
   return 0;
}
```

V programu jsou nejprve deklarovány 2 globální proměnné `a` a `b`, které nepatří do žádného uživatelem definovaného prostoru jmen a potom prostor jmen s názvem `ProstorJmen`. V tomto prostoru jmen je deklarována proměnná `a` a funkce `f()`. Definiční deklarace funkce `f()` leží mimo prostor jmen. Tělo funkce `f()` je součástí prostoru jmen `ProstorJmen`. To znamená, že proměnná `a` má v této funkci hodnotu 4. Standardní datový proud `cout` je kvalifikován prostorem jmen `std`, protože je, jakož i všechny další objekty standardní knihovny C++, součástí prostoru jmen `std`.

Výpis programu je následující:

```
lokalni a = 5
ProstorJmen::a = 4
globalni a = 2
funkce f: a*b = 12
```

Příklad

```
namespace Vnejsi {
   int a = 1;
   namespace Vnitrni {
      void f() { std::cout << "Funkce f = " << a << '\n'; }
      int a = 2;
      void g() { std::cout << "Funkce g = " << a << '\n'; }
   }
}

int main()
{
   std::cout << "Vnejsi a = " << Vnejsi::a << '\n';
   std::cout << "Vnitrni a = " << Vnejsi::Vnitrni::a << '\n';
   Vnejsi::Vnitrni::f();
   Vnejsi::Vnitrni::g();
   return 0;
}
```

V uvedeném příkladu jsou definovány dva prostory jmen, nadřízený `Vnejsi` a vnořený `Vnitrni`. V obou je definována proměnná `a`. Ve vnořeném prostoru jmen jsou definovány dvě funkce. V obou z nich se vypisuje hodnota proměnné `a`. Ve funkci `f()` ještě není známa deklarace proměnné `a` v prostoru jmen `Vnitrni`, a proto se použije proměnná `a` z nadřízeného prostoru jmen.

Složky vnitřního prostoru jmen použité ve funkci `main` se musí kvalifikovat jmény obou prostorů jmen.

Výpis programu bude následující:

```
Vnejsi a = 1
Vnitrni a = 2
Funkce f = 1
Funkce g = 2
```

Jako přátelé třídy, která je deklarována v prostoru jmen, se uvažují funkce a třídy, které jsou deklarovány ve stejném prostoru jmen.

Příklad

```
void f() { std::cout << "globalni funkce f\n"; } // #1
namespace PA {
   class TA {
      static int a;
      friend void f(); // přítelem je PA::f()
   };
   int TA::a = 10;
   void f() { std::cout << "a = " << TA::a << '\n'; } // #2
}
```

V uvedeném příkladu je v prostoru jmen `PA` definována třída `TA`. V této třídě je deklarována spřátelená funkce `f()`. Nejedná se o funkci `f()`, která je definována před prostorem jmen `PA` (příkaz `#1`), nýbrž o funkci `f()`, definovanou v témže prostoru jmen (příkaz `#2`). Kdyby příkaz #2 nebyl uveden, překladač by oznámil chybu. Deklarace spřátelené globální funkce `f()` v třídě `TA` se provede následovně:

```
namespace PA {
   class TA {
      friend void ::f();
......// ...
   };
}
```

Definice po částech

Definici prostoru jmen lze rozdělit i do několika částí, které mohou ležet i v několika různých zdrojových souborech. Např.:

```
namespace A { int a; }
namespace B {
   int f() { A::a + A::b; } // Chyba
}
namespace A { int b; }
```

Proměnné `a` a `b` leží ve stejném prostoru jmen `A`. Pro jejich používání ovšem platí pravidlo, že jméno se musí nejprve deklarovat a teprve potom ho lze použít. To znamená, že ve funkci `B::f()` lze použít proměnnou `A::a`, nikoli však proměnnou `A::b`. Problém lze vyřešit tak, že se v prostoru jmen `B` deklaruje prototyp funkce `f()` a její definiční deklarace se uvede až po deklaraci proměnné `A::b`, např. takto:

```
namespace A { int a; }
namespace B {
   int f();
}
```

```
namespace A { int b; }
namespace B {
   int f() { return A::a + A::b; } // OK
}
```

Typickým příkladem definice prostoru jmen, rozdělené do několika souborů je standardní C++ knihovna, jejíž všechny části jsou součástí prostoru jmen `std`, i když jsou deklarovány v různých hlavičkových a zdrojových souborech.

Dalším příkladem může být deklarace třídy v prostoru jmen. Definiční deklarace třídy a informativní deklarace složek třídy je uvedena v hlavičkovém souboru, definiční deklarace složek třídy je uvedena ve zdrojovém souboru.

Anonymní prostor jmen

Vynechá-li se v definici prostoru jmen identifikátor prostoru, definuje se *anonymní prostor jmen* (angl. *unnamed namespace*).

Jména deklarovaná v anonymním prostoru jmen se v programu používají bez kvalifikace, podobně jako globální proměnné. Všechny anonymní prostory jmen ve stejné překladové jednotce (modulu) se spojí v jeden prostor, kterému překladač přidělí jednoznačné interní jméno, které bude v každé překladové jednotce jiné. To znamená, že proměnné a funkce, které jsou v něm deklarovány, se budou chovat jako statické – nebude je možné používat v jiných překladových jednotkách.

Příklad

Soubor1.cpp:

```
static int a = 1;
int b = 2;

namespace {
   int c = 3;
   void f() { std::cout << a << b << c; }
}
```

Soubor2.cpp:

```
extern int a, b, c;
void g() { std::cout << a << b << c; } // Chyba
```

V zdrojovém souboru Soubor1.cpp je definována globální statická proměnná `a`, globální nestatická proměnná `b` a anonymní prostor jmen, obsahující proměnnou `c` a funkci `f()`. Ve funkci `f()` lze použít všechny 3 definované proměnné.

V zdrojovém souboru Soubor2.cpp je uvedena informativní deklarace proměnných `a`, `b` a `c`, které jsou použity ve funkci `g()`. Pokud se oba soubory přeloží, překladač chybu neoznámí. Avšak při sestavování (linkování) programu vznikne chyba – proměnné `a` a `c` nelze v souboru Soubor2.cpp použít.

Alias prostoru jmen

V jazyku C++ lze definovat přezdívku pro existující prostor jmen, neboli *alias* (angl. *namespace alias*). Definice přezdívky má následující syntaxi:

definice_přezdívky_prostoru_jmen:

namespace *alias_prostoru_jmen* **=** *kvalifikovaný_specifikátor_prostoru_jmen***;**

alias_prostoru_jmen:

identifikátor

kvalifikovaný_specifikátor_prostoru_jmen:

specifikátor_vnořeného_prostoru_jmen$_{nep}$ *jméno_prostoru_jmen*

specifikátor_vnořeného_prostoru_jmen:

jméno_prostoru_jmen **::** *specifikátor_vnořeného_prostoru_jmen*$_{nep}$

jméno_prostoru_jmen:

původní_jméno_prostoru_jmen

alias_prostoru_jmen

původní_jméno_prostoru_jmen:

identifikátor

Alias_prostoru_jmen je identifikátor nového označení existujícího prostoru jmen.

Kvalifikovaný_specifikátor_prostoru_jmen je jméno existujícího prostoru jmen. Jedná-li se o vnořený prostor jmen, kvalifikuje se jménem nadřízeného prostoru a binárním rozlišovacím operátorem. Jako jméno prostoru jmen lze použít i dříve definovaný alias.

Příklad

```
namespace Vnejsi {
   int a = 1;
   namespace Vnitrni {
      int a = 2;
      void f() { std::cout << "a = " << a << '\n'; }
   }
}

namespace AVnitrni = Vnejsi::Vnitrni;
namespace AV = AVnitrni;

int main()
{
   AVnitrni::f();
   std::cout << AV::a << '\n';
   return 0;
}
```

V uvedeném příkladu jsou definovány dva prostory jmen, nadřízený a vnořený. Potom je definován alias `AVnitrni` pro prostor `Vnejsi::Vnitrni` a alias `AV` pro alias `AVnitrni`. Funkce `f()` je kvalifikována přezdívkou `AVnitrni` a proměnná `a` přezdívkou `AV`. V obou případech se jedná o alias vnitřního prostoru jmen.

Funkci `f()` by bylo možné ve funkci `main` zavolat třemi možnými způsoby:

```
Vnejsi::Vnitrni::f();
AVnitrni::f();
AV::f();
```

Zpřístupnění prostoru jmen

Složka prostoru jmen, která je použita mimo její prostor jmen, se nemusí kvalifikovat jménem prostoru jmen, pokud se použije deklarace nebo direktiva `using`.

Deklarace using

Syntaxe:

deklarace_using:

using *specifikátor_vnořeného_jména identifikátor***;**

using :: *identifikátor***;**

specifikátor_vnořeného_jména:

jméno_třídy_nebo_prostoru_jmen **::** *specifikátor_vnořeného_jména*$_{nep}$

jméno_třídy_nebo_prostoru_jmen:

jmenovka_třídy

jméno_prostoru_jmen

Deklarace `using` slouží ke zpřístupnění *identifikátoru* v oblasti viditelnosti deklarace `using`. *Specifikátor_vnořeného_jména* je buď jméno prostoru jmen (původní nebo alias) případně kvalifikovaného jménem nadřízeného prostoru jmen nebo jmenovka třídy případně kvalifikovaná jmenovkou obklopující třídy. *Identifikátor* v prvním případě představuje jméno složky z uvedeného prostoru jmen nebo jmenovku třídy nebo jméno složky třídy. Ve druhém případě *identifikátor* je globální identifikátor (identifikátor globální proměnné, obyčejné funkce apod.), který je součástí globálního prostoru jmen.

Deklarace `using` se používá pro zpřístupnění složky deklarované v prostoru jmen nebo ve třídě nebo ke zpřístupnění globálního objektu. Zpřístupnění složky třídy se používá ke změně přístupových práv složky předka v potomkovi – viz 6. přednáška předmětu Jazyk C++ I.

Deklarace `using` se může vyskytnout jak na úrovni souboru tak i v nějakém bloku.

Pokud by se v předchozím příkladu na začátku těla funkce `main` uvedly deklarace:

```
using Vnejsi::Vnitrni::f;
using AV::a;
```

mohly by se identifikátory `f` a `a` použít ve funkci `main` bez kvalifikace:

```
f();
std::cout << a << '\n';
```

V jednom oboru viditelnosti deklarace nesmí být pomocí deklarace `using` zpřístupněno více identifikátorů stejného jména s výjimkou funkcí. Pokud by se v předchozím příkladu v těle funkce `main` uvedly deklarace

```
using Vnejsi::Vnitrni::a;
using Vnejsi::a;
```

překladač oznámí chybu vícenásobné deklarace identifikátoru `a`.

Pokud by ale program vypadal následovně:

```
namespace Vnejsi {
   int a = 1;
   namespace Vnitrni { int a = 2; }
}
using Vnejsi::Vnitrni::a; // #1
```

```
int main()
{
   using Vnejsi::a; // #2
   std::cout << a << '\n'; // #3
   return 0;
}
```

překladač chybu neoznámí. Deklarace #2 zastiňuje deklaraci #1 a tudíž v příkazu #3 se použije proměnná `Vnejsi::a`.

Pokud *identifikátor* v deklaraci `using` představuje jméno přetížené funkce, zpřístupní se všechny verze této funkce. Ve stejném oboru viditelnosti deklarace se může vyskytnout více deklarací `using` se stejným identifikátorem, který představuje funkci.

Příklad

```
namespace A {
   void f(int);
}
namespace B {
   void f(double);
   void f(char);
// void f(int);
}

void g()
{
   using A::f; // #1
   using B::f; // #2
   f('c'); // #3
   f(1); // #4
}
```

Ve funkci `g()` deklarace #2 zpřístupňuje jak funkci `B::f(double)`, tak i funkci `B::f(char)`. Příkaz #3 zavolá funkci `B::f(char)` a příkaz #4 funkci `A::f(int)`. Chyba nevznikne.

Pokud se navíc deklaruje funkce `B::f(int)`, podle normy jazyka C++ jsou příkazy #1 a #2 správné. Chyba nastane až při použití funkce `f(int)` bez kvalifikace v příkazu #4, kdy překladač nedokáže určit, kterou funkci `f(int)` má zavolat. V takovémto případě by se muselo volání funkce `f(int)` kvalifikovat jménem prostoru jmen. Překladač MS Visual C++ 2003 ale oznámí chybu již na příkazu #2.

Pokud se pomocí deklarace `using` zpřístupňuje jmenovka třídy, lze se potom pomocí jmenovky třídy odvolávat na statické složky této třídy a její výčtové konstanty bez kvalifikace jménem prostoru jmen. Zpřístupnění jména výčtového typu ještě neznamená zpřístupnění jeho výčtových konstant. Deklarace `using` zpřístupňující jednotlivé složky třídy nemůže být uvedena mimo tělo třídy.

Příklad

```
namespace A {
   struct TA {
      enum { b = 1 };
      static int a;
   };
   int TA::a = 10;
   enum TB { c = 2 };
}
```

```
void g()
{
   using A::TA;
   using A::TB;
   std::cout << "TA::a = " << TA::a << '\n';
   std::cout << "TA::b = " << TA::b << '\n';
   std::cout << "TB::c = " << TB::c << '\n'; // #1 Chyba
}
void h()
{
   using A::TA::a; // #2 Chyba
   using A::TA::b;
   using A::c;
   std::cout << "a = " << a << '\n'; // #3 Chyba
   std::cout << "b = " << b << '\n'; // A::TA::b
   std::cout << "c = " << c << '\n'; // A::c
}
```

Ve funkci `g()` se zpřístupňuje jmenovka struktury `TA`. Díky tomu se lze v této funkci odvolávat na statický atribut `a` a výčtovou konstantu `b` pouze pomocí jmenovky třídy `TA`. V této funkci je dále zpřístupněn identifikátor `TB` výčtového typu, což ale neznamená zpřístupnění jeho výčtové konstanty `c`, a proto příkaz #1 je chybný.

Ve funkci `h()` je příkaz #2 chybný, protože deklaraci `using` lze použít na složku třídy `TA` jen v těle potomka třídy `TA`. V důsledku toho není správný ani příkaz #3. Zpřístupnění výčtových konstant `b` a `c` je v pořádku, protože identifikátory `b` a `c` nejsou složkami třídy `TA`, nýbrž složkami výčtových typů. Výčtové konstanty `b` a `c` se v této funkci nemusí kvalifikovat žádným jménem.

Pomocí deklarace `using` lze zpřístupnit i zastíněný globální identifikátor.

Příklad

```
int i = 1;

namespace A {
   int i = 2;
}

void f()
{
   using A::i;
   std::cout << "A::i = " << i << '\n';
   {
      using ::i;
      std::cout << "globalni i = " << i << '\n';
   }
}
```

Direktiva using

Syntaxe:

direktiva_using:

using namespace *kvalifikovaný_specifikátor_prostoru_jmen***;**

Direktiva `using` slouží ke zpřístupnění všech složek, deklarovaných v uvedeném prostoru jmen v oblasti viditelnosti této direktivy. Složky ze zpřístupněného prostoru jmen lze použít bez kvalifikace, pokud nedojde k nejednoznačnosti. Případné nejednoznačnosti lze vyřešit pomocí kvalifikace.

Direktiva `using` se smí uvést v oblasti platnosti prostoru jmen (globálního nebo uživatelem definovaného) nebo bloku. Nesmí se tedy objevit v oblasti platnosti třídy nebo prototypu funkce.

Příklad

```
namespace A {
   int i;
   void f() { std::cout << "A::f\n"; }
}
namespace B {
   void f() { std::cout << "B::f\n"; }
}

int main()
{
   using namespace A;
   f(); // #1 OK
   using namespace B;
   f(); // #2 Chyba
   A::f(); // OK
   B::f(); // OK
   return 0;
}
```

V uvedeném příkladu ve funkci `main` se nejprve zpřístupní prostor jmen `A`, tj. zpřístupní se proměnná `A::i` a funkce `A::f()`. Příkaz #1 je tedy v pořádku. Po zpřístupnění i prostoru jmen `B` jsou zpřístupněné dvě funkce `f()`, proto volání funkce `f()` bez kvalifikace v příkazu #2 je chybné. Překladač neví, zda má volat `A::f()` nebo `B::f()`.

Direktiva `using` je tranzitivní. To znamená, že pokud se zpřístupní touto direktivou prostor jmen `B` a v prostoru jmen `B` je zpřístupněn prostor jmen `A`, lze v oboru viditelnosti direktivy `using namespace B;` používat bez kvalifikace nejen jména z prostoru `B`, ale i jména z prostoru `A`.

Příklad

```
namespace A {
   void f() { std::cout << "A::f\n"; }
}
namespace B {
   using namespace A;
   void g() { std::cout << "volani A::f() z B::g()\n"; f(); }
}

int main()
{
   using namespace B;
   f();
   g();
   return 0;
}
```

V uvedeném příkladu zpřístupněním prostoru jmen `B` ve funkci `main`, se zpřístupní i prostor jmen `A`, a tudíž funkce `f()` a `g()` se ve funkci `main` nemusí kvalifikovat identifikátorem jejich prostoru jmen.

PROSTORY JMEN – POKRAČOVÁNÍ

Koenigovo vyhledávání operátorů a funkcí

Pokud se operátor nemá volat jako obyčejná funkce nebo metoda, nelze při jejich zápisu ve výrazu použít kvalifikaci prostoru jmen. Proto jsou pro vyhledávání operátorů stanovena zvláštní pravidla.

Operátory se vyhledávají jak v kontextu jejich použití, tak i v kontextu jejich operandů. Kontext operandu představuje prostor jmen, ve kterém je operand deklarován, a kontext použití operátoru znamená prostor jmen, do kterého patří funkce (metoda), v jejímž těle je operátor použit. Na množinu nalezených operátorů se potom uplatní pravidla pro rozlišování přetížených operátorů.

Příklad

```
namespace PA {
   struct TA {
      int a;
      TA(int _a = 0) : a(_a) {}
   };
   TA& operator ++ (TA& A) { A.a++; return A; }
}
namespace PB {
    void f(PA::TA& A) { ++A; }
}
```

V uvedeném příkladu překladač ve funkci `PB::f` nalezne operátor inkrementace podle kontextu operandu `A`. Operátor je použit pro operand typu `TA`, který patří do prostoru jmen `PA`, do něhož patří i tento operátor.

Příklad

```
namespace PA {
   struct TA {
      int a;
      TA(int _a = 0) : a(_a) {}
   };
}
PA::TA& operator ++ (PA::TA& A) { A.a++; return A; } // #1


namespace PB {
   PA::TA& operator ++ (PA::TA& A) { A.a++; return A; } // #2
   void f(PA::TA& A) { ++A; }
}
```

V uvedeném příkladu překladač ve funkci `PB::f` nalezne vhodný operátor inkrementace, a to podle kontextu použití operátoru. Operátor inkrementace je použit v těle funkce `f`, která patří do prostoru jmen `PB`, v níž je definován i tento operátor #2. Operátor #1 by se z funkce `f` volal v případě, kdyby nebyl definován operátor #2. Pokud by operátor #1 byl definován v prostoru jmen `PA`, překladač by oznámil chybu, protože by nalezl dva stejné operátory inkrementace.

Obdobné pravidlo platí i pro volání funkce. Použije-li se např. jméno funkce bez kvalifikace a jejím skutečným parametrem bude hodnota objektového nebo výčtového typu, který je definovaný v nějakém prostoru jmen, bude se tato funkce hledat i v prostoru jmen, do něhož patří typ jejího skutečného parametru.

Příklad

```
namespace PA {
   enum  TA { a, b };
   void f(TA A) { std::cout << (A == a ? "a" : "b") << '\n'; } }

int main()
{
   PA::TA A = PA::b;
   f(A); // OK
   return 0;
}
```

V uvedeném příkladu lze ve funkci `main` volat funkci `f()` bez kvalifikace `PA::`, protože její skutečný parametr je typu `TA` z prostoru jmen `PA`.

Hlavičkové soubory knihovny jazyka C v C++

Hlavičkové soubory knihovny jazyka C lze v jazyce C++ použít ve dvou tvarech:

1. Standardním způsobem známým z jazyka C, např. `<stdio.h>`.
2. Vynecháním přípony a přidáním písmene `c` na začátek názvu hlavičkového souboru, např. `<cstdio>`.

Pokud je hlavičkový soubor knihovny jazyka C zahrnut ve zdrojovém textu jazyka C++ jedním z uvedených způsobů, překladač zahrne funkce a další složky uvedené v hlavičkovém souboru do prostoru jmen `std`. V prvním případě se nemusí složky uvedené v hlavičkovém souboru kvalifikovat prostorem jmen `std`, zatímco v druhém případě se kvalifikovat musí.

Norma jazyka C++ doporučuje používat druhý způsob.

V následující tabulce je seznam všech hlavičkových souborů knihovny jazyka C použitelných druhým způsobem:

`<cassert>`	`<cctype>`	`<cerrno>`
`<cfloat>`	`<ciso646>`	`<climits>`
`<clocale>`	`<cmath>`	`<csetjmp>`
`<csignal>`	`<cstdarg>`	`<cstddef>`
`<cstdio>`	`<cstdlib>`	`<cstring>`
`<ctime>`	`<cwchar>`	`<cwctype>`

VÝJIMKY

Výjimka (angl. *exception*) představuje situaci, která nastane v průběhu normálního chodu programu a způsobí, že program nemůže obvyklým způsobem dále pokračovat. Jinými slovy, výjimka je chyba v běhu programu.

Pokud vznikne výjimka, je třeba ji programově ošetřit. Otázkou je, na kterém místě. Chyba, např. zpřístupnění hodnoty prvního prvku v prázdném seznamu může být důsledkem logické chyby v úplně jiné části programu. Chyba se zpravidla ošetřuje na místě jejího původu vzniku. Je proto potřebné se snadno, rychle a bezpečně přesunout na úplně jiné místo v programu.

V prostředí MS Visual C++ 2003 jsou k dispozici tyto druhy výjimek:

- výjimky podle normy jazyka C++ s rozšířeními
- strukturované výjimky operačních systémů Microsoft
- výjimky knihoven Microsoft: MFC a .NET.

Snadný a rychlý přesun řízení programu z jedné funkce do jiné, přímo či nepřímo nadřazené, poskytuje v normě jazyka C tzv. „dlouhý skok“. Ten lze provést pomocí standardních funkcí

`setjmp` a `longjmp`. Ty zajistí zrušení lokálních proměnných ve všech do sebe vnořených blocích, které se tímto skokem opouští, ale nezajistí uvolnění alokovaných prostředků (uzavření souborů, uvolnění paměti).

Mechanismus práce s výjimkami v jazyce C++ zajišťuje kromě ošetření zásobníku (zrušení lokálních proměnných) také volání destruktorů všech lokálních instancí v opouštěných blocích. Kromě toho lze z místa chyby do místa jejího ošetření přenést řadu informací, které lze při ošetřování výjimky využít.

Norma jazyka C výjimky nezná, ale ve 32-bitových překladačích pro operační systémy Microsoft Windows jsou k dispozici tzv. *strukturované výjimky* (angl. *structured exception handling* – SEH), zavedené firmou Microsoft.

Výjimky knihoven Microsoft zapouzdřují mj. strukturované výjimky.

Výjimky podle normy jazyka C++

V jazyce C++, resp. v normě jazyka, lze pracovat pouze s tzv. synchronními (nebo také softwarovými) výjimkami, tj. s výjimkami, které vzniknou uvnitř programu a nejsou závislé na operačním systému. Pomocí výjimek v C++ tedy nelze zpracovávat výjimky jako např. dělení nulou, aritmetické přetečení, přístup na neplatnou adresu apod. Tyto výjimky se nazývají asynchronní (nebo také hardwarové) a lze je ošetřovat pomocí mechanismu strukturovaných výjimek nebo výjimek nadstavbové knihovny.

Všechny operace, které by mohly vyvolat výjimky, se musí provádět v tzv. *pokusném bloku* (angl. *try block*). Pokusný blok se skládá ze složeného příkazu (angl. *compound statement*) a z jednoho či několika *handlerů* (angl. *exception handler*).[1]

Pokud při provádění operací ve složeném příkazu pokusného bloku nevznikne výjimka, proběhnou řádně všechny příkazy tohoto bloku a po jeho ukončení bude program pokračovat za hlídaným blokem – handlery se přeskočí.

Jestliže v průběhu některé z operací ve složeném příkazu pokusného bloku nastane výjimka, provádění příkazů tohoto bloku se předčasně skončí v místě, kde výjimka vznikla a řízení se přesune do některého z handlerů. Pokud handler neukončí běh programu, může program po provedení tohoto handleru pokračovat za pokusným blokem, v němž se handler nachází.

Výjimka může vzniknout přímo ve složeném příkazu pokusného bloku nebo v některém z vnořených bloků či v některé z funkcí volaných ve složeném příkazu pokusného bloku. Pokusný blok může být vnořen do jiného pokusného bloku a výjimku, která vznikne ve vnořeném pokusném bloku, může ošetřit handler v nadřízeném pokusném bloku.

Když vznikne výjimka, začne systém hledat vhodný handler, který by ji ošetřil. Přitom postupuje podle posloupnosti vnořených pokusných bloků a volání funkcí od místa vzniku výjimky k funkci `main`. Dochází k tzv. *šíření výjimky* do nadřízeného bloku resp. do volající funkce.

Při vyvolání výjimky se vytváří datový objekt, který ponese informaci o povaze výjimky a okolnostech jejího vzniku. Často jde o instanci určitého objektového typu. Typ hodnoty, která je posílána handleru při vyvolání výjimky, se nazývá *typ výjimky*.

Syntaxe výjimek

pokusný_blok:

try *složený_příkaz seznam_handlerů*

[1] Podle skript Virius: Programování v C++ : Všechny operace, které by mohly vyvolat výjimky, se musí provádět v hlídaném bloku. Hlídaný blok se skládá z pokusného bloku a z jednoho či několika handlerů. Norma jazyka C++ ale pojem hlídaný blok (guarded block) nezná.

pokusný_blok_funkce:

try *inicializační_část_konstruktoru*$_{nep}$ *tělo_funkce seznam_handlerů*

seznam_handlerů:

handler seznam_handlerů$_{nep}$

handler:

catch (*deklarace_výjimky* **)** *složený_příkaz*

deklarace_výjimky:

deklarační_specifikátory deklarátor

deklarační_specifikátory abstraktní_deklarátor

deklarační_specifikátory

…

výraz_throw:

throw *přiřazovací_výraz*$_{nep}$

Deklarace_výjimky se podobá specifikaci jednoho formálního parametru v deklaraci funkce – obsahuje buď určení typu a identifikátor parametru nebo jen určení typu. Může specifikovat předávání hodnotou i odkazem a může obsahovat i výpustku.

Vyvolání výjimky (angl. *throwing exception*) se provádí pomocí výrazu *výraz_throw*. Hodnota výrazu *přiřazovací_výraz* ponese informace o výjimce. Typ výjimky je určen typem hodnoty tohoto výrazu. Hodnotu tohoto výrazu lze použít v handleru a podle ní se rozhodnout, jak se výjimka ošetří.

Z pokusného bloku včetně handleru lze vyskočit příkazem `return`, `break`, `continue` nebo `goto`. Skákat dovnitř pokusného bloku je zakázáno.

Příklad

```
class TIntVektor {
   int* a, n;
public:
   TIntVektor(int _n) : n(_n) { a = new int[n]; }
   ~TIntVektor() { delete[] a; }

   int&  operator [] (int index) { return a[index]; }
   int   Pocet() const { return n; }
   void  Uloz(const char* JmenoSouboru) const;
   void  Vypis() const; // vypis na obrazovku
};

void TIntVektor::Uloz(const char* JmenoSouboru) const
{
   FILE* f = fopen(JmenoSouboru, "wt");
   if (!f) throw 1;
   for (int i = 0; i < n; i++) {
      if (fprintf(f, "%d\n", a[i]) == EOF) throw 2;
   }
   fclose(f);
}

int main()
{
   try {
      TIntVektor A(10);
```

```
      for (int i = 0; i < A.Pocet(); i++) A[i] = i;
      A.Uloz("d:\\pom\\cisla.txt");
      cout << "Cisla byla ulozena do souboru\n"; // #1
      A.Vypis();                                 // #2
   }
   catch (int i) {
      switch (i) {
         case 1: cout << "Chyba pri otevreni souboru"; break;
         case 2: cout << "Chyba pri zapisu do souboru"; break;
      }
   }
   getch(); // #3
   return 0;
}
```

Tělo funkce `main` začíná pokusným blokem. V něm se vytvoří instance `A` představující pole 10 celých čísel a naplní se posloupností čísel 0 až 9. Potom se zavolá metoda `Uloz`, která uloží pole čísel do zadaného textového souboru.

Pokud uložení proběhne úspěšně, program bude pokračovat dalšími příkazy (tj. příkazy #1, #2, #3) a handler `catch(int i)` se přeskočí.

Pokud při uložení vznikne chyba, vyvolá se výjimka pomocí výrazu `throw 1` resp. `throw 2`. Tím vznikne výjimka typu `int` a zároveň se vytvoří objekt typu `int`. Zbylé příkazy metody `Uloz` za příkazem `throw` se již neprovedou a program začne hledat vhodný handler. V těle metody `Uloz` žádný nenajde a proto přejde do funkce, která metodu volala, tj. do funkce `main`. V ní nalezne vhodný handler `catch(int i)`, jehož příkazy se provedou. Příkazy #1 a #2 se vynechají. Protože program není v tomto handleru ukončen, pokračuje potom příkazem #3.

Výrazy `throw 1` a `throw 2` mohou být např. nahrazeny výrazy

```
throw "Chyba pri otevreni souboru"
throw "Chyba pri zapisu do souboru"
```

potom místo handleru `catch (int i)` by byl ve funkci `main` následující handler:

```
catch (const char* c) {
   cout << c << '\n';
}
```

Nejčastěji však typem výjimky je objektový typ. V uvedeném příkladu by mohla existovat např. třída `TVyjimka`:

```
class TVyjimka {
   char s[250];
public:
   TVyjimka(const char* _s) { strcpy(s, _s); }
   void  Vypis() const;
};

void TVyjimka::Vypis() const
{
   cout << s << '\n';
   cout << "Pro pokracovani stisknete Enter\n";
   getch();
}
```

Metoda `Uloz` by potom měla tento tvar:

```
void TIntVektor::Uloz(const char* JmenoSouboru) const
{
   FILE* f = fopen(JmenoSouboru, "wt");
   if (!f) throw TVyjimka("Chyba pri otevreni souboru");
   for (int i = 0; i < n; i++) {
      if (fprintf(f, "%d\n", a[i]) == EOF)
         throw TVyjimka("Chyba pri zapisu do souboru");
   }
   fclose(f);
}
```

Handler pro ošetření výjimky by byl potom následující:

```
catch (TVyjimka& c) {
   c.Vypis();
}
```

Handler

Handler musí bezprostředně následovat za složeným příkazem pokusného bloku nebo za jiným handlerem. Handler začíná klíčovým slovem `catch`, za kterým je v závorkách specifikován typ výjimky. Specifikace typu výjimky se podobá specifikaci jednoho formálního parametru funkce.

Při hledání odpovídajícího handleru se prochází handlery v pořadí jejích uvedení za složeným příkazem pokusného bloku.

Jméno parametru handleru se může vynechat. V tom případě se nelze v těle handleru odvolávat na objekt uvedený ve výrazu `throw`.

Parametr handleru lze předávat i jako konstantu nebo referenci. Předávání odkazem (referencí) se používá zejména u parametru objektového typu, protože umožňuje používat virtuální metody a nedochází ke zbytečnému kopírování objektu.

Specifikaci typu výjimky v deklaraci handleru lze nahradit výpustkou. Takový handler se nazývá *univerzální*. Zachytí totiž všechny výjimky jakéhokoliv typu. Musí být proto uveden jako poslední v pokusném bloku.

Např.

```
try {
   // ...
}
catch (TVyjimka& t) {
   // ...
}
catch (...) {
   cout << "Neznama vyjimka";
   getch();
   exit(1); // ukončí program
}
```

Nastane-li výjimka, hledá program vhodný handler, a to podle jeho typu (tj. typu jeho parametru). Na rozdíl od volání přetížených funkcí se však téměř neuplatňují typové konverze.

Handler odpovídá výrazu `throw`, který vyvolá výjimku typu `E`, jestliže:

- handler je typu (tj. má parametr typu) *cv* `T` nebo *cv* `T&` a `E` a `T` jsou stejného typu,
- handler je typu *cv* `T` nebo *cv* `T&` a `T` je jednoznačný veřejně přístupný předek třídy `E`,

- handler je typu *cv1* `T*` *cv2* a `E` je ukazatel na typ, který může být konvertován na typ handleru pomocí standardní konverze ukazatelů.

Kde *cv*, *cv1* nebo *cv2* je množina *cv*-modifikátorů, tj. `const`, `volatile`, `const volatile` nebo žádný modifikátor.

Příklad

```
class TVyjimka1 {
public:
   virtual void Vypis() const { cout << "Vyjimka typu 1\n"; }
};
class TVyjimka2 : public TVyjimka1 {
public:
   virtual void Vypis() const { cout << "Vyjimka typu 2\n"; }
};
class TVyjimka3 : protected TVyjimka2 {
public:
   virtual void Vypis() const { cout << "Vyjimka typu 3\n"; }
};
class TVyjimka4 : public TVyjimka1 {
public:
   virtual void Vypis() const { cout << "Vyjimka typu 4\n"; }
};

int main()
{
   try {
      f(); // throw TVyjimka2();
   }
   catch (TVyjimka1 t) {
      t.Vypis();
      getch();
   }
   return 0;
}
```

Handler zachytí výjimku typu `TVyjimka1` a všechny výjimky typu veřejně zděděného ze třídy `TVyjimka1`, tj. `TVyjimka2` a `TVyjimka4` a zavolá odpovídající metodu `Vypis`. Např. pokud vznikne výjimka typu `TVyjimka2`, na obrazovku se vypíše text "`Vyjimka typu 2`".

Pokud je výjimka objektového typu, který tvoří dědickou hierarchii, musí se handler typu předka uvést před handlerem typu potomka. Funkce `main` z předchozího příkladu může mít následující tvar:

```
int main()
{
   try {
      f();
   }
   catch (TVyjimka4& t) {
      t.Vypis(); getch();
   }
   catch (TVyjimka1& t) {
      t.Vypis(); getch();
   }
   return 0;
}
```

V tom případě handler typu `TVyjimka4` zachytí pouze výjimku typu `TVyjimka4` a handler typu `TVyjimka1` zachytí výjimky `TVyjimka1` a `TVyjimka2`. Pokud by se pořadí handlerů otočilo, handler typu `TVyjimka4` by se nikdy neprovedl.

V handleru lze znovu vyvolat tutéž výjimku, kterou handler zachytil. To se provede příkazem

```
throw;
```

Výjimka se pošle dál, do nadřízeného bloku. Tento způsob se využívá např. pokud v některé funkci je potřebné provést určitou operaci, pokud nastane libovolná výjimka, ale tuto výjimku dále neošetřovat, např.

```
f()
{
    try {
        // ...
    }
    catch (...) {
        // ...
        throw;
    }
}
```

Výjimku, která vznikne v handleru (např. i příkazem `throw;`), nemůže zachytit jiný handler stejného pokusného bloku, ale jen nadřízeného pokusného bloku.

Např.

```
void g()
{
    try {
        f();
    }
    catch (TVyjimka& t) { // #1
        t.Vypis();
    }
    catch (...) { // #2
        cout << "Neznama vyjimka";
    }
    return 0;
}
```

Pokud by ve funkci `f()` vznikla výjimka typu `TVyjimka`, zachytil by ji handler #1. Pokud by se v tomto handleru vyvolala výjimka stejného nebo jiného typu, nezachytil by ji handler #2, ale handler nadřízeného pokusného bloku.

Specifikace výjimek v deklaraci funkce

Jazyk C++ nabízí možnost specifikovat pro danou funkci výjimky, které se z ní mohou šířit. Tato informace je určena pro programátora, aby věděl, jaké výjimky může v jednotlivých funkcích očekávat.

Syntaxe:

deklarace_funkce_se_specifikací_výjimek:

hlavička_funkce **throw (** $seznam\_typů_{nep}$ **)**

seznam_typů:

typ

typ, seznam_typů

Seznam_typů je seznam označení typů výjimek, které mohou v dané funkci vzniknout a nejsou v ní ošetřeny. Pokud za hlavičkou funkce klíčové slovo `throw` není uvedeno, znamená to, že se z dané funkce může rozšířit jakákoli výjimka. Bude-li seznam typů prázdný, nesmí se z dané funkce rozšířit žádná výjimka. Specifikace výjimek může být uvedena i v deklaraci ukazatele na funkci nebo třídního ukazatele na metodu. Nesmí se ale uvádět v deklaraci `typedef`.

Příklady:

```
void f();
void g() throw();
void h() throw(TVyjimka, int);
typedef void (*tuf)() throw (int); // chyba
int (TA::*ug)() const throw ();
```

V těle funkce `f()` se může přímo nebo nepřímo vyvolat jakákoli výjimka a rozšířit se z ní. V těle funkce `g()` sice může vzniknout jakákoli výjimka, ale každá musí být zachycena a ošetřena, žádná výjimka se z této funkce nesmí rozšířit. V těle funkce `h()` může vzniknout jakákoli výjimka, ale z funkce se mohou rozšířit pouze výjimky typu `TVyjimka` a `int`. Je-li `TVyjimka` *třída*, může se z funkce `g()` rozšířit též instance *třídy*, která je veřejně a jednoznačně odvozená z třídy `TVyjimka`. Deklarace typu pomocí `typedef` není podle normy povolena. Poslední příklad představuje deklaraci třídního ukazatele na metodu třídy `TA`, z níž se nesmí rozšířit žádná výjimka.

Jestliže se z funkce se specifikací výjimek rozšíří výjimka typu, který není uveden v seznamu *seznam_typů*, vznikne tzv. *neočekávaná výjimka* (angl. *unexpected exception*). Tato výjimka způsobí ukončení programu – viz dále.

Překladač Visual C++ 2003 sice povoluje specifikaci výjimek v deklaracích funkcí, ale ignoruje je. Překladač zobrazí varování, že specifikaci ignoruje. Při použití specifikace `throw()` překladač varování nezobrazí.

Všechny deklarace včetně definice určité funkce nebo metody musí mít specifikovaný stejný seznam výjimek.

Jestliže virtuální metoda obsahuje specifikaci výjimek, musí tuto specifikaci obsahovat i její předefinované verze v odvozených třídách. Pokud virtuální metoda specifikaci výjimek neobsahuje, předefinovaná virtuální metoda může specifikaci výjimek obsahovat.

Příklad

```
struct TA {
   virtual void d();
   virtual void f() throw (int);
   virtual void g() throw (int);
   virtual void h() throw (int);
};

struct TB : TA {
   virtual void d() throw (int);          // OK
   virtual void f() throw (int, double); // chyba
   virtual void g();                      // chyba
   virtual void h() throw (int);          // OK
};
```

Obdobně, ukazateli na funkci, který obsahuje specifikaci výjimek, lze přiřadit pouze funkci nebo ukazatel na funkci, obsahující stejný seznam výjimek. Ukazateli na funkci, který neobsahuje specifikaci výjimek, se může přiřadit funkce nebo ukazatel na funkci, který specifikaci výjimek obsahuje.

Příklad

```
void (*uf1)(int i);
void (*uf2)(int i) throw (double);
void f()
{
   uf1 = uf2; // OK
   uf2 = uf1; // chyba
}
```

Pokud se ve funkci se specifikací výjimek vyskytne přímo výraz `throw`, vyvolávající nepovolenou výjimku, překladač by měl oznámit varování. Překladač Visual C++ 2003 oznámí varování jen v případě, že se jedná o prázdnou specifikaci výjimek. Např.

```
void g() throw()
{
   throw 1;
}
```

Pokud však ve funkci se specifikací výjimek vzniká výjimka nepřímo, tj. v jiné funkci, kterou tato funkce volá, překladač Visual C++ 2003 ani Borland C++ 6 varování neoznámí. Např.:

```
void h() { throw 1; }
void g() throw () { h(); } // žádné varování
```

VÝJIMKY – POKRAČOVÁNÍ

Výjimky podle normy jazyka C++ – pokračování

Pokusný blok funkce

Má-li být celé tělo funkce součástí pokusného bloku, lze definici funkce napsat např. takto:

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

Norma jazyka C++ však pro tento případ umožňuje definovat funkci tak, že se vypustí složené závorky těla funkce:

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

Konstruktory a destruktory

Pokud vznikne výjimka v konstruktoru instance *třídy*, nebude se pro ni volat destruktor. Budou se ale volat destruktory jejích nestatických složek a předků, které již byly zcela zkonstruovány a nebyla započata jejich destrukce voláním jejich destruktorů. Případné operace destrukce částečně nezkonstruované *třídy* je potřebné provést v handleru v těle konstruktoru.

Příklad

```
class TA {  };
class TB {  };
class TC : public TA { #1 };
class TD : public TB, public TC { #2 };

int main()
{
   try {
      TD D;
   }
   catch (int) {
      // ...
   }
}
```

Pokud vznikne výjimka v těle konstruktoru instance `D` třídy `TD` (#2), budou se volat destruktory pro předky `TA`, `TB` a `TC`. Destruktor instance `D` se nezavolá.

Pokud při konstrukci instance `D` třídy `TD` nastane výjimka v těle konstruktoru předka `TC` (#1), v té chvíli již bude zkonstruován předek `TB` a `TA`. Pro tyto dva předky se tedy zavolají destruktory.

Předek `TC` není ještě zcela zkonstruován, proto se jeho destruktor nezavolá. Také instance `D` ještě není hotova, a proto se nebude volat ani její destruktor.

Únik výjimky z těla konstruktoru nebo destruktoru lze zajistit vložením pokusného bloku do jeho těla. Problém ale nastane, pokud by se výjimka vyvolala z inicializační části konstruktoru. V tomto případě lze použít pokusný blok funkce. Např. v předchozí přednášce deklarovaná třída `TIntVektor` by mohla mít následující definici konstruktoru:

```
TIntVektor::TIntVektor(int _n)
try
:n(_n), a(new int[n])
{
}
catch (...) {
   // ...
}
```

V takovémto případě:

- Handlery pokusného bloku funkce v konstruktoru nesmějí obsahovat příkaz `return`. Překladač Visual C++ 2003 příkaz `return` povolí, ale při běhu programu vznikne chyba.
- Handlery pokusného bloku funkce v konstruktoru nebo destruktoru nesmějí obsahovat příkaz `goto`, kterým se skočí do těla tohoto konstruktoru resp. destruktoru.
- Před vstupem do handleru pokusného bloku funkce v konstruktoru nebo destruktoru jsou plně zkonstruované nestatické složky a předkové instance již zrušeny, takže se v těchto handlerech nemá smysl na ně odkazovat.
- Po ukončení handleru pokusného bloku funkce v konstruktoru nebo destruktoru se zachycená výjimka pošle dál (jako kdyby se v handleru uvedl příkaz `throw;`). To znamená, že výjimku, která se rozšířila z inicializační části konstruktoru, nelze v konstruktoru ošetřit tak, aby se dále nešířila.

Příklad

Je dána třída `TA`, obsahující dvě dynamicky alokovaná pole `a`, `b`.

```
class TA {
   int   *a, *b;
   int   na, nb;
public:
   TA(int _na, int _nb);
   ~TA();
};

TA::TA(int _na, int _nb)
try
   : a(0), b(0), na(_na), nb(_nb)
{
   a = new int[na];
   b = new int[nb];
   // ...
}
catch (...) {
   if (a) delete[] a;
   if (b) delete[] b;
   cout << "chyba 1\n";
}
```

```
TA::~TA()
{
   cout << "destruktor TA\n";
   delete[] a;
   delete[] b;
}

int main()
{
   try {
      TA A(10, 0xEFFFFFFF);
   }
   catch (...) {
      cout << "chyba 2\n";
   }
   return 0;
}
```

Pokud v uvedeném příkladu vznikne výjimka v konstruktoru třídy `TA`, nezavolá se její destruktor, ale přitom mohlo být pole `a` nebo i pole `b` již alokováno. Musí se proto v handleru pokusného bloku funkce v konstruktoru již alokovaná pole dealokovat. K tomu, aby se zjistilo, které pole již bylo alokováno, se v inicializační části konstruktoru nejprve ukazatele `a`, `b` inicializují na nulu. Po provedení handleru pokusného bloku funkce v konstruktoru se výjimka rozšíří do nadřízeného bloku, tj. do funkce `main`, kde se musí opět zachytit a definitivně ošetřit.

Výpis programu bude následující:

```
chyba 1
chyba 2
```

Když vznikne výjimka a program hledá vhodný handler, prochází zásobník s lokálními automatickými proměnnými, ruší je a přitom volá destruktory lokálních automatických instancí objektových typů. Tomu se říká *úklid zásobníku* (angl. *stack unwinding*). V průběhu tohoto procesu se nesmí rozšířit další výjimka.

To znamená, že v destruktoru se v průběhu hledání handleru nesmí rozšířit výjimka. Pokud se to stane, program zavolá funkci `terminate()` a skončí. V destruktoru při úklidu zásobníku ale může vzniknout výjimka, která je v něm ošetřena.

V normě jazyka C++ existuje standardní funkce `uncaught_exception()`, která vrací `true`, pokud je volána v průběhu úklidu zásobníku při výjimce a `false` v ostatních případech. Tuto funkci lze využít v destruktoru. Funkce `uncaught_exception()` je deklarována v hlavičkovém souboru `<exception>` v prostoru jmen `std`.

Příklad

```
class TA {
public:
   ~TA() {
      if (uncaught_exception())
         cout << "uklid zasobniku v destruktoru TA\n";
      else
         cout << "destruktor TA\n";
   }
};
```

```
class TB {
   int b;
public:
   TB(int _b) { if (_b) b = _b; else throw 1;  }
};

int main()
{
   try {
      TA A; TB B(0); // #1
   }
   catch (int i) { cout << "vyjimka " << i << '\n'; }
   try {
      TA A; TB B(1); // #2
   }
   catch (int i) { cout << "vyjimka " << i << '\n'; }
   return 0;
}
```

V pokusném bloku #1 vznikne výjimka v konstruktoru třídy `TB` a začne proces úklidu zásobníku, při němž dojde k zrušení lokální instance `A`. V pokusném bloku #2 výjimka nevznikne. Výpis programu bude následující:

```
uklid zasobniku v destruktoru TA
vyjimka 1
destruktor TA
```

Alokace paměti

Pokud se nepodaří alokace objektu pomocí operátoru `new` a operátor nebyl volán s parametrem `nothrow`, vyvolá se standardní výjimka typu `bad_alloc`.

Pokud se vytváří dynamická instance třídy pomocí operátoru `new`, např.

```
TA* A = new TA();
```

a v konstruktoru třídy `TA` se vyvolá výjimka, paměť kterou již operátor `new` alokoval se automaticky uvolní voláním operátoru `delete`. Totéž platí i pro alokaci polí.

Výjimkový objekt

Po vyvolání výjimky pomocí výrazu `throw` dojde k vytvoření globální kopie objektu, uvedeného za slovem `throw`. Jedná-li se o objektový typ, volá se jeho kopírovací konstruktor. Pokud parametr handleru není reference, objekt se znovu zkopíruje – vytvoří se lokální kopie tohoto globálního objektu. Pro objektový typ se tedy znovu zavolá jeho kopírovací konstruktor. Výjimkový objekt existuje, dokud se neošetří nějakým handlerem. Pokud se v handleru výjimka znovu nevyvolá příkazem `throw;`, po ukončení handleru se globální objekt zruší, tj. pro objektový typ se zavolá jeho destruktor. Příkazem `throw;` se nevytváří nový výjimkový objekt.

Pokud se nepodaří zkonstruovat globální instance objektového typu výjimky, program zavolá funkci `terminate()` a skončí. Globální instance totiž nelze uzavřít do pokusného bloku.

Příklad

Je dána třída `TVyjimka`, obsahující dynamicky alokovaný řetězec textu výjimky. Třída `TVyjimka` musí proto obsahovat kopírovací konstruktor, destruktor a kopírovací operátor přiřazení.

```
class TVyjimka {
   char* s;
public:
   TVyjimka(const char* _s) {
      s = new char[strlen(_s)+1]; strcpy(s, _s);
      cout << "Konstruktor\n"; }
   TVyjimka(const TVyjimka& t) {
      s = new char[strlen(t.s)+1]; strcpy(s, t.s);
      cout << "Kopirovaci konstruktor\n"; }
   ~TVyjimka() { delete [] s; }
   // je nutné též definovat kopírovací operátor přiřazení
   void Vypis() const { cout << s << '\n'; }
};

int f(int a, int b)
{
   try {
      if (b == 0) throw TVyjimka("Deleni nulou"); // #1
      return a/b;
   }
   catch (const TVyjimka& t) {
      // ...
      throw;
   }
}
int main()
{
   try {
      cout << f(10, 0);
   }
   catch (const TVyjimka& t) {
      t.Vypis();
   }
   system("pause");
   return 0;
}
```

V uvedeném příkladu dojde k vyvolání výjimky ve funkci `f`. Příkaz #1 v překladači Visual C++ 2003 způsobí volání pouze konstruktoru `TVyjimka(const char* _s)`. V překladači Borland C++ 6 dojde navíc k vyvolání kopírovacího konstruktoru.

Pokud by se místo příkazu #1 uvedly příkazy

```
if (b == 0) { TVyjimka t("Deleni nulou"); throw t; }
```

dojde ve výrazu `throw t` k volání kopírovacího konstruktoru třídy `TVyjimka` v obou překladačích.

Neošetřené a neočekávané výjimky

Neošetřená výjimka (angl. *unhandled exception*) vznikne, pokud:

- výjimku nezachytí žádný z handlerů,
- se rozšíří výjimka z destruktoru během úklidu zásobníku,
- se rozšíří výjimka z konstruktoru nebo destruktoru globální instance *třídy*.

V takovýchto případech program zavolá funkci

```
void terminate();
```

Funkce `terminate()` normálně zavolá funkci

```
void abort();
```

která ukončí program a vypíše nějaké hlášení, např. „Abnormal program termination“.

Chování funkce `terminate()` lze však změnit pomocí standardní funkce `set_terminate`:

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();
```

Této funkci se předá ukazatel na funkci, která se zavolá místo funkce `abort`. Funkce `set_terminate()` vrací ukazatel na předchozí funkci. Funkce volaná místo funkce `abort` musí ukončit program bez návratu do volané funkce.

Pokud se z nějaké funkce rozšíří výjimka, která není uvedena ve specifikaci výjimek této funkce, jedná se o *neočekávanou výjimku* (angl. *unexpected exception*). V případě vzniku neočekávané výjimky program zavolá funkci

```
void unexpected();
```

která normálně zavolá funkci `terminate()`. Pomocí funkce `set_unexpected` lze předepsat volání jiné funkce:

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();
```

Funkce volaná místo funkce `unexpected()` nesmí obsahovat příkaz `return`. Může však obsahovat výraz `throw`, tj. místo ukončení programu může výjimku transformovat v jinou. Pokud tato výjimka je uvedena ve specifikaci výjimek funkce, kvůli níž neočekávaná výjimka vznikla, bude program pokračovat v hledání vhodného handleru. Jestliže ve specifikaci výjimek nově vyvolaná výjimka není uvedena:

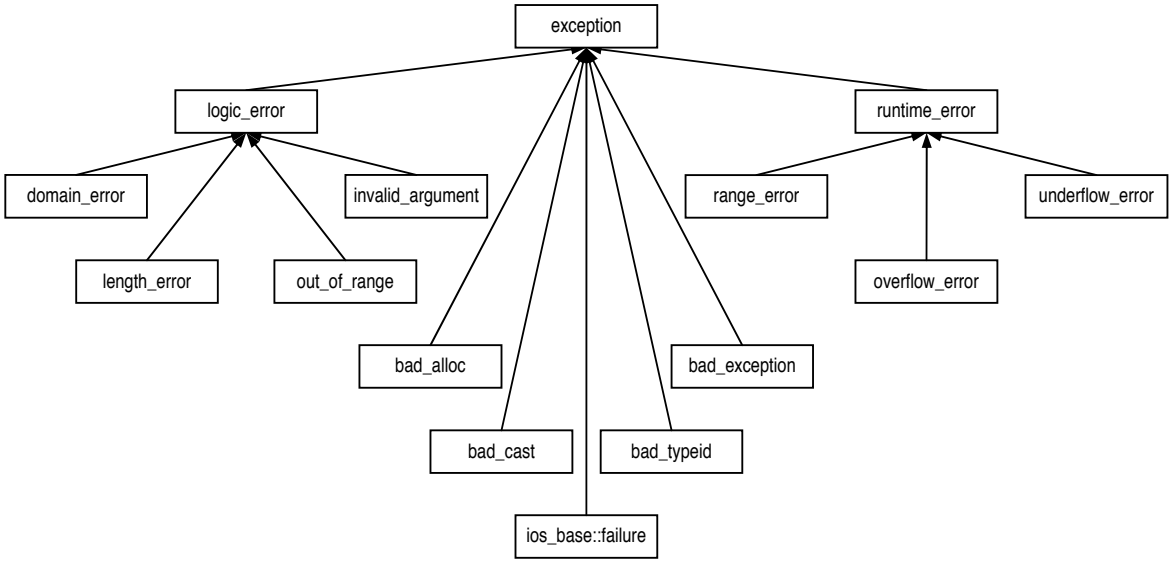
- a specifikace výjimek obsahuje standardní výjimku typu `bad_exception`, nahradí se nově vyvolaná výjimka výjimkou `bad_exception` a program začne hledat vhodný handler;
- a specifikace výjimek neobsahuje standardní výjimku typu `bad_exception`, zavolá se funkce `terminate()`.

Překladač Visual C++ 2003 neočekávané výjimky nepodporuje. Překladač C++ Builder 6 neočekávané výjimky sice podporuje, ale nepodporuje transformaci výjimky na výjimku typu `bad_exception`.

Prototypy funkcí `set_terminate()`, `set_unexpected()` a další nástroje používané při práci s výjimkami jsou uvedeny v hlavičkovém souboru `<exception>` v prostoru jmen `std`.

Standardní výjimky

Funkce a metody, které jsou uvedeny ve standardní knihovně C++, jakož i některé operace, prováděné v jazyce C++, mohou vyvolávat výjimky různých typů (tříd), odvozených od společného předka `exception`. Jejich dědická hierarchie je znázorněna na následujícím obrázku.



Hierarchie standardních tříd výjimek

Třída `exception` má být v hlavičkovém souboru `<exception>` deklarována takto:

```
class exception {
public:
   exception() throw();
   exception(const exception&) throw();
   exception& operator = (const exception&) throw();
   virtual ~exception() throw();
   virtual const char* what() const throw();
};
```

Metoda `what()` může vracet nulově zakončený jedno- nebo dvou-bytový řetězec znaků popisující typ výjimky.

Význam dalších výjimek je následující:

- `bad_alloc` (hlavičkový soubor `<new>`) – vyvolá se v případě neúspěšné alokace pomocí operátoru `new`, který nebyl volán s parametrem `nothrow`. Obsahuje stejné metody, jako třída `exception`.
- `bad_cast` (hlavičkový soubor `<typeinfo>`) – vyvolá se, pokud se nemůže provést přetypování reference na jednu třídu na referenci na jinou třídu pomocí operátoru `dynamic_cast`. Obsahuje stejné metody, jako třída `exception`.
- `bad_exception` (hlavičkový soubor `<exception>`) – význam viz kapitola „Neošetřené a neočekávané výjimky“. Obsahuje stejné metody, jako třída `exception`.
- `bad_typeid` (hlavičkový soubor `<typeinfo>`) – vznikne, pokud se operátoru `typeid` zadá dereferencovaný ukazatel s hodnotou nula.
- `ios_base::failure` (hlavičkový soubor `<ios>`) – základní třída pro výjimky, vzniklé při operacích vstupu a výstupu pomocí objektových datových proudů. Oproti třídě `exception`,

obsahuje navíc explicitní konstruktor s parametrem typu `string` (řetězec znaků). Obsah tohoto řetězce vypíše virtuální metoda `what()`.

- `logic_error` (hlavičkový soubor `<stdexcept>`) – základní třída pro skupinu logických chyb, které jsou důsledkem chyb v interní logice programu. Těmto chybám lze zabránit.
- `runtime_error` (hlavičkový soubor `<stdexcept>`) – základní třída pro skupinu běhových chyb. Jedná se o chyby, které nelze předvídat, neboť jsou důsledkem poruch periferních zařízení, chyb uživatele programu apod.

Všechny třídy odvozené z třídy `logic_error` nebo `runtime_error` včetně těchto tříd jsou deklarovány v hlavičkovém souboru `<stdexcept>` a mají oproti třídě `exception` explicitní konstruktor s parametrem typu `string` (řetězec znaků). Obsah tohoto řetězce vypíše virtuální metoda `what()`.

Třídy odvozené z třídy `logic_error`:

- `domain_error` – vyvolá se při vzniku doménové chyby. Např. ve třídě `numpunct` a `moneypunct` při převodu řetězce znaků na číslo.
- `invalid_argument` – je vyvolána z různých metod a funkcí, kterým byl předán neplatný parametr.
- `length_error` – vznikne při pokusu vytvořit objekt, jehož požadovaná velikost překračuje maximální povolenou velikost. Např. ve třídě `string` se vyvolá tato výjimka, pokud požadovaný počet znaků řetězce překračuje maximální možnou velikost řetězce.
- `out_of_range` – vyvolá se v případě, kdy hodnota skutečného parametru metody nebo funkce je mimo očekávaný rozsah. Např. ve třídě `string` se vyvolá tato výjimka, pokud se metodě `insert` předá pozice, na kterou se má vložit jiný řetězec, jejíž hodnota je mimo rozsah 0 až (počet znaků řetězce – 1).

Třídy odvozené z třídy `runtime_error`:

- `overflow_error` – vyvolá se, pokud dojde k aritmetickému přetečení. Např. v šabloně třídy `bitset<N>` (bitová množina) metoda `to_ulong()` vyvolá tuto výjimku, pokud počet bitů množiny překračuje počet bitů datového typu `unsigned long`.
- `range_error` – vyvolá se, pokud dojde k chybě v rozsahu při interních výpočtech.
- `underflow_error` – vyvolá se, pokud dojde k aritmetickému podtečení.

Cena výjimek

Výjimky poskytují jazyku C++ aparát, pomocí něhož lze elegantně zvládat chybové situace. Není to ovšem zadarmo. Přeložené programy s výjimkami jsou rozsáhlejší, i když mohou být rychlejší. Obsahují několik tabulek, do kterých si program bude ukládat potřebné informace. Navíc obsahuje nové standardní funkce, které práci s výjimkami umožňují. Také ošetření zásobníku u funkcí se specifikovanými výjimkami je poněkud složitější.

To vše je cena, která se musí zaplatit, a to i v případě, že v programu ve skutečnosti žádná výjimka nenastane.

Jestliže dojde k výjimce, musí program:

- vyhledat podle typu výjimky odpovídající handler
- uklidit zásobník
- při rozšíření výjimky z funkce zkontrolovat, zda typ výjimky odpovídá specifikaci dovolených typů.

To vše je časově náročnější, proto řada současných překladačů C++ umožňuje používání výjimek zakázat.

Strukturované výjimky

Strukturované výjimky (angl. structured exception handling, zkr. SEH), byly navrženy firmou Microsoft pro aplikace spustitelné pod 32-bitovými operačními systémy Windows. Strukturované výjimky jsou součástí Win32 API a obsahují mechanismus pro práci se synchronními (softwarovými) i asynchronními (hardwarovými) výjimkami. Jsou primárně určeny pro jazyk C, ale lze je využívat i v jazyce C++.

Strukturované výjimky umožňují jak zachytit a ošetřit výjimku, tak i předepsat *koncovku* pokusného bloku (angl. *termination handler*) – skupinu operací, která se provede vždy na závěr bloku, bez ohledu na to, jakým způsobem je opouštěn.

Práce se strukturovanými výjimkami je opět založena na *pokusném bloku*, který je složen z klíčového slova `__try`, *složeného příkazu pokusného bloku* a *handleru* nebo *koncovky*. Za složeným příkazem pokusného bloku může být uveden buď jen jeden handler nebo jedna koncovka.

Vznikle-li výjimka, přeruší se operace ve složeném příkazu pokusného bloku a systém bude hledat vhodný handler. Pokud ho nenajde v bloku, ve kterém výjimka vznikla, přejde do bloku nadřízeného. Podobně, pokud se vhodný handler nenajde ve funkci, ve které výjimka vznikla, rozšíří se výjimka do funkce, která ji volala atd. Narozdíl od výjimek v C++ operace ve složeném příkazu pokusného bloku se pouze přerušily a za jistých okolností se k nim lze vrátit.

Program postupně zkouší všechny handlery, na které při šíření výjimky narazí. Každý handler obsahuje *vstupní filtr*. To je výraz, který se při vstupu do handleru vyhodnotí a podle něj se program rozhodne pro některou z následujících možností:

- Handler výjimku ošetří. Pokud tím neukončí program, vrátí se pak řízení za blok, který výjimku ošetřil. Pokud není handler v téže funkci jako místo, kde výjimka vznikla, ošetří se také řádně zásobník, tj. zruší se lokální automatické proměnné. Volání destruktorů závisí na podpoře strukturovaných výjimek překladačem C++.
- Handler výjimku odmítne. Výjimka se bude šířit dále, systém bude hledat vhodný handler v nadřízených blocích.
- Handler rozhodne, že nejde o chybu a přikáže výjimku ignorovat. Řízení se po vyhodnocení vstupním filtrem vrátí na místo, kde výjimka vznikla.

Překladač Borland C++ a Visual C++ 2003 umožňují použití pokusného bloku s koncovkou i pro výjimky jazyka C++.

Syntaxe strukturované výjimky s koncovkou

pokusný_blok:

__try *složený_příkaz koncovka*

koncovka:

__finally *složený_příkaz*

Pokusný blok strukturované výjimky začíná klíčovým slovem `__try`, koncovku označuje klíčové slovo `__finally`. Obě tato klíčová slova začínají dvěma podtržítky.

Výjimka způsobí předčasné ukončení příkazů pokusného bloku, což může mít za následek např. neuvolnění alokované paměti, neuzavření souborů apod. Tento problém řeší koncovka pokusného bloku, která se provede (téměř) vždy.

Koncovka se provede, jestliže složený příkaz pokusného bloku skončí:

- normálně, tedy provedením posledního příkazu složeného příkazu pokusného bloku – řízení přešlo řádně přes uzavírací složenou závorku složeného příkazu pokusného bloku,
- vyvoláním výjimky (strukturované výjimky i výjimky jazyka C++)
- provedení některého z příkazů k přesunu řízení: `break`, `continue`, `goto`, `return`,

- voláním funkce `longjmp`.

Koncovka pokusného bloku se neprovede, jestliže se v něm zavolá některá z funkcí pro ukončení běhu programu nebo vlákna (threadu), např. `ExitThread`, `ExitProcess`, `TerminateThread`, `TerminateProcess`, `exit` nebo `terminate`.

V překladači Borland C++ 6 může pokusný blok začínat i klíčovým slovem `try`.

Příklad

```
int main()
{
   FILE *f;
   int Pole[100];
   f = fopen("data.txt", "rt");
   __try {
      NactiPole(Pole, f); // v této funkci může vzniknout výjimka
   }
   __finally {
      fclose(f);
   }
   // ...
   return 0;
}
```

Příkaz `fclose(f);` se provede, ať ve funkci `NactiPole` vznikne výjimka nebo ne.

V bloku koncovky lze použít funkci:

```
int AbnormalTermination();
```

Ta vrací 0, pokud složený příkaz pokusného bloku, ke kterému se koncovka vztahuje, skončil normálně a nenulovou hodnotu, pokud skončil abnormálně. Za normální ukončení se považuje případ uvedený v první odrážce výše uvedeného seznamu, ostatní tři odrážky se považují za abnormální ukončení (tj. výjimka, příkaz `break`, `continue`, `goto`, `return`, funkce `longjmp`). Funkce `AbnormalTermination` má být klíčovým slovem, ale překladač Visual C++ 2003 ji nezná. Musí se zahrnout hlavičkový soubor `<windows.h>`.

V překladači Visual C++ 2003 existuje klíčové slovo `__leave`, které se může vyskytnout jako příkaz jen ve složeném příkazu pokusného bloku s koncovkou a způsobí okamžité ukončení pokusného bloku a přechod do koncovky. Tento příkaz se považuje za normální ukončení složeného příkazu pokusného bloku, tj. funkce `AbnormalTermination` vrací 0.

Příklad

```
int main()
{
   FILE *f;
   int *Pole;
   __try {
      f = fopen("data.txt", "rt");
      Pole = new int[20];
      NactiPole(Pole, 20, f); // v této funkci může vzniknout výjimka
   }
   __finally {
      if (AbnormalTermination()) delete[] Pole;
      fclose(f);
   }
   // ...
   return 0;
}
```

Pokud složený příkaz pokusného bloku skončí normálně, zavře se pouze soubor, pokud však skončí výjimkou, dealokuje se navíc i pole.

Univerzální handler pokusného bloku jazyka C++ při použití překladače Visual C++ 2003 zachytává i asynchronní výjimky. Překladač Borland C++ 6 toto nepodporuje.

Příklad

```
int main()
{
   int a, b, c;
   try {
      cout << "Zadej dve cisla: ";
      cin >> a >> b;
      c = a/b; // může vzniknout asynchronní výjimka dělení nulou
      cout << "Jejich podil je: " << c << '\n';
   }
   catch (...) { // #1
      cout << "Vyjimka\n";
   }
   return 0;
}
```

V případě, že proměnná `b` bude mít nulovou hodnotu, výraz `a/b` způsobí asynchronní výjimku celočíselného dělení nulou, kterou v případě použití překladače Visual C++ 2003 zachytí univerzální handler #1.

Překladač Visual C++ 2003 má následující omezení v použití strukturovaných výjimek (SEH) v C++:

- V jedné funkci nemůže být uveden pokusný blok SEH (__try __except, __try __finally) spolu s pokusným blokem C++ (try catch).
- V pokusném bloku SEH nemůže být alokován objektový typ.

  Např. v pokusném bloku SEH nemůže být uveden příkaz

  ```
  Pole = new TA[10];
  ```

  pokud `TA` je objektový typ.

  Takovýto příkaz ale může být uveden ve funkci, která se zavolá z pokusného bloku SEH.

DYNAMICKÁ IDENTIFIKACE TYPŮ, PŘETYPOVÁNÍ

Vztah mezi cv-modifikátory

V následujícím textu je uveden operátor < porovnávající dva cv-modifikátory. Množina cv-modifikátorů `cv1` je menší než `cv2`, pokud platí některá z těchto podmínek:

cv1	<	*cv2*
žádný cv-modifikátor	<	`const`
žádný cv-modifikátor	<	`volatile`
žádný cv-modifikátor	<	`const volatile`
`const`	<	`const volatile`
`volatile`	<	`const volatile`

Dynamická identifikace typů

Dynamická identifikace typů (angl. *runtime-type identification* nebo *runtime-type information*, zkr. RTTI) slouží k zjišťování skutečného typu proměnných nebo výrazů za běhu programu.

RTTI se opírá o:

- operátor `typeid` – umožňuje určit typ objektu za běhu programu,
- operátor `dynamic_cast`,
- třídu `type_info`.

Operátor `typeid` lze použít jen s vložením hlavičkového souboru `<typeinfo>`. Syntaxe jeho použití je následující:

typeid(*výraz***)**

typeid(*označení_typu***)**

V prvním případě se zjišťuje typ *výrazu*. Nejčastěji se bude jednat o dereferencovaný ukazatel nebo referenci na nějakou instanci. Ve druhém případě se výsledek používá zejména při porovnávání.

Použije-li se operátor `typeid` na výraz, který představuje hodnotu neobjektového typu nebo instanci nepolymorfního objektového typu, vyhodnotí se již v době překladu. Použije-li se na dereferencovaný ukazatel na polymorfní typ nebo na referenci na instanci polymorfního typu, bude se vyhodnocovat dynamicky, tj. až za běhu programu a výsledkem je typ skutečného objektu, na který se ukazatel nebo reference odkazuje. Je-li parametrem operátoru dereferencovaný ukazatel s hodnotou 0, který ukazuje na polymorfní typ, operátor vyvolá výjimku typu `bad_typeid`.

Operátor `typeid` vrací konstantní referenci na instanci třídy `type_info`, jejíž deklarace je podle normy následující:

```
class type_info {
private:
   type_info(const type_info& rhs);
   type_info& operator=(const type_info& rhs);
public:
   virtual ~type_info();
   bool operator == (const type_info& rhs) const;
   bool operator != (const type_info& rhs) const;
   bool before(const type_info& rhs) const;
   const char* name() const;
};
```

Operátory `==` a `!=` slouží pro porovnávání výsledků operátorů `typeid`. Metoda `name()` vrací ukazatel na řetězec znaků představující označení typu. Metoda `before()` vrací `true`, jestliže typ `*this` je před typem `rhs` z hlediska lexikálního pořadí označení typů, tj. porovnává řetězce označení typů podle abecedy.

Kopírovací konstruktor a kopírovací operátor přiřazení jsou soukromé metody, tudíž nelze instanci této třídy kopírovat. Ve specifikaci normy není uveden nekopírovací konstruktor, ale překladače Visual C++ 2003 i C++ Builder 6 nepovolují vytvoření instance této třídy.

Operátor `typeid` ignoruje cv-modifikátory na nejvyšší úrovni typu výrazu. Je-li např. definována třída `TA` a její instance `A` a `B`

```
struct TA { int a, b; };
TA A;
const TA B = { 10, 20 };
const TA& BR = B;
```

následující výrazy budou nabývat hodnot uvedených v komentářích:

```
typeid(A) == typeid(B);                 // true
typeid(TA) == typeid(const TA);         // true
typeid(TA) == typeid(B);                // true
typeid(&A) == typeid(const TA*);        // false
typeid(&B) == typeid(const TA*);        // true
typeid(&B) == typeid(const TA* const); // true
typeid(A) == typeid(BR);                // true
```

Příklad

V uvedeném příkladu je definována třída `TZakladni`, která je předkem třídy `TOdvozena`. Obě třídy jsou polymorfní, protože obsahují virtuální metodu `f()`.

```
class TZakladni {
public:
   int a;
   virtual int f() const { return a; }
};
class TOdvozena : public TZakladni {
public:
   int b;
   int f() const { return a+b; }
};
```

```
int main()
{
   TZakladni Zakladni;
   TOdvozena Odvozena;
   TZakladni *u1 = &Zakladni, *u2 = &Odvozena;
   try {
      cout << "u2 ukazuje na objekt typu " << typeid(*u2).name() << '\n';
      cout << "u2 je typu " << typeid(u2).name() << '\n';
      cout << "Prvni v abecede je trida, na kterou ukazuje " <<
               (typeid(*u1).before(typeid(*u2)) ? "u1" : "u2") << '\n';
      u2 = 0;
      cout << "u2 ukazuje na objekt typu " << typeid(*u2).name() << '\n';
   }
   catch (bad_typeid&) {
      cout << "Dereferencovani nuloveho ukazatele v typeid\n";
   }
   return 0;
}
```

Program vypíše následující texty:

```
u2 ukazuje na objekt typu class TOdvozena
u2 je typu class TZakladni *
Prvni v abecede je trida, na kterou ukazuje u2
Dereferencovani nuloveho ukazatele v typeid
```

Jestliže funkce `f()` nebude virtuální, bude se typ dereferencovaného ukazatele určovat staticky v době překladu a výstup programu bude následující:

```
u2 ukazuje na objekt typu class TZakladni
u2 je typu class TZakladni *
Prvni v abecede je trida, na kterou ukazuje u2
u2 ukazuje na objekt typu class TZakladni
```

Výjimka typu `bad_typeid` se v tomto případě nevyvolá.

Překladače umožňují zapnutí nebo vypnutí použití RTTI. Je-li použití RTTI zapnuto, program je delší.

Přetypování

Norma jazyka C++ nabízí kromě klasického operátoru *(typ)*, převzatého z jazyka C, čtyři nové operátory `const_cast`, `dynamic_cast`, `static_cast` a `reinterpret_cast`. Tyto operátory, si rozdělují úlohy klasického operátoru *(typ)* a nabízí další možnosti. Doporučuje se používat nové operátory místo klasického operátoru mj. z důvodu zpřehlednění programu a jednoduššího vyhledávání operací přetypování v programech.

Syntaxe použití všech těchto operátorů je stejná, např. pro operátor `static_cast`:

static_cast<*označení_typu***>(***výraz***)**

Za klíčovým slovem operátoru je v lomených závorkách označení cílového typu, za nímž následuje v kulatých závorkách přetypovávaný výraz.

Operátor static_cast

Slouží pro běžná přetypování, která mohou proběhnout v C++ automaticky a pro některé další případy. Přesněji řečeno, výraz `static_cast<T>(v)` je správný, je-li správná deklarace s inicializací:

```
T t(v);
```

Efekt takovéto explicitní konverze je stejný jako kdyby se provedla deklarace proměnné `t` s inicializací výrazem `v` a potom by se dočasná proměnná `t` použila jako výsledek konverze. Je-li typ `T` reference, výsledkem operátoru je l-hodnota, jinak je výsledkem r-hodnota.

Operátor `static_cast` lze tedy použít např. ke konverzi:

- potomka na předka, ukazatele na potomka na ukazatel na předka apod. pokud je taková konverze dovolena (předek je jednoznačný a v místě konverze přístupný),
- výčtového typu na celé číslo,
- ukazatele nebo čísla na logickou hodnotu,
- mezi různými číselnými typy,
- pole určitých typů na ukazatel téhož typu,
- z určitého typu na objektový typ, který má definován odpovídající konverzní konstruktor.

Příklad

```
void f(int);
void f(double);
int i;

f(static_cast<double>(i));
```

Proměnná `i` se přetypovala na typ `double`, aby se mohla zavolat funkce `f(double)`.

Dále lze operátor `static_cast` použít k následujícím nestandardním konverzím:

- Libovolnou hodnotu lze konvertovat na typ `void`. Tím se hodnota `v` „zahodí“.
- Celé číslo lze konvertovat na výčtový typ. Pokud leží konvertovaná hodnota v rozsahu cílového typu, nezmění se, jinak je výsledná hodnota nedefinovaná.
- Lze konvertovat nevirtuálního předka na potomka, pokud je správná i opačná konverze. Přesněji, l-hodnotu typu „*cv1* `TA`“, kde `TA` je objektový typ, lze konvertovat na typ „reference na *cv2* `TB`“, kde `TB` je potomek třídy `TA`, pokud je dovolena standardní konverze z typu „ukazatel na `TB`“ na „ukazatel na `TA`“ a `TA` není virtuální předek třídy `TB` a zároveň *cv2* `>=` *cv1*, kde *cv1* a *cv2* jsou cv-modifikátory. Výsledkem je l-hodnota typu „*cv2* `TB`“. Obdobné pravidlo platí pro konverzi ukazatelů, výsledkem je potom r-hodnota typu „ukazatel na *cv2* `TB`“.
- Výraz typu „třídní ukazatel v `TB` na typ *cv1* `T`“ lze konvertovat na hodnotu typu „třídní ukazatel v `TA` na typ *cv2* `T`“, pokud je přípustná standardní konverze z typu „třídní ukazatel v `TA` na typ `T`“na typ „třídní ukazatel v `TB` na typ `T`“ a pokud `TA` není virtuální předek `TB` nebo virtuální předek předka `TB` a zároveň *cv2* `>=` *cv1*. Konverze má smysl, pokud konvertovaný výraz ukazuje na složku, která je obsažena jak v třídě `TB`, tak i v `TA`.

Nelze konvertovat ukazatel na pole, logickou hodnotu na ukazatel apod.

Příklad

```
class TA {
protected:
   int a;
public:
   TA(int _a) : a(_a) {}
};

class TB {
protected:
   int b;
public:
   TB(int _b) : b(_b) {}
};

class TC : public TA, private TB {
public:
   TC(int i) : TA(i), TB(i*10) {}
};

void f(TB& B) { ... }
void f(TC& C) { ... }
void f(TC* C) { ... }

int main()
{
   TA A(5), *UA = &A;
   TC C(4);
   TA *UA2 = &C, &RA = C;

   f(static_cast<TC*>(UA2)); // #1
   f(static_cast<TC*>(UA));  // #2
   f(static_cast<TC&>(RA));  // #3
   f(static_cast<TC&>(A));   // #4
   f(static_cast<TB&>(C));   // #5 – Chyba
   f(static_cast<TC>(A));    // #6 - Chyba
   return 0;
}
```

V uvedeném příkladu jsou definovány třídy `TA`, `TB` a `TC`. Třída `TC` má veřejně přístupného předka `TA` a soukromého předka `TB`.

V příkazu #1 se přetypovává ukazatel na třídu `TA` na ukazatel na třídu `TC`, tedy ukazatel na předka na ukazatel na potomka. Protože třída `TA` je jednoznačným, veřejně přístupným nevirtuálním předkem třídy `TC`, je toto přetypování možné. Navíc má smysl, protože `UA2` obsahuje ve skutečnosti adresu instance třídy `TC`.

Přetypování v příkazu #2 je sice formálně správné (překladač neoznámí chybu), důsledek může být však tragický, protože `UA` obsahuje ve skutečnosti adresu instance třídy `TA` a přetypování na potomka je nesmyslné. Vlastní přetypování za běhu programu chybu neoznámí, ale složky potomka budou mít nedefinované hodnoty.

V příkazu #3 se provádí přetypování reference `RA` na podobjekt `TA` na referenci na celý objekt `TC`. Přetypování je v pořádku formálně i logicky.

Příkaz #4 provádí přetypování samostatné instance `A` na referenci na `TC`, což je formálně správné, ale podobně jako v příkazu #2 může vést k těžko odhalitelným chybám za běhu programu.

Přetypování v příkazu #5 je chybné. Překladač oznámí chybu, protože třída `TB` je soukromý, a tudíž ve funkci `main()` nepřístupný, předek.

Příkaz #6 není správný, protože nelze přetypovat potomka na předka bez použití reference nebo ukazatele.

Operátor dynamic_cast

Operátor lze použít pouze k přetypování na ukazatele nebo referenci v rámci dědické hierarchie objektových typů.

Výraz `dynamic_cast<T>(v)` provede konverzi hodnoty výrazu `v` na typ `T`. Cílový typ `T` musí být ukazatel nebo reference na plně definovaný objektový typ nebo `void*`.

Je-li `T` ukazatel, hodnotou výrazu `v` musí být ukazatel na plně definovanou třídu. Výsledkem bude r-hodnota typu `T`.

Je-li `T` reference, výraz `v` musí být l-hodnota plně definované třídy a výsledkem bude l-hodnota typu, na který se odkazuje `T`.

Je-li cílový typ `T` stejný jako typ hodnoty výrazu `v`, nic se nestane.

Je-li výsledkem výrazu `v` ukazatel s hodnotou 0, bude výsledkem přetypování ukazatel s hodnotou 0 typu `T`.

Operátor `dynamic_cast` lze použít k obyčejnému přetypování potomka na předka. Tedy, je-li `T` typ „ukazatel na *`cv1`* `TA`“ a `v` je typu „ukazatel na *`cv2`* `TB`“, přičemž `TA` je předkem `TB`, je výsledkem ukazatel na jediný podobjekt typu `TA` v objektu typu `TB`, na který ukazuje `v`. Podobně, je-li `T` „reference na *`cv1`* `TA`“ a `v` je typu „*`cv2`* `TB`“, přičemž `TA` je předkem `TB`, je výsledkem l-hodnota jediného podobjektu typu `TA` v objektu typu `TB`, na který se odkazuje `v`. V obou případech (ukazatele i reference) musí být třída `TA` jednoznačným přístupným předkem třídy `TB` a zároveň *`cv2`* `<=` *`cv1`*.

Ve všech dalších případech musí být `v` ukazatel na polymorfní typ nebo l-hodnota polymorfního typu.

Je-li `T` typ `void*`, výraz `v` musí být ukazatel. Výsledkem bude ukazatel na celý objekt (potomek na nejvyšším stupni dědické hierarchie), na který ukazuje `v`. To znamená, že se číselná hodnota ukazatele může změnit. Zde se operátor `dynamic_cast` liší od operátoru `static_cast` nebo klasického operátoru *(typ)*, které vrací ukazatel s nezměněnou hodnotou, pouze se změní typ na `void*`. Jinak se použije RTTI za účelem zjištění, zda je možné požadovanou konverzi provést.

Postup přetypování pomocí RTTI je následující:

- Je-li výraz `v` ukazatel na jediný zděděný veřejně přístupný podobjekt v objektu typu `T`, bude výsledkem ukazatel na tento objekt typu `T`. Totéž platí pro reference.
- Pokud `v` ukazuje na zděděný veřejně přístupný podobjekt nějakého celého objektu, který obsahuje jednoznačného veřejně přístupného předka typu `T`, výsledkem je podobjekt typu `T` v celém objektu, na nějž ukazuje `v`. Totéž platí pro reference.
- Jinak se přetypování nepodaří.

Pokud se přetypování nepodaří, operátor `dynamic_cast`:

- vrátí hodnotu 0, je-li cílovým typem `T` ukazatel;
- vyvolá výjimku typu `bad_cast`, je-li cílovým typem `T` reference.

Příklad

```
class TA {
protected:
   int a;
public:
   TA(int _a) : a(_a) {}
   virtual void Hlaseni() { cout << "Trida TA\n"; }
};

class TB {
protected:
   int b;
public:
   TB(int _b) : b(_b) {}
   virtual void Hlaseni() { cout << "Trida TB\n"; }
   void f() { cout << b << '\n'; }
};

class TC : public TA, public TB {
public:
   TC(int i) : TA(i), TB(i*10) {}
   void Hlaseni() { cout << "Trida TC\n"; }
   void g() { cout << a << '\t' << b << '\n'; }
};

void h(TB* B)
{
   TC* C = dynamic_cast<TC*>(B);
   if (C) C->g();
}

void h2(TA& A)
{
   try {
      dynamic_cast<TB&>(A).f();
   }
   catch (bad_cast) {
      cout << "Vyjimka typu bad_cast\n";
   }
}

int main()
{
   TA A(1);
   TB B(2), *UB = &B;
   TC C(3), *UC = &C;
   h(UB); // #1 - metoda g() se nezavolá
   UB = UC;
   h(UB); // #2
   h2(C); // #3
   h2(A); // #4 - vyvolá výjimku
   cout << static_cast<void*>(UB) << '\n'; \\ #5
   cout << dynamic_cast<void*>(UB) << '\n'; \\ #6
   return 0;
}
```

V uvedeném příkladu jsou definovány polymorfní třídy `TA`, `TB` a `TC`. Třída `TC` má veřejně přístupné předky `TA` a `TB`.

Funkce `h` má formální parametr typu ukazatel na `TB`, který se snaží přetypovat na ukazatel na potomka `TC`. Pokud se přetypování podaří, zavolá se metoda `g()` třídy `TC`. V příkazu #1 se volá funkce `h` s parametrem `UB`, který ve skutečnosti ukazuje na instanci `B` třídy `TB` a tudíž se přetypování ve funkci `h` nepodaří. V příkazu #2 přetypování proběhne úspěšně, protože parametr `UB` ve skutečnosti ukazuje na instanci `C` třídy `TC`.

Funkce `h2` má parametr typu reference na `TA`, který se snaží přetypovat na referenci na `TB`, pro níž se volá metoda `f()`. Pokud se přetypování nepodaří, vyvolá se výjimka, která se v této funkci ošetří. Příkaz #3 proběhne úspěšně – instanci `C` třídy `TC` lze přetypovat na podobjekt `TB`. V příkazu #4 se přetypování nepodaří a vyvolá se výjimka.

Příkazy #5 a #6 vypisují adresu ukazatele `UB` přetypovaného na `void*`. Pomocí `static_cast` se vypíše současná adresa ukazatele `UB`, zatímco s použitím operátoru `dynamic_cast` se vypíše adresa celého objektu, na který `UB` ukazuje, tedy adresa instance `C`.

Pokud by metoda `Hlaseni()` třídy `TB` nebyla virtuální, překladač by oznámil chybu při použití operátoru `dynamic_cast` ve funkci `h` a v příkazu #6.

Operátor const_cast

Slouží pro přidání nebo odebrání cv-modifikátorů. V přetypování `const_cast<T>(v)` se cílový typ `T` smí lišit od výrazu `v` jen v cv-modifikátorech.

Příklad

```
class TA {
   int a;
public:
   TA(int _a) : a(_a) {}
   int f() { return a += 10; }
   int f() const { return a+10; }
};

int main()
{
   TA A(10);
   cout << A.f() << '\n';
   cout << const_cast<const TA&>(A).f() << '\n';
   return 0;
}
```

Třída `TA` obsahuje 2 metody `f()`, jednu z nich pro konstantní a druhou pro nekonstantní instance. Má-li se zavolat konstantní metoda `f()` pro nekonstantní instanci `A`, musí se použít přetypování pomocí `const_cast`.

Operátor reinterpret_cast

Operátor slouží k různým „nečistým“ přetypováním, např. přetypování ukazatele na celé číslo (pokud se ukazatel do celého čísla vejde), převod čísla na ukazatel, převod ukazatele na jednu třídu na ukazatel na úplně jinou nesouvisející třídu, převod ukazatele na oblast dat na ukazatel na funkci apod.

Příklad

```
struct TA { int a, b; };
struct TB { int c, d; };

void Vypis(const TA& A) { cout << A.a << ", " << A.b << '\n'; }

int main()
{
   TB B = { 10, 20 };
   Vypis(reinterpret_cast<const TA&>(B));
   return 0;
}
```

Struktury `TA` a `TB` nemají mezi sebou žádnou souvislost. Protože však mají stejnou strukturu, může mít smysl použít k výpisu jejich atributů stejnou funkci `Vypis`, která požaduje parametr typu konstantní reference na `TA`. Pokud se má funkce `Vypis` volat pro instanci `B` třídy `TB`, musí se instance `B` přetypovat na konstantní referenci na `TA`.

VSTUPY A VÝSTUPY

Jazyk C++ obsahuje všechny nástroje pro vstupní a výstupní operace, které jsou dostupné v jazyku C, a navíc jeho součástí jsou prostředky založené na objektových datových typech.

Pro vstup a výstup se v jazyku C++, ale i v jazyku C používají tzv. *datové proudy*. Datový proud se stará o přenos dat od *zdroje* ke *spotřebiči*. Zdrojem může být program a spotřebičem soubor, obrazovka aj. nebo naopak. Součástí proudu bývá zpravidla *vyrovnávací paměť* (angl. *stream buffer*). Při přenosu může docházet k transformaci dat, např. k převodu z binární do znakové (textové) podoby a naopak.

Standardní datové proudy

V jazyce C a C++ existují tři standardní datové proudy:

- *stdin* – slouží pro vstup ze standardního vstupního souboru. Tím je zpravidla konzola počítače, tedy klávesnice. Může být ale přesměrován prostředky operačního systému (z příkazového řádku při spuštění programu).
- *stdout* – je určen pro výstup do standardního výstupního souboru. Tím je zpravidla konzola, tedy obrazovka monitoru. Může být ale přesměrován prostředky operačního systému.
- *stderr* – je určen pro výstup do standardního souboru chyb. Tím je zpravidla opět konzola. Tento proud však nelze pod operačním systémem DOS/Windows přesměrovat.

Všechny tyto proudy se automaticky otevírají při spuštění programu a zavírají při jeho ukončení.

Objektové datové proudy

Objektová koncepce datových proudů přináší řadu výhod, mezi něž patří možnost definovat vstupní operátor >> a výstupní operátor >> pro své vlastní datové typy, definovat vlastní manipulátor apod.

Datové proudy jazyka C++ jsou založeny na dvou hierarchiích objektových typů, které jsou znázorněny na následujícím obrázku.

Všechny objekty související s objektovými datovými proudy jsou součástí prostoru jmen `std`.

Základem je třída `ios_base`. Od ní jsou přímo a nepřímo odvozeny různé šablony třídy. Všechny tyto šablony kromě paměťových datových proudů (`basic_istringstream`, `basic_ostringstream`, `basic_stringstream`) mají stejné parametry, např. deklarace šablony třídy `basic_ios` je následující:

```
template <class charT, class traits = char_traits<charT> >
class basic_ios;
```

Prvním parametrem je znakový typ, a to `char` nebo `wchar_t`. Druhým parametrem je třída, která popisuje některé vlastnosti znaků. Implicitně se zde používá šablona `char_traits`, jejíž instance jsou třídy popisující chování znakového typu.

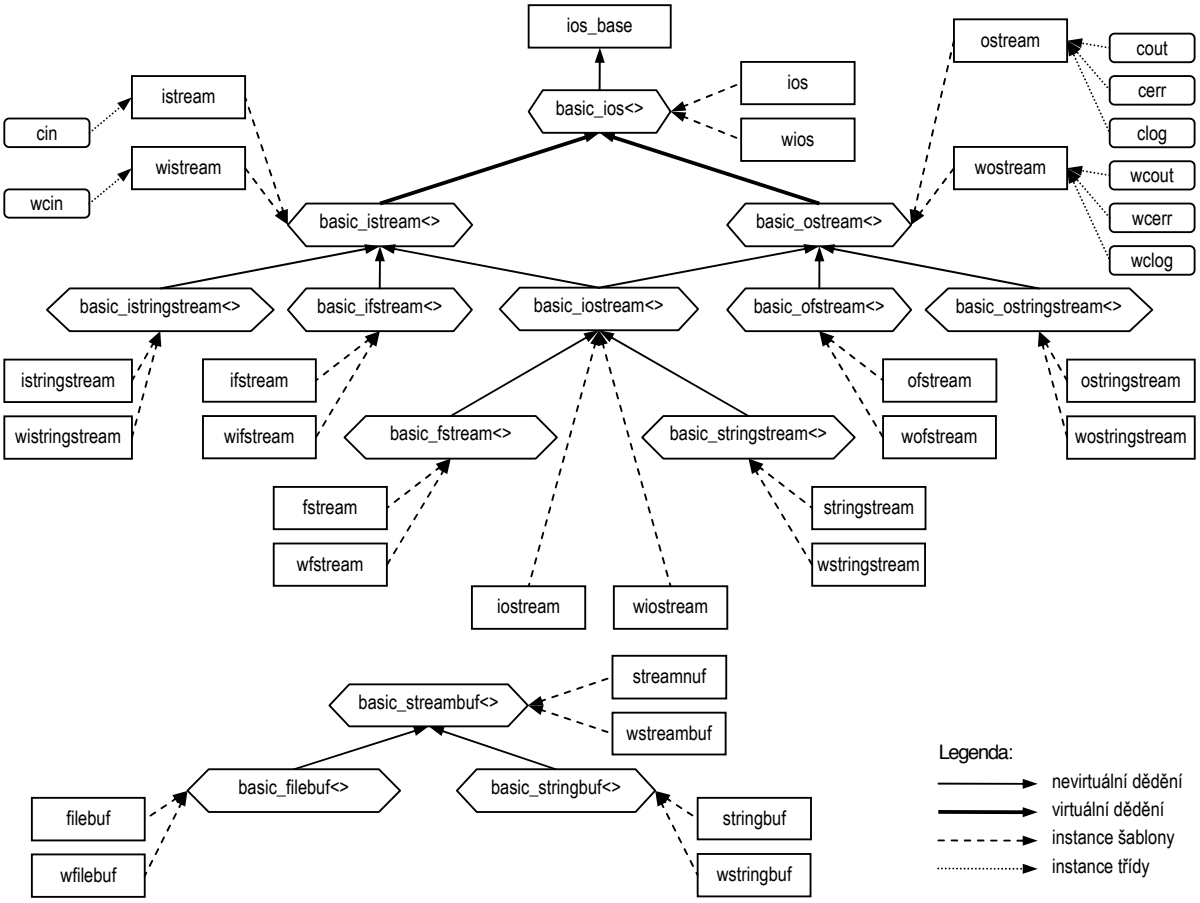
Všechny šablony třídy mají prefix `basic_`. Každá takováto šablona má dvě instance – jednu pro typ `char` a druhou pro typ `wchar_t`. Instance pro typ `char` má název `xxxx` a pro typ `wchar_t` `wxxxx`, kde `xxxx` je název šablony třídy bez prefixu `basic_`. Např. šablona třídy `basic_ios` má dvě instance deklarované takto:

```
typedef basic_ios<char>      ios;
typedef basic_ios<wchar_t>   wios;
```

V hlavičkovém souboru `<iostream>` jsou deklarovány instance tříd pro práci se standardními datovými proudy:

```
extern istream cin; // stdin
extern ostream cout; // stdout
extern ostream cerr; // stderr
extern ostream clog; // stderr
extern wistream wcin; // stdin
extern wostream wcout; // stdout
extern wostream wcerr; // stderr
extern wostream wclog; // stderr
```

U každého z nich je v komentáři uveden standardní datový proud, pro který je objektový datový proud určen. Proudy `cerr` a `wcerr` mají vyrovnávací paměť, zatímco proudy `clog` a `wclog` vyrovnávací paměť nemají.



Hierarchie objektových datových proudů

VSTUPY A VÝSTUPY – POKRAČOVÁNÍ

Třída ios_base

Třída je definována v hlavičkovém souboru `<ios>`. Ve třídě `ios_base` jsou definovány vlastnosti, které jsou společné všem objektovým datovým proudům. Obsahuje definici těchto vnořených typů:

- třída `failure`, odvozená ze třídy `exception`,
- tři typy bitových masek – `fmtflags`, `iostate`, `openmode`,
- výčtový typ `seekdir`,
- třída `Init`.

Třída `failure` je typ výjimky, která se může vyvolat při vzniku chyby během operací s datovým proudem.

Typ bitové masky (angl. *bitmask type*) může být podle normy C++ implementován jako výčtový typ (pro který jsou přetížené jisté operátory), celočíselný typ nebo jako instance šablony třídy `bitset`. V prostředí Visual C++ 2003 a C++ Builder 6 jsou typy `fmtflags`, `iostate`, `openmode` deklarovány jako typy `int` pomocí `typedef`. Pro typ bitové masky lze používat operátory pro práci s bity. Pro každý z typů `fmtflags`, `iostate`, `openmode` obsahuje třída neveřejný atribut, jehož název není normou C++ specifikovaný. K těmto atributům se přistupuje pomocí dále popsaných metod a manipulátorů.

Typ `fmtflags` může obsahovat kombinaci prvků, představující formátovací příznaky. Jejich seznam je uveden v následující tabulce. Jedná se o veřejně přístupné statické konstanty třídy `ios_base`.

Příznak	Význam
`dec`	Vstupy a výstupy celočíselných hodnot budou v desítkové soustavě.
`hex`	Vstupy a výstupy celočíselných hodnot budou v šestnáctkové soustavě.
`oct`	Vstupy a výstupy celočíselných hodnot budou v osmičkové soustavě.
`fixed`	Výstup reálných čísel bude ve tvaru s pevnou řádovou čárkou.
`scientific`	Výstup reálných čísel bude v semilogaritmickém tvaru.
`left`	Výstupní hodnota bude zarovnána k levému okraji vyhrazeného prostoru. Výplňové znaky budou vpravo od hodnoty.
`right`	Výstupní hodnota bude zarovnána k pravému okraji vyhrazeného prostoru. Výplňové znaky budou vlevo od hodnoty.
`internal`	Znaménko u výstupní hodnoty nebo prefix `0x` u hexadecimální výstupní hodnoty bude zarovnáno k levému okraji a vlastní hodnota k pravému okraji vyhrazeného prostoru. Výplňové znaky budou mezi znaménkem a vlastní hodnotou.
`showbase`	Výstupní hodnota se zobrazí s prefixem označujícím číselnou soustavu, a to `0x` pro šestnáctkovou a `0` pro osmičkovou soustavu.
`showpoint`	Při výstupu reálného čísla se vždy zobrazí desetinná čárka a tolik číslic za desetinnou čárkou, kolik je zadaná hodnota přesnosti minus počet číslic před desetinnou čárkou. Příznak nemá význam, je-li nastaven příznak `fixed` nebo `scientific`.
`showpos`	Kladná výstupní číselná hodnota se zobrazí se znaménkem `+`.

Příznak	Význam
`skipws`	Při určitých vstupních operacích se přeskočí bílé znaky, nacházející se před vstupní hodnotou.
`boolaplha`	Hodnoty typu `bool` budou vstupovat a vystupovat jako řetězec `"false"` resp. `"true"`.
`unitbuf`	Vyprázdní proud po každé výstupní operaci.
`uppercase`	Při výstupu hodnoty v šestnáctkové soustavě zobrazí velká písmena `A` až `F`.

Typ `fmtflags` dále definuje tyto konstanty:

Konstanta	Hodnota
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

V atributu typu `fmtflags` jsou implicitně nastaveny pouze příznaky `dec` a `skipws`. Hodnoty jsou implicitně zarovnávány napravo vyhrazeného prostoru. Pro práci s formátovacími příznaky slouží tyto metody:

```
fmtflags flags() const;
```

Vrací hodnotu atributu formátovacích příznaků.

```
fmtflags flags(fmtflags fmtfl);
```

Do atributu formátovacích příznaků uloží hodnotu `fmtfl`. Vrací předchozí hodnotu atributu.

```
fmtflags setf(fmtflags fmtfl);
```

V atributu formátovacích příznaků nastaví příznaky dané parametrem `fmtfl`. Vrací předchozí hodnotu atributu.

```
fmtflags setf(fmtflags fmtfl, fmtflags mask) const;
```

Z atributu formátovacích příznaků odebere příznaky definované maskou `mask` a potom nastaví příznak(y), jehož hodnota je výsledkem výrazu `fmtfl & mask`. Jako parametr `mask` se používá zpravidla jedna z konstant: `adjustfield`, `basefield`, `floatfield` – v takovém případě se pomocí této metody nastaví jeden z formátovacích příznaků obsažených v dané masce a zbývající příznaky dané masky se z atributu odeberou.

```
void unsetf(fmtflags mask);
```

Z atributu odebere příznaky dané parametrem `mask`.

Pro formátovaný výstup slouží dále následující metody:

```
streamsize precision() const;
streamsize precision(streamsize prec);
```

První verze vrací aktuální hodnotu přesnosti (počtu desetinných míst) reálných čísel na výstupu, druhá verze nastaví přesnost na `prec` a vrací předchozí hodnotu přesnosti. Implicitní přesnost je 6. Přesnost se použije, je-li nastaven jeden z příznaků `scientific` nebo `fixed`. Pokud jeden z těchto příznaků není nastaven, reálná čísla budou vystupovat se skutečným počtem desetinných míst.

Typ `streamsize` má být podle normy C++ synonymum pro nějaký celočíselný typ se znaménkem. V prostředí Visual C++ 2003 a C++ Builder 6 je synonymem pro typ `int`.

```
streamsize width() const;
streamsize width(streamsize wide);
```

První verze vrací aktuální hodnotu šířky vyhrazeného prostoru. Druhá verze nastaví šířku na `wide` a vrací předchozí hodnotu šířky. Pro výstupní hodnotu se jedná o minimální šířku, pro vstup je šířka použita pro pole znaků – viz některá z dalších přednášek. Nastavená šířka se použije pouze pro vstup/výstup nejbližší příští hodnoty a potom se opět nastaví na nulu. Pokud má výstupní hodnota více znaků než zadaná šířka, hodnota se vypíše celá.

Příklad

```
int main()
{
   bool b;
   cin.setf(ios_base::boolalpha);
   cout << "Zadej hodnotu b (true, false): ";
   cin >> b;
   cout.setf(ios_base::boolalpha);
   cout << "Logicka hodnota b = " << b << '\n';
   cout.unsetf(ios_base::boolalpha);
   cout << "Celociselna hodnota b = " << b << '\n';
   return 0;
}
```

V uvedeném příkladu se pro proud `cin` nastaví příznak `boolalpha` a z klávesnice se načte logická hodnota. Uživatel musí zadat buď text `false` nebo `true`. Hodnota v proměnné `b` se potom vypíše na obrazovku ve tvaru logické a celočíselné hodnoty.

Příklad

Následující funkce vypíše na obrazovku tabulku funkce sinus obsahující n hodnot x a sin(x). Hodnota x je v radiánech.

```
void TabulkaSinus(int pdm, int n)
{
   int i, w = pdm+4;
   double x;
   ios_base::fmtflags flags = cout.flags();
   int presnost = cout.precision();
   cout.precision(pdm);
   cout.setf(ios_base::fixed, ios_base::floatfield);
   cout.setf(ios_base::right, ios_base::adjustfield);
   cout.width(w); cout << "x";
   cout.width(w); cout << "sin(x)" << '\n';
   for (i = 0, x = 0.0; i < n; i++, x += M_PI/n) {
      cout.width(w); cout << x;
      cout.width(w); cout << sin(x) << '\n';
   }
   cout.flags(flags);
   cout.precision(presnost);
}
```

Na začátku funkce se zapamatuje aktuální stav nastavených formátovacích příznaků a přesnosti, který se na závěr funkce obnoví. Hodnoty jsou vypisovány s přesností na `pdm` desetinných míst. Pro každou výstupní hodnotu je použita šířka `(pdm+4)` znaků se zarovnáním napravo.

Pokud by se funkce zavolala s parametry `TabulkaSinus(4, 5)`, výpis programu by byl následující:

```
     x  sin(x)
0.0000  0.0000
0.6283  0.5878
1.2566  0.9511
1.8850  0.9511
2.5133  0.5878
```

Typ `iostate` obsahuje stavové příznaky. Jejich seznam je uveden v následující tabulce.

Příznak	Význam
`goodbit`	Indikuje stav bez chyb. Příznak má hodnotu 0.
`badbit`	Indikuje, že nějaká operace, jiná než vstupní nebo výstupní, byla neúspěšná.
`eofbit`	Indikuje, že vstupní operace dosáhla konce vstupní sekvence (např. zjištěn konec souboru).
`failbit`	Indikuje, že nějaká vstupní nebo výstupní operace byla neúspěšná.

Metody pro práci se stavovými příznaky jsou definovány v potomkovi `basic_ios`.

Typ `openmode` obsahuje následující příznaky pro otevření datového proudu (souboru).

Příznak	Význam
`app`	Před každým zápisem se přesune ukazatel na konec proudu.
`ate`	Otevře proud a ukazatel se přesune na konec proudu.
`binary`	Otevře proud jako binární.
`in`	Otevře proud pro čtení.
`out`	Otevře proud pro zápis.
`trunc`	Pokud má proud nenulovou délku (velikost), změní se jeho velikost na nulu.

Výčtový typ `seekdir` může obsahovat následující příznaky, sloužící pro přesun ukazatele v proudu (souboru).

Příznak	Význam
`beg`	Požadavek na přesunutí ukazatele na pozici relativní vzhledem k začátku proudu.
`cur`	Požadavek na přesunutí ukazatele na pozici relativní vzhledem k aktuální pozici ukazatele v proudu.
`end`	Požadavek na přesunutí ukazatele na pozici relativní vzhledem ke konci proudu.

Příznaky pro otevření proudu a přesun ukazatele v proudu jsou využívány v šablonách určených pro práci se soubory, řetězci znaků a vyrovnávací pamětí.

Vnořená třída `Init` slouží ke konstrukci a destrukci tříd `cout`, `cin`, `clog`, `cerr` a jejich ekvivalentů pro typ `wchar_t`. Obsahuje statický atribut `init_cnt`, který udává počet zkonstruovaných instancí této třídy.

Typy deklarované v <ios>

V hlavičkovém souboru `<ios>` jsou deklarované dva typy:

```
typedef OFF_T streamoff;
typedef SZ_T streamsize:
```

Typ `streamoff` se používá pro uchování pozice v datovém proudu. Typ **OFF_T** je závislý na implementaci. V prostředí Visual C++ 2003 a C++ Builder 6 je použit typ `long`.

Typ **`SZ_T`** má být synonymum celočíselného typu, který dokáže reprezentovat velikost vyrovnávací paměti a znaky přenášené ve vstupních a výstupních operacích. V prostředí Visual C++ 2003 je použit typ `int`. V prostředí C++ Builder 6 je použit typ `ptrdiff_t`, což je standardní datový typ knihovny jazyka C, deklarovaný v hlavičkovém souboru `<stddef.h>` jako synonymum pro typ `int`.

Šablona třídy basic_ios

Šablona je definována v hlavičkovém souboru `<ios>`. Je odvozena od třídy `ios_base` a slouží jako virtuální předek šablon tříd `basic_istream` a `basic_ostream` a jejich potomků.

Šablona obsahuje ukazatel na instanci šablony `basic_streambuf`, tj. ukazatel na objekt vyrovnávací paměti. Tento ukazatel je možné získat nebo nastavit pomocí dvou metod `rdbuf`:

```
basic_streambuf<charT, traits>* rdbuf() const;
basic_streambuf<charT, traits>* rdbuf(basic_streambuf<charT,traits> *sb);
```

První z nich vrací aktuální hodnotu ukazatele, druhá jej nastavuje a vrací jeho původní hodnotu. Tento ukazatel je také parametrem konstruktoru:

```
explicit basic_ios(basic_streambuf<charT,traits>* sb);
```

Třída obsahuje následující metody pro práci se stavovými příznaky:

Metody pro zjištění, zda daný příznak je nastaven pro připojenou vyrovnávací paměť:

```
bool good() const; // příznak goodbit
bool bad() const; // příznak badbit
bool eof() const; // příznak eofbit
bool fail() const; // příznak failbit
```

```
void clear(iostate state = goodbit);
```

Nastaví kombinaci příznaků `state`. Pokud se použije implicitní parametr, zruší se všechny chybové stavové příznaky. Pokud metoda `rdbuf()` vrací nulu, metoda `clear` přidá k hodnotě `state` příznak `badbit`.

```
iostate rdstate() const;
```

Vrací hodnotu aktuálně nastavených stavových příznaků.

```
void setstate(iostate state);
```

V atributu stavových příznaků nastaví příznaky dané parametrem `state`. Metoda volá metodu `clear(rdstate() | state)`.

```
operator void*() const;
```

Vrací nulu, pokud metoda `fail` vrací `true`, jinak vrací nenulovou hodnotu.

```
bool operator !() const;
```

Vrací výsledek volání metody `fail`.

V prostředí Visual C++ 2003 jsou metody `good`, `bad`, `eof`, `fail`, `rdstate`, `operator void*` a `operator !` definovány v třídě `ios_base`. Metody `clear` a `setstate` jsou definovány jak ve třídě `ios_base`, tak i v šabloně této třídy.

V prostředí C++ Builder 6 jsou výše uvedené metody definovány v třídě `ios_base`, kromě nekonstantních metod `clear` a `setstate`.

Pokud dojde k určité chybě, implicitně se nastaví odpovídající chybový stavový příznak. Program automaticky příznak nezruší, musí se zrušit zavoláním metody `clear`.

Příklad

```
int NactiInt()
{
   int i;
   char c;
   do {
      cin >> i;
      if (cin) return i; // #1 - volá se cin.operator void*()
      cout << "Chyba zadej znovu\n";
      cin.clear();
      cin.unsetf(ios_base::skipws);
      do {
         cin >> c;
      }
      while (c != '\n');
      cin.setf(ios_base::skipws);
   } while (true);
}
```

Funkce `NactiInt` slouží k načtení čísla typu `int`, které vrací. V příkazu #1 se testuje, zda čtení proběhlo bez chyb, tj. zda není nastaven stavový příznak `failbit`. Pokud se chyba nevyskytla, funkce vrátí načtenou hodnotu. Jestliže došlo k chybě, stalo se tak pravděpodobně proto, že se ve vstupním proudu nacházejí nečíselné znaky, které je nutné před zopakováním čtení z proudu vyjmout. Před vyjímáním znaků z proudu se ale musí nejprve zrušit chybové stavové příznaky voláním metody `clear`. Znaky jsou z proudu vyjímány opakovaně, dokud se nevyjme znak nového řádku, který odpovídá stisknutí klávesy ENTER. Extrahování znaků z proudu se provádí při vypnutém přeskakování bílých znaků (zrušen příznak `skipws`), jinak by se nenačetl znak nového řádku, který je také bílým znakem.

Pokud dojde k chybě, lze předepsat, aby se kromě nastavení chybového stavového příznaku vyvolala výjimka typu `ios_base::failure`. Pravidla vyvolávání výjimek se řídí dvěma metodami:

```
iostate exceptions() const;
void exceptions(iostate except);
```

První z nich vrací aktuální nastavení výjimek. Implicitně vrací nulu.

Druhá varianta určuje případy, kde se výjimka vyvolá. Tyto případy jsou dány stavovými příznaky zadanými pomocí parametru `except`. Metoda na závěr volá `clear(rdstate())`, tj. znovu se nastaví aktuální stavové příznaky a pokud je v nich obsažen chybový stavový příznak, který je součástí parametru `except`, vyvolá se výjimka.

Příklad

```
int main()
{
   int i;
   cin.exceptions(ios_base::failbit); // #1
   try {
      cout << "Zadej cele cislo: "; cin >> i;
      cout << "Bylo nacteno cislo " << i << '\n';
   }
   catch (ios_base::failure& t) {
      cout << "Doslo k vyjimce " << t.what() << '\n';
   }
   return 0;
}
```

V uvedeném příkladu se nejprve příkazem #1 nastaví vyvolání výjimky, pokud se v proudu `cin` nastaví stavový příznak `failbit`. Potom se ve složeném příkazu pokusného bloku provádí čtení. Pokud při čtení dojde k chybě, vyvolá se výjimka typu `ios_base::failure`, která se zachytí a vypíše se text výjimky pomocí metody `what()`.

Pro formátovaný výstup obsahuje šablona `basic_ios` dvě metody `fill`:

```
char_type fill() const;
char_type fill(char_type fillch);
```

První verze vrací aktuální výplňový znak, kterým se vyplní prázdná část vyhrazeného prostoru. Druhá verze nastaví výplňový znak a vrací předchozí výplňový znak. Implicitním výplňovým znakem je mezera.

Typ `char_type` je vnořený typ šablony `basic_ios`, deklarovaný jako synonymum pro typový parametr `charT` této šablony.

Příklad

```
int main()
{
   const int w = 20;
   cout << "TextA";
   cout.width(w);
   cout.fill('.');
   cout << 10 << '\n';
   cout << "TextB";
   cout.width(w);
   cout << 150 << '\n';
   return 0;
}
```

Uvedený program vypíše následující tabulku:

```
TextA..................10
TextB.................150
```

Šablona dále obsahuje metody pro převod znaků:

```
char_type widen(char c) const;
char narrow(char_type c, char dfault) const;
```

Metoda `widen` konvertuje jednobajtový znak `c` na typ `char_type`. Metoda `narrow` převede znak `c` typu `char_type` na jednobajtový znak. Pokud znak `c` v jednobajtové znakové sadě není uveden, metoda vrací znak `dfault`.

Šablona třídy fpos

Šablona třídy `fpos` je definována v hlavičkovém souboru `<ios>`. Slouží pro uchování pozice ukazatele v datovém proudu (souboru). Obsahuje jeden typový parametr `stateT`, který uchovává stav konverze znaků.

Šablona obsahuje dva neveřejné atributy, jeden pro uchování pozice ukazatele a druhý typu `stateT`. Jejich názvy norma C++ nespecifikuje. Atribut pro pozici v datovém proudu je typu `streamoff`.

Šablona má podle normy obsahovat dvě metody `state` a dále pak takové metody, aby bylo možné provést určité normou specifikované operace. V prostředí Visual C++ 2003 je definována následovně:

```
template <class stateT> class fpos {
public:
    fpos(streamoff _Off = 0);
    fpos(stateT _State, fpos_t _Fileposition);
    operator streamoff() const;
    bool operator ==(const fpos<stateT>&);
    bool operator != (const fpos<stateT>&);
    fpos<stateT> operator + (streamoff);
    fpos<stateT> operator - (streamoff);
    stremoff operator - (streamoff);
    fpos<stateT>& operator += (streamoff);
    fpos<stateT>& operator -= (streamoff);
    stateT state() const;
    void state(stateT);
private:
    // atributy
};
```

Z definice je zřejmé, že šablona obsahuje metody pro práci s hodnotou typu `streamoff`, a to konverzní konstruktor, operátor přetypování, operátor rovnosti a nerovnosti a aritmetické operátory. Dále obsahuje dvě metody `state`. První z nich vrací aktuální hodnotu atributu typu `stateT` a druhá nastavuje hodnotu tohoto atributu. Uvedená definice šablony normě vyhovuje.

V hlavičkovém souboru `<iosfwd>` jsou deklarovány instance šablony `fpos` pro typ `char` a `wchat_t` takto:

```
typedef fpos<char_traits<char>::state_type> streampos;
typedef fpos<char_traits<wchar_t>::state_type> wstreampos;
```

Vnořený typ `state_type` v instancích šablon `char_traits` je deklarován jako synonymum pro typ `mbstate_t`. Typ `mbstate_t` má být definován v hlavičkovém souboru `<cwchar>` tak, aby mohl reprezentovat všechny konverzní stavy, které se mohou vyskytnout ve vícebajtové znakové sadě (angl. *multi-byte character set*, zkr. MBCS). Přesnou definici tohoto typu norma C++ neuvádí. V prostředí Visual C++ 2003 je typ `mbstate_t` deklarován jako synonymum pro typ `int`.

Šablona třídy basic_istream

Šablona je definována v hlavičkovém souboru `<istream>`. Je virtuálně odvozena od šablony `basic_ios` a je základem vstupních proudů.

Typy

Šablona obsahuje následujících deklarace typů:

```
typedef charT                      char_type;
typedef typename traits::int_type  int_type;
typedef typename traits::pos_type  pos_type;
typedef typename traits::off_type  off_type;
typedef traits                     traits_type;
```

Typ `pos_type` reprezentuje absolutní pozici ukazatele v datovém proudu. V instanci šablony třídy `char_traits<char>` je deklarován jako synonymum typu `streampos` a v instanci `char_traits<wchar_t>` jako `wstreampos`.

Typ `off_type` reprezentuje relativní pozici ukazatele v datovém proudu. V instancích šablony třídy `char_traits<char>` a `char_traits<wchar_t>` je deklarován jako synonymum typu `streamoff`.

Typ `traits::int_type` má být dle normy typ nebo třída, která je schopna reprezentovat všechny znaky konvertované z korespondujícího typu `char_type` včetně znaku konce souboru. V instanci šablony třídy `char_traits<char>` je deklarován jako synonymum pro typ `int` a v instanci `char_traits<wchar_t>` jako `wint_t`. Typ `wint_t` je standardním datovým typem jazyka C a je deklarován jako synonymum pro typ `wchar_t` v hlavičkovém souboru `<stddef.h>`.

Metody pro práci s ukazatelem v proudu

Pozice ukazatele v datovém proudu se počítá od nuly.

Pro zjištění a nastavení pozice ukazatele v proudu slouží tyto metody:

```
pos_type tellg();
```

Vrací aktuální pozici ukazatele v datovém proudu. Pokud `fail() == true`, vrací `pos_type(-1)`.

```
basic_istream<charT, traits>& seekg(pos_type& pos);
```

Nastaví absolutní pozici ukazatele na `pos` a vrací `*this`.

```
basic_istream<charT, traits>&
seekg(off_type& off, ios_base::seekdir dir);
```

Nastaví pozici ukazatele na `off` relativně k místu zadanému parametrem `dir`: začátek proudu, konec proudu, aktuální pozice v proudu. Vrací `*this`.

Příklad

Funkce `NactiInt` uvedená v kapitole „Šablona třídy `basic_ios`“ by mohla být upravena takto:

```
int NactiInt()
{
   int i;
   do {
      cin >> i;
      if (cin) return i;
      cin.clear();
      cin.seekg(0, ios_base::end); // #1
      cin.clear();// #2
      cout << "Chyba, zadej znovu\n";
   } while (true);
}
```

Příkaz #1 přesune ukazatel na konec proudu. Tento příkaz pro proud cin v prostředí Visual C++ 2003 nastaví stavový příznak `failbit`, který se musí zrušit příkazem #2. V prostředí C++ Builder 6 příkaz #2 není zapotřebí.

Metody a funkce pro formátovaný vstup

Šablona obsahuje přetížené operátory `>>` pro jednotlivé základní datové typy kromě znakových typů. Tyto operátory slouží pro čtení hodnoty určitého datového typu z daného vstupního datového proudu. Nazývají se *extraktory* resp. *metody pro formátovaný vstup* (angl. *extractors* resp. *formatted input functions*). Mají stejný tvar lišící se typem parametru, např. pro typ `double` vypadá prototyp takto:

```
basic_istream<charT, traits>& operator >> (double& f);
```

Operátory vrací referenci na tuto šablonu a tudíž je lze zřetězovat, např. načtení hodnot dvou proměnných z konzoly lze zapsat následovně:

```
cin >> a >> b;
```

Nejprve se načte hodnota do proměnné `a` a potom do proměnné `b`, přičemž proměnné mohou být odlišného typu.

Extraktory implicitně přeskakují bílé znaky.

Existuje také extraktor, jehož pravým operandem je ukazatel na vyrovnávací paměť proudu:

```
basic_istream<charT, traits>& operator >>
(basic_streambuf<char_type, traits>* sb);
```

Tento extraktor čte znaky a ukládá je do výstupního proudu, se kterým je spojena vyrovnávací paměť `sb`.

Součástí hlavičkového souboru `<istream>` jsou přetížené operátory `>>`, definované jako obyčejné funkce, tři pro pole znaků a tři pro znakové typy:

```
template<class charT, class traits> basic_istream<charT, traits>&
   operator >> (basic_istream<charT, traits>&, charT*);
template<class traits> basic_istream<char, traits>&
   operator >> (basic_istream<char, traits>&, unsigned char*);
template<class traits> basic_istream<char, traits>&
   operator >> (basic_istream<char, traits>&, signed char*);

template<class charT, class traits> basic_istream<charT, traits>&
   operator >> (basic_istream<charT, traits>&, charT&);
template<class traits> basic_istream<char, traits>&
   operator >> (basic_istream<char, traits>&, unsigned char&);
template<class traits> basic_istream<char, traits>&
   operator >> (basic_istream<char, traits>&, signed char&);
```

Všechny uvedené operátory mají levý operand typu `basic_istream<charT, traits>`, pravý operand se liší a udává typ, jehož hodnota se má načíst. Všechny vrací referenci na levý operand.

Operátory pro načtení pole znaků čtou znaky a ukládají je do pole. Pokud je nastaven formátovací příznak `skipws`, přeskočí se úvodní bílé znaky. Čtení skončí, pokud:

- se narazí na bílý znak – bílý znak se z proudu nevyjme,
- se přečte `width()-1` znaků v případě, že `width() > 0`,

- se dojde na konec proudu.

Pokud není nastaven příznak `skipws` a proud začíná bílým znakem, žádný znak se nenačte. Pokud operátor nevyjme žádný znak z proudu, volá `setstate(failbit)`.

Příklad

```
int main()
{
   char s[10];
   cout << "Zadej retezec znaku: ";
   cin.width(sizeof s);
   cin >> s; // nacte se maximalne 9 znaku
   cout << "Nacteny retezec: \"" << s << "\"" << '\n';
   return 0;
}
```

V uvedeném příkladu se načte řetězec maximálně 9 znaků dlouhý a vypíše se na obrazovku. Úvodní bílé znaky se přeskočí.

Operátory pro načtení znaku načtou jeden znak. Pokud je nastaven formátovací příznak `skipws`, přeskočí se úvodní bílé znaky a načte se první nebílý znak. Pokud příznak není nastaven, načte se první bílý znak.

Metody pro neformátovaný vstup

Pro čtení znaků z datového proudu obsahuje šablona také metody pro neformátovaný vstup (angl. *unformatted input functions*). Tyto metody nepřeskakují bílé znaky. Jedná se o následující metody.

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

Vrací počet znaků naposledy vyjmutých z proudu pomocí některé z metod pro neformátovaný vstup.

```
int_type get();
```

Přečte jeden znak z proudu, který je návratovou hodnotou metody. Pokud v proudu žádný znak není, volá `setstate(failbit)`.

```
basic_istream<charT, traits>& get(char_type& c);
```

Přečte jeden znak z proudu do parametru `c` a vrací `*this`. Pokud v proudu žádný znak není, volá `setstate(failbit)`.

```
basic_istream<charT, traits>& get(char_type* s, streamsize n,
                                  char_type delim);
```

Přečte maximálně `n-1` znaků, které uloží do pole `s`. Čtení může skončit dříve, pokud se:

- zjistí znak `delim`, který se ponechá ve vstupním proudu,
- dojde na konec proudu – v takovém případě volá `setstate(eofbit)`.

Pokud do `s` neuloží žádný znak, volá `setstate(failbit)`. V každém případě přidá na konec pole `s` nulový znak. Vrací `*this`.

```
basic_istream<charT, traits>& get(char_type* s, streamsize n);
```

Vrací `get(s, n, widen('\n'))`.

```
basic_istream<charT, traits>& get(basic_streambuf<char_type, traits>& sb,
                                  char_type delim);
```

Čte znaky a ukládá je do výstupního proudu, se kterým je spojena vyrovnávací paměť `sb`. Čtení skončí, pokud:

- se zjistí znak `delim`, který se ponechá ve vstupním proudu,
- se dojde na konec proudu – v takovém případě volá `setstate(eofbit)`,
- dojde k chybě při ukládání do vyrovnávací paměti výstupního proudu.

Pokud do `sb` neuloží žádný znak, volá `setstate(failbit)`. Vrací `*this`.

```
basic_istream<charT, traits>&
get(basic_streambuf<char_type, traits>& sb);
```

Vrací `get(sb, widen('\n'))`.

```
basic_istream<charT, traits>&
getline(char_type* s, streamsize n, char_type delim);
```

Čte znaky a ukládá je do pole `s`. Čtení skončí, pokud se:

- zjistí znak `delim`, který se vyjme ze vstupního proudu, ale do `s` se neuloží,
- dojde na konec proudu – v takovém případě volá `setstate(eofbit)`,
- do pole `s` uloží `n-1` znaků – pokud je v proudu další znak, volá `setstate(failbit)`.

Pokud do `s` neuloží žádný znak, volá `setstate(failbit)`. V každém případě přidá na konec pole `s` nulový znak. Vrací `*this`.

```
basic_istream<charT, traits>& getline(char_type* s, streamsize n);
```

Vrací `getline(s, n, widen('\n'))`.

```
basic_istream<charT, traits>&
ignore(int n = 1, int_type delim = traits::eof());
```

Čte znaky, které nikam neukládá. Čtení skončí, pokud se:

- načte `n` znaků,
- dojde na konec proudu – v takovém případě volá `setstate(eofbit)`,
- zjistí znak `delim`, který se vyjme ze vstupního proudu (implicitně znak konce souboru) – tato podmínka se nevyskytne, pokud je znak `delim` roven znaku konce souboru.

Vrací `*this`. Parametr `n` nesmí být roven `INT_MAX`.

```
int_type peek();
```

Pokud metoda `good()` vrací `false`, vrací `traits::eof()`, jinak vrací hodnotu následujícího znaku v proudu, přičemž tento znak není z proudu vyjmut.

```
basic_istream<charT, traits>& read(char_type* s, streamsize n);
```

Pokud metoda `good()` vrací `false`, volá `setstate(failbit)`. Jinak čte znaky a ukládá je do pole `s`. Čtení skončí, pokud se:

- do pole `s` uloží `n` znaků,
- dojde na konec proudu – v takovém případě volá `setstate(failbit|eofbit)`.

Vrací `*this`. Pole `s` nebude zakončené nulou. Metoda se používá pro čtení jakýchkoliv dat do oblasti paměti, na kterou ukazuje `s`.

```
streamsize readsome(char_type* s, streamsize n);
```

Pokud metoda `good()` vrací `false`, volá `setstate(failbit)`. Jinak čte znaky uložené v připojené vyrovnávací paměti a ukládá je do pole `s`. Výsledek metody závisí na výsledku volání `rdbuf()->in_avail()`. Metoda `in_avail()` vrací hodnotu typu `streamsize` udávající počet znaků, které se nacházejí ve vyrovnávací paměti a nebyly ještě přečteny.

- je rovna –1: volá `setstate(eofbit)` a z proudu nevyjme žádný znak,
- je rovna 0: z proudu nevyjme žádný znak,
- je větší než 0: z proudu vyjme `min(rdbuf()->in_avail(), n)`.

Vrací počet vyjmutých znaků, které se uloží do pole `s`. Pole `s` nebude zakončené nulou.

```
basic_istream<charT, traits>& putback(char_type c);
```

Pokud metoda `good()` vrací `false`, volá `setstate(failbit)`. Jinak vloží znak do vstupního proudu. Při dalším čtení z tohoto proudu se načte nejprve tento vložený znak. Pokud se vložení znaku nepodaří, volá `setstate(badbit)`. Vrací `*this`. Počet znaků, které lze za sebou vložit do proudu je omezený.

```
basic_istream<charT, traits>& unget();
```

Pokud metoda `good()` vrací `false`, volá `setstate(failbit)`. Jinak do vstupního proudu vrátí naposledy načtený znak, resp. posune ukazatel ve vyrovnávací paměti o jeden znak dozadu. Pokud operaci nelze provést, volá `setstate(badbit)`.

```
int sync();
```

Synchronizuje interní vstupní vyrovnávací paměť s externí posloupností znaků, tj. zruší operace, provedené metodami `putback` a `unget`. Pokud `rdbuf() == 0`, vrací –1. Pokud `rdbuf() != 0`, vrací 0 a v případě chyby volá `setstate(badbit)`.

Příklad

```
int main()
{
   char  s[10];
   int   count;
   bool  b = true;
   while (cin.good()) {
      if (b) cout << "Zadej retezec: ";
      else   b = true;
      cin.getline(s, sizeof s);
      count = cin.gcount();
      if (cin.eof()) cout << "Konec proudu\n";
      else if (cin.fail()) {
         cout << "Radek obsahuje vice nez " << sizeof s - 1 <<
                 " znaku\n";
         cin.clear(cin.rdstate() & ~ios_base::failbit);
                   // odebere se failbit
         b = false;
      }
      else count--;
      cout << "Pocet nactenych znaku: " << count << '\n';
      cout << "Nacteny retezec: \"" << s << "\"" << '\n';
   }
   cout << "\nKonec programu - klavesa Enter\n";
   cin.clear();
   cin.get();
   return 0;
}
```

V uvedeném příkladu se opakovaně čte řetězec znaků z klávesnice do pole `s` a obsah pole `s` se vypisuje na obrazovku. Čtení jednoho řetězce končí novým řádkem, koncem souboru nebo přečtením `sizeof(s)-1` znaků. Čtení řetězců skončí, pokud se dojde na konec proudu (uživatel stiskne kombinaci kláves **Ctrl+Z** a **Enter**). Program se ukončí po stisknutí klávesy **Enter**.

Příklad

Příklad obsahuje upravenou metodu `NactiInt`, jejíž původní verze je uvedena v kapitole „Šablona třídy `basic_ios`".

```
int NactiInt()
{
   int i;
   do {
      cin >> i;
      if (cin) return i;
      cin.clear();
      cin.ignore(INT_MAX-1, '\n');
      cout << "Chyba, zadej znovu\n";
   } while (true);
}

int main()
{
   cout << "Zadej cele cislo: ";
   int i = NactiInt();
   cout << "Bylo nacteno cislo " << i << '\n';
   cout << "\nKonec programu - klavesa Enter\n";
   if (cin.rdbuf()->in_avail()) cin.ignore(INT_MAX-1, '\n'); // #1
   cin.get(); // #2
   return 0;
}
```

Příkaz #1 v případě, že vyrovnácí pamět proudu `cin` není prázdná, vyjme z vyrovnávací paměti všechny znaky, které nikam neuloží. Potom se teprve provede příkaz #2. Kdyby se před provedením příkazu #2 nevyprázdnila vyrovnávací paměť a obsahovala by nějaký znak, příkaz #2 by vyjmul z vyrovnávací paměti další znak a nečekal by na stisknutí klávesy **Enter**.

VSTUPY A VÝSTUPY – POKRAČOVÁNÍ

Šablona třídy basic_ostream

Šablona je definována v hlavičkovém souboru `<ostream>`. Je virtuálně odvozena od šablony `basic_ios` a je základem výstupních proudů.

Typy

Šablona obsahuje deklarace typů `char_type`, `int_type`, `pos_type`, `off_type` a `traits_type`. Deklarace je stejná jako v šabloně `basic_istream`.

Metody pro práci s ukazatelem v proudu

Pro zjištění a nastavení pozice ukazatele v proudu slouží tyto metody:

```
pos_type tellp();
```

Vrací aktuální pozici ukazatele v datovém proudu. Pokud `fail() == true`, vrací `pos_type(-1)`.

```
basic_ostream<charT, traits>& seekp(pos_type& pos);
```

Nastaví absolutní pozici ukazatele na `pos` a vrací `*this`.

```
basic_ostream<charT, traits>& seekp(off_type& off,
                                    ios_base::seekdir dir);
```

Nastaví pozici ukazatele na `off` relativně k místu zadanému parametrem `dir`: začátek proudu, konec proudu, aktuální pozice v proudu. Vrací `*this`.

V proudu, který je určen pro vstup i výstup (např. `fstream`), lze používat pro zjištění aktuální pozice ukazatele a změnu jeho pozice jak metody `tellp` a `seekp`, tak i metody `tellg` a `seekg`.

Metody a funkce pro formátovaný výstup

Šablona obsahuje přetížené operátory `<<` pro jednotlivé základní datové typy kromě znakových typů. Tyto operátory slouží pro zápis hodnoty určitého datového typu do daného výstupního datového proudu. Nazývají se *insertory* resp. *metody pro formátovaný výstup* (angl. *inserters* resp. *formatted output functions*). Mají stejný tvar lišící se typem parametru, např. pro typ `double` vypadá prototyp takto:

```
basic_ostream<charT, traits>& operator << (double f);
```

Operátory vrací referenci na tuto šablonu a tudíž je lze zřetězovat, podobně jako u extraktorů.

Existuje také operátor, jehož pravým operandem je ukazatel na vyrovnávací paměť proudu:

```
basic_ostream<charT, traits>& operator <<
(basic_streambuf<char_type, traits>* sb);
```

Tento operátor čte znaky ze vstupního proudu, se kterým je spojena vyrovnávací paměť `sb` a zapisuje je do výstupního proudu. Čtení skončí, pokud:

- se dojde na konec vstupního proudu,
- dojde k chybě při vkládání znaků do výstupního proudu,
- dojde k chybě při čtení znaků z vyrovnávací paměti.

Pokud se do výstupního proudu nevloží žádný znak, volá `setstate(failbit)`.

Součástí hlavičkového souboru `<ostream>` jsou přetížené operátory `<<`, definované jako obyčejné funkce, pět pro znakové typy a pět pro pole znaků:

```
template<class charT, class traits> basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& out, charT c);
template<class charT, class traits> basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& out, char c);
template<class traits> basic_ostream<char, traits>&
   operator << (basic_ostream<char, traits>& out, char c);
template<class traits> basic_ostream<char, traits>&
   operator << (basic_ostream<char, traits>& out, signed char c);
template<class traits> basic_ostream<char, traits>&
   operator << (basic_ostream<char, traits>& out, unsigned char c);

template<class charT, class traits> basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& cout, const charT* s);
template<class charT, class traits> basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& out, const char* s);
template<class traits> basic_ostream<char, traits>&
   operator << (basic_ostream<char, traits>& out, const char* s);
template<class traits> basic_ostream<char, traits>&
   operator << (basic_ostream<char, traits>& out, const signed char* s);
template<class traits> basic_ostream<char, traits>&
   operator << (basic_ostream<char,traits>& out,const unsigned char* s);
```

Tyto funkce se chovají podobně jako metody `<<` s tím rozdílem, že pro jednobajtové znaky `c` typu `char` a `s` typu `const char*` se volá `out.widen(c)`, pokud se vkládají do proudu `out` s jiným typovým parametrem než `char`. V případě pole znaků se do výstupního proudu vloží celý řetězec znaků `s`, přesněji `traits::length(s)` znaků.

Metody pro neformátovaný výstup

Pro zápis znaků do datového proudu obsahuje šablona také metody pro neformátovaný výstup (angl. *unformatted output functions*). Jedná se o následující metody.

`basic_ostream<charT, traits>&` **`put`**`(char_type c);`

Do výstupního proudu vloží znak `c`. Pokud znak nelze do proudu vložit, volá `setstate(badbit)`. Vrací `*this`.

`basic_ostream<charT, traits>&` **`write`**`(const char_type* s, streamsize n);`

Do výstupního proudu vkládá znaky z pole `s`. Vkládání skončí, pokud:

- se vloží `n` znaků,
- při vkládání vznikne chyba – v takovém případě volá `setstate(badbit)`.

Vrací `*this`. Metoda neskončí, pokud narazí v poli `s` na nulu, je určena pro výstup jakýchkoliv dat z oblasti paměti, na kterou ukazuje `s`.

`basic_ostream<charT, traits>&` **`flush`**`();`

Jestliže `rdbuf() != 0`, volá `rdbuf()->pubsync()` a v obou případech vrací `*this`. Jestliže `pubsync()` vrací –1, volá `setstate(badbit)`. Metoda `pubsync()` vyprázdní vyrovnávací paměť.

Příklad

I když vstupní datový proud nemá metodu `flush()`, lze vyprázdnit jeho vyrovnávací paměť pomocí metody `pubsync()` třídy `basic_streambuf`. Funkci `main()` z předchozího příkladu lze napsat také následujícím způsobem:

```
int main()
{
   cout << "Zadej cele cislo: ";
   int i = NactiInt();
   cout << "Bylo nacteno cislo " << i << '\n';
   cout << "\nKonec programu - klavesa Enter\n";
   cin.rdbuf()->pubsync();
   cin.get();
   return 0;
}
```

Manipulátory

Manipulátory jsou funkce, které lze volat s použitím operátoru >> nebo <<. Např. v příkazu

```
cout << setw(10) << a;
```

je použit manipulátor `setw`, který nastaví šířku vyhrazeného prostoru stejně, jako kdyby se volala metoda `cout.width(10)`.

Manipulátory jsou obyčejné funkce deklarované v různých hlavičkových souborech, v závislosti na tom, pro jaké proudy je lze použit. Mohou být bez parametrů nebo s parametry a lze definovat i vlastní manipulátory.

Manipulátor	Hlavičkový soubor	Význam
`dec`	`<ios>`	Nastaví formátovací příznak `dec` voláním metody `setf(ios_base::dec, ios_base::basefield)` třídy `ios_base`.
`hex`	`<ios>`	Nastaví formátovací příznak `hex` voláním metody `setf(ios_base::hex, ios_base::basefield)` třídy `ios_base`.
`oct`	`<ios>`	Nastaví formátovací příznak `oct` voláním metody `setf(ios_base::oct, ios_base::basefield)` třídy `ios_base`.
`fixed`	`<ios>`	Nastaví formátovací příznak `fixed` voláním metody `setf(ios_base::fixed, ios_base::floatfield)` třídy `ios_base`.
`scientific`	`<ios>`	Nastaví formátovací příznak `scientific` voláním metody `setf(ios_base::scientific, ios_base::floatfield)` třídy `ios_base`.
`left`	`<ios>`	Nastaví formátovací příznak `left` voláním metody `setf(ios_base::left, ios_base::adjustfield)` třídy `ios_base`.
`right`	`<ios>`	Nastaví formátovací příznak `right` voláním metody `setf(ios_base::right, ios_base::adjustfield)` třídy `ios_base`.

Manipulátor	Hlavičkový soubor	Význam
`internal`	`<ios>`	Nastaví formátovací příznak `internal` voláním metody `setf(ios_base::internal, ios_base::adjustfield)` třídy `ios_base`.
`showbase`	`<ios>`	Nastaví formátovací příznak `showbase` voláním metody `setf(ios_base::showbase)` třídy `ios_base`.
`noshowbase`	`<ios>`	Odebere formátovací příznak `showbase` voláním metody `unsetf(ios_base::showbase)` třídy `ios_base`.
`showpoint`	`<ios>`	Nastaví formátovací příznak `showpoint` voláním metody `setf(ios_base::showpoint)` třídy `ios_base`.
`noshowpoint`	`<ios>`	Odebere formátovací příznak `showpoint` voláním metody `unsetf(ios_base::showpoint)` třídy `ios_base`.
`showpos`	`<ios>`	Nastaví formátovací příznak `showpos` voláním metody `setf(ios_base::showpos)` třídy `ios_base`.
`noshowpos`	`<ios>`	Odebere formátovací příznak `showpos` voláním metody `unsetf(ios_base::showpos)` třídy `ios_base`.
`skipws`	`<ios>`	Nastaví formátovací příznak `skipws` voláním metody `setf(ios_base::skipws)` třídy `ios_base`.
`noskipws`	`<ios>`	Odebere formátovací příznak `skipws` voláním metody `unsetf(ios_base::skipws)` třídy `ios_base`.
`boolalpha`	`<ios>`	Nastaví formátovací příznak `boolalpha` voláním metody `setf(ios_base::boolalpha)` třídy `ios_base`.
`noboolalpha`	`<ios>`	Odebere formátovací příznak `boolalpha` voláním metody `unsetf(ios_base::boolalpha)` třídy `ios_base`.
`unitbuf`	`<ios>`	Nastaví formátovací příznak `unitbuf` voláním metody `setf(ios_base:: unitbuf)` třídy `ios_base`.
`nounitbuf`	`<ios>`	Odebere formátovací příznak `unitbuf` voláním metody `unsetf(ios_base:: unitbuf)` třídy `ios_base`.
`uppercase`	`<ios>`	Nastaví formátovací příznak `uppercase` voláním metody `setf(ios_base::uppercase)` třídy `ios_base`.
`nouppercase`	`<ios>`	Odebere formátovací příznak `uppercase` voláním metody `unsetf(ios_base::uppercase)` třídy `ios_base`.
`setprecision (int n)`	`<iomanip>`	Nastaví přesnost (počet desetinných míst) reálných čísel voláním metody `precision(n)` třídy `ios_base.`
`setw(int n)`	`<iomanip>`	Nastaví šířku vyhrazeného prostoru voláním metody `width(n)` třídy `ios_base`.
`setbase (int base)`	`<iomanip>`	Nastaví typ číselné soustavy podle hodnoty `base`: 0 – implicitní stav, 8 – nastaví příznak `oct`, 10 – nastaví příznak `dec`, 16 – nastaví příznak `hex`.
`setfill (char_type c)`	`<iomanip>`	Nastaví výplňový znak, kterým se vyplní prázdná část vyhrazeného prostoru voláním metody `fill(n)` šablony třídy `basic_ios`.

Manipulátor	Hlavičkový soubor	Význam
`setiosflags (ios_base:: fmtflags mask)`	`<iomanip>`	Nastaví kombinaci formátovacích příznaků voláním metody `setf(mask)` třídy `ios_base`.
`resetiosflags (ios_base:: fmtflags mask)`	`<iomanip>`	Odebere kombinaci formátovacích příznaků voláním metody `setf(ios_base::fmtflags(0), mask)` třídy `ios_base`.
`ws`	`<istream>`	Vyjme ze vstupního proudu úvodní bílé znaky. Vyjímání končí, pokud: • se vyskytne nebílý znak • se dojde na konec proudu – v takovém případě nastaví příznak `eofbit`.

`endl`	`<ostream>`	Vloží znak pro přechod na nový řádek do výstupního proudu a vyprázdní vyrovnávací paměť tohoto proudu voláním metod:
		`os.put(os.widen('\n'));`
		`os.flush();`
		kde `os` je instance třídy šablony `basic_ostream` nebo jejího potomka.

`ends`	`<ostream>`	Do výstupního proudu vloží nulový znak voláním metody `put(charT())` šablony třídy `basic_ostream`.
`flush`	`<ostream>`	Vyprázdní vyrovnávací paměť sdruženou s výstupním proudem voláním metody `flush()` šablony třídy `basic_ostream`.

Příklad

```
void Tabulka(const char** Texty, double* Hodnoty, int n)
{
   int i, p = cout.precision();
   char c = cout.fill();
   ios_base::fmtflags f = cout.flags();

   cout << setfill('.') << fixed << setprecision(2);
   for (i = 0; i < n; i++) {
      cout << left << setw(15) << Texty[i] <<
              right << setw(15) << Hodnoty[i] << endl;
   }
   cout << setiosflags(f) << setprecision(p) << setfill(c);
}

int main()
{
   const char* Texty[] = { "Aaa", "Bbbbb", "Cc" };
   double      Hodnoty[] = { 1.5, 101.236, 10.55 };
   Tabulka(Texty, Hodnoty, sizeof Texty/sizeof Texty[0]);
   cin.get();
   return 0;
}
```

V uvedeném příkladu se vypíše tabulka textů v levém sloupci a k nim odpovídajících hodnot v pravém sloupci s výplňovým znakem tečka pomocí funkce `Tabulka`. Na začátku funkce `Tabulka` se zapamatuje aktuální stav atributů přesnosti, výplňového znaku a formátovacích příznaků, který se na závěr funkce obnoví. Výstup programu bude následující:

```
Aaa.....................1.50
Bbbbb..................101.24
Cc......................10.55
```

Šablona třídy basic_iostream

Šablona je definována v hlavičkovém souboru <istream>. Má dva veřejně přístupné předky: basic_istream a basic_ostream. Kromě konstruktorů a destruktoru neobsahuje další metody. Je základní šablonou pro vstupní i výstupní operace.

Šablona třídy basic_ifstream

Šablona je definována v hlavičkovém souboru <fstream>. Má jednoho veřejně přístupného předka basic_istream. Slouží pro čtení ze souboru typu FILE. Využívá funkce ze standardní knihovny jazyka C pro práci se soubory. Tyto funkce jsou deklarované v hlavičkovém souboru <stdio.h> resp. <cstdio>.

Typy

Šablona obsahuje deklarace typů char_type, int_type, pos_type, off_type a traits_type. Deklarace je stejná jako v šabloně basic_istream.

Konstruktory

Šablona obsahuje dva konstruktory:

```
basic_ifstream();
explicit basic_ifstream(const char* s,
                        ios_base::openmode mode= ios_base::in);
```

Implicitní konstruktor inicializuje předka basic_istream předáním ukazatele na vyrovnávací paměť typu basic_filebuf<charT, traits> a žádný soubor neotevře.

Druhá verze konstruktoru provede totéž co první verze a navíc volá rdbuf()->open(s, mode | in). Metoda open šablony basic_filebuf otevře soubor mající jméno s v režimu mode | in voláním obyčejné funkce std::fopen(s, modstr). Parametr modstr je určen z hodnoty výrazu mode & ~ios_base::ate podle následující tabulky:

Kombinace příznaků parametru *mode*					Parametr *modstr*
binary	**in**	**out**	**trunc**	**app**	
		+			"w"
		+		+	"a"
		+	+		"w"
	+				"r"
	+	+			"r+"
	+	+	+		"w+"
+		+			"wb"
+		+		+	"ab"
+		+	+		"wb"
+	+				"rb"
+	+	+			"r+b"
+	+	+	+		"w+b"

Jestliže parametr `mode` neobsahuje některou z kombinací příznaků, otevření souboru se neprovede. Pokud se soubor otevře a `(mode & ios_base::ate) != 0`, metoda `open` šablony `basic_filebuf` přesune ukazatel souboru na konec souboru.

Jestliže nelze soubor otevřít nebo je již otevřený, metoda `rdbuf()->open` vrací nulu a konstruktor nastaví příznak `failbit`.

Šablona využívá vyrovnávací paměť typu šablony třídy `basic_filebuf<charT, traits>`, která je odvozena od šablony třídy `basic_streambuf` a obsahuje řadu předefinovaných virtuálních metod, které pracují se souborem.

Šablona nemá uživatelem deklarovaný destruktor, tj. je použit implicitně deklarovaný destruktor, kterým se mj. zavolá destruktor šablony třídy `basic_filebuf`. Ten uzavře případně otevřený soubor, s nímž je proud sdružen. To znamená, že soubor není nutné explicitně zavírat, pokud není potřebné otevřít jiný soubor.

V prostředí Visual C++ 2003 šablona obsahuje ještě jeden konstruktor:

```
explicit basic_ifstream(FILE* _File);
```

Tento konstruktor sdruží proud přesněji jeho vyrovnávací paměť se souborem typu `FILE`, na který ukazuje parametr `_File`. Parametr `_File` je předán do konstruktoru šablony třídy `basic_filebuf`. Destruktor v tomto případě neuzavře soubor.

Metody

```
basic_filebuf<charT, traits>* rdbuf() const;
```

Vrací ukazatel na vyrovnávací paměť, se kterou je proud sdružen.

```
bool is_open() const;
```

Vrací výsledek volání `rdbuf()->is_open()`, tj. vrací `true`, pokud je soubor otevřen.

```
void open(const char* s, ios_base::openmode mode = ios_base::in);
```

Otevře soubor stejným způsobem jako druhý konstruktor.

```
void close();
```

Volá `rdbuf()->close()`. Metoda `close` šablony `basic_filebuf` zavře soubor voláním obyčejné funkce `std::fclose` a pokud je již soubor uzavřen nebo pokud nelze soubor uzavřít, vrací nulu a v tom případě metoda `close` šablony `basic_ifstream` nastaví příznak `failbit`.

Šablona třídy basic_ofstream

Šablona je definována v hlavičkovém souboru `<fstream>`. Má jednoho veřejně přístupného předka `basic_ostream`. Slouží pro zápis do souboru typu `FILE`. Má stejné metody včetně konstruktorů jako šablona `basic_ifstream`. Liší se pouze v implicitní hodnotě parametru `mode`, který je použit v konstruktoru a v metodě `open`:

```
explicit basic_ofstream(const char* s,
                        ios_base::openmode mode= ios_base::out);
void open(const char* s, ios_base::openmode mode = ios_base::out);
```

Konstruktor i metoda `open` volají `rdbuf()->open(s, mode | out)`. Jestliže metoda `rdbuf()->open` vrací nulu, nastavují příznak `failbit`.

V prostředí Visual C++ 2003 šablona obsahuje ještě jeden konstruktor:

```
explicit basic_ofstream(FILE* _File);
```

Má stejný význam jako konstruktor `basic_ifstream(FILE* _File)`.

Příklad

V následujícím příkladu je ukázka zápisu a čtení pole reálných čísel z/do textového souboru pomocí obyčejných funkcí `Uloz` a `Nacti`.

```
void Uloz(const char* JmSoub, const float* Pole, int n, int PocDesMist)
{
   ofstream os(JmSoub);
   if (!os) { // dtto: if (!os.is_open())
      throw ios_base::failure("Chyba pri otevreni souboru");
   }
   os << n << endl;
   os << setprecision(PocDesMist) << fixed;
   for (int i = 0; i < n; i++) os << Pole[i] << ' ';
}

void Nacti(const char* JmSoub, float*& Pole, int& n)
{
   ifstream is(JmSoub);
   if (!is) { // dtto: if (!is.is_open())
      throw ios_base::failure("Chyba pri otevreni souboru");
   }
   is >> n;
   Pole = new float[n];
   for (int i = 0; i < n; i++) is >> Pole[i];
}
```

Soubory se nemusí zavírat explictiním voláním metody `os.close()` resp. `is.close()`. Toto volání provede automaticky destruktor instance `os` resp. `is` při opuštění funkce `Uloz` resp. `Nacti`.

Příklad

V následujícím příkladu je ukázka čtení a zápisu bodů z/do binárního souboru.

```
class TBod {
   enum { MaxNazev = 20 };
   char Nazev[MaxNazev];
   int  X, Y;
public:
   void Nacti(istream& is)
        { is.read(reinterpret_cast<char*>(this), sizeof(TBod)); }
   void Uloz(ostream& os) const
        { os.write(reinterpret_cast<const char*>(this), sizeof(TBod)); }
};

void UlozBody(const TBod* Body, int n, const char* JmSoub)
{
   ofstream os(JmSoub, ios_base::binary); #1
   if (!os) throw ios_base::failure("Chyba pri otevreni souboru");
   for (int i = 0; i < n; i++) Body[i].Uloz(os);
   if (!os) throw ios_base::failure("Chyba pri zapisu do souboru");
}
```

```
void NactiBody(TBod*& Body, int& n, const char* JmSoub)
{
   ifstream is(JmSoub, ios_base::binary); #2
   if (!is) throw ios_base::failure("Chyba pri otevreni souboru");
   is.seekg(0, ios_base::end); #3
   n = is.tellg()/sizeof(TBod); #4
   is.seekg(0); #5
   Body = new TBod[n];
   for (int i = 0; i < n; i++) Body[i].Nacti(is);
   if (!is) throw ios_base::failure("Chyba pri cteni ze souboru");
}
```

Příkaz #1 je totožný s příkazem

```
ofstream os(JmSoub, ios_base::binary | ios_base::out);
```

Není-li příznak `ios_base::out` uveden, doplní se automaticky. Obdobně se přidá příznak `ios_base::in` v příkazu #2.

Příkazy #3 až #5 slouží pro zjištění počtu záznamů v souboru. První příkaz přesune ukazatel na konec souboru. Pomocí metody `tellg` se potom zjistí velikost souboru. Podělením velikosti souboru velikostí třídy `TBod` dostaneme počet záznamů v souboru. Na konec se musí přesunout ukazatel na začátek souboru příkazem #5.

Šablona třídy basic_fstream

Šablona je definována v hlavičkovém souboru `<fstream>`. Má jednoho veřejně přístupného předka `basic_iostream`. Slouží pro čtení i zápis z/do souboru typu `FILE`. Má stejné metody včetně konstruktorů jako šablona `basic_ifstream`. Liší se pouze v implicitní hodnotě parametru `mode`, který je použit v konstruktoru a v metodě `open`:

```
explicit basic_fstream(const char* s,
                  ios_base::openmode mode= ios_base::in | ios_base::out);
void open(const char* s,
          ios_base::openmode mode= ios_base::in | ios_base::out);
```

Konstruktor i metoda `open` volají `rdbuf()->open(s, mode)`. Jestliže metoda `rdbuf()->open` vrací nulu, nastavují příznak `failbit`.

V prostředí Visual C++ 2003 šablona obsahuje ještě jeden konstruktor:

```
explicit basic_fstream(FILE* _File);
```

Má stejný význam jako konstruktor `basic_ifstream(FILE* _File)`.

Rozšiřování možností vstupů a výstupů

Vlastní vstupní a výstupní operátory

Pro formátované vstupy a výstupy slouží přetížené operátory >> a <<. Pro vestavěné typy jsou definovány jako metody šablon tříd `basic_istream` a `basic_ostream`. Pro uživatelské typy se musí definovat jako obyčejné funkce nebo jako metody třídy odvozené od šablony `basic_istream` resp. `basic_ostream`.

Má-li operátor vstupu >> pro typ `X` fungovat podobně jako vestavěné operátory >>, musí se definovat buď jako šablona obyčejné operátorové funkce:

```
template <class charT, class traits>
basic_istream<charT, traits>& operator >>
(basic_istream<charT, traits>& is, X& x)
{
   // Čtení údaje(ů) typu X a uložení do x
   return is;
}
```

nebo jako obyčejná operátorová funkce, např. pro instanci `istream`:

```
istream& operator >> (istream& is, X& x)
{
   // Čtení údaje(ů) typu X a uložení do x
   return is;
}
```

Podobně operátor výstupu << pro typ `X` se musí definovat jedním z těchto způsobů

```
template <class charT, class traits>
basic_ostream<charT, traits>& operator <<
(basic_ostream<charT, traits>& os, const X& x)
{
   // Zápis údaje(ů) typu X z paramteru x
   return os;
}

ostream& operator << (ostream& os, const X& x)
{
   // Zápis údaje(ů) typu X z paramteru x
   return os;
}
```

Proud se musí předávat a vracet vždy odkazem, neboť jinak by se překladač snažil vytvořit jeho kopii, a to je nejen zbytečné, ale i nemožné, protože kopírovací konstruktor šablony třídy `basic_ios` je soukromý. Parametr `x` pro operátor výstupu << lze předávat hodnotou nebo odkazem včetně konstantní reference, zatímco pro operátor vstupu >> musí být typu nekonstantní reference na `X`.

Příklad

V následujícím příkladu je definována třída `TBod`, která má dvě spřátelené operátorové funkce pro čtení ze vstupního proudu `istream` a zápis do výstupního proudu `ostream`. Příklad dále obsahuje obyčejné funkce `NactiBody` a `UlozBody`, které slouží pro čtení a zápis bodů z/do textového souboru, využívající operátorové funkce pro vstup a výstup údajů třídy `TBod`.

```
#define Oddelovac ';'

class TBod {
   enum { MaxNazev = 20 };
   char Nazev[MaxNazev];
   int X, Y;
public:
   friend ostream& operator << (ostream& os, const TBod& t);
   friend istream& operator >> (istream& is, TBod& t);
};
```

```
ostream& operator << (ostream& os, const TBod& t)
{
   os << t.Nazev << Oddelovac << t.X << Oddelovac << t.Y << endl;
   return os;
}

istream& operator >> (istream& is, TBod& t)
{
   char s[20];
   is.getline(t.Nazev, sizeof t.Nazev, Oddelovac);
   is.getline(s, sizeof s, Oddelovac);
   t.X = atoi(s);
   is.getline(s, sizeof s, '\n');
   t.Y = atoi(s);
   return is;
}

void UlozBody(const TBod* Body, int n, const char* JmSoub)
{
   ofstream os(JmSoub);
   if (!os) throw ios_base::failure("Chyba pri otevreni souboru");
   os << n << endl;
   for (int i = 0; i < n; i++) os << Body[i];
   if (!os) throw ios_base::failure("Chyba pri zapisu do souboru");
}

void NactiBody(TBod*& Body, int& n, const char* JmSoub)
{
   ifstream is(JmSoub);
   if (!is) throw ios_base::failure("Chyba pri otevreni souboru");
   is >> n;
   is.ignore(INT_MAX-1, '\n');
   Body = new TBod[n];
   for (int i = 0; i < n; i++) is >> Body[i];
   if (!is) throw ios_base::failure("Chyba pri cteni ze souboru");
}
```

Vlastní manipulátory bez parametrů

Manipulátory bez parametrů jsou funkce s tímto prototypem:

```
třída& manip(třída&);
```

kde *třída* je jméno třídy nebo šablony třídy a *manip* je jméno manipulátoru.

Např. manipulátor `boolalpha` je deklarován v hlavičkovém souboru `<ios>` následovně:

```
ios_base& boolalpha(ios_base& str);
```

Manipulátor `endl` je v hlavičkovém souboru `<ostream>` deklarován následovně:

```
template<class charT, class traits> basic_ostream<charT, traits>&
   endl(basic_ostream<charT, traits>& os);
```

K tomu, aby je bylo možné použít na pozici pravého operandu operátoru >> nebo <<, jsou v šabloně třídy `basic_istream` a `basic_ostream` definovány metody operátoru >> resp. << s parametrem typu ukazatel na funkci.

V šabloně třídy `basic_istream` jsou definovány následující 3 metody:

```
basic_istream<charT, traits>& operator >>
   (basic_istream<charT, traits>& (*pf)(basic_istream<charT, traits>&));
basic_istream<charT, traits>& operator >>
   (basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
basic_istream<charT, traits>& operator >> (ios_base& (*pf)(ios_base&));
```

Obdobně jsou definovány 3 metody v šabloně třídy `basic_ostream`:

```
basic_ostream<charT, traits>& operator <<
   (basic_ostream<charT, traits>& (*pf)(basic_ostream<charT, traits>&));
basic_ostream<charT, traits>& operator <<
   (basic_ios<charT, traits>& (*pf)(basic_ios<charT, traits>&));
basic_ostream<charT, traits>& operator << (ios_base& (*pf)(ios_base&));
```

Všechny tyto operátory mají parametr typu ukazatel na funkci `pf`, která má jeden parametr typu reference na určitou třídu či šablonu a vrací referenci na tuto třídu (šablonu). Všechny provádí totéž, volají `pf(*this)` a vrací `*this`.

Příkazem

```
cin >> boolalpha;
```

se pro instanci `cin` zavolá operátor `>>` s parametrem typu ukazatel na funkci `boolalpha`, která má parametr typu `ios_base`, tedy zavolá se operátor

```
basic_istream<charT, traits>& operator >> (ios_base& (*pf)(ios_base&));
```

Příklad

V následujícím příkladu je definován manipulátor, který vloží do výstupního proudu 10 mezer.

```
template<class charT, class traits>
basic_ostream<charT, traits>& m10 (basic_ostream<charT, traits>& os)
{
   for (int i = 0; i < 10; i++) os << os.widen(' ');
}
```

Příkaz

```
cout << "aaa" << m10 << "bbb";
```

nebo

```
wcout << "aaa" << m10 << "bbb";
```

způsobí výpis následujícího textu:

```
aaa          bbb
```

Pokud manipulátor `m10` bude využíván jen pro jednobajtovou znakovou sadu, lze ho definovat jako obyčejnou funkci následovně:

```
ostream& m10(ostream& os)
{
   for (int i = 0; i < 10; i++) os << ' ';
   return os;
}
```

VSTUPY A VÝSTUPY – POKRAČOVÁNÍ

Rozšiřování možností vstupů a výstupů – pokračování

Vlastní manipulátory s parametry

Manipulátory s parametry jsou funkce s tímto prototypem:

```
smanip manip(parametry);
```

kde:

smanip ..... není normou specifikovaný typ a může být různý pro jednotlivé manipulátory,

manip..... jméno manipulátoru,

parametry ..... libovolné parametry manipulátoru.

Manipulátory, které jsou součástí normy C++, mají pouze jeden parametr.

Řešení v prostředí Visual C++ 2003

V prostředí Visual C++ 2003 je typ *smanip* pro manipulátory, které volají metodu třídy `ios_base` s jedním parametrem, definován jako šablona struktury v hlavičkovém souboru `<iomanip>` přibližně takto:

```
template<class _Arg> struct _Smanip {
   _Smanip(void (*_Left)(ios_base&, _Arg), _Arg _Val)
      : _Pfun(_Left), _Manarg(_Val) { }

   void (*_Pfun)(ios_base&, _Arg);
   _Arg _Manarg;
};
```

Šablona má jeden typový parametr `_Arg`, který představuje typ parametru metody třídy `ios_base`, která se má volat pro daný manipulátor. Šablona obsahuje dva atributy a konstruktor, který tyto atributy inicializuje. Atribut `_Pfun` představuje ukazatel na funkci, která vrací `void` a má dva parametry: referenci na třídu `ios_base` a parametr typu `_Arg`. Atribut `_Manarg` představuje hodnotu typu `_Arg`.

Tato šablona je např. použita pro manipulátor `setw`, jehož definice je přibližně následující:

```
_Smanip<streamsize> setw(streamsize wide)
{ return _Smanip<streamsize>(&swfun, wide); }
```

Ve funkci `setw` se vytvoří instance struktury `_Smanip<streamsize>`, která bude obsahovat ukazatel na funkci `swfun` a hodnotu parametru typu `streamsize`, který se použije při volání metody `width` třídy `ios_base`. Tato instance je výsledkem funkce `setw`.

Funkce `swfun` provede vlastní volání metody `width` třídy `ios_base` s parametrem `wide`:

```
static void swfun(ios_base& iostr, streamsize wide)
{ iostr.width(wide); }
```

Aby bylo možné manipulátory, které vrací instanci šablony `_Smanip`, použít v operátoru `>>` resp. `<<`, jsou definovány následující šablony obyčejných operátorových funkcí:

```
template<class _Elem, class _Traits, class _Arg>
inline basic_istream<_Elem, _Traits>&
operator >> (basic_istream<_Elem, _Traits>& _Istr,
             const _Smanip<_Arg>& _Manip)
{
   (*_Manip._Pfun)(_Istr, _Manip._Manarg);
   return (_Istr);
}

template<class _Elem, class _Traits, class _Arg>
inline basic_ostream<_Elem, _Traits>&
operator<< (basic_ostream<_Elem, _Traits>& _Ostr,
            const _Smanip<_Arg>& _Manip)
{
   (*_Manip._Pfun)(_Ostr, _Manip._Manarg);
   return (_Ostr);
}
```

Příkazem

```
cout << setw(10);
```

se nejprve zavolá funkce `setw`, která vrátí instanci `_Smanip<streamsize>` a pro ni se zavolá operátorová funkce `<<`. Ta zavolá funkci `swfun`, která zavolá `cout.width(10)`.

Obdobně pomocí jiných šablon jsou definovány další standardní manipulátory, mající parametry.

Řešení v prostředí C++ Builder 6

V prostředí C++ Builder 6 jsou manipulátory s jedním parametrem řešeny elegantnějším způsobem než v prostředí Visual C++ 2003. Typ *smanip* pro manipulátory, které volají metodu třídy `ios_base` s jedním parametrem, je definován jako šablona následující struktury v hlavičkovém souboru `<iomanip>`:

```
template <class _Arg>
struct _Ios_Manip_1 {
   typedef _Arg (ios_base::*__f_ptr_type)(_Arg);

   _Ios_Manip_1(__f_ptr_type __f, const _Arg& __arg)
    : _M_f(__f), _M_arg(__arg) {}

   void operator()(ios_base& __ios) const
     { (__ios.*_M_f)(_M_arg); }

   __f_ptr_type _M_f;
   _Arg _M_arg;
};
```

Šablona má jeden typový parametr `_Arg`, který představuje typ parametru metody třídy `ios_base`, která se má volat pro daný manipulátor. Šablona obsahuje deklaraci typu `__f_ptr_type`, což je třídní ukazatel na metodu třídy `ios_base`, která vrací `_Arg` a má jeden parametr typu `_Arg`. Dále obsahuje dva atributy: jeden typu `__f_ptr_type` a druhý typu `_Arg`. Tyto atributy jsou inicializovány konstruktorem. Nejpodstatnější složkou je operátor volání funkce, který zajišťuje volání určité metody třídy `ios_base` pro daný manipulátor pomocí třídního ukazatele.

Tato šablona je např. použita pro manipulátor `setw` a `setprecision`. Definice manipulátoru `setw` je následující:

```
inline _Ios_Manip_1<streamsize> setw(int __n)
{
  _Ios_Manip_1<streamsize>::__f_ptr_type __f = &ios_base::width;
   return _Ios_Manip_1<streamsize>(__f, __n);
}
```

Ve funkci `setw` se vytvoří instance struktury `_Ios_Manip_1<streamsize>`, která bude obsahovat ukazatel na metodu `width` třídy `ios_base` a hodnotu parametru typu `streamsize`, který se použije při volání metody `width`. Tato instance je výsledkem funkce `setw`.

Aby bylo možné manipulátory, které vrací instanci šablony `_Ios_Manip_1`, použít v operátoru >> resp. <<, jsou definovány následující šablony obyčejných operátorových funkcí:

```
template <class _CharT, class _Traits, class _Arg>
inline basic_istream<_CharT, _Traits>&
operator >> (basic_istream<_CharT, _Traits>& __in,
             const _Ios_Manip_1<_Arg>& __f)
{ __f(__in); return __in; }

template <class _CharT, class _Traits, class _Arg>
inline basic_ostream<_CharT, _Traits>&
operator << (basic_ostream<_CharT, _Traits>& __os,
             const _Ios_Manip_1<_Arg>& __f)
{ __f(__os); return __os; }
```

Příkazem

```
cout << setw(10);
```

se nejprve zavolá funkce `setw`, která vrátí instanci `_Ios_Manip_1<streamsize>` a pro ni se zavolá operátorová funkce <<. Ta zavolá operátor volání funkce instance `_Ios_Manip_1<streamsize>`, který zavolá `cout.width(10)`.

Lze definovat i vlastní manipulátory s parametry, a to buď pomocí šablony, kterou bude možné využít i pro další vlastní manipulátory stejného typu nebo pomocí třídy určené jen pro jeden konkrétní manipulátor.

Příklad

V následujícím příkladu je definován manipulátor `endlx`, který vloží do výstupního proudu `n` nových řádků, a to pomocí struktury.

```
struct s_endlx {
public:
   s_endlx(int _n) : n(_n) {}
   int n;
};

inline s_endlx endlx(int n)
{
   return s_endlx(n);
}
```

```
template <class charT, class traits>
basic_ostream<charT, traits>&
operator << (basic_ostream<charT, traits>& os, const s_endlx& t)
{
   for (int i = 0; i < t.n; i++) os << endl;
   return os;
}
```

Struktura `s_endlx` slouží k uchování parametru `n` manipulátoru. Obsahuje jeden veřejně přístupný atribut `n` a konstruktor, který jej inicializuje. Manipulátor `endlx` vytvoří instanci této struktury a tu také vrací. Dále je definována šablona obyčejné operátorové funkce `<<`, která provede vlastní výpis `n` nových řádků.

Příkaz

```
cout << "aaa" << endlx(2) << "bbb";
```

provede následující výpis:

```
aaa


bbb
```

Příklad

Pokud by programátor chtěl vytvořit více vlastních manipulátorů, které by měly jeden parametr libovolného typu a byly by určeny pro výstupní proud `basic_ostream`, musel by definovat např. následující šablonu třídy a následující funkce.

```
template <class charT, class traits, class arg>
class o_manip_1 {
public:
   typedef void (*fptr)(basic_ostream<charT, traits>&, arg);

   o_manip_1(fptr _f, const arg& _a) : f(_f), a(_a) {}
   void operator()(basic_ostream<charT, traits>& os) const { f(os, a); }
protected:
   fptr  f;
   arg   a;
};

template <class charT, class traits>
inline void _endlx(basic_ostream<charT, traits>& os, int n)
{
   for (int i = 0; i < n; i++) os << endl;
}

template <class charT, class traits>
inline o_manip_1<charT, traits, int> endlx(int n) // #1
{
   return o_manip_1<charT, traits, int>(_endlx, n);
}

inline o_manip_1<char, char_traits<char>, int> endlx(int n) // #2
{
   return o_manip_1<char, char_traits<char> , int>(_endlx, n);
}
```

```
template <class charT, class traits, class arg>
basic_ostream<charT, traits>&
operator << (basic_ostream<charT, traits>& os,
             const o_manip_1<charT, traits, arg>& t)
{
   t(os);
   return os;
}
```

Šablona třídy `o_manip_1` má dva atributy, oba jsou inicializovány jejím konstruktorem. Atribut `f` obsahuje ukazatel na funkci, která má dva parametry, první typu reference na `basic_ostream`, druhý typu `Arg`, což je typový parametr této šablony, představující typ parametru manipulátoru. Šablona dále obsahuje operátor volání funkce, který zavolá funkci, na níž ukazuje atribut `f`. Dále je definována šablona obyčejné funkce `_endlx`, která provede vlastní činnost manipulátoru. Její adresa je obsažena v atributu `f` šablony `o_manip_1`. Manipulátor `endlx` je definován jako šablona obyčejné funkce #1, která vytvoří instanci šablony `o_manip_1` a tu také vrací. Potom je definována přetížená obyčejná funkce `endlx` #2 pro instanci `o_manip_1<char, char_traits<char>, int>`, aby se volání manipulátoru `endlx` nemuselo kvalifikovat skutečnými hodnotami jeho typových parametrů.

Princip použití manipulátoru `endlx` je stejný jako v předchozím příkladu. Příkazem

```
cout << "aaa" << endlx(2) << "bbb";
```

se nejprve zavolá přetížená funkce `endlx`, potom šablona obyčejné operátorové funkce `<<`, v níž se zavolá operátor volání funkce šablony `o_manip_1`, který nakonec zavolá šablonu funkce `_endlx`.

Pokud by se však nedefinovala přetížená obyčejná funkce `endlx` #2 muselo by se volání manipulátoru `endlx` kvalifikovat skutečnými hodnotami jeho typových parametrů následovně:

```
cout << "aaa" << endlx<char, char_traits<char> >(2) << "bbb";
```

Paměťové datové proudy

Paměťové datové proudy uchovávají ve vyrovnávací paměti `basic_stringbuf` řetězec znaků založený na šabloně třídy `basic_string`. Jedná se o šablony tříd `basic_istringstream`, `basic_ostringstream`, `basic_stringstream`, které jsou definovány v hlavičkovém souboru `<sstream>`. Všechny tyto šablony mají stejné parametry, např. deklarace šablony třídy `basic_istringstream` je následující:

```
template <class charT, class traits = char_traits<charT>,
          class Allocator = allocator<charT> >
class basic_istringstream;
```

Od ostatních šablon souvisejících s objektovými datovými proudy mají tyto šablony navíc parametr `Allocator`. Jedná se o třídu sloužící pro alokaci a dealokaci paměti, která se používá v šabloně třídy `basic_string`. Implicitně je použita šablona třídy `allocator`. Tyto tři parametry mají také šablony `basic_stringbuf` a `basic_string`.

Pro otevření datového proudu lze používat pouze příznaky `ios_base::in` a `ios_base::out`. Jiné příznaky jsou ignorovány.

Typy

Všechny tři šablony paměťových datových proudů obsahují deklaraci typů `char_type`, `int_type`, `pos_type`, `off_type` a `traits_type`. Deklarace je stejná jako v šabloně `basic_istream`.

Atributy

Šablony obsahují jeden soukromý atribut `sb`, který představuje vyrovnávací paměť typu `basic_stringbuf<charT, traits, Allocator>`. Jméno tohoto atributu není závazné.

Metody

Všechny šablony paměťových datových proudů obsahují tyto tři metody.

```
basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
```

Vrací `&sb`, tj. ukazatel na vyrovnávací paměť, se kterou je proud sdružen.

```
basic_string<charT, traits, Allocator> str() const;
```

Vrací `rdbuf()->str()`, tj. řetězec znaků, který je obsažen v paměťovém proudu přesněji v jeho vyrovnávací paměti bez ohledu na aktuální pozici ukazatele v proudu.

```
void str(const basic_string<charT,traits,Allocator>& s);
```

Volá metodu `rdbuf()->str(s)`, která uloží do vyrovnávací paměti řetězec znaků `s`. Pozice ukazatele v proudu se nezmění.

Šablona třídy basic_istringstream

Šablona má jednoho veřejně přístupného předka `basic_istream`. Slouží pro čtení dat z paměťového datového proudu. Přečtením nějakého údaje z paměťového datového proudu nedojde k odstranění údaje z tohoto proudu ale jen k posunu ukazatele v proudu za načtený údaj.

Kromě složek společných pro všechny šablony paměťových datových proudů obsahuje dva konstruktory:

```
explicit basic_istringstream(ios_base::openmode which = ios_base::in);
explicit basic_istringstream(const basic_string<charT, traits,
                             Allocator>& str,
                             ios_base::openmode which = ios_base::in);
```

První verze inicializuje předka `basic_istream` předáním ukazatele na vyrovnávací paměť `&sb` a atribut `sb` inicializuje konstruktorem `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)`, který vytvoří prázdnou vyrovnávací paměť.

Druhá verze provede totéž, ale atribut `sb` inicializuje konstruktorem `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)`, který vytvoří vyrovnávací paměť a zkopíruje do ní řetězec `str`. Pozice ukazatele v proudu bude nulová (začátek proudu).

Šablona třídy basic_ostringstream

Šablona má jednoho veřejně přístupného předka `basic_ostream`. Slouží pro zápis dat do paměťového datového proudu.

Obsahuje stejné konstruktory jako šablona `basic_istringstream`, od nichž se liší pouze v implicitní hodnotě parametru `which`:

```
explicit basic_ostringstream(ios_base::openmode which = ios_base::out);
explicit basic_ostringstream(const basic_string<charT, traits,
                             Allocator>& str,
                             ios_base::openmode which = ios_base::out);
```

První verze inicializuje předka `basic_ostream` předáním ukazatele na vyrovnávací paměť `&sb` a atribut `sb` inicializuje konstruktorem `basic_stringbuf<charT, traits, Allocator>(which | ios_base::out)`, který vytvoří prázdnou vyrovnávací paměť.

Druhá verze provede totéž, ale atribut `sb` inicializuje konstruktorem `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)`, který vytvoří vyrovnávací paměť a zkopíruje do ní řetězec `str`. Pozice ukazatele v proudu bude nulová (začátek proudu).

Příklad

```
int main()
{
   int a = 10;
   ostringstream oss("Text1 ");
   oss.seekp(0, ios_base::end); #1
   oss << "Text2 " << a << endl; #2
   cout << oss.str();
   cin.get();
   return 0;
}
```

V uvedeném příkladu se musí provést příkaz #1 pro přesun ukazatele na konec proudu, jinak by provedením příkazu #2 došlo k přepsání textu `"Text1 "`. Na obrazovku se vypíše následující text:

```
Text1 Text2 10
```

Šablona třídy basic_stringstream

Šablona má jednoho veřejně přístupného předka `basic_iostream`. Slouží pro čtení i zápis dat z/do paměťového datového proudu.

Obsahuje stejné konstruktory jako šablona `basic_istringstream`, od nichž se liší pouze v implicitní hodnotě parametru `which`:

```
explicit basic_stringstream
         (ios_base::openmode which = ios_base::out|ios_base::in);
explicit basic_stringstream
         (const basic_string<charT, traits, Allocator>& str,
          ios_base::openmode which = ios_base::out|ios_base::in);
```

První verze inicializuje předka `basic_iostream` předáním ukazatele na vyrovnávací paměť `&sb` a atribut `sb` inicializuje konstruktorem `basic_stringbuf<charT, traits, Allocator>(which)`, který vytvoří prázdnou vyrovnávací paměť.

Druhá verze provede totéž, ale atribut `sb` inicializuje konstruktorem `basic_stringbuf<charT, traits, Allocator>(str, which)`, který vytvoří vyrovnávací paměť a zkopíruje do ní řetězec `str`. Pozice ukazatele v proudu bude nulová (začátek proudu).

STANDARDNÍ KNIHOVNA C++

Součástí normy jazyka C++ je i *standardní knihovna jazyka C++* (angl. *Standard C++ Library*). Tato knihovna obsahuje komponenty pro podporu jazyka C++, diagnostiku, všeobecné utility, řetězce znaků, lokalizační nástroje, kontejnery, iterátory, algoritmy, práci s čísly a komponenty pro vstup a výstup. Všechny komponenty jsou součástí prostoru jmen `std`.

Knihovna jazyka C++ obsahuje především knihovnu různých šablon tříd a obyčejných funkcí, které jsou základem generického programování v C++. Generické programování je jeden z programovacích stylů. Mezi programovací styly patří:

- *strukturované programování* – programování pomocí funkcí, které operují nad daty – neobjektový přístup,
- *objektově orientované programování*,
- *generické programování* – je založeno na vytváření abstraktních vzorů funkcí a tříd pomocí generických konstrukcí (šablon).

V prostředí C++ Builder 6 existují dvě implementace standardní C++ knihovny:

- Rogue Wave C++ Standard Template Library – starší knihovna zahrnutá pro účely zpětné kompatibility. Pokud se má použít v prostředí C++ Builder 6, musí se definovat makro `_USE_OLD_RW_STL`.
- STLport 4.5 (Standard Template Library portable) – „open source“ knihovna podporovaná řadou kompilátorů a platforem, vyvinutou firmou Silicon Graphics, Inc. (SGI). Je rozsáhlejší než knihovna Rogue Wave. V prostředí C++ Builder 6 je implicitní knihovnou.

V prostředí Visual C++ 2003 je k dispozici pouze jedna verze C++ knihovny, jejímž autorem je P. J. Plauger. Knihovnu STLport lze v prostředí Visual C++ 2003 také použít. Je k dispozici na webových stránkách http://www.stlport.com/ a http://www.sgi.com/tech/stl/

Všechny tyto knihovny vyhovují normě jazyka C++. Liší se množstvím dalších nabízených komponent, v názvosloví a v použitých algoritmech.

Nejpropracovanější knihovnou je knihovna STLport, která kromě toho, že obsahuje řadu dalších komponent, kategorizuje jednotlivé komponenty a zavádí novou terminologii. V těchto přednáškách je standardní knihovna jazyka C++ popisována pomocí terminologie knihovny STLport.

Koncepty a modely

Pojem *koncept* a s ním související pojmy jsou zavedeny v knihovně STLport pro určité názvy, které se ve standardní knihovně jazyka C++ vyskytují.

Jazyk C++ umožňuje definováním šablony funkce popsat velkou množinu funkcí lišících se typem parametrů. Neumožňuje však rozhodování, zda daný typ je vhodný pro příslušnou šablonu. Např. pro šablonu `minimum`

```
template <class T> T minimum(T a, T b)
{ return a < b ? a : b; }
```

je vhodný typ `int`, kdežto třída `TA`, která je definována takto

```
class TA { };
```

není vhodným typem pro šablonu `minimum`, protože pro ni není definován operátor <.

U takovéto funkce je zřejmé, jaké mají být její parametry. Ale u složitých šablon to zřejmé být nemusí. Proto se pro parametr šablony specifikuje množina požadavků, které musí splňovat a této množině se přidělí nějaké jméno – vznikne *koncept*.

Koncept (angl. *concept*) je tedy množina požadavků na typové parametry generických konstrukcí.

Model konceptu (angl. *model of a concept*) je typ, který vyhovuje danému konceptu.

Zjemnění konceptu (angl. *refinement of a concept*) – koncept *X* je zjemněním konceptu *Y*, pokud množina požadavků konceptu *Y* je podmnožinou požadavků konceptu *X*. Koncept *X* tedy vyhovuje konceptu *Y* a obsahuje navíc další požadavky. Určitý koncept může být zjemněním i několika jiných konceptů. Zjemnění konceptu je podobné dědičnosti tříd.

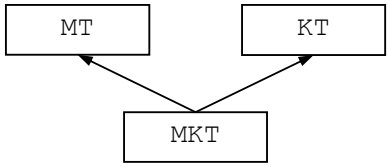
Pojmy vysvětlíme na uvedeném příkladu. Šablona funkce `minimum` předpokládá, že pro její parametr `T` existuje operátor <. Koncept popisující tento požadavek nazveme *minimalizovatelné typy* (zkr. *MT*) a definujeme ho např. takto. Koncept *MT* je množina typů `T` takových, že pokud `x` a `y` jsou typu `T`, pak výraz `x < y`:

a) je syntakticky v pořádku (lze ho přeložit) – musí být k dispozici operátor <,
b) znamená, že `x` je menší než `y` (přetížený operátor < může mít jiný význam, než by se očekávalo),
c) je typu, který lze implicitně konvertovat na typ `bool`.

Modelem konceptu *MT* je např. typ `int`. Ale modelem konceptu *MT* není třída `TA`.

Šablona funkce `minimum` vrací instanci typu `T`. Ta se vytváří pomocí kopírovacího konstruktoru typu `T`, což je další požadavek této šablony. Tento koncept nazveme *kopírovatelné typy* (zkr. *KT*) a definujeme jej např. takto. Koncept *KT* je libovolný základní datový typ nebo třída, která má definován veřejně přístupný kopírovací konstruktor. Modelem konceptu *KT* je nejen typ `int`, ale i třída `TA`, protože obsahuje implicitně deklarovaný kopírovací konstruktor.

Parametr `T` šablony funkce `minimum` musí splňovat požadavky dvou konceptů. Proto zavedeme koncept *minimalizovatelné kopírovatelné typy* (zkr. *MKT*). Koncept *MKT* je *zjemněním* konceptu *MT* a *KT*. Modelem konceptu *MKT* je z typu `int` a `TA` pouze typ `int`. Vztahy mezi koncepty lze znázorňovat orientovaným acyklickým grafem podobně jako dědickou hierarchii tříd. Vztah mezi koncepty *MT, KT* a *MKT* je znázorněn na obr. 1.



Obr. 1 Zjemnění konceptů MT a KT

Aby bylo zřejmé, jaký koncept pro daný typový parametr šablony je požadován, uvádí se název konceptu jako název daného typového parametru. Šablona funkce `minimum` by byla definována takto:

```
template <class MKT> MKT minimum(MKT a, MKT b)
{ return a < b ? a : b; }
```

V knihovně STLport jsou definovány koncepty kontejnerů, iterátorů a dalších objektů.

Časová složitost algoritmů

Norma jazyka C++ specifikuje pro jednotlivé operace a algoritmy časovou složitost. Existují tři základní druhy časové složitosti algoritmů:

- maximální – označuje se symbolem *O*(*výraz*),
- průměrná – označuje se symbolem Θ(*výraz*),

- minimální – označuje se symbolem Ω(*výraz*).

Nejčastěji se u algoritmů udává složitost *O*(*výraz*).

Podle typu výrazu uvedeného v závorkách za symbolem časové složitosti rozeznáváme např. tyto časové složitosti:

- konstantní (angl. *constant time complexity*) – jako *výraz* je uvedena konstanta, např. *O*(1),
- lineární (angl. *linear time complexity*) – jako *výraz* je použita proměnná, např. počet prvků, se kterými se má provést určitá operace, např. *O*(*N*),
- logaritmická (angl. *logarithmic time complexity*) – např. *O*(log *N*), přesněji $O(\log_2 N)$,
- exponenciální (angl. *exponential time complexity*) – např. $O(2^N)$.

Druhy uspořádání

V normě jazyka C++ jsou definovány tři druhy uspořádání posloupnosti prvků:

- *parciální uspořádání* (angl. *partial ordering*),
- *ostré slabé uspořádání* (angl. *strict weak ordering*),
- *úplné uspořádání* (angl. *total ordering*).

Pro *parciální uspořádání* musí relace < (ve všeobecnosti relace *R*) splňovat vlastnosti uvedené v následující tabulce.

Vlastnost	Platí
areflexivita	není pravda, že `x < x`
asymetrie	jestliže `x < y`, potom není pravda, že `y < x`
tranzitivita	jestliže `x < y` a zároveň `y < z`, potom `x < z`

Pro *ostré slabé uspořádání* musí relace < splňovat všechny vlastnosti parciálního uspořádání a navíc následující vlastnost:

Vlastnost	Platí
tranzitivita ekvivalence	jestliže `x` je ekvivalentní k `y` a `y` je ekvivalentní k `z`, potom `x` je ekvivalentní k `z`. Dva objekty x a y jsou *ekvivalentní*, pokud platí `!(x < y) && !(y < x)`

Pro *úplné uspořádání* musí relace < splňovat všechny vlastnosti ostrého slabého uspořádání a navíc musí platit, že ekvivalence je totéž co rovnost, tj.

jestliže `!(x < y) && !(y < x)`, potom `x == y`.

Vlastnostem parciálního uspořádání vyhovuje taková množina prvků, v níž mezi některými prvky není definován operátor <. Např. chceme porovnávat řetězce znaků, přičemž operátor < je definován jen pro znaky anglické abecedy, pro znak "č" není definován. Platí:

```
"ab" < "ac" = true
"ab" < "ač" = false
"ab" > "ač" = false
"ač" < "ad" = false
"ač" > "ad" = false
```

Takto definovaná relace < vyhovuje vlastnostem areflexivity, asymetrie a tranzitivity, ale nevyhovuje vlastnosti tranzitivy ekvivalence: `"ab"` je ekvivalentní k `"ač"`, `"ač"` je ekvivalentní k

"ad", ale "ab" není ekvivalentní k "ad". Z toho vyplývá, že tato relace < vyhovuje parciálnímu uspořádání, ale nevyhovuje ostrému slabému uspořádání.

Vlastnostem parciálního uspořádání vyhovují také vztahy mezi cv-modifikátory.

Vlastnostem ostrého slabého uspořádání vyhovuje např. pole prvků typu třídy `TStudent`. Třída `TStudent` má definován operátor < a operátor ==. Pomocí operátoru < porovnává studenty podle příjmení a potom podle jména studenta, ale pomocí operátoru == porovnává studenty podle jejich osobního čísla. V poli studentů může existovat více studentů se stejným jménem a příjmením, ale ne se stejným osobním číslem a tudíž neplatí vlastnost ekvivalence. Pokud by jak operátor <, tak i operátor == porovnával studenty podle jejich osobního čísla, vyhovovalo by toto pole vlastnostem úplného uspořádání.

Všeobecné koncepty

Norma jazyka C++ popisuje všeobecné požadavky na typové parametry šablon. Knihovna STLport pro tyto požadavky zavádí koncepty.

V této kapitole jsou použity následující symboly:

`X` ..... typ, který je modelem daného konceptu,

`x, y, z` ..... objekty typu `X`.

Koncept `Assignable`

Typ je modelem konceptu `Assignable`, pokud je možné kopírovat objekty tohoto typu a přiřadit hodnotu do objektu tohoto typu.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Po provedení výrazu
`X(x)`	`X`	`X(x)` je kopií `x`
`X x(y);` `X x = y;`		`x` je kopií `y`
`x = y`	`X&`	`x` je kopií `y`

Norma jazyka C++ definuje pro koncept `Assignable` pouze poslední z uvedených požadavků, tj. `x = y`.

Koncept `DefaultConstructible`

Typ je modelem konceptu `DefaultConstructible`, jestliže má implicitní konstruktor, který, je-li to možné, provede konstrukci objektu bez inicializace jeho složek.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ
`X()`	`X`
`X x;`	

Koncept `EqualityComparable`

Typ je modelem konceptu `EqualityComparable`, jestliže objekty tohoto typu lze porovnávat pomocí operátoru ==, který musí splňovat vlastnosti relace rovnosti.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam
`x == y`	lze implicitně konvertovat na typ `bool`	
`x != y`	lze implicitně konvertovat na typ `bool`	ekvivalent výrazu `!(x == y)`.

Relace == má vlastnosti uvedené v následující tabulce.

Vlastnost relace ==	Platí
identita	jestliže `&x == &y`, `potom x == y`
reflexivita	`x == x`
symetrie	jestliže `x == y`, potom `y == x`
tranzitivita	jestliže `x == y` a zároveň `y == z`, potom `x == z`

Koncept `LessThanComparable`

Typ je modelem konceptu `LessThanComparable`, jestliže objekty tohoto typu lze porovnávat pomocí operátoru <, který musí splňovat vlastnosti *parciálního uspořádání*.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam
`x < y`	lze implicitně konvertovat na typ `bool`	
`x > y`	lze implicitně konvertovat na typ `bool`	ekvivalent výrazu `y < x`
`x <= y`	lze implicitně konvertovat na typ `bool`	ekvivalent výrazu `!(y < x)`
`x >= y`	lze implicitně konvertovat na typ `bool`	ekvivalent výrazu `!(x < y)`

Relace < má vlastnosti uvedené v následující tabulce.

Vlastnost relace <	Platí
areflexivita	není pravda, že `x < x`
asymetrie	jestliže `x < y`, potom není pravda, že `y < x`
tranzitivita	jestliže `x < y` a zároveň `y < z`, potom `x < z`

Norma jazyka C++ vyžaduje, aby tento koncept vyhovoval vlastnostem *ostrého slabého uspořádání*. V knihovně STLport je pro tyto vlastnosti definován samostatný koncept `StrictWeakOrdering`.

Koncept `CopyConstructible`

Tento koncept je definován v normě jazyka C++, ale ne v knihovně STLport.

Model konceptu musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Požadavek
`X(x)`		`x` je ekvivalentem výrazu `X(x)`
`X(u)`		`u` je ekvivalentem výrazu `X(u)`
`x.~X()`		
`&x`	`X*`	výsledkem je adresa objektu `x`
`&u`	`const X*`	výsledkem je adresa objektu `u`

Legenda:

u..... objekt typu `const X`.

STANDARDNÍ KNIHOVNA C++ – POKRAČOVÁNÍ

Relační operátory

Koncept `EqualityComparable` požaduje, aby operátor `!=` byl definován pomocí operátoru ==. Podobně koncept `LessThanComparable` specifikuje, jak definovat >, <= a >= pomocí operátoru <. Pokud pro daný typ je potřebné definovat operátory `!=`, >, <= a >=, lze využít šablony definované v hlavičkovém souboru `<utility>`.

Hlavičkový soubor `<utility>` obsahuje šablony následujících čtyř operátorových funkcí, které používají operátor == nebo <.

```
namespace std {
   namespace rel_ops {
      template<class T> bool operator!=(const T& x, const T& y)
         { return !(x == y); }
      template<class T> bool operator> (const T& x, const T& y)
         { return y < x; }
      template<class T> bool operator<=(const T& x, const T& y)
         { return !(y < x); }
      template<class T> bool operator>=(const T& x, const T& y);
         { return !(x < y); }
   }
}
```

Tyto šablony jsou definovány ve vnořeném prostoru jmen `rel_ops`.

V určité třídě stačí tedy definovat pouze operátor == a <, pro ostatní relační operátory se použijí instance uvedených šablon operátorových funkcí, pokud se do zdrojového souboru, v němž je daná třída definována, zahrne hlavičkový soubor `<utility>` a zpřístupní se složky prostoru jmen `rel_ops` např. pomocí direktivy `using`.

Příklad

```
#include <utility> // #1
using namespace rel_ops; // #2

class TA {
   int   a, b;
public:
   TA(int _a, int _b) : a(_a), b(_b) {}
   bool operator < (const TA& t) const { return a < t.a; }
   bool operator == (const TA& t) const { return a == t.a; }
};

TA A(0), B(1);
```

Třída `TA` má pouze operátorové funkce < a ==. Výrazy `A > B`, `A != B` budou přesto správné, protože jsou ve zdrojovém textu zahrnuty řádky #1 a #2.

Šablona pair

Pro heterogenní páry hodnot je v hlavičkovém souboru `<utility>` definována následující šablona struktury `pair`:

```
template <class T1, class T2>
struct pair {
   typedef T1 first_type;
   typedef T2 second_type;
   T1 first;
   T2 second;
   pair();
   pair(const T1& x, const T2& y);
   template<class U, class V> pair(const pair<U, V> & p);
};
```

Popis složek

`pair``();`

Inicializuje atributy takto: `first(T1())`, `second(T2())`.

`pair``(const T1& x, const T2& y);`

Inicializuje atributy parametry `x` a `y`.

`template<class U, class V>` **`pair`**`(const pair<U, V> & p);`

Inicializuje atributy této šablony odpovídajícími atributy instance šablony `pair<U, V>` za předpokladu, že lze provést implicitní konverzi z typu `U` na `T1` a z typu `V` na `T2`.

V hlavičkovém souboru `<utility>` je dále definována šablona obyčejné funkce `make_pair` a šest šablon relačních operátorů pro porovnání dvou instancí šablon `pair`:

```
template <class T1, class T2>
pair<T1, T2> make_pair(const T1& x, const T2& y);

// jedna z operátorových funkcí
template <class T1, class T2>
bool operator==(const pair<T1,T2>&, const pair<T1,T2>&);
```

Funkce `make_pair` vytvoří instanci `pair<T1, T2>(x, y)`, kterou vrací.

Šablona auto_ptr

Šablona `auto_ptr` uchovává ukazatel na objekt obdržený pomocí operátoru `new`. Tento objekt automaticky ruší (dealokuje) při své destrukci. Šablona je definována v hlavičkovém souboru `<memory>` takto:

```
template<class X> class auto_ptr {
   template <class Y> struct auto_ptr_ref;
   X* _p;
public:
   typedef X element_type;
   explicit auto_ptr(X* p = 0) throw();
   auto_ptr(auto_ptr&) throw();
   template<class Y> auto_ptr(auto_ptr<Y>&) throw();
   auto_ptr& operator=(auto_ptr&) throw();
   template<class Y> auto_ptr& operator=(auto_ptr<Y>&) throw();
   ~auto_ptr() throw();

   X& operator*() const throw();
   X* operator->() const throw();
   X* get() const throw();
```

```
    X* release() throw();
    void reset(X* p =0) throw();
    auto_ptr(auto_ptr_ref<X>) throw();
    template<class Y> operator auto_ptr_ref<Y>() throw();
    template<class Y> operator auto_ptr<Y>() throw();
};
```

Šablona obsahuje jeden neveřejný atribut `_p` – ukazatel na objekt typu `X`. Jméno `_p` není normou specifikováno. Šablona představuje striktní vlastnictví určitého objektu. Při kopírování `auto_ptr` se přenese vlastnictví objektu na cílový `auto_ptr`. Jestliže více než jeden `auto_ptr` vlastní stejný objekt, chování programu je nedefinovatelné. Pokud `auto_ptr` nevlastní objekt, ukazatel na objekt v šabloně `auto_ptr` je nulový.

Typy

`template <class Y> struct auto_ptr_ref;`

Šablona struktury `auto_ptr_ref` je privátní složkou a uchovává referenci na `auto_ptr`.

Konstruktory

`explicit` **`auto_ptr`**`(X*p=0) throw();`

Inicializuje atribut _p parametrem `p`. Parametr `p` musí být ukazatel na `X` nebo na třídu odvozenou z třídy `X` nebo je nulový.

`auto_ptr``(auto_ptr& a) throw();`

Kopírovací konstruktor. Tato šablona (`this`) převezme vlastnictví ukazatele na objekt `X`, který vlastnila instance `a`.

`template<class Y>` **`auto_ptr`**`(auto_ptr<Y>& a) throw();`

Vytvoří instanci šablony `auto_ptr<X>` z instance šablony `auto_ptr<Y>` za předpokladu, že `Y*` lze implicitně zkonvertovat na `X*` (bez použití operátoru přetypování). Jedná se zejména o konverzi z potomka na předka. Tato šablona (`this`) převezme vlastnictví ukazatele na objekt `X`, který vlastnila instance `a`.

Metody

`auto_ptr&` **`operator=`**`(auto_ptr& a) throw();`

Zkopíruje instanci `a` do instance `*this`. Jestliže tato šablona (`this`) vlastní ukazatel na objekt `X`, nejprve jej dealokuje. Potom tato šablona (`this`) převezme vlastnictví ukazatele na objekt `X`, který vlastnila instance `a`.

`template<class Y> auto_ptr&` **`operator=`**`(auto_ptr<Y>& a) throw();`

Provede totéž co kopírovací operátor přiřazení za předpokladu, že `Y*` lze implicitně zkonvertovat na `X*` (bez použití operátoru přetypování).

`X*` **`get`**`() const throw();`

Vrací ukazatel na objekt `X`, který je součástí této šablony, tj. vrací `_p`.

`X&` **`operator*`**`() const throw();`

Vrací referenci na objekt `X`, který tato šablona vlastní, tj. vrací `*get()`.

`X*` **`operator->`**`() const throw();`

Vrací ukazatel na objekt `X`, který je součástí této šablony, tj. vrací `get()`.

`X* release() throw();`

Uvolní vlastnictví ukazatele na objekt `X`, tj. ukazatel vynuluje, ale nedealokuje. Vrací původní hodnotu ukazatele.

`void reset(X* p =0) throw();`

Jestliže tato šablona (`this`) vlastní ukazatel na objekt `X`, nejprve jej dealokuje. Potom do tohoto ukazatele (atributu `_p`) uloží hodnotu parametru `p`.

`template<class Y> operator auto_ptr<Y>() throw();`

Šablona operátoru přetypování, která vytvoří instanci šablony `auto_ptr<Y>` z instance šablony `this`. Tato šablona uvolní vlastnictví ukazatele na objekt `X` voláním `release()`. Jeho vlastnictví převezme instance `auto_ptr<Y>`, kterou operátor vrací.

Příklad

```
class TA {
   int a, b;
public:
   TA(int _a, int _b) : a(_a), b(_b) {}
   int GetA() const { return a; }
   int GetB() const { return b; }
};
void f(const TA& A)
{ cout << A.GetA() << ' ' << A.GetB() << endl; }

void g(const TA* A)
{ cout << A->GetA() << ' ' << A->GetB() << endl; }

int main()
{
   auto_ptr<TA> A(new TA(10, 20));
// auto_ptr<TA> A = new TA(10, 20); // nelze provést
   cout << A->GetA() + A->GetB() << endl; // volání operátoru ->
   f(*A); // volání operátoru *
   g(A.get());
   auto_ptr<TA> B;
   B = A; // vlastnictví převezme B, A obsahuje nulový ukazatel
          // volání operátoru =
   auto_ptr<TA> C(new TA(1, 2));
   TA* D = C.release();
   delete D;
   return 0;
}
```

V uvedeném příkladu se alokuje instance třídy `TA`, kterou obsahuje instance `A` šablony `auto_ptr`. Ukazatel na `TA` z instance `A` později přebírá instance `B`. Po ukončení funkce `main`, dojde automaticky k dealokaci ukazatele, který vlastní instance `B`.

ITERÁTORY

Iterátor (angl. *iterator)* je zevšeobecněním pojmu ukazatel: je to nějaký objekt, který ukazuje na jiný objekt. Slouží zejména k procházení nějakého rozsahu objektů. Jestliže iterátor „ukazuje“ na jeden prvek z nějakého rozsahu, potom je možné ho např. inkrementovat a přesunout se na následující prvek z daného rozsahu.

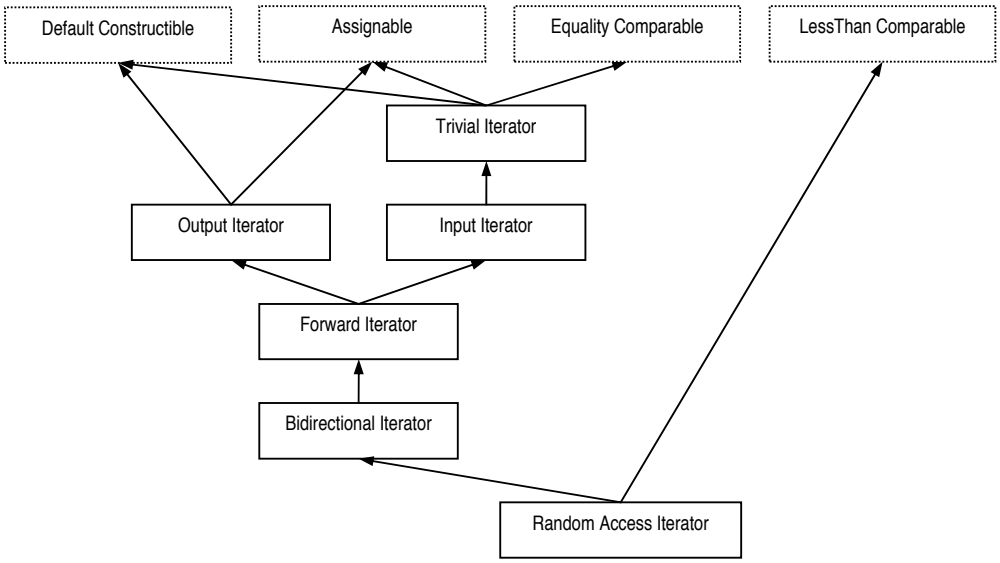
Iterátory jsou centrální součástí generického programování, protože představují rozhraní mezi kontejnery (např. `vector`) a generickými algoritmy. Tyto algoritmy mají zpravidla jako své parametry iterátory a ne samotné kontejnery, takže kontejnery musí poskytovat přístup ke svým prvkům pomocí iterátorů. Algoritmy zpravidla nejsou definovány jako metody nějakého kontejneru, ale jsou definovány jako šablony obyčejných funkcí tak, aby je bylo možné použít pro různé typy kontejnerů.

Iterátor je koncept. Knihovna STLport definuje 6 konceptů iterátorů:

- *triviální iterátor* (angl. *Trivial Iterator*),
- *vstupní iterátor* (angl. *Input Iterator*),
- *výstupní iterátor* (angl. *Output Iterator*),
- *dopředný iterátor* (angl. *Forward Iterator*),
- *dvousměrný iterátor* (angl. *Bidirectional Iterator*),
- *iterátor s náhodným přístupem* (angl. *Random Access Iterator*).

Norma jazyka C++ definuje stejné iterátory, neobsahuje pouze definici triviálního iterátoru.

Iterátory jsou na sobě hierarchicky závislé, tj. některý koncept iterátoru je zjemněním konceptu jiného iterátoru nebo všeobecného konceptu – hierarchie konceptů iterátorů je znázorněna na obr. 1.



Obr. 1 Hierarchie konceptů iterátorů

Typ objektu, na který iterátor ukazuje, se nazývá *typ hodnoty* (angl. *value type*) a získá se dereferencováním iterátoru. Jedná se o typ objektu, který se vkládá do kontejneru. Objekt, který je typu *typ hodnoty*, se také nazývá *hodnota prvku kontejneru*.

Iterátor `i` je *měnitelný* (angl. *mutable*), jestliže výsledkem operace `*i` je nekonstantní reference na *typ hodnoty*. To znamená, že lze modifikovat objekt, na který iterátor ukazuje. Opakem měnitelného iterátoru je iterátor *konstantní* (angl. *constant* nebo *immutable*).

Iterátor je *koncový* (angl. *past-the-end*), jestliže ukazuje za poslední prvek nějakého kontejneru.

Iterátor `i` je *dereferencovatelný* (angl. *dereferenceable*), jestliže je definován výraz `*i`. Koncový iterátor není *dereferencovatelný*.

Iterátor `i` je *inkrementovatelný* (angl. *incrementable*), jestliže je platný výraz `++i`. *Koncový* iterátor není inkrementovatelný.

Iterátor `j` je *dosažitelný* (angl. *reachable*) z iterátoru `i`, jestliže existuje konečný počet opakování výrazu `++i`, po jejichž provedení bude platit `i == j`. Jestliže iterátor `j` je dosažitelný z iterátoru `i`, potom oba iterátory ukazují na stejný kontejner.

Většina algoritmů pracuje s dvojicí iterátorů, které se zpravidla označují `first` a `last`. Ty představují *rozsah* (angl. *range*) všech iterátorů od `first` do `last` vyjma `last`. Rozsah se označuje intervalem `[first, last)`. Rozsah je prázdný, pokud `first == last`. Rozsah je platný, pokud iterátor `last` je dosažitelný z iterátoru `first`. Výsledky algoritmů, které jsou volány s neplatným rozsahem, jsou nedefinovatelné.

Počet prvků rozsahu iterátoru je typu, který se nazývá podle normy C++ *typ rozdílu* (angl. *difference type*) a podle STLport *typ vzdálenosti (angl. distance type*). Jedná se o celočíselný typ se znaménkem.

V následujícím textu této kapitoly jsou použity tyto symboly:

`X` ..... typ, který je modelem konceptu iterátoru,

`T` ..... *typ hodnoty* iterátoru,

`Distance` .... *typ vzdálenosti* mezi iterátory `X`,

`x`, `y`, `i`, `j` ..... objekty typu `X`,

`t` ..... objekt typu `T`,

`n` ..... objekt typu `Distance`.

Triviální iterátor

Triviální iterátor může být dereferencován, aby se zpřístupnila reference objektu, na který iterátor „ukazuje“. Může být měnitelný nebo konstantní. Není přímo využit žádným algoritmem. Je základním konceptem jiných iterátorů. Je zjemněním konceptů `DefaultConstructible`, `Assignable`, `EqualityComparable`.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`*x`	lze implicitně konvertovat na typ `T`	Dereference.
`*x = t`		Dereference s přiřazením. Výraz je požadován jen pro měnitelný iterátor. Po provedení výrazu musí `*x` být kopií `t`.
`x->m`		Přístup ke složce `m` typu `T`. Ekvivalent výrazu `(*x).m`.

Všechny operace mají konstantní časovou složitost.

Koncept musí splňovat následující vlastnost:

Vlastnost	Platí
identita	`x == y` jen v tom případě, pokud platí `&*x == &*y`

Modelem triviálního konceptu je např. ukazatel na objekt, který není součástí pole objektů.

Vstupní iterátor

Vstupní iterátor slouží k procházení prvků kontejneru za účelem čtení objektů obsažených v kontejneru (nemodifikuje objekty kontejneru). Může být:

- dereferencován, aby se zpřístupnila reference objektu, na který iterátor ukazuje,
- inkrementován, aby se přesunul na následující iterátor nějaké posloupnosti.

Je zjemněním triviálního iterátoru. Je konstantní.

Model konceptu kromě výrazů definovaných v konceptu konstantního triviálního iterátoru, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`*i`		Iterátor `i` musí být dereferencovatelný
`++i`	`X&`	Prefixová inkrementace. Před provedením výrazu musí být iterátor `i` dereferencovatelný. Po provedení výrazu musí být iterátor `i` dereferencovatelný nebo koncový.
`(void)i++`		Postfixová inkrementace. Před provedením výrazu musí být iterátor `i` dereferencovatelný. Po provedení výrazu musí být iterátor `i` dereferencovatelný nebo koncový. Ekvivalent výrazu `(void)++i`
`*i++`	`T`	Postfixová inkrementace a dereference. Po provedení výrazu musí být iterátor `i` dereferencovatelný nebo koncový. Ekvivalent příkazů `{ T t = *i; ++i; return t;}`

Všechny operace mají konstantní časovou složitost.

Protože je iterátor konstantní, jeho model nevyhovuje výrazu `*i = t`.

Po provedení výrazu `++i` nelze použít kopii staré hodnoty `i`. Pokud platí `i == j`, nemusí platit `++i == ++j`. Algoritmy nesmí dvakrát procházet přes stejný iterátor. Musí se jednat o tzv. *jednoprůchodové* (angl. *single-pass*) algoritmy. Iterátor totiž může při inkrementaci vyjmout prvek z kontejneru, např. ze vstupního datového proudu.

Modelem vstupního iterátoru je např. šablona třídy `istream_iterator`.

Vstupní iterátor používá např. algoritmus `find`, který může mít např. takovouto definici:

```
template<class InputIterator, class EqualityComparable>
InputIterator find(InputIterator first, InputIterator last,
                   const EqualityComparable& value)
{
   while (first != last && *first != value) ++first;
   return first;
}
```

Algoritmus `find` provádí sekvenční hledání hodnoty `value` v rozsahu `[first, last)`. Vrací iterátor, který ukazuje na hodnotu `value`. Jestliže hodnotu `value` nenajde, vrací `last`.

Algoritmus `find` jakož i další generické algoritmy jsou definovány v hlavičkovém souboru `<algorithm>`.

Vstupním iterátorům vyhovují např. obyčejné ukazatele (jsou modelem iterátoru s náhodným přístupem).

Příklad

V následujícím příkladu se hledá hodnota 7 v poli 10 celých čísel:

```
int pole[10];
// naplnění pole
int* i = find(pole, pole+10, 7);
if (i == pole+10) cout << "Hodnota 7 v poli neexistuje";
else cout << "Hodnota 7 se poli nachazi na indexu " << i-pole;
```

Výrazy pole a pole+10 představují měnitelné iterátory. Konstantní iterátory jsou typu ukazatel na konstantní `int`, např.:

```
const int *first = pole;
const int *last = pole+10;
const int *i = find(first, last, 7);
```

Iterátory různých kontejnerů knihovny C++ také vyhovují vstupním iterátorům (vyhovují iterátorům, které jsou zjemněním vstupního iterátoru). Kontejnery obsahují dvě metody `begin()` a `end()`. Metoda `begin()` vrací iterátor, který ukazuje na první prvek kontejneru, metoda `end()` vrací koncový iterátor.

Příklad

V následujícím příkladu se hledá hodnota 7 v poli celých čísel `pole`:

```
vector<int> pole;
// naplnění vektoru
vector<int>::iterator it = find(pole.begin(), pole.end(), 7);
if (it == pole.end())cout << "Hodnota 7 v poli neexistuje";
else cout << "Hodnota 7 se v poli nachazi na indexu " << it-pole.begin();
```

Každý kontejner, který poskytuje iterátory, obsahuje vnořený typ `iterator` pro měnitelný iterátor a typ `const_iterator` pro konstantní iterátor.

Výstupní iterátor

Výstupní operátor obsahuje mechanismus pro uložení posloupnosti hodnot do kontejneru, ale přístup k hodnotám této posloupnosti není požadován.

Je zjemněním konceptů `DefaultConstructible` a `Assignable`.

Model konceptu kromě výrazů definovaných v konceptu `DefaultConstructible` a `Assignable`, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`X(x)`	`X`	Kopírovací konstruktor. Výraz `*X(x) = t` je ekvivalentem výrazu `*x = t`.
`X y(x);` `X y = x;`		Kopírovací konstruktor. Po provedení výrazu musí platit: `*y = t` je ekvivalentem výrazu `*x = t`.
`*x = t`	není používaný	Dereference s přiřazením.

Výraz	Návratový typ	Význam, požadavky
`++x`	`X&`	Prefixová inkrementace. Před provedením výrazu musí být iterátor `x` dereferencovatelný. Po provedení výrazu `x` ukazuje na následující místo, do kterého může být uložena hodnota.
`(void)x++`	`void`	Postfixová inkrementace. Před provedením výrazu musí být iterátor `x` dereferencovatelný. Po provedení výrazu `x` ukazuje na následující místo, do kterého může být uložena hodnota. Ekvivalent výrazu `(void)++x`.
`*x++ = t;`	není používaný	Postfixová inkrementace s přiřazením. Po provedení výrazu `x` ukazuje na následující místo, do kterého může být uložena hodnota. Ekvivalent příkazů `{*x = t; ++x; }`

Všechny operace mají konstantní časovou složitost.

Norma C++ místo výrazu `(void)x++` předepisuje výraz `x++`, který má mít takový návratový typ, který lze konvertovat na typ `const X&`.

Dereference výstupního iterátoru smí být použita jen na levé straně přiřazovacího výrazu, tj. `*x = t`. Dereference nesmí být použita např. ke čtení hodnoty objektu, na který iterátor ukazuje. Algoritmy nesmí dvakrát procházet přes stejný iterátor. Musí se jednat o *jednoprůchodové* algoritmy.

Měla by být aktivní pouze jedna kopie výstupního iterátoru v určitém okamžiku, tj. po zkopírování iterátoru `x` do `y` (např. `X y = x;`) a použití iterátoru `y` se již nesmí použít iterátor `x`.

Před každou inkrementací iterátoru musí být dereference s přiřazením, tj. složený příkaz
`{*x = t; ++x; *x = t2; ++x}` je správný, ale příkaz
`{*x = t; ++x; ++x; *x = t2;}` správný není.

Pro zajištění výrazu `*x = t` se mohou v modelu tohoto konceptu definovat např. dva přetížené operátory:

- operátor `*` vrací `X&`
- operátor = vrací `X&` a má parametr typu `const T&`.

Modelem výstupního iterátoru jsou např. šablony tříd `ostream_iterator`, `insert_iterator`, `front_insert_iterator`, `back_insert_iterator`.

Výstupní iterátor používá např. algoritmus `copy`, který může mít např. takovouto definici:

```
template <class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result)
{
   while (first != last) *result++ = *first++;
   return result;
}
```

Algoritmus `copy` kopíruje prvky z rozsahu `[first, last)` do rozsahu začínajícího iterátorem `result`. Algoritmus předpokládá (ale nekontroluje), že se všechny prvky vstupního rozsahu vejdou do výstupního rozsahu. Vrací iterátor, který ukazuje za poslední prvek zkopírovaný do výstupního rozsahu.

Výstupním iterátorům vyhovují jak obyčejné ukazatele, tak i iterátory kontejnerů knihovny C++.

Příklad

V následujícím příkladu se kopíruje obyčejné pole do kontejneru typu `vector`:

```
int pole[10];
vector<int> vektor(10); // alokuje se vektor 10 celých čísel
// naplnění pole
copy(pole, pole + 10, vektor.begin());
```

Dopředný iterátor

Dopředný iterátor umožňuje procházet posloupnost prvků od začátku ke konci, ale ne opačně. Algoritmy, které ho používají, mohou být tzv. *víceprůchodové* (angl. *multipass*) – mohou procházet přes stejný iterátor vícekrát. Dopředný iterátor může být měnitelný nebo konstantní.

Je zjemněním konceptu vstupního a výstupního iterátoru.

Model konceptu ruší omezení dvou výrazů inkrementace vstupního iterátoru – viz následující tabulka.

Výraz	Návratový typ	Význam, požadavky
`++i`	`X&`	Prefixová inkrementace. Před provedením výrazu musí být iterátor `i` dereferencovatelný. Po provedení výrazu musí být iterátor `i` dereferencovatelný nebo koncový. Platí `&i == &++i`.
`i++`	`X`	Postfixová inkrementace. Před provedením výrazu musí být iterátor `i` dereferencovatelný. Po provedení výrazu musí být iterátor `i` dereferencovatelný nebo koncový. Ekvivalent příkazů `{ X tmp = i; ++i; return tmp; }`

Všechny operace mají konstantní časovou složitost.

Podle normy C++ má mít výraz `i++` takový návratový typ, který lze konvertovat na typ `const X&`.

Dopředný iterátor má následující odlišnosti od vstupního operátoru:

- Po provedení inkrementace `i` lze použít kopii staré hodnoty `i` (výraz `i++` má návratový typ `X`).
- Pokud `i`, `j` jsou dereferencovatelné a `i == j`, platí `++i == ++j`.
- Platí `&i == &++i`, tj. po provedení inkrementace se nemění adresa objektu iterátoru.

Modelem dopředného iterátoru je např. iterátor kontejneru `slist` (jednosměrný zřetězený seznam).

Dopředné iterátory používá např. algoritmus `replace`, který může mít např. takovouto definici:

```
template <class ForwardIterator, class T>
void replace (ForwardIterator first, ForwardIterator last,
              const T& old_value, const T& new_value)
{
   while (first != last) {
      if (*first == old_value) *first = new_value;
      ++first;
   }
}
```

Iterátory `first` a `last` nemohou být výstupní ze dvou důvodů:

- dereference iterátoru `*first` je použita pro porovnání,
- může se vícekrát za sebou vyskytnout inkrementace bez uložení hodnoty do kontejneru.

Algoritmus `replace` nahrazuje všechny výskyty hodnoty `old_value` v rozsahu `[fist, last)` hodnotou `new_value`.

Dopředným iterátorům vyhovují jak obyčejné ukazatele, tak i iterátory kontejnerů knihovny C++.

Příklad

V následujícím příkladu se nahradí všechny výskyty hodnoty 7 ve vektoru celých čísel hodnotou 11:

```
vector<int> vektor(10);
// naplnění vektoru
replace(vektor.begin(),vektor.end(), 7, 11);
```

ITERÁTORY – POKRAČOVÁNÍ

Dvousměrný iterátor

Dvousměrný iterátor umožňuje procházet posloupnost prvků od začátku ke konci nebo naopak, tj. může být inkrementován i dekrementován. Může být měnitelný nebo konstantní.

Je zjemněním dopředného iterátoru.

Model konceptu kromě výrazů definovaných v konceptu dopředného iterátoru musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`--i`	`X&`	Prefixová dekrementace. Před provedením výrazu musí být iterátor `i` dereferencovatelný nebo koncový. Po provedení výrazu musí být iterátor `i` dereferencovatelný. Platí `&i == &--i`. Pokud platí `i == j`, musí platit `--i == --j`. Pokud `j` je dereferencovatelný iterátor a platí `i == ++j`, potom musí platit `--i == j`.
`i--`	`X`	Postfixová dekrementace. Před provedením výrazu musí být iterátor `i` dereferencovatelný nebo koncový. Ekvivalent příkazů `{ X tmp = i; --i; return tmp; }`

Všechny operace mají konstantní časovou složitost.

Podle normy C++ má mít výraz `i--` takový návratový typ, který lze konvertovat na typ `const X&`.

Koncept má následující vlastnost.

Vlastnost	Platí

symetrie inkrementace a dekrementace	jestliže iterátor `i` je dereferencovatelný, potom platí:
	`--(++i) == i`
	`++(--i) == i`

Modelem dvousměrného iterátoru je např. iterátor kontejneru `list` (obousměrný zřetězený seznam).

Dvousměrné iterátory používá např. algoritmus `reverse_copy`, který může mít např. takovouto definici:

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
                            BidirectionalIterator last,
                            OutputIterator result)
{
   while (first != last) *result++ = *--last;
   return result;
}
```

Algoritmus `reverse_copy` kopíruje prvky z rozsahu `[first, last)` odzadu (od posledního po první prvek rozsahu) do rozsahu začínajícího iterátorem `result`. Algoritmus předpokládá (ale nekontroluje), že se všechny prvky vstupního rozsahu vejdou do výstupního rozsahu. Vrací iterátor, který ukazuje za poslední prvek zkopírovaný do výstupního rozsahu.

Dvousměrným iterátorům vyhovují jak obyčejné ukazatele, tak i některé iterátory kontejnerů knihovny C++.

Příklad

V následujícím příkladu se reverzně kopírují prvky obousměrného zřetězeného seznamu do vektoru:

```
list<int> seznam;
for (i = 0; i < 3; i++) seznam.push_back(i);
vector<int> vektor(seznam.size());
reverse_copy(seznam.begin(), seznam.end(), vektor.begin());
```

Metody `size()` a `push_back()` jsou uvedeny u různých kontejnerů. Metoda `size()` vrací skutečný počet prvků kontejneru, metoda `push_back()` přidá objekt na konec kontejneru. Seznam bude obsahovat hodnoty (0, 1, 2) a vektor (2, 1, 0).

Iterátor s náhodným přístupem

Iterátor s náhodným přístupem umožňuje kromě inkrementace a dekrementace přesunutí na libovolný prvek posloupnosti s konstantní časovou složitostí. Nabízí stejné operace jako ukazatele jazyka C. Může být měnitelný nebo konstantní.

Je zjemněním dvousměrného iterátoru a konceptu `LessThanComparable`.

Model konceptu kromě výrazů definovaných v konceptu dvousměrného iterátoru musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`i += n`	`X&`	Posunutí iterátoru o `n` prvků dopředu. Pokud `n > 0`, ekvivalent k `n`-krát provedení výrazu `++i`. Pokud `n < 0`, ekvivalent k `n`-krát provedení výrazu `--i`.

`i + n`
nebo
`n + i`
	`X`	Posunutí iterátoru o `n` prvků dopředu.
		Ekvivalent příkazů
		`{ X tmp = i; return tmp += n; }`.

`i -= n`	`X&`	Posunutí o `n` prvků dozadu. Ekvivalent výrazu `i += (-n)`.
`i - n`	`X`	Posunutí o `n` prvků dozadu. Ekvivalent příkazů `{ X tmp = i; return tmp -= n; }`.
`i - j`	`Distance`	Rozdíl. Buď iterátor `i` musí být dosažitelný z iterátoru `j` nebo iterátor `j` musí být dosažitelný z `i`. Vrací takové celé číslo `n`, pro které platí `i == j + n`.
`i[n]`	lze implicitně konvertovat na typ `T`	Hodnota prvku kontejneru mající index `n`. Ekvivalent výrazu `*(i + n)`.

Výraz	Návratový typ	Význam, požadavky
`i[n] = t`	lze implicitně konvertovat na typ `T`	Přiřazení hodnoty do prvku kontejneru majícího index `n`. Je požadováno jen pro měnitelný iterátor. Ekvivalent výrazu `*(i + n) = t`. Po provedení výrazu je `i[n]` kopií `t`.
`i < j`	lze implicitně konvertovat na typ `bool`	Buď iterátor `i` musí být dosažitelný z iterátoru `j` nebo iterátor `j` musí být dosažitelný z `i` nebo platí obojí. Ekvivalent výrazu `j - i > 0`.

`i > j`
`i <= j`
`i >= j`
	lze implicitně konvertovat na typ `bool`	Platí obdobná pravidla dosažitelnosti iterátoru jako pro výraz `i < j`.
		Lze definovat jako ekvivalenty operátoru < (viz koncept `LessThanComparable`).

Všechny operace mají konstantní časovou složitost.

Koncept má vlastnosti uvedené v následující tabulce.

Vlastnost	Platí
symetrie součtu a rozdílu	jestliže je platný výraz `i + n`, potom provedením příkazu `{ i += n; i -= n }` nebo výrazu `(i + n) - n` se `i` nezmění.
relace mezi vzdáleností a součtem	jestliže je platný výraz `i - j`, potom `i == j + (i - j)`
dosažitelnost a vzdálenost	jestliže iterátor `i` je dosažitelný z iterátoru `j`, potom `i - j >= 0`
ostré slabé uspořádání	relace < má vlastnosti ostrého slabého uspořádání.

Modelem tohoto iterátoru je např. obyčejný ukazatel `T*` a iterátor kontejneru `vector`.

Iterátory s náhodným přístupem používají třídící algoritmy nebo algoritmus `random_shuffle`, který náhodně promíchá prvky rozsahu `[first, last)`. Definice tohoto algoritmu je složitá, proto je zde uvedena jeho jednodušší definice v podobě algoritmu `mixup`:

```
template <class RandomAccessIterator>
void mixup(RandomAccessIterator first, RandomAccessIterator last)
{
   while (first < last) {
      iter_swap(first, first + rand() % (last - first));
      ++first;
   }
}
```

Algoritmus `iter_swap` vymění hodnoty dvou prvků, na které ukazují iterátory, přičemž se nemění pozice iterátoru v kontejneru, ale hodnota prvku, na který iterátor ukazuje. Uvedený algoritmus provádí opakovaně výměnu dvou prvků prostřednictvím iterátoru `first` a náhodně vybraného iterátoru z rozsahu `[first, last)`. Cyklus skončí, pokud se iterátor `first` dostane na pozici iterátoru `last`. Výraz `(rand() % (last - first))` vrací náhodné celé číslo v rozsahu `0` až `(last - first)-1`.

Šablona iterator_traits

Některé algoritmy, používající iterátory, potřebují určit *typ hodnoty*, *typ vzdálenosti* (*typ rozdílu*) nebo kategorii iterátoru.

Pro iterátory definované jako třídy se uvedené požadavky řeší deklarováním vnořených typů, např. `value_type`. Protože však modelem konceptů iterátorů jsou i obyčejné ukazatele, které nemají vnořené typy, zavádí se šablona `iterator_traits`, která je v hlavičkovém souboru `<iterator>` definována následovně:

```
template<class Iterator> struct iterator_traits {
   typedef typename Iterator::difference_type difference_type;
   typedef typename Iterator::value_type value_type;
   typedef typename Iterator::pointer pointer;
   typedef typename Iterator::reference reference;
   typedef typename Iterator::iterator_category iterator_category;
};
```

Šablona má jeden typový parametr `Iterator`, který je modelem určitého konceptu iterátoru. Je-li to třída, musí mít deklarovány uvedené vnořené typy. Aby mohla být tato šablona použita i pro obyčejné ukazatele, jsou dále definovány dvě její parciální specializace – pro obyčejné ukazatele na nekonstantní objekty:

```
template<class T> struct iterator_traits<T*> {
   typedef ptrdiff_t difference_type;
   typedef T value_type;
   typedef T* pointer;
   typedef T& reference;
   typedef random_access_iterator_tag iterator_category;
};
```

a pro obyčejné ukazatele na konstantní objekty:

```
template<class T> struct iterator_traits<const T*> {
   typedef ptrdiff_t difference_type;
   typedef T value_type;
   typedef const T* pointer;
   typedef const T& reference;
   typedef random_access_iterator_tag iterator_category;
};
```

Koncept výstupního iterátoru nevyžaduje typ `difference_type` a `value_type` – oba mají být deklarovány jako `void` a tím i typ `pointer` a `reference`.

Typ hodnoty iterátoru je např. potřebný v algoritmu `iter_swap`, který vymění hodnoty dvou objektů, na které ukazují iterátory `a` a `b`.

```
template <class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b)
{
   iterator_traits<ForwardIterator1>::value_type tmp = *a;
   *a = *b;
   *b = tmp;
}
```

Pro iterátor `I`, definovaný jako třída představuje zápis `iterator_traits<I>::value_type` synonymum pro `I::value_type`. Zatímco pro iterátor obyčejného ukazatele `T*` představuje zápis `iterator_traits<T*>::value_type` synonymum pro `T`.

Kategorie iterátorů

Daný algoritmus může být lépe naprogramován pro určitý druh iterátoru než pro ostatní. Potom existují dvě nebo více implementací algoritmu pro jednotlivé kategorie iterátorů. Překladač zvolí tu verzi algoritmu, která odpovídá typu iterátoru.

Pro tyto účely jsou v hlavičkovém souboru `<iterator>` definovány následující struktury:

```
struct input_iterator_tag {};
struct output_iterator_tag {};
struct forward_iterator_tag: public input_iterator_tag {};
struct bidirectional_iterator_tag: public forward_iterator_tag {};
struct random_access_iterator_tag: public bidirectional_iterator_tag {};
```

Struktury neobsahují žádné složky a jsou definovány mj. proto, aby byla zřejmá jejich dědičnost. Ta je shodná se zjemněním konceptů iterátorů.

Pro každý iterátor typu `Iterator` musí být definován typ `iterator_category` přístupný pomocí šablony `iterator_traits`:

```
iterator_traits<Iterator>::iterator_category
```

Kategorie iterátoru se může např. využít v algoritmu `reverse`. Ten má dva parametry – dvousměrné iterátory `first` a `last`. Algoritmus otočí pořadí prvků v rozsahu `[first, last)` tak, že první prvek bude poslední, poslední prvek bude první, druhý prvek bude předposlední atd. Pro iterátory s náhodným přístupem lze provést požadovaný úkol efektivněji, proto může mít dvě verze, jednu pro dvousměrné iterátory a druhou pro iterátory s náhodným přístupem. Vytvoří se tudíž dvě pomocné šablony funkcí `__reverse` s třetím parametrem udávajícím kategorii iterátoru:

```
template <class BidirectionalIterator>
void __reverse(BidirectionalIterator first, BidirectionalIterator last,
               bidirectional_iterator_tag)
{
   while (true) {
      if (first == last || first == --last) return;
      else iter_swap(first++, last);
   }
}

template <class RandomAccessIterator>
void __reverse(RandomAccessIterator first, RandomAccessIterator last,
               random_access_iterator_tag)
{
   while (first < last) iter_swap(first++, --last);
}
```

Vlastní algoritmus `reverse` potom bude mít následující definici:

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last)
{
   __reverse(first, last,
       iterator_traits<BidirectionalIterator>::iterator_category());
}
```

Při volání funkce `__reverse` je třetím parametrem instance typu, představující kategorii iterátoru – vytváří se dočasná nepojmenovaná instance pomocí implicitního konstruktoru.

Základní třídy iterátorů

Pokud programátor vytváří svůj vlastní iterátor, může jej odvodit ze šablony struktury `iterator`, definované v hlavičkovém souboru `<iterator>` takto:

```
template<class Category, class T, class Distance = ptrdiff_t,
         class Pointer = T*, class Reference = T&>
struct iterator {
   typedef T value_type;
   typedef Distance difference_type;
   typedef Pointer pointer;
   typedef Reference reference;
   typedef Category iterator_category;
};
```

Pokud programátor odvodí svůj iterátor od této šablony, bude mít jeho iterátor deklarovány vnořené typy, které jsou pro iterátory požadovány. Šablona obsahuje 5 typových parametrů. První z nich je kategorie iterátoru. To by měla být jedna z předdefinovaných struktur `input_iterator_tag` , `output_iterator_tag` atd. Význam dalších typových parametrů je zřejmý z uvedené definice.

Vlastní iterátor, který by byl modelem konceptu dvousměrného iterátoru, by mohl být definován např. takto:

```
class TMyIterator : public iterator<bidirectional_iterator_tag, double>
{
   // implementace
};
```

V knihovně STLport jsou dále definovány odvozené třídy ze šablony `iterator` pro jednotlivé kategorie iterátorů: `input_iterator`, `output_iterator`, `forward_iterator`, `bidirectional_iterator` a `random_access_iterator`. Norma C++ je ale nespecifikuje. Jejich definice je následující:

```
template <class T, class Distance> struct input_iterator :
   public iterator<input_iterator_tag, T, Distance, T*, T&> {};
struct output_iterator :
   public iterator<output_iterator_tag,void,void,void,void> {};
template <class T, class Distance> struct forward_iterator :
   public iterator<forward_iterator_tag, T, Distance, T*, T&> {};
template <class T, class Distance> struct bidirectional_iterator :
   public iterator<bidirectional_iterator_tag, T, Distance, T*, T&> {};
template <class T, class Distance> struct random_access_iterator :
   public iterator<random_access_iterator_tag, T, Distance, T*, T&> {};
```

Všechny z nich kromě výstupního iterátoru jsou definovány jako šablony se dvěma typovými parametry: `T` pro typ hodnoty, `Distance` pro typ vzdálenosti (rozdílu). Výstupní iterátor je definován jako obyčejná struktura, protože ani typ hodnoty ani typ vzdálenosti není pro výstupní iterátor požadovaný.

Funkce pro základní operace s iterátory

Kromě různých generických algoritmů, které mají jako své parametry iterátory, existují pro základní operace s iterátory dvě šablony funkcí `advance` a `distance`. Jsou deklarovány v hlavičkovém souboru `<iterator>` následovně:

```
template <class InputIterator, class Distance>
void advance(InputIterator& i, Distance n);
```

```
template<class InputIterator>
typename iterator_traits<InputIterator>::difference_type
distance(InputIterator first, InputIterator last);
```

Funkce `advance` `n`-krát inkrementuje (pro kladné `n`) nebo dekrementuje (pro záporné `n`) iterátor `i`.

Funkce `distance` vrací počet inkrementací iterátoru `first` potřebných k tomu, aby iterátor `first` dosáhl pozice iterátoru `last`, neboli vrací vzdálenost (rozdíl) rozsahu `[first, last)`.

Pro iterátory s náhodným přístupem obě funkce využívají operátory +, – a požadovaná operace má konstantní časovou složitost. Pro ostatní iterátory obě funkce využívají operátor ++ a doba provedení operace závisí na počtu inkrementací – operace má lineární časovou složitost.

Příklad

```
int main()
{
   list<int>   seznam;
   int         i;
   list<int>::iterator it;

   for (i = 0; i < 10; i++) seznam.push_back(i*2);
   for (it = seznam.begin(); it != seznam.end(); ++it)
      cout << *it << ' ';
   cout << endl;
   it = seznam.begin();
   cout << "Je-li it rovno seznam.begin(), ukazuje it na cislo "
        << *it << endl;
   advance(it, 4);
   cout << "Po provedeni advance(it, 4) ukazuje it na cislo " << *it
        << endl;
   i = distance(seznam.begin(), seznam.end());
   cout << "Vzdalenost mezi seznam.begin() a seznam.end() je " << i
        << endl;
   cin.get();
   return 0;
}
```

Výstup programu bude následující:

```
0 2 4 6 8 10 12 14 16 18
Je-li it rovno seznam.begin(), ukazuje it na cislo 0
Po provedeni advance(it, 4) ukazuje it na cislo 8
Vzdalenost mezi seznam.begin() a seznam.end() je 10
```

Třídy iterátorů

Inverzní iterátor

Inverzní iterátor (angl. *reverse iterator*) je tzv. *adaptér* iterátoru, který prochází prvky kontejneru v opačném pořadí než normální iterátor.

Adaptér (angl. *adaptor*) je třída nebo funkce, která konvertuje jedno rozhraní na jiné rozhraní. Adaptér iterátoru konvertuje určité rozhraní na rozhraní používané iterátory.

Inverzní iterátor lze použít pro dvousměrný iterátor nebo iterátor s náhodným přístupem. Operátor inkrementace aplikovaný na objekt inverzního iterátoru provede totéž co operátor dekrementace aplikovaný na objekt korespondujícího normálního iterátoru.

Základní vztah mezi inverzním iterátorem a jeho korespondujícím normálním iterátorem `i` je dán následující rovností:

```
&*(reverse_iterator(i)) == &*(i - 1)
```

Tento vztah je dán faktem, že koncový normální iterátor ukazuje za poslední platný prvek normální posloupnosti, přičemž k němu odpovídající inverzní iterátor musí ukazovat na první platný prvek opačné posloupnosti, tedy ukazuje na poslední platný prvek normální posloupnosti. Naopak k normálnímu iterátoru, který ukazuje na první platný prvek normální posloupnosti, odpovídá inverzní iterátor, který ukazuje za poslední platný prvek opačné posloupnosti, tedy ukazuje před první platný prvek normální posloupnosti.

Inverzní iterátor je definován jako šablona třídy `reverse_iterator` v hlavičkovém souboru `<iterator>`:

```
template <class Iterator>
class reverse_iterator : public
   iterator<typename iterator_traits<Iterator>::iterator_category,
            typename iterator_traits<Iterator>::value_type,
            typename iterator_traits<Iterator>::difference_type,
            typename iterator_traits<Iterator>::pointer,
            typename iterator_traits<Iterator>::reference> {
protected:
   Iterator current;
public:
   explicit reverse_iterator(Iterator x);
   Iterator base() const;
   // metody a vnořené typy
};
```

Šablona má jeden typový parametr `Iterator`, představující normální iterátor, který šablona adaptuje. Je odvozena ze šablony `iterator`. Kromě metod (zejména operátorových funkcí) a vnořených typů, které jsou součástí normálního iterátoru, obsahuje explicitní konstruktor s jedním parametrem typu `Iterator`. Tento konstruktor slouží k inicializaci chráněného atributu `current` typu `Iterator`. Atribut představuje normální iterátor, který šablona adaptuje na inverzní. Dále obsahuje konstantní metodu `base()`, která vrací hodnotu tohoto atributu.

Knihovna STLport definuje dvě verze inverzního iterátoru:

- šablonu `reverse_iterator` pro iterátor s náhodným přístupem
- šablonu `reverse_bidirectional_iterator` pro dvousměrný iterátor.

Norma C++ specifikuje pouze šablonu `reverse_iterator` určenou jak pro iterátor s náhodným přístupem, tak pro dvousměrný iterátor.

Kontejnery, které používají dvousměrné iterátory nebo iterátory s náhodným přístupem mají metody `rbegin()` a `rend()`. Metody nemají žádné parametry a vrací inverzní iterátory. Konkrétně metoda `rbegin()` vrací instanci `reverse_iterator(end())` a metoda `rend()` vrací instanci `reverse_iterator(begin())`. Symbol `reverse_iterator` je vnořený typ kontejneru, který představuje šablonu `reverse_iterator` (resp. `reverse_bidirectional_iterator`).

Příklad

```
int main()
{
   vector<int> vektor(10);
   int         i;

   srand(static_cast<unsigned>(time(0)));
   for (i = 0; i < 10; i++) vektor[i] = rand()%10;
   cout << "Puvodni vektor: ";
   for (vector<int>::iterator it = vektor.begin();
        it != vektor.end(); ++it) cout << *it << ' ';
   cout << endl;
   cout << "Inverzni vektor: ";
   vector<int>::reverse_iterator first(vektor.rbegin());
   // dtto: vector<int>::reverse_iterator first(vektor.end());
   // dtto: reverse_iterator<vector<int>::iterator> first(vektor.end());
   vector<int>::reverse_iterator last(vektor.rend());
   // dtto: vector<int>::reverse_iterator last(vektor.begin());
   // dtto: reverse_iterator<vector<int>::iterator> last(vektor.begin());
   for (; first != last; ++first) cout << *first << ' ';
   cout << endl;

   vector<int>::reverse_iterator rit = find(vektor.rbegin(),
                                            vektor.rend(), 4);
   if (rit == vektor.rend()) cout << "Cislo 4 se ve vektoru nenachazi";
   else cout << "Posledni vyskyt cisla 4 je na pozici "
             << vektor.size()-(rit-vektor.rbegin());
   cin.get();
   return 0;
}
```

Uvedený program vypíše vektor náhodných celých čísel v původním a inverzním pořadí pomocí inverzních iterátorů a potom pomocí inverzních iterátorů hledá poslední výskyt čísla 4 ve vektoru. Výstup programu bude např. následující:

```
Puvodni vektor: 9 1 2 2 4 9 4 7 5 6
Inverzni vektor: 6 5 7 4 9 4 2 2 1 9
Posledni vyskyt cisla 4 je na pozici 7
```

Vstupní iterátor datového proudu

Vstupní iterátor datového proudu je šablona třídy `istream_iterator`, která čte objekty typu `T` pomocí operátoru `>>` ze vstupního datového proudu `basic_istream` resp. z jeho potomka. Šablona je modelem konceptu vstupního iterátoru.

Při každé inkrementaci přečte jeden objekt typu `T` z datového proudu a uloží jej. Přečtený objekt je přístupný pomocí dereference iterátoru. Jestliže se dosáhne konce datového proudu, iterátor obsahuje speciální hodnotu, která indikuje koncový iterátor. Šablona je definována v hlavičkovém souboru `<iterator>` následovně:

```
template <class T, class charT = char, class traits = char_traits<charT>,
          class Distance = ptrdiff_t>
class istream_iterator:
   public iterator<input_iterator_tag, T, Distance, const T*, const T&> {
public:
   typedef charT char_type;
   typedef traits traits_type;
   typedef basic_istream<charT, traits> istream_type;
   istream_iterator();
   istream_iterator(istream_type& s);
   istream_iterator(const istream_iterator<T,charT,traits,Distance>& x);
   ~istream_iterator();
   const T& operator*() const;
   const T* operator->() const;
   istream_iterator<T,charT,traits,Distance>& operator++();
   istream_iterator<T,charT,traits,Distance> operator++(int);
   // neveřejné atributy
};
```

Šablona má čtyři typové parametry:

- `T` – typ objektu, který je čten ze vstupního proudu
- `charT` – znakový typ vstupního proudu, implicitně `char`
- `traits` – třída, která popisuje některé vlastnosti znaků, implicitně `char_traits<charT>`
- `Distance` – typ vzdálenosti (rozdílu), implicitně `ptrdiff_t`.

Je odvozena ze šablony `iterator`. Kromě metod (zejména operátorových funkcí) potřebných pro vstupní iterátor obsahuje deklaraci tří vnořených typů `char_type`, `traits_type` a `istream_type` (viz definice šablony) a tři konstruktory: implicitní, kopírovací a konstruktor s parametrem `istream_type`.

Implicitní konstruktor vytvoří koncový iterátor. Konstruktor s parametrem `istream_type`, vytvoří iterátor, který je sdružen se zadaným vstupním datovým proudem, z něhož čte data.

Příklad

V následujícím příkladu se čtou celá čísla z textového souboru a ukládají se do vektoru, který se potom vypíše na obrazovku. Textový soubor obsahuje nejprve počet čísel a potom jednotlivá čísla oddělená bílým znakem.

```
int main()
{
   int         i;
   ifstream    is("data.txt");

   is >> i; // pocet cisel
   vector<int> vektor(i);
   copy(istream_iterator<int>(is), istream_iterator<int>(),
        vektor.begin());
   cout << "Nacteny vektor:\n";
   for (i = 0; i != vektor.size(); i++) cout << vektor[i] << ' ';
   cout << endl;
   cin.get();
   return 0;
}
```

Pro čtení pole čísel ze souboru se použil algoritmus `copy`. Ten kopíruje prvky z rozsahu `[first, last)` do rozsahu začínajícího iterátorem `result`. Rozsah `[first, last)` je dán instancemi

šablony `istream_iterator` s typovým parametrem `int`, tj. budou se číst hodnoty typu `int`. Pro parametr `first` byla vytvořena instance s parametrem vstupního datového proudu `is`. Pro parametr `last` byla použita instance vytvořená pomocí implicitního konstruktoru, který poskytuje koncový iterátor. Parametr `result` obsahuje iterátor, kterým začíná vektor. Algoritmus `copy` tedy opakovaně čte čísla typu `int`, dokud se nevyskytne konec souboru a ukládá je do vektoru. Vektor musí mít alespoň takovou velikost, kolik je čísel v souboru.

Výstupní iterátor datového proudu

Výstupní iterátor datového proudu je šablona třídy `ostream_iterator`, která zapisuje objekty typu `T` pomocí operátoru `<<` do výstupního datového proudu `basic_ostream` resp. do jeho potomka. Šablona je modelem konceptu výstupního iterátoru. Za každý zapsaný objekt typu `T` může být zapsán ještě řetězec znaků, který je parametrem konstruktoru.

Šablona je definována v hlavičkovém souboru `<iterator>` následovně:

```
template <class T, class charT = char,
          class traits = char_traits<charT> >
class ostream_iterator:
   public iterator<output_iterator_tag, void, void, void, void>
{
public:
   typedef charT char_type;
   typedef traits traits_type;
   typedef basic_ostream<charT,traits> ostream_type;
   ostream_iterator(ostream_type& s);
   ostream_iterator(ostream_type& s, const charT* delimiter);
   ostream_iterator(const ostream_iterator<T,charT,traits>& x);
   ~ostream_iterator();
   ostream_iterator<T,charT,traits>& operator=(const T& value);
   ostream_iterator<T,charT,traits>& operator*();
   ostream_iterator<T,charT,traits>& operator++();
   ostream_iterator<T,charT,traits>& operator++(int);
private:
   basic_ostream<charT,traits>* out_stream;
   const charT*                 delim;
};
```

Šablona má tři typové parametry, které mají stejný význam jako u šablony `istream_iterator`. Je odvozena ze šablony `iterator`. Kromě operátorových funkcí potřebných pro výstupní iterátor a neveřejných atributů obsahuje deklaraci tří vnořených typů `char_type`, `traits_type` a `ostream_type` (viz definice šablony) a tři konstruktory.

Konstruktor

```
ostream_iterator(ostream_type& s);
```

vytvoří iterátor, který je sdružen s výstupním datovým proudem `s`, do něhož bude zapisovat data. Za každý zapsaný objekt nezapisuje jiný text.

Konstruktor

```
ostream_iterator(ostream_type& s, const charT* delimiter);
```

oproti předchozímu konstruktoru zapisuje za každý objekt typu `T` řetězec znaků `delimiter`.

Třetí konstruktor je kopírovací konstruktor.

Operátorová funkce dereference a dvě operátorové funkce inkrementace pouze vrací `*this`. Hlavní činnost provádí operátor přiřazení, jehož definice může být následující:

```
ostream_iterator<T,charT,traits>& operator=(const T& value)
{
   *out_stream << value;
   if (delim != 0) *out_stream << delim;
   return *this;
}
```

Z uvedené definice je zřejmé, že do výstupního datového proudu, na který ukazuje atribut `out_stream`, zapíše hodnotu `value` a případně řetězec znaků, na který ukazuje atribut `delim`. Názvy atributů norma C++ nepředepisuje.

Následující příkaz vypíše na obrazovku vektor celých čísel, přičemž za každé číslo včetně posledního vloží mezeru.

```
copy(vektor.begin(), vektor.end(), ostream_iterator<int>(cout, " "));
```

Iterátory vyrovnávací paměti datového proudu

Kromě vstupního a výstupního iterátoru sdruženého s datovým proudem jsou v hlavičkovém souboru `<iterator>` definovány i vstupní a výstupní iterátor sdružený přímo s vyrovnávací pamětí datového proudu. Jedná se o šablony `istreambuf_iterator` a `ostreambuf_iterator`. Jedním z jejich konstruktorů je konstruktor s parametrem typu `basic_streambuf<charT,traits>*`, který inicializuje ukazatel na sdruženou vyrovnávací paměť. Použití těchto iterátorů je obdobné jako vstupního a výstupního iterátoru datového proudu.

Vkládací iterátory

Mnoho algoritmů ukládá výsledky do výstupního iterátoru. Přiřazení hodnoty do dereferencovaného výstupního iterátoru, např. `*result = value` přepíše hodnotu prvku, na který iterátor `result` ukazuje. V následujícím příkladu se pomocí algoritmu `copy` zkopírují hodnoty z vektoru `a` do vektoru `b`:

```
vector<int> a(10);
vector<int> b(10);
// ...
copy(a.begin(), a.end(), b.begin());
```

Vektor `b` musí ale být před kopírováním dimenzovaný alespoň na tolik prvků, kolik jich obsahuje vektor `a`, přičemž prvních 10 původních hodnot vektoru b je použitím uvedeného algoritmu `copy` přepsáno.

I prvky lineárního zřetězeného seznamu mohou být v tomto stylu přepisovány, pokud obsahují dostatečný počet prvků, např.:

```
list<int> c;
// naplnění seznamu c alespoň 10 čísly
copy(a.begin(), a.end(), c.begin());
```

Avšak u zřetězených seznamů je častější vkládání prvků než jejich přepisování. Jsou dynamicky zvětšovány tak, jak jsou do nich prvky přidávány a dopředu se neprovádí jejich alokace na určitý počet prvků jako je tomu u vektorů.

Pro účely vkládání prvků do kontejneru při použití algoritmů jako je např. `copy` slouží *vkládací iterátory* (angl. *insert iterators*). V následujícím příkladu se vloží prvky vektoru `a` do prázdného zřetězeného seznamu `d`:

```
list<int> d;
copy(a.begin(), a.end(), front_insert_iterator<list<int> >(d));
```

Vkládací iterátory jsou adaptéry, které jsou modelem výstupních iterátorů. Existují tři druhy vkládacích iterátorů:

- `front_insert_iterator`
- `back_insert_iterator`
- `insert_iterator`.

Jedná se o šablony, definované v hlavičkovém souboru `<iterator>`.

Všechny tři vkládací iterátory se konstruují s parametrem typu kontejner, do kterého mají vkládat prvky. Operátor přiřazení vloží prvek do kontejneru na určité místo podle typu vkládacího iterátoru:

- `front_insert_iterator` vloží prvek na začátek kontejneru,
- `back_insert_iterator` vloží prvek na konec kontejneru,
- `insert_iterator` vloží prvek na pozici, na kterou ukazuje iterátor, předaný jako parametr konstruktoru.

Definice šablony `front_insert_iterator` je následující:

```
template <class Container>
class front_insert_iterator :
   public iterator<output_iterator_tag,void,void,void,void> {
protected:
   Container* container;
public:
   typedef Container container_type;
   explicit front_insert_iterator(Container& x);
   front_insert_iterator<Container>&
      operator=(typename Container::const_reference value);
   front_insert_iterator<Container>& operator*();
   front_insert_iterator<Container>& operator++();
   front_insert_iterator<Container> operator++(int);
};
```

Šablona má jeden typový parametr `Container`, který představuje kontejner, do kterého se mají prvky vkládat. Atribut `container` je inicializován explicitním konstruktorem.

Operátor přiřazení má parametr `value` typu `Container::const_reference`, což je konstantní reference na hodnotu prvku kontejneru. Kontejner musí mít deklarován vnořený typ `const_reference`, takže nelze použít jako kontejner obyčejné pole. Operátor přiřazení volá metodu `push_front` kontejneru příkazem:

```
container->push_front(value);
```

Tato metoda vloží prvek `value` na začátek kontejneru. Metoda je součástí jen takových typů kontejnerů, které umožňují vkládání prvků na začátek s konstantní časovou složitostí. Je to např. kontejner `list`. Metodu `push_front` neobsahuje např. kontejner `vector`.

Ostatní operátorové funkce vrací `*this` stejně jako je tomu u výstupního iterátoru datového proudu.

Šablona `back_insert_iterator` má stejnou definici jako šablona `front_insert_iterator`. Liší se pouze příkazem operátoru přiřazení, který obsahuje příkaz:

```
container->push_back(value);
```

Metoda `push_back` přidá prvek na konec kontejneru. Metoda je součástí např. kontejneru `list` a `vector`.

Šablona `insert_iterator` má následující definici:

```
template <class Container>
class insert_iterator :
   public iterator<output_iterator_tag,void,void,void,void> {
protected:
   Container* container;
   typename Container::iterator iter;
public:
   typedef Container container_type;
   insert_iterator(Container& x, typename Container::iterator i);
   insert_iterator<Container>&
      operator=(typename Container::const_reference value);
   insert_iterator<Container>& operator*();
   insert_iterator<Container>& operator++();
   insert_iterator<Container>& operator++(int);
};
```

Oproti šabloně `front_insert_iterator` má konstruktor dva parametry:

```
insert_iterator(Container& x, typename Container::iterator i);
```

Parametr `i` představuje iterátor, na jehož pozici se má vložit prvek. Parametr i inicializuje chráněný atribut `iter`. Operátor přiřazení obsahuje tři příkazy:

```
iter = container->insert(iter, value);
++iter;
return *this;
```

První příkaz volá metodu `insert` sdruženého kontejneru, která vloží prvek `value` na pozici iterátoru `iter` a vrací iterátor, který ukazuje na vložený prvek. Druhý příkaz posune tento iterátor o jednu pozici dopředu tak, aby ukazoval za vložený prvek. Tato metoda je např. součástí různých kontejnerů včetně kontejneru `list` a `vector`.

Příklad

```
int main() {
   int pole1[] = { 3, 2, 1 };
   int pole2[] = { 4, 5, 6 };
   int pole3[] = { 7, 8, 9 };
   list<int> seznam;
   copy(pole1, pole1+3, front_insert_iterator<list<int> >(seznam));
   copy(pole3, pole3+3, back_insert_iterator<list<int> >(seznam));
   list<int>::iterator iter = find(seznam.begin(), seznam.end(), 7);
   copy(pole2, pole2+3, insert_iterator<list<int> >(seznam, iter));
   copy (seznam.begin(), seznam.end(), ostream_iterator<int>(cout, " "));
   cout << endl;
   cin.get();
   return 0;
}
```

V uvedeném příkladu jsou nejprve hodnoty 3, 2 a 1 vloženy na začátek prázdného seznamu. Každá hodnota je vložena vždy na začátek seznamu, takže seznam bude po této operaci obsahovat čísla v pořadí 1, 2 a 3. Dále jsou hodnoty 7, 8 a 9 vloženy na konec seznamu. Potom se pomocí algoritmu `find` hledá pozice v seznamu, kde se nachází hodnota 7 a na tuto pozici, tj. před číslo 7, se potom vloží čísla 4, 5 a 6. Po těchto operacích bude seznam obsahovat uspořádanou posloupnost čísel 1, 2 až 9 a výstup programu bude následující:

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

Je rozdíl mezi těmito iterátory:

```
insert_iterator<list<int> >(seznam, seznam.begin())
```

a iterátorem

```
front_insert_iterator<list<int> >(seznam))
```

V prvním případě je první číslo vloženo na začátek seznamu, druhé číslo za první číslo, třetí číslo za druhé číslo atd. Kdežto ve druhém případě je každé číslo vloženo vždy na začátek. Takže vložením posloupnosti 1, 2 a 3 do prázdného seznamu bude seznam v prvním případě obsahovat posloupnost 1, 2 a 3 a ve druhém případě 3, 2 a 1.

Aby se při definici instance vkládacího iterátoru nemusel uvádět typ kontejneru v lomených závorkách za jménem šablony, musí se definovat tzv. *vytvořující funkce*. To je šablona obyčejné funkce, která vytvoří instanci šablony. Má stejné typové parametry jako šablona, jejíž instanci vytváří a stejné formální parametry jako jsou parametry konstruktoru šablony. Pro uvedené iterátory existuji vytvořující funkce:

- `front_inserter`
- `back_inserter`
- `inserter`.

Tyto vytvořující funkce mají podobnou definici. Např. funkce `front_inserter` má následující definici:

```
template <class Container>
front_insert_iterator<Container> front_inserter(Container& x)
{
   return front_insert_iterator<Container>(x);
}
```

V uvedeném příkladu by mohly příkazy volající algoritmus `copy` vypadat takto:

```
copy(pole1, pole1+3, front_inserter(seznam));
copy(pole3, pole3+3, back_inserter(seznam));
copy(pole2, pole2+3, inserter(seznam, iter));
```

FUNKTORY

Funktor (angl. *functor* nebo *function object*) je jakýkoli objekt, který lze použít po vzoru funkce s kulatými závorkami. Funktorem je tedy ukazatel nebo reference na obyčejnou funkci (včetně operátorové funkce) nebo instance třídy, která má přetížený operátor volání funkce.

Řada generických algoritmů má jako svůj parametr funktor. Např. algoritmus `for_each`, který může mít následující definici:

```
template <class InputIterator, class UnaryFunction>
UnaryFunction for_each(InputIterator first, InputIterator last,
                       UnaryFunction f)
{
   for (; first != last; ++first)
      f(*first);
   return f;
}
```

Algoritmus `for_each` pro každý prvek rozsahu `[first, last)` volá funktor `f`, který nesmí modifikovat prvek rozsahu. V normě C++ je druhý typový parametr této šablony pojmenován `Function`, ale přesnější je pojmenování `UnaryFunction` uvedené v knihovně STLport, protože funktor je volán s jedním parametrem.

Příklad

V následujícím příkladu se pro každý prvek pole celých čísel volá funkce `vypis`, která vypíše číslo na obrazovku, přičemž záporné číslo vypíše v kulatých závorkách bez znaménka minus.

```
void vypis(int a)
{
   if (a < 0) cout << "(" << -a << ") ";
   else cout << a << ' ';
}

int main()
{
   int pole[5] = { -1, -2, 0, 1, 2 };
   for_each(pole, pole+5, vypis); // #1
   return 0;
}
```

Výstup programu bude následující:

```
(1) (2) 0 1 2
```

Algoritmus `for_each` má jako třetí parametr identifikátor funkce `vypis`, který se automaticky zkonvertuje na ukazatel na funkci `vypis`. Stejného efektu se dosáhne zápisem:

```
for_each(pole, pole+5, &vypis);
```

Příkaz `#1` lze také nahradit dvojicí příkazů, kdy se nejprve definuje a inicializuje ukazatel na funkci a potom se zavolá funkce `for_each`:

```
void (*uf)(int) = vypis; // dtto: void (*uf)(int) = &vypis;
for_each(pole, pole+5, uf);
```

Funkci `for_each` je možné také volat s parametrem typu reference na funkci. Příkaz `#1` lze nahradit např. takto:

```
void (&rf)(int) = vypis;
for_each(pole, pole+5, rf);
```

Použití instance třídy pro funktor místo obyčejné funkce má význam zejména tehdy, pokud jsou potřebné další parametry pro vykonání funktoru. Ty jsou součástí třídy jako atributy a inicializují se zpravidla pomocí konstruktoru dané třídy.

Příklad

Pokud by se měla na obrazovku vypsat jen čísla, která spadají do zadaného intervalu, lze definovat třídu `TVypisInterval`, která obsahuje dva atributy, představující hranice intervalu, konstruktor, který je inicializuje a operátor volání funkce, který provede výpis čísla spadajícího do intervalu:

```
class TVypisInterval {
   int d, h;
public:
   TVypisInterval(int _d, int _h) : d(_d), h(_h) {}
   void     operator()(int a) const
               { if (a >= d && a <= h) cout << a << ' '; }
};
```

Výpis čísel dříve uvedeného pole patřících do intervalu `<-1, 1>` se potom provede příkazem:

```
for_each(pole, pole+5, TVypisInterval(-1,1));
```

Výstup programu bude následující:

```
-1 0 1
```

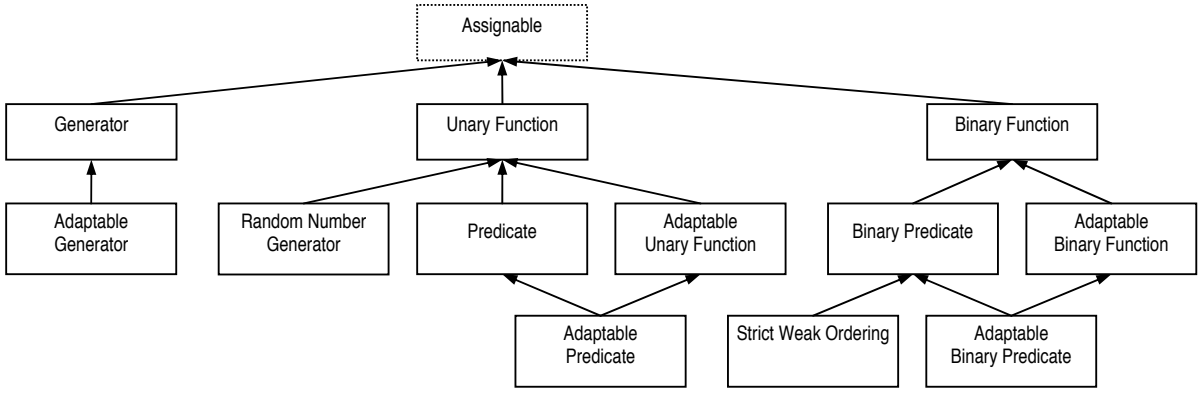
Třída `TVypisInterval` by mohla být určena i pro jiné prvky než jen celá čísla, pokud by byla definována jako šablona s jedním typovým parametrem udávajícím typ atributů `d`, `h` a parametru `a`.

Koncepty funktorů

Knihovna STLport definuje pro funktory 12 konceptů. Jsou na sobě hierarchicky závislé, tj. některý koncept funktoru je zjemněním jiného. Jejich hierarchická závislost je znázorněna na obr. 1. Jedná se o následující koncepty:

- *generátor* (angl. *Generator*) – funktor, který lze volat bez parametrů,
- *unární funkce* (angl. *Unary Function*) – funktor, který lze volat s jedním parametrem,
- *binární funkce* (angl. *Binary Function*) – funktor, který lze volat se dvěma parametry,
- *predikát* (angl. *Predicate*) – unární funkce, jejíž návratový typ lze implicitně konvertovat na typ `bool`,
- *binární predikát* (angl. *Binary Predicate*) – binární funkce, jejíž návratový typ lze implicitně konvertovat na typ `bool`,
- *adaptabilní* (angl. *Adaptable*) *generátor*, *unární funkce*, *binární funkce*, *predikát* a *binární predikát*,

- *generátor náhodných čísel* (angl. *Random Number Generator*),
- *ostré slabé uspořádání* (angl. *Strict Weak Ordering*).



Obr. 1 Hierarchie konceptů funktorů

Koncept generátoru, unární a binární funkce nepředepisuje typ návratové hodnoty funktoru. Ten může být závislý na algoritmu, který ho využívá.

Algoritmy využívající funktory, mají parametr funktoru volaný hodnotou, tj. při volání algoritmu dochází ke kopii funktoru. Z tohoto důvodu všechny funktory jsou zjemněním všeobecného konceptu `Assignable`, tj. lze je kopírovat pomocí kopírovacího konstruktoru nebo kopírovacího operátoru přiřazení.

Obecně může existovat *n-nární funkce*, která má *n* parametrů (např. ternární funkce má 3 parametry). Více než dva parametry funktoru nejsou ale v standardní knihovně C++ použity, proto nejsou pro ně definovány ani koncepty.

Generátor

Generátor používá např. algoritmus `generate`, který má následující prototyp:

```
template <class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last,
              Generator gen);
```

Algoritmus `generate` přiřazuje výsledek volání generátoru `gen` každému prvku rozsahu `[first, last)`.

Příklad

V následujícím příkladu se inicializuje vektor 10 celých čísel posloupností 1 až 10.

```
class TPosloupnost {
   int x;
public:
   TPosloupnost(int _x = 0) : x(_x) {}
   int operator()() { return ++x; }
};
```

```
int main()
{
   vector<int> vektor(10);
   generate(vektor.begin(), vektor.end(), TPosloupnost());
   copy(vektor.begin(), vektor.end(), ostream_iterator<int>(cout, " "));
   return 0;
}
```

Výstup programu bude následující:
```
1 2 3 4 5 6 7 8 9 10
```

Binární funkce

Koncept binární funkce vyžaduje např. jedna z verzí algoritmu `transform`, která má následující prototyp:

```
template <class InputIterator1, class InputIterator2,
          class OutputIterator, class BinaryFunction>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryFunction binary_op);
```

Tento algoritmus provede operaci `binary_op(*i1, *i2)` pro každý iterátor `i1` z rozsahu `[first1, last1)` a každý iterátor `i2` z rozsahu začínajícího iterátorem `first2`. Výsledek operace přiřadí do `*o`, kde `o` je výstupní iterátor z rozsahu začínajícího iterátorem `result`. To znamená, že pro každé `n`, pro které platí `0 <= n < last1 - first1`, algoritmus provede přiřazení `*(result + n) = binary_op(*(first1 + n), *(first2 + n))`. Algoritmus vrací výstupní iterátor, následující za posledním výstupním iterátorem, do kterého byl uložen výsledek operace, tj. vrací `result + (last1 - first1)`.

Příklad

V následujícím příkladu se vynásobí prvky pole celých čísel `A` a prvky pole reálných čísel `B`. Výsledky násobení jsou vypsány na obrazovku pomocí výstupního iterátoru datového proudu.

```
inline double Nasobeni(int a, double b) { return a*b; }

int main()
{
   int A[4] = { 1, 2, 3, 4 };
   double B[4] = { 0.5, 1.2, 2.5, 3.4 };
   transform(A, A+4, B, ostream_iterator<double>(cout, " "), Nasobeni);
   cin.get();
   return 0;
}
```

Výstup programu bude následující:
```
0.5 2.4 7.5 13.6
```

Predikát

Predikát je zjemněním konceptu unární funkce, jejíž návratový typ lze implicitně konvertovat na typ `bool`. Požaduje ho např. algoritmus `find_if`, který má následující prototyp:

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last,
                      Predicate pred);
```

Algoritmus `find_if` vrací první iterátor `i` z rozsahu `[first, last)`, pro který predikát `pred(*i)` je pravdivý. Pokud žádný takový iterátor neexistuje, algoritmus vrací `last`.

Příklad

Je definována šablona třídy `TJeMensiNez`, která obsahuje operátor volání funkce vracející `true`, pokud hodnota parametru operátoru je menší než hodnota atributu této šablony:

```
template <class T> class TJeMensiNez {
   T x;
public:
   TJeMensiNez(const T& _x) : x(_x) {}
   bool operator()(const T& t) const { return t < x; }
};
```

Šablonu je možné využít k hledání prvního záporného čísla ve vektoru celých čísel:

```
vector<int> vektor(5);
// naplnění vektoru
vector<int>::iterator it = find_if(vektor.begin(), vektor.end(),
                                   TJeMensiNez<int>(0));
if (it == vektor.end()) cout << "Ve vektoru neni zaporne cislo";
else                    cout << "Prvni zaporne cislo je " << *it;
```

Binární predikát

Binární predikát je zjemněním konceptu binární funkce, jejíž návratový typ lze implicitně konvertovat na typ `bool`. Používá ho např. jedna z verzí algoritmu `equal`, která má následující prototyp:

```
template <class InputIterator1, class InputIterator2,
          class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, BinaryPredicate binary_pred);
```

Algoritmus `equal` vrací `true`, pokud se všechny prvky rozsahu `[first1, last1)` rovnají prvkům rozsahu začínajícího iterátorem `first2`. Rovnost dvou prvků se testuje pomocí binárního predikátu `binary_pred`. Přesněji řečeno, algoritmus vrací `true`, pokud pro každý iterátor `i` z rozsahu `[first1, last1)` predikát `binary_pred(*i, *(first2 + (i - first1)))` je pravdivý.

Generátor náhodných čísel

Generátor náhodných čísel je unární funkce, která vrací pseudonáhodné celé číslo z intervalu `<0, n)`, kde `n` je parametr funkce. Typ parametru funkce musí být shodný s jejím návratovým typem – musí se jednat o celočíselný typ. Parametr `n` musí být kladné číslo. Generovaná čísla odpovídají rovnoměrnému rozdělení pravděpodobnosti.

Generátor náhodných čísel používá např. jedna z verzí algoritmu `random_shuffle`, která má následující prototyp:

```
template <class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first,
                    RandomAccessIterator last,
                    RandomNumberGenerator& rand);
```

Algoritmus `random_shuffle` náhodně promíchá prvky rozsahu `[first, last)` pomocí generátoru náhodných čísel `rand`.

Druhá verze algoritmu `random_shuffle` má jen dva parametry `first`, `last` a používá interní generátor pseudonáhodných čísel.

Ostré slabé uspořádání

Koncept ostré slabé uspořádání je zjemněním konceptu binárního predikátu. Vrací `true`, pokud hodnota prvního parametru předchází hodnotě druhého parametru. Oba parametry musí být stejného typu. Koncept má vlastnosti ostrého slabého uspořádání – viz 7. přednáška.

Funktor vyhovující konceptu ostrého slabého uspořádání vyžaduje např. jedna z verzí algoritmu třídění `sort`, která má následující prototyp:

```
template <class RandomAccessIterator, class StrictWeakOrdering>
void sort(RandomAccessIterator first, RandomAccessIterator last,
          StrictWeakOrdering comp);
```

Algoritmus `sort` utřídí prvky rozsahu `[first, last)` vzestupně. Pro porovnání dvou prvků volá funktor `comp(*i, *j)`, kde `i` a `j` jsou iterátory z rozsahu `[first, last)`.

Druhá verze algoritmu `sort` má jen dva parametry `first`, `last` a pro porovnání prvků používá výraz `*i < *j`.

Příklad

Je dán vektor bodů typu `TBod`. Třída `TBod` obsahuje dva atributy: souřadnici `x` a `y`. Vektor se utřídí pomocí funkce `MensiBod` vzestupně nejprve podle souřadnice `x` a potom podle souřadnice `y`.

```
struct TBod {
   int x, y;
   TBod(int _x, int _y) : x(_x), y(_y) {}
};

inline bool MensiBod(const TBod& a, const TBod& b)
{
   return a.x < b.x || (a.x == b.x && a.y < b.y);
}

int main()
{
   vector<TBod> SeznamBodu;
   SeznamBodu.push_back(TBod(3, 4));
   SeznamBodu.push_back(TBod(3, 2));
   SeznamBodu.push_back(TBod(2, 5));
   SeznamBodu.push_back(TBod(1, 1));
   sort(SeznamBodu.begin(), SeznamBodu.end(), MensiBod);
   // ...
   return 0;
}
```

Po provedení funkce `sort` bude vektor obsahovat body v pořadí (1, 1), (2, 5), (3, 2) a (3, 4).

Adaptabilní funktory a adaptéry funktorů

Adaptabilní funktory jsou funktory, které obsahují deklaraci vnořených typů pro návratový typ a parametry funktoru.

Adaptabilní generátor musí obsahovat pouze deklaraci vnořeného typu `result_type`, který představuje návratový typ generátoru.

Adaptabilní unární funkce a *adaptabilní predikát* musí obsahovat deklaraci dvou vnořených typů:

- `result_type` – návratový typ funktoru,
- `argument_type` – typ parametru funktoru.

Adaptabilní binární funkce a *adaptabilní binární predikát* musí obsahovat deklaraci tří vnořených typů:

- `result_type` – návratový typ funktoru,
- `first_argument_type` – typ prvního parametru funktoru,
- `second_argument_type` – typ druhého parametru funktoru.

Význam adaptabilních funktorů spočívá v tom, že je mohou využít adaptéry funktorů.

Adaptér funktoru (angl. *function object adapter*) je adaptabilní funktor, který transformuje rozhraní nějakého funktoru. Z toho vyplývá, že lze adaptéry řetězit a určitému algoritmu lze předat jako parametr místo funktoru např. adaptér adaptéru funktoru.

Příklad

V kapitole „Predikát“ byl uveden příklad, v němž se pomocí funktoru (šablony třídy) `TJeMensiNez` hledalo první číslo menší než zadané číslo v určité posloupnosti, konkrétně první záporné číslo ve vektoru. Kdybychom chtěli v posloupnosti hledat první číslo, které je větší nebo rovno než zadané číslo, mohli bychom buď napsat další funktor, který by místo operátoru < obsahoval operátor >=, nebo můžeme vytvořit adaptér funktoru, který provede negaci výsledku zadaného funktoru, v našem případě negaci operátoru <.

K tomu musíme nejprve z funktoru `TJeMensiNez` vytvořit adaptabilní funktor, např. `TAdaptJeMensiNez`, který obsahuje deklaraci vnořených typů `result_type` a `argument_type`:

```
template <class T> class TAdaptJeMensiNez {
public:
   typedef bool result_type;
   typedef T argument_type;
private:
   T x;
public:
   TAdaptJeMensiNez(const T& _x) : x(_x) {}
   bool operator()(const T& t) const { return t < x; }
};
```

Potom definujeme adaptér unární funkce `TNegace`:

```
template <class AdaptableUnaryFunction> class TNegace {
public:
   typedef bool result_type;
   typedef typename AdaptableUnaryFunction::argument_type argument_type;
private:
   AdaptableUnaryFunction f;
public:
   TNegace(AdaptableUnaryFunction _f) : f(_f) {}
   bool operator()(argument_type t) { return !f(t); }
};
```

Adaptér obsahuje atribut `f` typu adaptabilní unární funkce, který je inicializován pomocí konstruktoru. Protože adaptér je adaptabilní funktor, obsahuje deklaraci vnořených typů `result_type` a `argument_type`. Typ `argument_type` je synonymem stejně pojmenovaného typu funktoru unární funkce. Operátor volání funkce neguje výsledek volání funktoru `f`.

Hledání prvního nezáporného čísla ve vektoru by se provedlo příkazem:

```
vector<int>::iterator it = find_if(vektor.begin(), vektor.end(),
        TNegace<TAdaptJeMensiNez<int> >(TAdaptJeMensiNez<int>(0)));
```

Abychom nemuseli u adaptéru uvádět typ funktoru v lomených závorkách, vytvoříme ještě vytvořující funkci adaptéru:

```
template <class AdaptableUnaryFunction>
inline TNegace<AdaptableUnaryFunction> Negace(AdaptableUnaryFunction f)
{
   return TNegace<AdaptableUnaryFunction>(f);
}
```

Příkaz pro hledání se potom zjednoduší na zápis:

```
vector<int>::iterator it = find_if(vektor.begin(), vektor.end(),
                                   Negace(TAdaptJeMensiNez<int>(0)));
```

Bázové třídy adaptabilních funktorů

Aby se u adaptabilních funktorů nemusely deklarovat požadované vnořené typy, jsou v hlavičkovém souboru `<functional>` definovány dvě šablony tříd, od kterých mohou být adaptabilní funktory odvozeny. Jedná se o následující definice bázové šablony třídy pro adaptabilní unární a binární funkci:

```
template <class Arg, class Result> struct unary_function {
   typedef Arg argument_type;
   typedef Result result_type;
};

template <class Arg1, class Arg2, class Result> struct binary_function {
   typedef Arg1 first_argument_type;
   typedef Arg2 second_argument_type;
   typedef Result result_type;
};
```

Pro adaptabilní generátor bázová třída neexistuje. Šablona `unary_function` má dva typové parametry: první představuje typ parametru funktoru a druhý typ výsledku. Šablona `binary_function` má tři typové parametry: první a druhý jsou typy parametrů funktoru a třetí je typ výsledku.

Příklad

Předchozí příklad by mohl být upraven dále uvedeným způsobem. Funktor `TAdaptJeMensiNez` by mohl být odvozen od šablony `unary_function` takto:

```
template <class T>
class TAdaptJeMensiNez : public unary_function<T, bool> {
private:
   T  x;
public:
   TAdaptJeMensiNez(const T& _x) : x(_x) {}
   bool operator()(const T& t) const { return t < x; }
};
```

Obě verze funktoru `TAdaptJeMensiNez` jsou správně definované adaptabilní funktory, nicméně všechny funktory v standardní knihovně C++ jsou potomkem uvedených bázových tříd, proto se tento způsob doporučuje i pro uživatelem definované funktory.

Obdobně lze odvodit od šablony `unary_function` také adaptér funktoru `TNegace`. V tomto případě se ale musí vnořené typ `argument_type` kvalifikovat jménem typového parametru `AdaptableUnaryFunction`. I v tomto případě norma jazyka C++ dává přednost odvození adaptéru od bázové třídy. Definice adaptéru `TNegace` je následující:

```
template <class AdaptableUnaryFunction> class TNegace :
   public unary_function<typename AdaptableUnaryFunction::argument_type,
                         bool> {
private:
   AdaptableUnaryFunction f;
public:
   TNegace(AdaptableUnaryFunction _f) : f(_f) {}
   bool operator ()
      (typename AdaptableUnaryFunction::argument_type t) const
      { return !f(t); }
};
```

FUNKTORY – POKRAČOVÁNÍ

Standardní funktory

Pro operátory vestavěných datových typů nelze získat adresu, která by se mohla použít jako ukazatel na funkci při volání nějakého generického algoritmu. Pro tento účel se musí volat prostřednictvím tříd (resp. šablon tříd) funktorů. Knihovna obsahuje řadu předdefinovaných adaptabilních funktorů, které implementují různé aritmetické, relační nebo logické operátory. Všechny jsou definovány v hlavičkovém souboru `<functional>`. Jedná se o šablony struktur, odvozené od šablony `unary_function` nebo `binary_function`. Např. pro aritmetický operátor součtu je definována šablona `plus` následovně:

```
template <class T> struct plus : binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x + y; }
};
```

Z definice je zřejmě, že je lze používat nejen pro vestavěné datové typy, ale i uživatelem definované. Definice ostatních standardních funktorů je obdobná. Jejich seznam je uveden v tab. 1.

Tab. 1 Seznam standardních funktorů, které implementují operátory

Jméno funktoru	Operace	Předek
Aritmetické operace		
`plus`	`x + y`	`binary_function<T, T, T>`
`minus`	`x - y`	`binary_function<T, T, T>`
`multiplies`	`x * y`	`binary_function<T, T, T>`
`divides`	`x / y`	`binary_function<T, T, T>`
`modulus`	`x % y`	`binary_function<T, T, T>`
`negate`	`-x`	`unary_function<T,T>`
Operace porovnávání		
`equal_to`	`x == y`	`binary_function<T, T, bool>`
`not_equal_to`	`x != y`	`binary_function<T, T, bool>`
`greater`	`x > y`	`binary_function<T, T, bool>`
`less`	`x < y`	`binary_function<T, T, bool>`
`greater_equal`	`x >= y`	`binary_function<T, T, bool>`
`less_equal`	`x <= y`	`binary_function<T, T, bool>`
Logické operace		
`logical_and`	`x && y`	`binary_function<T, T, bool>`
`logical_or`	`x || y`	`binary_function<T, T, bool>`
`logical_not`	`!x`	`unary_function<T,bool>`

Příklad

```
int main()
{
   list<int> seznam1, seznam2;
   for (int i = 0; i < 5; i++) seznam1.push_back(i);
   for (int i = 10; i < 15; i++) seznam2.push_back(i);
   transform(seznam1.begin(), seznam1.end(),
             seznam2.begin(), seznam1.begin(), plus<int>());
   copy(seznam1.begin(), seznam1.end(),
        ostream_iterator<int>(cout, " "));
   return 0;
}
```

V uvedeném příkladu se ke každému prvku seznamu `seznam1` přičte hodnota prvku seznamu `seznam2` pomocí algoritmu `transform`. Tento algoritmus může mít výstupní iterátor shodný s jedním ze vstupních iterátorů – v tomto případě výstupní iterátor může být `seznam1.begin()` nebo `seznam2.begin()`.

Výstup programu bude následující:

```
10 12 14 16 18
```

Příklad

V následujícím příkladu se hodnota každého prvku vektoru logických hodnot zneguje pomocí algoritmu `transform`:

```
int main()
{
   vector<bool> vektor;
   vektor.push_back(true);
   vektor.push_back(false);
   vektor.push_back(true);
   cout << boolalpha;
   cout << "vektor pred negaci: ";
   copy(vektor.begin(), vektor.end(), ostream_iterator<bool>(cout, " "));
   cout << endl;
   transform(vektor.begin(), vektor.end(),
             vektor.begin(), logical_not<bool>());
   cout << "vektor po negaci: ";
   copy(vektor.begin(), vektor.end(), ostream_iterator<bool>(cout, " "));
   cout << endl;
   cin.get();
   return 0;
}
```

Výstup programu bude následující:

```
vektor pred negaci: true false true
vektor po negaci: false true false
```

V příkladu byla použita druhá verze algoritmu `transform`, která má následující prototyp:

```
template <class InputIterator, class OutputIterator, class UnaryFunction>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryFunction op);
```

Algoritmus provede operaci `op(*i)` pro každý iterátor `i` z rozsahu `[first, last)`. Výsledek operace přiřadí do `*o`, kde `o` je výstupní iterátor z rozsahu začínajícího iterátorem `result`. To znamená, že pro každé `n`, pro které platí `0 <= n < last - first`, algoritmus provede přiřazení

`*(result + n) = op(*(first + n))`. Algoritmus vrací výstupní iterátor, následující za posledním výstupním iterátorem, do kterého byl uložen výsledek operace, tj. vrací `result + (last - first)`.

Standardní adaptéry funktorů

Standardní knihovna C++ nabízí řadu předdefinovaných adaptérů funktorů. Všechny jsou definovány v hlavičkovém souboru `<functional>`.

Negátory

Negátor (angl. *negator*) je adaptér funktoru, který neguje výsledek volání adaptabilního unárního nebo binárního predikátu. Existují dvě verze negátoru:

- `unary_negate` – je modelem adaptabilního predikátu a neguje predikát,
- `binary_negate` – je modelem adaptabilního binárního predikátu a neguje binární predikát.

Negátor `unary_negate` má následující definici:

```
template <class AdaptablePredicate>
class unary_negate : public
   unary_function<typename AdaptablePredicate::argument_type, bool> {
protected:
   AdaptablePredicate pred; // jmeno atributu muze byt libovolne
public:
   explicit unary_negate(const AdaptablePredicate& _pred) : pred(_pred){}
   bool operator()
      (const typename AdaptablePredicate::argument_type& x) const
      { return !pred(x); }
};
```

Definice negátoru `binary_negate` je obdobná:

```
template <class AdaptableBinaryPredicate>
class binary_negate : public
  binary_function<typename AdaptableBinaryPredicate::first_argument_type,
          typename AdaptableBinaryPredicate::second_argument_type, bool>
{
protected:
   AdaptableBinaryPredicate pred; // jmeno atributu muze byt libovolne
public:
   explicit binary_negate(const AdaptableBinaryPredicate& _pred) :
      pred(_pred){}
   bool operator()
      (const typename AdaptableBinaryPredicate::first_argument_type& x,
       const typename AdaptableBinaryPredicate::second_argument_type& y)
      const
   { return !pred(x, y); }
};
```

Negátory vytváří kopii predikátu, který negují. Kopie predikátu se ukládá do neveřejného atributu `pred` v konstruktoru. Oba negátory mají své vytvořující funkce:

- `not1` – pro negátor `unary_negate`,
- `not2` – pro negátor `binary_negate`.

Příklad

Hledání prvního nezáporného čísla ve vektoru celých čísel pomocí dříve definovaného adaptabilního predikátoru `TAdaptJeMensiNez` se může provést příkazem:

```
vector<int>::iterator it = find_if(vektor.begin(), vektor.end(),
                                   not1(TAdaptJeMensiNez<int>(0)));
```

Vazače

Vazač (angl. *binder*) je adaptér funktoru, který transformuje binární funkci na unární tak, že jeden z parametrů binární funkce je fixní (vázaný). Vazač je tedy modelem adaptabilní unární funkce. Existují dva druhy vazačů:

- `binder1st` – má vázaný první parametr binární funkce,
- `binder2nd` – má vázaný druhý parametr binární funkce.

Definice vazače `binder1st` je následující:

```
template <class ABF>
class binder1st : public
   unary_function<typename ABF::second_argument_type,
                  typename ABF::result_type> {
protected:
   ABF op;
   typename ABF::first_argument_type value;
public:
   binder1st(const ABF &x, const typename ABF::first_argument_type& y) :
      op(x), value(y) {}
   typename ABF::result_type operator()
      (const typename ABF::second_argument_type& x) const
      { return op(value, x); }
};
```

Typový parametr vazače má být `AdaptableBinaryFunction`. Aby uvedená definice byla přehlednější, je pro typový parametr zvolena zkratka `ABF`. Konstruktor vazače má dva parametry. První parametr `x` představuje binární funkci, kterou má vazač transformovat a druhý parametr `y` představuje první parametr binární funkce, který bude fixní. Parametry `x` a `y` inicializují chráněné atributy `op` a `value`. Operátor volání funkce vrací výsledek volání binární funkce, které předá jako první parametr vázaný parametr `value` a jako druhý parametr předá parametr operátoru volání funkce `x`.

Definice vazače `binder2nd` je podobná.

Pro oba vazače existují tyto vytvořující funkce:

- `bind1st` – pro vazač `binder1st`,
- `bind2nd` – pro vazač `binder2nd`.

Příklad

Příkazem

```
vector<int>::iterator it =
   find_if(vektor.begin(), vektor.end(), bind2nd(less<int>(), 0));
```

se hledá první záporné číslo ve vektoru celých čísel, tj. pro každý prvek `x` vektoru se testuje pravdivost výrazu `x < 0`.

Zatímco příkazem

```
vector<int>::iterator it =
   find_if(vektor.begin(), vektor.end(), bind1st(less<int>(), 0));
```

se hledá první kladné číslo ve vektoru, tj. pro každý prvek `x` vektoru se testuje pravdivost výrazu `0 < x`.

Adaptéry obyčejných funkcí

Aby mohla být obyčejná funkce modelem adaptabilního funktoru, musí obsahovat deklarace vnořených typů. To se provede vytvořením adaptéru, který požadované vnořené typy deklaruje. Existují dva adaptéry obyčejných funkcí:

- `pointer_to_unary_function` – adaptuje obyčejnou unární funkci,
- `pointer_to_binary_function` – adaptuje obyčejnou binární funkci.

Definice adaptéru `pointer_to_unary_function` je následující:

```
template <class Arg, class Result>
class pointer_to_unary_function : public unary_function<Arg, Result> {
protected:
   Result (*f)(Arg);
public:
   explicit pointer_to_unary_function(Result (* _f)(Arg)) : f(_f) {}
   Result operator()(Arg x) const { return f(x); }
};
```

Adaptér `pointer_to_unary_function` je odvozen od šablony `unary_function`, která deklaruje typy požadované pro adaptabilní unární funkci. Konstruktor přebírá ukazatel na obyčejnou unární funkci, kterým inicializuje atribut `f`. Operátor volání funkce vrací výsledek volání funkce, na kterou ukazuje `f`.

Adaptér `pointer_to_binary_function` má podobnou definici:

```
template <class Arg1, class Arg2, class Result>
class pointer_to_binary_function :
   public binary_function<Arg1, Arg2, Result> {
protected:
   Result (*f)(Arg1, Arg2);
public:
   explicit pointer_to_binary_function(Result (* _f)(Arg1, Arg2)) :
      f(_f) {}
   Result operator()(Arg1 x, Arg2 y) const { return f(x, y); }
};
```

Pro oba adaptéry je k dispozici přetížená vytvořující funkce `ptr_fun`.

Příklad

Je definována struktura bodu `TBod` s atributy `x` a `y`, pro níž je definován přetížený operátor výstupu `<<` a funkce `ShodneBody`, která vrací `true`, pokud se dva body rovnají. Ve funkci `main` se pomocí algoritmu `replace_if` v seznamu bodů `Body` nahradí každý bod mající souřadnice [3, 4] bodem [0, 0].

```
struct TBod {
   int x, y;
   TBod(int _x, int _y) : x(_x), y(_y) {}
};
```

```
ostream& operator << (ostream& os, const TBod& t)
{
   os << '(' << t.x << ',' << t.y << ") ";
   return os;
}

inline bool ShodneBody(TBod t1, TBod t2)
{ return t1.x == t2.x && t1.y == t2.y; }

int main()
{
   list<TBod> Body;
   Body.push_back(TBod(1, 2));
   Body.push_back(TBod(3, 4));
   Body.push_back(TBod(5, 6));
   Body.push_back(TBod(3, 4));
   replace_if(Body.begin(), Body.end(),
             bind2nd(ptr_fun(ShodneBody), TBod(3, 4)), TBod(0, 0)); // #1
   copy(Body.begin(), Body.end(), ostream_iterator<TBod>(cout));
   cin.get();
   return 0;
}
```

Algoritmus `replace_if` má následující deklaraci:

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

Tento algoritmus nahradí každý prvek z rozsahu `[first, last)`, pro který predikát `pred` je pravdivý, prvkem `new_value`. Přesněji řečeno, pro každý iterátor `i` z rozsahu `[first, last)`, pro který `pred(*i)` je pravdivý, provede přiřazení `*i = new_value`.

Funkce `ShodneBody` nemůže být přímo parametrem adaptéru `binder2nd`, protože neobsahuje vnořené typy. Proto se musí zapouzdřit do adaptéru `pointer_to_binary_function` pomocí vytvořující funkce `ptr_fun`.

Výstup programu bude následující:

```
(1,2) (0,0) (5,6) (0,0)
```

Pokud by příkaz `#1` byl nahrazen zápisem:

```
replace_if(Body.begin(), Body.end(),
           not1(bind2nd(ptr_fun(ShodneBody), TBod(3, 4))), TBod(0, 0));
```

v seznamu se nahradí každý bod, který nemá souřadnice [3, 4] bodem [0, 0] a program potom vypíše takovýto text:

```
(0,0) (3,4) (0,0) (3,4)
```

Pokud bychom v uvedeném programu použili místo funkce `ShodneBody` funkci `ShodneBodyRef`, která by měla parametry typu konstantní reference na `TBod`:

```
inline bool ShodneBodyRef(const TBod& t1, const TBod& t2);
```

překladač by oznámil chybu ve vazači `binder2nd`, protože by parametrem jeho konstruktoru a operátoru volání funkce byla reference na referenci. V takovém případě bychom museli definovat

vlastní vazač, např. `binder2nd_ref` a jeho vytvořující funkci `bind2nd_ref` následujícím způsobem:

```
template <class ABF>
class binder2nd_ref : public
   unary_function<typename ABF::first_argument_type,
                  typename ABF::result_type> {
protected:
   ABF op;
   typename ABF::second_argument_type value;
public:
   binder2nd_ref(const ABF &x, typename ABF::second_argument_type y) :
      op(x), value(y) {}
   typename ABF::result_type operator()
     (typename ABF::first_argument_type x) const { return op(x, value); }
};

template <class ABF>
inline binder2nd_ref<ABF> bind2nd_ref(const ABF &x,
                                      typename ABF::second_argument_type y)
{
   return binder2nd_ref<ABF>(x, y);
}
```

Z důvodu přehlednosti je místo typového parametru `AdaptableBinaryFunction` použita jeho zkratka `ABF`.

Adaptéry metod objektových typů

Adaptéry metod umožňují použít na místě volání funktoru volání metody určité třídy. Obsahují třídní ukazatel na určitou metodu, který je inicializován konstruktorem. Operátor volání funkce volá prostřednictvím třídního ukazatele příslušnou metodu, a to pro instanci, která je prvním parametrem operátoru volání funkce. Existuje osm typů těchto adaptérů, jejichž seznam je uveden v tab. 2. Liší se typem metody, na který třídní ukazatel ukazuje, počtem parametrů volané metody (0 nebo 1) a typem prvního parametru operátoru volání funkce (ukazatel nebo reference). Adaptéry jsou buď potomkem šablony `unary_function` nebo `binary_function` v závislosti na počtu parametrů operátoru volání funkce a volané metody.

Tab. 2 Seznam standardních adaptérů metod objektových typů

Adaptér	Třídní ukazatel	Operátor volání funkce
`mem_fun_t<S,T>`	`S (T::*f)()`	`S operator()(T* p) const { return (p->*f)(); }`
`mem_fun1_t<S,T,A>`	`S (T::*f)(A)`	`S operator()(T* p, A x) const { return (p->*f)(x); }`
`mem_fun_ref_t<S,T>`	`S (T::*f)()`	`S operator()(T& p) const { return (p.*f)(); }`
`mem_fun1_ref_t <S,T,A>`	`S (T::*f)(A)`	`S operator()(T& p, A x) const { return (p.*f)(x); }`
`const_mem_fun_t<S,T>`	`S (T::*f)() const`	`S operator()(const T* p) const { return (p->*f)(); }`
`const_mem_fun1_t <S,T,A>`	`S (T::*p)(A) const`	`S operator()(const T* p, A x) const { return (p->*f)(x); }`

Adaptér	Třídní ukazatel	Operátor volání funkce
`const_mem_fun_ref_t <S,T>`	`S (T::*p)() const`	`S operator()(const T& p) const { return (p.*f)(); }`
`const_mem_fun1_ref_t <S,T,A>`	`S (T::*p)(A) const`	`S operator()(const T& p, A x) const { return (p.*f)(x); }`

Definice např. adaptéru `mem_fun_t` je následující:

```
template <class S, class T> class mem_fun_t :
   public unary_function<T*, S> {
private:
   S (T::*f)();
public:
   explicit mem_fun_t(S (T::*p)()) : f(p) {}
   S operator()(T* p) const { return (p->*f)(); }
};
```

Pro uvedené adaptéry existují dvě přetížené vytvořující funkce:

- `mem_fun` – pro adaptéry mající jako první parametr operátoru volání funkce ukazatel na instanci, tj. `mem_fun_t`, `mem_fun1_t`, `const_mem_fun_t` a `const_mem_fun1_t`,
- `mem_fun_ref` – pro adaptéry, mající jako první parametr operátoru volání funkce referenci na instanci, tj. `mem_fun_ref_t`, `mem_fun1_ref_t`, `const_mem_fun_ref_t` a `const_mem_fun1_ref_t`.

Příklad

Je dán seznam bodů typu `TBod`. Pomocí algoritmu `for_each` a metody `Vypis` třídy `TBod` se na obrazovku vypíší souřadnice všech bodů seznamu. Potom se pomocí algoritmu `count_if` a metody `RovnoX` zjistí počet bodů, které mají souřadnici x = 3.

```
class TBod {
   int x, y;
public:
   TBod(int _x, int _y) : x(_x), y(_y) {}
   void Vypis() const { cout << '(' << x << ',' << y << ") "; }
   int RovnoX(int _x) const { return x == _x; }
};

int main()
{
   list<TBod> Body;
   Body.push_back(TBod(1, 2));
   Body.push_back(TBod(3, 4));
   Body.push_back(TBod(5, 6));
   Body.push_back(TBod(3, 5));
   for_each(Body.begin(), Body.end(), mem_fun_ref(TBod::Vypis));
   int n = count_if(Body.begin(), Body.end(),
                    bind2nd(mem_fun_ref(TBod::RovnoX), 3));
   cout << endl << n;
   cin.get();
   return 0;
}
```

Výpis programu bude následující:

```
(1,2) (3,4) (5,6) (3,5)
2
```

Algoritmus `count_if` má následující prototyp:

```
template <class InputIterator, class Predicate>
iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, Predicate pred);
```

Tento algoritmus vrací počet prvků z rozsahu `[fist, last)`, pro které predikát `pred` je pravdivý. Přesněji řečeno, vrací počet iterátorů `i` z rozsahu `[fist, last)`, pro které `pred(*i)` vrací `true`.

Nestandardní funktory a adaptéry funktorů

Součástí knihovny STLport jsou oproti normě jazyka C++:

- funktory: `identity`, `project1st`, `project2nd`, `select1st` a `select2nd`,
- adaptéry funktorů: `unary_compose` a `binary_compose`.

KONCEPTY KONTEJNERŮ – ÚVOD

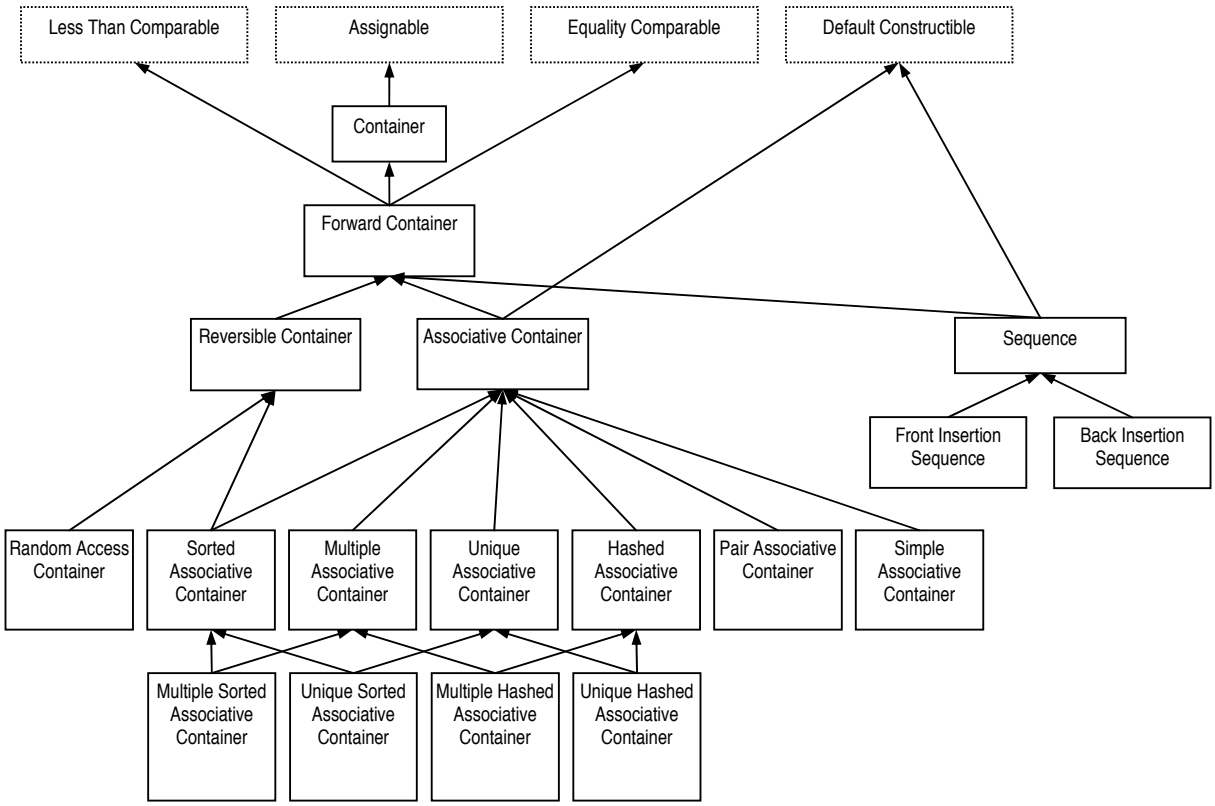
Kontejner (*angl. container*) je objekt, který obsahuje jiné objekty, tzv. *prvky* (angl. *elements*) a obsahuje metody pro zpřístupnění jeho prvků.

Knihovna STLport definuje pro kontejnery 18 konceptů:

- *kontejner* (angl. *Container*),
- *dopředný kontejner* (angl. *Forward Container*),
- *reverzibilní kontejner* (angl. *Reversible Container*),
- *kontejner s náhodným přístupem* (angl. *Random Access Container*),
- *sekvence* (angl. *Sequence*),
- *sekvence s vkládáním na začátek* (angl. *Front Insertion Sequence*),
- *sekvence s vkládáním na konec* (angl. *Back Insertion Sequence*),
- *asociativní kontejner* (angl. *Associative Container*),
- *jednoduchý asociativní kontejner* (angl. *Simple Associative Container*),
- *párový asociativní kontejner* (angl. *Pair Associative Container*),
- *tříděný asociativní kontejner* (angl. *Sorted Associative Container*),
- *hašovací asociativní kontejner* (*kontejner s transformací klíče*) (angl. *Hashed Associative Container*),
- *jednoznačný asociativní kontejner* (angl. *Unique Associative Container*),
- *víceznačný asociativní kontejner* (angl. *Multiple Associative Container*),
- *jednoznačný tříděný asociativní kontejner* (angl. *Unique Sorted Associative Container*),
- *víceznačný tříděný asociativní kontejner* (angl. *Multiple Sorted Associative Container*),
- *jednoznačný hašovací asociativní kontejner* (*jednoznačný kontejner s transformací klíče*) (angl. *Unique Hashed Associative Container*),
- *víceznačný hašovací asociativní kontejner* (*víceznačný kontejner s transformací klíče*) (angl. *Multiple Hashed Associative Container*).

V normě jazyka C++ jsou definovány společné požadavky na všechny kontejnery, které odpovídají konceptu dopředného kontejneru a konceptu `DefaultConstructible`. Dále jsou v normě definovány požadavky na sekvenci (přibližně odpovídají konceptu Sekvence) a asociativní kontejner.

Jejich hierarchická závislost je znázorněna na obr. 1.



Obr. 1 Hierarchie konceptů kontejnerů

V následujícím textu této a dalších kapitol popisujících koncepty kontejnerů jsou použity tyto symboly:

`X` ..... typ, který je modelem daného konceptu kontejneru,

`T` ..... typ `X::value_type` – typ prvku kontejneru `X`, tzv. *typ hodnoty* (angl. *value type)* kontejneru,

`a`, `b` ..... objekty (instance) typu `X`,

`t` ..... objekt typu `T`,

`n` ..... objekt typu, který lze implicitně konvertovat na typ `X::size_type`,

`p`, `q` ..... objekty typu `X::iterator`,

`k` ..... objekt typu `X::key_type`,

`d` ..... objekt typu `X::mapped_type`,

`c` ..... objekt typu `X::key_compare`.

Všeobecné koncepty kontejnerů

Koncept Kontejner

Základním konceptem kontejnerů je *Kontejner (angl. Container).* Ostatní koncepty kontejnerů jsou přímo nebo nepřímo jeho zjemněním.

Kontejner musí mít mj. sdružený typ `iterator`, který slouží k procházení prvků kontejneru. Není zaručena neměnnost pořadí prvků kontejneru při jeho iteraci. Není garantováno, že v určitém

okamžiku bude aktivní pouze jeden iterátor daného kontejneru. Kontejner vlastní své prvky. Prvky zaniknou se zánikem kontejneru. Pokud však kontejner vlastní ukazatele na nějaké objekty, při zániku kontejneru zaniknou ukazatele, ale objekty, na které ukazatele ukazují, zaniknout nemusí.

Velikost (angl. *size*) *kontejneru* je počet prvků kontejneru.

Koncept kontejneru je zjemněním konceptu `Assignable`.

Kontejner musí mít deklarovány vnořené typy odpovídající výrazům v následující tabulce.

Výraz	Význam, požadavky
`X::value_type`	Typ prvku kontejneru, tj. *typ hodnoty* kontejneru. Musí být modelem konceptu `Assignable`.
`X::iterator`	Typ iterátoru. Musí být modelem vstupního iterátoru nebo jeho zjemnění. Musí se dát konvertovat na `X::const_iterator`. *Typ hodnoty* iterátoru musí být *typem hodnoty* (prvku) kontejneru.
`X::const_iterator`	Typ konstantního iterátoru. Slouží jen k procházení prvků kontejneru, ne k jejich modifikaci. Musí být modelem minimálně vstupního iterátoru. *Typ hodnoty* iterátoru musí být *typem hodnoty* (prvku) kontejneru.
`X::reference`	Typ, který se chová jako reference na *typ hodnoty* kontejneru.
`X::const_reference`	Typ, který se chová jako konstantní reference na *typ hodnoty* kontejneru.
`X::pointer`	Typ, který se chová jako ukazatel na *typ hodnoty* kontejneru.
`X::difference_type`	Celočíselný typ se znaménkem představující *typ vzdálenosti* dvou iterátorů `X::iterator` resp. `X::const_iterator`. Zpravidla se jedná o synonymum pro typ `ptrdiff_t`.
`X::size_type`	Celočíselný typ bez znaménka, který může reprezentovat nezápornou hodnotu typu `X::difference_type`. Zpravidla se jedná o synonymum pro typ `size_t`.

Model konceptu kromě výrazů definovaných v konceptu, kterého je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`X(a)`		Kopírovací konstruktor. Po provedení výrazu platí `X().size() == a.size()`, kde `X()` obsahuje kopii každého prvku kontejneru `a`.
`X b(a);`		Kopírovací konstruktor. Po provedení výrazu platí `b.size() == a.size()`, kde `b` obsahuje kopii každého prvku kontejneru `a`.
`b = a`		Kopírovací operátor přiřazení. Po provedení výrazu platí `b.size() == a.size()`, kde `b` obsahuje kopii každého prvku kontejneru `a`.
`a.~X()`		Destruktor. Každý prvek kontejneru `a` je zrušen a paměť pro ně je dealokována.

Výraz	Návratový typ	Význam, požadavky
`a.begin()`	`iterator` pro nekonstantní `a`, `const_iterator` pro konstantní `a`	Vrací iterátor ukazující na první prvek kontejneru.
`a.end()`	`iterator` pro nekonstantní `a`, `const_iterator` pro konstantní `a`	Vrací koncový iterátor.
`a.size()`	`size_type`	Vrací velikost kontejneru. Platí: `a.size() >= 0 && a.size() <= a.max_size()`. Vrací stejnou hodnotu jako vzdálenost mezi iterátory `a.begin()` a `a.end()`.
`a.max_size()`	`size_type`	Vrací maximální možnou velikost kontejneru.
`a.empty()`	lze implicitně konvertovat na typ `bool`	Ekvivalent výrazu `a.size() == 0`, ale může být zapsáno efektivněji.
`a.swap(b)`	`void`	Ekvivalent výrazu `swap(a, b)`. Algoritmus `swap` provede výměnu obsahu dvou proměnných.

V normě C++ není definován vnořený typ `pointer`.

Dopředný kontejner

Dopředný kontejner je kontejner, u kterého se pořadí prvků při iteraci nemění. Tím pádem mohou být dva dopředné kontejnery porovnávány pomocí relačních operátorů `==`, `!=`, `<`, `<=`, `>` a `>=`. Dopředný kontejner je tedy zjemněním nejen konceptu Kontejner, ale i konceptů `LessThanComparable` a `EqualityComparable`. Při porovnávání dvou kontejnerů se porovnávají jejich prvky, a proto prvky kontejneru musí být modelem nejen konceptu `Assignable`, ale i konceptu `LessThanComparable` a `EqualityComparable`.

Iterátor dopředného kontejneru musí být modelem dopředného iterátoru. Tím pádem může být kontejner používán *víceprůchodovými* algoritmy. V jednom okamžiku může být aktivních více iterátorů stejného kontejneru.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`a == b`	lze implicitně konvertovat na typ `bool`	Vrací `true`, pokud `a.size() == b.size()` a každý prvek kontejneru `a` se rovná odpovídajícímu prvku kontejneru `b`.
`a < b`	lze implicitně konvertovat na typ `bool`	Vrací `true`, pokud první prvek kontejneru `a` je menší než první prvek kontejneru `b`. Jestliže se první prvky rovnají, vrací true, pokud druhý prvek kontejneru `a` je menší než druhý prvek kontejneru `b` atd. Je ekvivalentem lexikografického porovnávání.

Reverzibilní kontejner

Reverzibilní kontejner je dopředný kontejner, jehož iterátor je dvousměrný.

Musí mít deklarovány vnořené typy odpovídající výrazům v následující tabulce.

Výraz	Význam, požadavky
`X::reverse_iterator`	Typ inverzního iterátoru, který je adaptérem iterátoru `X::iterator`.
`X::const_reverse_iterator`	Typ konstantního inverzního iterátoru, který je adaptérem iterátoru `X::const_iterator`.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`a.rbegin()`	`reverse_iterator` pro nekonstantní `a`, `const_reverse_iterator` pro konstantní `a`	Ekvivalent výrazu `X::reverse_iterator(a.end())`.
`a.rend()`	`reverse_iterator` pro nekonstantní `a`, `const_reverse_iterator` pro konstantní `a`	Ekvivalent výrazu `X::reverse_iterator(a.begin())`.

Modelem tohoto konceptu je kontejner `list` . Kontejner `list` je navíc modelem konceptů sekvence s vkládáním na začátek a sekvence s vkládáním na konec.

Kontejner s náhodným přístupem

Kontejner s náhodným přístupem je reverzibilní kontejner, jehož iterátor je iterátor s náhodným přístupem.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedenému v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`a[n]`	`reference` pro nekonstantní `a`, `const_reference` pro konstantní `a`	Vrací `n`-tý prvek od začátku kontejneru. Výsledkem je stejný prvek, jako kdyby se `n` krát inkrementoval iterátor `a.begin()`. Musí platit: `0 <= n < a.size()`. Operace má konstantní časovou složitost.
`a.at(n)`	`reference` pro nekonstantní `a`, `const_reference` pro konstantní `a`	Má stejný význam, jako výraz `a[n]` s tím rozdílem, že se kontroluje, zda `n` je platným indexem prvku pole. V případě, že není, vyvolá se výjimka `out_of_range`.

Modelem tohoto konceptu jsou kontejnery `vector` a `deque`. Kontejner `vector` je navíc modelem konceptu sekvence s vkládáním na konec a kontejner `deque` je navíc modelem konceptů sekvence s vkládáním na začátek a sekvence s vkládáním na konec.

KONCEPTY KONTEJNERŮ – SEKVENCE

Sekvence

Sekvence je kontejner, jehož velikost se může měnit a jehož prvky jsou uchovávány v lineárním pořadí. Sekvence je tedy koncept nějaké lineární datové struktury.

Je zjemněním konceptu dopředného kontejneru a konceptu `DefaultConstructible`.

Typ prvků sekvence musí být modelem konceptů `Assignable`, `LessThanComparable`, `EqualityComparable` a `DefaultConstructible`.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`X(n, t)` nebo `X a(n, t);`		Konstruktor, který vytvoří sekvenci `n` kopií objektu `t`. Po provedení výrazu platí: `size() == n` resp. `a.size() == n`. Parametr `n` musí být `>= 0`.
`X(n)` nebo `X a(n);`		Vytvoří posloupnost `n` prvků inicializovaných implicitním konstruktorem `T()`. Po provedení výrazu platí: `size() == n` resp. `a.size() == n`. Parametr `n` musí být `>= 0`.
`X()` nebo `X a;`		Implicitní konstruktor. Ekvivalent výrazu `X(0)`. Po provedení výrazu platí: `size() == 0` resp. `a.size() == 0`.
`X(i, j)` nebo `X a(i, j);`		Parametry `i` a `j` jsou vstupní iterátory, jejichž *typ hodnoty* je konvertovatelný na typ `T`. Vytvoří sekvenci, která je kopií rozsahu `[i, j)`. Po provedení výrazu platí: `size()` resp. `a.size()` je rovno vzdálenosti rozsahu `[i, j)`.
`a.front()`	`reference` pro nekonstantní `a`, `const_reference` pro konstantní `a`	Vrací referenci na první prvek kontejneru. Ekvivalent výrazu `*(a.begin())`. Kontejner nesmí být prázdný. Operace má konstantní časovou složitost.
`a.insert(p, t)`	`X::iterator`	Kopie objektu `t` se vloží před prvek, na který ukazuje iterátor `p`. Vrací iterátor, který ukazuje na vložený prvek.
`a.insert(p, n, t)`	`void`	Vloží `n` kopií objektu `t` před prvek, na který ukazuje iterátor `p`.
`a.insert(p, i, j)`	`void`	Vloží kopie prvků z rozsahu `[i, j)` před prvek, na který ukazuje iterátor `p`. Parametry `i` a `j` jsou vstupní iterátory, jejichž typ hodnoty je konvertovatelný na typ `T`.
`a.erase(p)`	`iterator`	Odstraní prvek, na který ukazuje iterátor `p`, z kontejneru. Vrací iterátor, který ukazuje na prvek následující po odstraněném prvku.

Výraz	Návratový typ	Význam, požadavky
`a.erase(p,q)`	`iterator`	Odstraní prvky rozsahu `[p, q)` z kontejneru. Vrací iterátor, který ukazuje na prvek následující po posledním odstraněném prvku.
`a.clear()`	`void`	Odstraní všechny prvky z kontejneru. Ekvivalent výrazu `a.erase(a.begin(), a.end())`.
`a.resize(n, t)`	`void`	Změní velikost kontejneru na `n` prvků. Případné nové prvky inicializuje kopií objektu `t`. Pokud `n < a.size()`, odstraní `(a.size() - n)` prvků z konce kontejneru. Po provedení výrazu platí: `a.size() == n`.
`a.resize(n)`	`void`	Ekvivalent výrazu `a.resize(n, T())`.

V normě C++ jsou výrazy „`X(n)`“ a „`X a(n);`“ uvedeny již ve společných požadavcích na všechny kontejnery. V normě C++ nejsou definovány metody `resize`, nicméně všechny kontejnery odpovídající konceptu sekvence, je obsahují.

V normě C++ jsou v požadavcích na sekvenci variantně (podle typu kontejneru) specifikovány požadavky obsažené v konceptech sekvence s vkládáním na začátek, sekvence s vkládáním na konec a kontejneru s náhodným přístupem.

Příklad

V uvedeném příkladu se používají kontejnery `list` a `vector`. Tyto kontejnery nejsou sice přímo modelem konceptu sekvence, ale jsou modelem konceptů, které jsou zjemněním konceptu sekvence.

V příkladu se vytvoří seznam celých čísel `B`, do kterého se pomocí konstruktoru zkopírují prvky obyčejného pole `Pole`, které jsou typu `char`. Při jejich kopírováním se provede implicitní konverze z typu `char` na typ `int`. Potom se vytvoří vektor celých čísel `A`, který obsahuje 5 prvků s hodnotou nula. Na začátek seznamu `B` se dále vloží první 3 prvky vektoru `A` a na konec 3 prvky s hodnotou 5. Seznam `B` se na závěr vypíše na obrazovku.

```
int main()
{
   char        Pole[4] = { 1, 2, 3, 4 };
   list<int>   B(Pole, Pole+4);
   vector<int> A(5, 0);
   B.insert(B.begin(), A.begin(), A.begin()+3);
   B.insert(B.end(), 3, 5);
   copy(B.begin(), B.end(), ostream_iterator<int>(cout, " "));
   cin.get();
   return 0;
}
```

Výpis programu bude následující:

```
0 0 0 1 2 3 4 5 5 5
```

Sekvence s vkládáním na začátek

Sekvence s vkládáním na začátek je sekvence, která umožňuje vkládání prvku na začátek kontejneru a odstranění prvního prvku z kontejneru s konstantní časovou složitostí.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`a.push_front(t)`	`void`	Vloží kopii objektu `t` na začátek kontejneru. Ekvivalent výrazu `a.insert(a.begin(), t)`.
`a.pop_front()`	`void`	Odstraní první prvek kontejneru. Kontejner nesmí být prázdný. Ekvivalent výrazu `a.erase(a.begin())`.

Modelem tohoto konceptu jsou kontejnery `list` a `deque`. Tyto kontejnery jsou navíc modelem dalších konceptů. Modelem tohoto konceptu je i `slist`, který je součástí pouze knihovny STLport.

Sekvence s vkládáním na konec

Sekvence s vkládáním na konec je sekvence, která umožňuje přidání prvku na konec kontejneru, odstranění posledního prvku z kontejneru a získání hodnoty posledního prvku kontejneru s konstantní časovou složitostí.

Model konceptu kromě výrazů definovaných v konceptech, kterých je zjemněním, musí vyhovovat výrazům uvedeným v následující tabulce.

Výraz	Návratový typ	Význam, požadavky
`a.back()`	`reference` pro nekonstantní `a`, `const_reference` pro konstantní `a`	Vrací referenci na poslední prvek kontejneru. Ekvivalent výrazu `*(--a.end())`.
`a.push_back(t)`	`void`	Vloží kopii objektu `t` na konec kontejneru. Ekvivalent výrazu `a.insert(a.end(), t)`.
`a.pop_back()`	`void`	Odstraní poslední prvek kontejneru. Kontejner nesmí být prázdný. Ekvivalent výrazu `a.erase(--a.end())`.

Modelem tohoto konceptu jsou kontejnery `vector`, `deque` a `list`. Tyto kontejnery jsou modelem i dalších konceptů.

MODELY KONCEPTŮ SEKVENCE

Modelem konceptů sekvence jsou následující kontejnery:

Třída kontejneru	Je modelem konceptů
`vector` `vector<bool>`	kontejner s náhodným přístupem sekvence s vkládáním na konec
`deque`	kontejner s náhodným přístupem sekvence s vkládáním na začátek sekvence s vkládáním na konec
`list`	reverzibilní kontejner sekvence s vkládáním na začátek sekvence s vkládáním na konec
`slist`	sekvence s vkládáním na začátek

Třída kontejneru	Je modelem konceptů
`basic_string`	kontejner s náhodným přístupem sekvence s vkládáním na konec

Norma jazyka C++ nespecifikuje jakou abstraktní datovou strukturu ten který kontejner představuje. Nicméně v popisech jednotlivých implementací standardní C++ knihovny je u některých kontejnerů druh abstraktní datové struktury uveden.

Všechny kontejnery, které jsou modelem sekvence, uvedené v této kapitole, obsahují typový parametr šablony `Allocator`, což je šablona třídy, která zajišťuje potřebné alokace a dealokace. Implicitně je to šablona třídy `allocator`, která pro většinu případů vyhovuje. Např. deklarace šablony třídy `vector` je následující:

```
template <class T, class Allocator = allocator<T> >
class vector;
```

Všechny konstruktory těchto kontejnerů kromě jejich kopírovacího konstruktoru mají poslední parametr implicitní, který je typu `const Allocator&` a jeho implicitní hodnotou je instance `Allocator()`. Zadaná instance alokátoru se použije při vytvoření dané instance kontejneru. Jejich deklarace např. pro šablonu třídy `vector` je následující:

```
explicit vector(const Allocator& = Allocator());
explicit vector(size_type n, const T& value = T(),
                const Allocator& = Allocator());
template <class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

Všechny tyto kontejnery dále obsahují následující deklaraci vnořeného typu `allocator_type`:

```
typedef Allocator allocator_type;
```

Dále obsahují metodu `get_allocator`, která vrací instanci třídy `Allocator`, se kterou je kontejner svázán. Její deklarace je následující:

```
allocator_type get_allocator() const;
```

Šablona třídy vector

Šablona třídy `vector` je sekvence, která má iterátor s náhodným přístupem. I když norma jazyka C++ nespecifikuje vnitřní strukturu kontejneru, jedná se o šablonu abstraktní datové struktury jednorozměrného pole neboli vektoru. Vektor je dynamicky alokovaný a po jeho vytvoření lze do něj prvky přidávat nebo z něj odstraňovat. Vložení a odstranění prvku na konci vektoru má konstantní časovou složitost, zatímco vložení a odstranění prvku v jiném místě vektoru má lineární časovou složitost.

Vektor může být alokován na více prvků, než kolik jich právě obsahuje. Celkový počet prvků, které vektor může obsahovat bez potřeby realokace, se nazývá *kapacita* vektoru. Kapacitu vektoru lze jen zvýšit. Kapacita se nesníží ani při odstranění všech prvků z vektoru. Pokud je kapacita vektoru rovna velikosti vektoru a má se do něj vložit další prvek, kapacita vektoru se určitým způsobem zvýší. V knihovně STLport se zdvojnásobí, v knihovně Visual C++ 2003 se zvýší o 50%. Při realokaci vektoru (při změně jeho kapacity) se zneplatní všechny iterátory a ukazatele na prvky vektoru.

Šablona je definována v hlavičkovém souboru `<vector>`. Deklarace šablony je následující:

```
template <class T, class Allocator = allocator<T> >
class vector;
```

Šablona má dva typové parametry. Parametr `T` představuje typ prvku vektoru. Parametr `Allocator` byl vysvětlen dříve.

Kromě metod uvedených v konceptech, kterých je vektor modelem, obsahuje následující metody.

```
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Vymaže vektor a zkopíruje do něj prvky z rozsahu `[first, last)`.

```
void assign(size_type n, const T& u);
```

Vymaže původní vektor a vytvoří nový vektor o `n` prvcích, které inicializuje kopií objektu `u`.

```
size_type capacity() const;
```

Vrací kapacitu vektoru. Platí: `capacity() >= size()`.

```
void reserve(size_type n);
```

Pokud `n <= capacity()`, neprovede žádnou operaci. Jinak zvětší kapacitu vektoru na `n`. Po provedení metody `reserve(n)` bude platit `capacity()== n`. Velikost vektoru, tj. výsledek metody `size()` se nezmění. Nezmění se ani hodnoty a pořadí prvků ve vektoru. Pokud `n > max_size()`, vyvolá výjimku `length_error`.

Příklad

Je dán vektor `C`, kterému se nastaví kapacita na 4 prvky. Zkopírují se do něj 4 prvky pole `A` a vypíše se jeho velikost, kapacita a prvky na obrazovku. Potom se na jeho konec přidají 3 prvky ze vstupního paměťového datového proudu `B` a znovu se provede stejný výpis. Přidáním prvků proudu `B` se zdvojnásobí kapacita vektoru `A` na 8 prvků.

```
int main()
{
   int            A[4] = { 1, 2, 3, 4 };
   istringstream  B("5 6 7");
   vector<int>    C;
   C.reserve(4);
   C.assign(A, A+4);
   copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
   cout << "\nVelikost vektoru: " << C.size() << ", kapacita: " <<
           C.capacity() << endl;
   C.insert(C.end(), istream_iterator<int>(B), istream_iterator<int>());
   copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
   cout << "\nVelikost vektoru: " << C.size() << ", kapacita: " <<
           C.capacity() << endl;
   cin.get();
   return 0;
}
```

Výpis programu v prostředí Visual C++ 2003 bude následující:

```
1 2 3 4
Velikost vektoru: 4, kapacita: 4
1 2 3 4 5 6 7
Velikost vektoru: 7, kapacita: 9
```

Šablonu třídy `vector` je možné použít pro vytvoření datové struktury matice, např. matice celých čísel:

```
vector< vector<int> > Matice;
```

Pro lepší práci s maticí je vhodné zapouzdřit tuto strukturu do třídy, která by měla metody poskytující rozhraní matice, např. konstruktor pro vytvoření matice, metodu pro změnu velikosti matice, metody pro získání počtu řádků a sloupců matice, operátor volání funkce nebo indexování pro zpřístupnění prvku matice.

Šablona třídy vector<bool>

Šablona třídy `vector<bool>` je parciální specializací šablony třídy `vector` pro prvky typu `bool`. Každý prvek tohoto vektoru zabírá jeden bit. Zkrácená definice šablony je následující:

```
template <class Allocator> class vector<bool, Allocator> {
public:
   typedef bool const_reference;

   class reference {
      friend class vector;
      reference();
      // neveřejné atributy
   public:
      ~reference();
      operator bool() const;
      reference& operator=(const bool x);
      reference& operator=(const reference& x);
      void flip();
   };

   static void swap(reference x, reference y);
   void flip();

   // další složky
};
```

Šablona obsahuje vnořenou třídu `reference`, která představuje referenci na prvek vektoru. Konstantní reference je deklarována jako synonymum pro typ `bool`. Reference má tyto složky:

- neveřejný konstruktor – nelze tedy vytvořit instanci tohoto typu mimo třídu `reference` (kromě šablony `vector`, která je přítelem reference),
- operátor přetypování na typ `bool`,
- operátor přiřazení z hodnoty typu `bool`,
- kopírovací operátor přiřazení,
- metoda `flip()` – neguje (překlopí) bit, který je s referencí spjat, tj. z hodnoty 0 vytvoří hodnotu 1 a naopak.

Oproti primární definici šablony `vector` obsahuje šablona `vector<bool>` dvě další metody.

```
static void swap(reference x, reference y);
```

Vymění hodnoty (bity) dvou referencí

```
void flip();
```

Neguje (překlopí) všechny bity (hodnoty prvků) vektoru.

Příklad

Je dán vektor logických hodnot, který se naplní třemi hodnotami. Hodnota prvního prvku se neguje a vektor se vypíše na obrazovku.

```
int main()
{
   vector<bool> A(3);
   A[0] = true;
   A[1] = false;
   A[2] = true;
   vector<bool>::reference r = A[0]; #1
   r.flip();#2
   cout << boolalpha;
   copy(A.begin(), A.end(), ostream_iterator<bool>(cout, " "));
   cin.get();
   return 0;
}
```

Příkazy #1 a #2 je možné nahradit příkazem `A[0].flip();`.

Výstup programu bude následující:

```
false false true
```

Šablona třídy deque

Šablona třídy `deque` představuje obousměrnou frontu. Jméno `deque` (vyslovuje se stejně jako anglické slovo „deck“) je zkratkou textu „double-ended queue“. Je podobná šabloně třídy `vector`. Oproti vektoru je navíc modelem sekvence s vkládáním na začátek, tj. lze s konstantní časovou složitostí vložit nebo odstranit prvek na začátku fronty. S konstantní časovou složitostí lze tedy provést i operace vložení a odstranění prvku na konci fronty a operaci náhodného přístupu k prvku. Operace vložení a odstranění prvku uvnitř fronty má lineární časovou složitost. Ačkoli šablona `deque` provádí s konstantní časovou složitostí stejné operace jako šablona `vector`, šablona `vector` danou operaci provede zpravidla rychleji, protože její implementace je jednodušší než šablony `deque`.

Interní reprezentace obousměrné fronty závisí na typu použité knihovny. Může se jednat o jednorozměrné pole s platnými prvky od určitého po určitý index pole. Nebo se může jednat o pole ukazatelů na pole prvků.

Všechny iterátory fronty se zneplatní při vložení prvku na jakoukoli pozici do fronty (včetně na začátek a konec) a při odstranění prvku ze středu fronty. Pokud se odstraní prvek ze začátku nebo konce fronty, zneplatní se jen iterátor, který ukazuje na odstraněný prvek.

Šablona je definována v hlavičkovém souboru `<deque>`. Deklarace šablony je následující:

```
template <class T, class Allocator = allocator<T> >
class deque;
```

Význam typových parametrů je stejný jako u šablony `vector`. Narozdíl od vektoru nemá metody `capacity` a `resize`, ale jako vektor obsahuje dvě metody `assign` – jejich význam je stejný jako u vektoru.

Šablona třídy list

Šablona třídy `list` je sekvence, která má dvousměrný iterátor. Představuje abstraktní datovou strukturu obousměrného zřetězeného seznamu. Konkrétní typ seznamu záleží na implementaci v dané knihovně. Je modelem konceptu reverzibilního kontejneru, sekvence s vkládáním na začátek a na konec.

Vložení a odstranění prvků z kterékoli pozice v seznamu má konstantní časovou složitost. Zpřístupnění prvku uvnitř seznamu má lineární časovou složitost. Iterátory seznamu zůstanou platné při vložení prvku na jakoukoli pozici do seznamu. Při odstranění prvku ze seznamu se zneplatní pouze iterátor, který ukazuje na odstraněný prvek.

Šablona třídy `list` je definována v hlavičkovém souboru `<list>` a její deklarace je následující:

```
template <class T, class Allocator = allocator<T> >
class list;
```

Význam typových parametrů je stejný jako u šablony `vector`.

Kromě metod uvedených v konceptech, kterých je seznam modelem, obsahuje následující metody.

```
template <class InputIterator>
void assign(InputIterator first, InputIterator last);
```

Popis viz šablona `vector`.

```
void assign(size_type n, const T& u);
```

Popis viz šablona `vector`.

```
void splice(iterator position, list<T,Allocator>& x);
```

Před prvek, na který ukazuje iterátor `position`, vloží všechny prvky ze seznamu `x`. Ze seznamu `x` prvky odstraní – seznam `x` bude prázdný. Prvky se nekopírují, ale pouze přesouvají. Operace má konstantní časovou složitost.

```
void splice(iterator position, list<T,Allocator>& x, iterator i);
```

Před prvek, na který ukazuje iterátor `position`, vloží ze seznamu `x` prvek, na který ukazuje iterátor `i`. Ze seznamu `x` prvek odstraní. Prvek se nekopíruje, ale přesouvá. Seznam `x` může být shodný se seznamem, na který ukazuje `this`. Operace nezmění seznam, pokud platí: `position == i` nebo `position == ++i`. Operace má konstantní časovou složitost.

```
void splice(iterator position, list<T,Allocator>& x, iterator first,
            iterator last);
```

Před prvek, na který ukazuje iterátor `position`, vloží prvky z rozsahu `[fist, last)` seznamu `x`. Ze seznamu `x` prvky uvedeného rozsahu odstraní. Prvky se nekopírují, ale pouze přesouvají. Seznam `x` může být shodný se seznamem, na který ukazuje `this`, ale iterátor `position` nesmí patřit do rozsahu `[fist, last)`. Pokud `&x == this`, má operace konstantní časovou složitost, jinak má lineární časovou složitost.

```
void remove(const T& value);
```

Ze seznamu odstraní každý prvek, pro který platí `*i == value`, kde `i` je iterátor z celého rozsahu prvků seznamu.

```
template <class Predicate> void remove_if(Predicate pred);
```

Ze seznamu odstraní každý prvek, pro který predikát `pred(*i)` je pravdivý, kde `i` je iterátor z celého rozsahu prvků seznamu. Pořadí prvků, které nejsou odstraněny, se nezmění.

```
void unique();
```

Ze seznamu v každé souvislé skupině shodných prvků odstraní všechny prvky kromě prvního, na které ukazuje iterátor `i` v rozsahu `[first+1, last)`, pro které platí `*i == *(i - 1)`. Operace má lineární časovou složitost.

```
template <class BinaryPredicate>
void unique(BinaryPredicate binary_pred);
```

Ze seznamu v každé souvislé skupině shodných prvků odstraní všechny prvky kromě prvního, na které ukazuje iterátor `i` v rozsahu `[first+1, last)`, pro které binární predikát `pred(*i, *(i - 1))` je pravdivý.

```
void merge(list<T,Allocator>& x);
```

Spojí utříděný seznam `*this` s utříděným seznamem `x` do seznamu `*this`. Po provedení sloučení bude seznam `x` prázdný. Prvky ze seznamu `x` se nekopírují, ale pouze přesouvají. Uspořádání seznamů vyhovuje vlastnostem ostrého slabého uspořádání. Prvky seznamů jsou porovnávány pomocí operátoru <. Pokud je prvek seznamu `*this` stejný s prvkem seznamu `x`, prvek seznamu `x` se vloží za prvek seznamu `*this`. Všechny iterátory obou seznamů zůstanou platné. Operace má lineární časovou složitost.

```
template <class StrictWeakOrdering>
void merge(list<T,Allocator>& x, StrictWeakOrdering comp);
```

Provede stejnou operaci jako metoda `merge` s jedním parametrem. Rozdíl je v tom, že prvky seznamu jsou porovnávány pomocí funktoru ostrého slabého uspořádání (viz 10. přednáška).

```
void sort();
```

Utřídí seznam, tak, že prvky seznamu porovnává pomocí operátoru <, který vyhovuje vlastnostem ostrého slabého uspořádání. Všechny iterátory seznamu zůstanou platné a budou ukazovat na původní prvky. Algoritmus provede přibližně (N log N) porovnání prvků, kde N je velikost seznamu.

```
template <class StrictWeakOrdering> void sort(StrictWeakOrdering comp);
```

Provede totéž co metoda `sort()` s tím rozdílem, že prvky jsou porovnávány pomocí funktoru ostrého slabého uspořádání (viz 10. přednáška).

```
void reverse();
```

Otočí pořadí prvků v seznamu. Všechny iterátory seznamu zůstanou platné a budou ukazovat na původní prvky. Operace má lineární časovou složitost.

Příklad

V uvedeném příkladu jsou použity různé metody šablony třídy `list` pro seznam celých čísel a po každé provedené operaci je seznam vypsán na obrazovku pomocí funkce `Vypis`.

```
void Vypis(list<int>& t, const char* text)
{
   cout << text;
   copy(t.begin(), t.end(), ostream_iterator<int>(cout, " "));
   cout << endl;
}
```

```
int main()
{
   int pole[] = { 6, 3, 2, 2, 7, 5, 2, 2, 4, 3, 8 };
   list<int> A(pole, pole + sizeof pole/sizeof pole[0]);
   list<int> B(3, 1);
   Vypis(A, "Seznam A: ");
   Vypis(B, "Seznam B: ");
   A.splice(++A.begin(), B);
   Vypis(A, "Seznam A po provedeni splice: ");
   A.remove_if(bind2nd(greater<int>(), 6));
   Vypis(A, "Seznam A po odstraneni prvku > 6: ");
   A.unique();
   Vypis(A, "Seznam A po provedeni unique: ");
   A.sort();
   Vypis(A, "Utrideny seznam A: ");
   A.reverse();
   Vypis(A, "Prevraceny seznam A: ");
   cin.get();
   return 0;
}
```

Výpis programu bude následující:

```
Seznam A: 6 3 2 2 7 5 2 2 4 3 8
Seznam B: 1 1 1
Seznam A po provedeni splice: 6 1 1 1 3 2 2 7 5 2 2 4 3 8
Seznam A po odstraneni prvku > 6: 6 1 1 1 3 2 2 5 2 2 4 3
Seznam A po provedeni unique: 6 1 3 2 5 2 4 3
Utrideny seznam A: 1 2 2 3 3 4 5 6
Prevraceny seznam A: 6 5 4 3 3 2 2 1
```

Šablona třídy slist

Šablona třídy `slist` je sekvence, která má dopředný iterátor. Představuje abstraktní datovou strukturu jednosměrného zřetězeného seznamu. Šablona není součástí normy jazyka C++, je uvedena pouze v knihovně STLport. Je modelem konceptu sekvence s vkládáním na začátek.

Bližší informace viz nápověda ke knihovně STLport.

Šablona třídy basic_string

Šablona třídy `basic_string` je sekvence znaků neboli řetězec znaků. Je modelem stejných konceptů jako šablona třídy `vector`, tj. konceptu kontejneru s náhodným přístupem a sekvence s vkládáním na konec. Šablona je definována v hlavičkovém souboru `<string>`. Její deklarace je následující:

```
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_string;
```

Parametr `charT` představuje typ znaků řetězce, zpravidla `char` nebo `wchar_t`. Parametr `traits` je třída, která popisuje vlastnosti množiny znaků. Implicitně se zde používá šablona struktury `char_traits`. Třetí parametr `Allocator` má stejný význam jako stejně pojmenovaný parametr šablony třídy `vector`.

Norma jazyka C++ definuje požadavky na třídu, které odpovídá šablona struktury `char_traits`. V knihovně STLport existuje pro tyto požadavky koncept `CharacterTraits`. Třída, která je modelem tohoto konceptu musí obsahovat:

- Deklaraci vnořených typů `char_type`, `int_type`, `off_type`, `pos_type`, `state_type`.
- Statické metody pro různé operace se znaky nebo posloupností znaků, např.:
  - rovnost dvojice znaků `c` a `d`: `eq(c, d)`,
  - zjištění, zda znak `c` je menší než znak `d`: `lt(c, d)`,
  - hledání znaku v posloupnosti znaků: `find`,
  - zjištění délky posloupnosti znaků `p`: `length(p)`,
  - získání koncového znaku v dané množině znaků: `eof()`.

Jedna ze statických metod je metoda `compare` s následující deklarací:

```
int compare(const char_type* p, const char_type* q, size_t n);
```

Metoda `compare` porovnává prvních `n` znaků dvou řetězců `p` a `q`. Vrací nulu, pokud se řetězce rovnají, zápornou hodnotu, pokud `p < q` a kladnou hodnotu, pokud `p > q`. Pro řetězce znaků obsahující české znaky, by bylo možné definovat svojí verzi této metody příp. i metody `lt` v šabloně třídy odvozené z `char_traits`.

Šablona třídy `char_traits` je definována v hlavičkovém souboru `<string>`, ve kterém jsou definovány i dvě její explicitní specializace: `char_traits<char>` a `char_traits<wchar_t>`.

Podle normy jazyka C++ reference, ukazatelé a iterátory, které se odkazují na prvky šablony `basic_string` se mohou stát neplatnými (ale nemusí – závisí na implementaci) následujícími operacemi:

- obyčejnými funkcemi `swap`, `operator >>` a `getline`, které pracují se šablonou `basic_string`,
- metodami `swap`, `data`, `c_str` šablony `basic_string`,
- voláním jakékoli nekonstantní metody šablony `basic_string` kromě metod `operator[]`, `at`, `begin`, `rbegin`, `end` a `rend`.

V knihovně STLport a Visual C++ 2003 zůstávají iterátory platné i po volání metod `c_str()` a `data()`.

V hlavičkovém souboru `<string>` jsou deklarovány dvě instance šablony `basic_string` pro typ `char` a `wchar_t`:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

Řetězec může být alokován na více znaků, než kolik jich právě obsahuje. Má tzv. *kapacitu*. Kapacitu řetězce narozdíl od kapacity vektoru, lze zvýšit i snížit. Snížit ji lze jen na hodnotu velikosti řetězce, kterou vrací metoda `size()`. Není definováno, o kolik se zvýší kapacita řetězce, pokud je rovna velikosti řetězce a má se do řetězce vložit další znak.

Kromě vnořených typů, které jsou specifikovány v konceptech, kterých je šablona `basic_string` modelem, obsahuje deklaraci vnořeného typu `traits_type`:

```
typedef traits traits_type;
```

Šablona obsahuje statický konstantní atribut `npos`, který představuje největší možnou hodnotu typu `size_type`. Největší možný index prvku v řetězci může být `npos - 1`. Je definován takto:

```
static const size_type npos = -1;
```

Metody mohou vyvolat výjimku `length_error`, pokud by nová velikost řetězce byla `>= npos`. V metodách se vyskytuje parametr `pos` typu `size_type`, který udává pozici (index) znaku v řetězci. První znak v řetězci má pozici 0. Metody mohou vyvolat výjimku `out_of_range`, pokud jejich parametr `pos` je větší než velikost řetězce, ke kterému se parametr `pos` vztahuje. V následujícím popisu metod nejsou tyto případy vyvolání výjimek `length_error` a `out_of_range` uváděny.

Kromě metod uvedených v konceptech, kterých je řetězec modelem, obsahuje následující metody.

Konstruktory

```
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos, const Allocator& a = Allocator());
```

Zkopíruje ***rlen*** znaků řetězce `str` od pozice `pos`. Hodnota ***rlen*** `= min(n, str.size() - pos)`. Jestliže se použijí implicitní parametry, jedná se o kopírovací konstruktor.

```
basic_string(const charT* s, size_type n,
             const Allocator& a = Allocator());
```

Zkopíruje `n` znaků z řetězce `s`.

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

Vytvoří řetězec z řetězce `s`. Ekvivalent konstruktoru `basic_string(s, traits::length(s), a)`.

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

Vytvoří řetězec obsahující `n` znaků hodnoty `c`.

Kapacitní metody

```
size_type length() const;
```

Vrací `size()`.

```
size_type capacity() const;
```

Vrací kapacitu řetězce. Platí: `capacity() >= size()`.

```
void reserve(size_type n = 0);
```

Voláním metody se požaduje změna kapacity řetězce na hodnotu `n`. Pokud `n < capacity()`, metoda sníží kapacitu řetězce na hodnotu `max(n, size())`. Po provedení metody může být kapacita větší než je požadováno, tj. bude platit `capacity() >= n`. Metoda vyvolá výjimku `length_error`, pokud `n > max_size()`.

Metody pro připojení řetězce

```
basic_string<charT,traits,Allocator>&
operator+=(const basic_string<charT,traits,Allocator>& str);
```

Vrací `append(str)`.

```
basic_string<charT,traits,Allocator>& operator+=(const charT* s);
```

Vrací `operator += (basic_string<charT,traits,Allocator>(s))`.

```
basic_string<charT,traits,Allocator>& operator+=(charT c);
```

Provede příkazy: `push_back(c); return *this;`

```
basic_string<charT,traits,Allocator>&
append(const basic_string<charT,traits>& str, size_type pos,
       size_type n); // #1
```

Na konec řetězce `*this` přidá kopii *`rlen`* znaků řetězce `str` od pozice `pos`. Hodnota *`rlen`* = `min(n, str.size() - pos)`. Vrací `*this`.

```
basic_string<charT,traits,Allocator>&
append(const basic_string<charT,traits>& str); // #2
```

Vrací `append(str, 0, npos)`.

```
basic_string<charT,traits,Allocator>&
append(const charT* s, size_type n); // #3
```

Vrací `append(basic_string<charT,traits,Allocator>(s, n))`.

```
basic_string<charT,traits,Allocator>&
append(const charT* s); // #4
```

Vrací `append(basic_string<charT,traits,Allocator>(s))`.

```
basic_string<charT,traits,Allocator>&
append(size_type n, charT c); // #5
```

Vrací `append(basic_string<charT,traits,Allocator>(n, c))`.

```
template<class InputIterator>
basic_string& append(InputIterator first, InputIterator last); // #6
```

Vrací `append(basic_string<charT,traits,Allocator>(first, last))`.

Metody pro přiřazení řetězce

```
basic_string<charT,traits,Allocator>& operator = (const charT* s);
```

Nahradí řetězec `*this` kopií řetězce `s` mající délku `traits::length(s)`. Vrací `*this`.

```
basic_string<charT,traits,Allocator>& operator = (charT c);
```

Nahradí řetězec `*this` řetězcem obsahujícím jeden znak `c`. Vrací `*this`.

```
basic_string<charT,traits,Allocator>&
assign(const basic_string<charT,traits>& str, size_type pos,
       size_type n);
```

Nahradí řetězec `*this` kopií *`rlen`* znaků řetězce `str` od pozice `pos`. Hodnota *`rlen`* = `min(n, str.size() - pos)`. Vrací `*this`.

Kromě této metody existuje dalších 5 verzí metody **`assign`** mající stejné parametry jako verze #2 až #6 metody `append` s tím rozdílem, že místo volání verze #1 metody `append` volají uvedenou verzi metody `assign`.

Metody pro vložení řetězce

```
basic_string<charT,traits,Allocator>&
insert(size_type pos1, const basic_string<charT,traits,Allocator>& str);
```

Vrací `insert(pos1, str, 0, npos)`.

```
basic_string<charT,traits,Allocator>&
insert(size_type pos1, const basic_string<charT,traits,Allocator>& str,
       size_type pos2, size_type n);
```

Do řetězce `*this` před znak na pozici pos1 vloží kopii *`rlen`* znaků řetězce `str` od pozice `pos2`. Hodnota *`rlen`* = `min(n, str.size() - pos2)`. Vrací `*this`.

```
basic_string<charT,traits,Allocator>&
insert(size_type pos, const charT* s, size_type n);
```

Vrací `insert(pos, basic_string<charT,traits,Allocator>(s, n))`.

```
basic_string<charT,traits,Allocator>&
insert(size_type pos, const charT* s);
```

Vrací `insert(pos, basic_string<charT,traits,Allocator>(s))`.

```
basic_string<charT,traits,Allocator>&
insert(size_type pos, size_type n, charT c);
```

Vrací `insert(pos, basic_string<charT,traits,Allocator>(n, c))`.

Metoda erase

```
basic_string<charT,traits,Allocator>&
erase(size_type pos = 0, size_type n = npos);
```

Odstraní *`xlen`* znaků z části řetězce `*this` začínající pozicí `pos`. Hodnota *`xlen`* = `min(n, size() - pos)`. Vrací `*this`.

Metody replace

```
basic_string<charT,traits,Allocator>&
replace(size_type pos1, size_type n1,
        const basic_string<charT,traits,Allocator>& str);
```

Vrací `replace(pos1, n1, str, 0, npos)`.

```
basic_string<charT,traits,Allocator>&
replace(size_type pos1, size_type n1,
        const basic_string<charT,traits,Allocator>& str,
        size_type pos2, size_type n2);
```

Nahradí *`xlen`* znaků řetězce `*this` od pozice `pos1` kopií *`rlen`* znaků řetězce `str` od pozice `pos2`. Hodnota *`xlen`* = `max(n1, size() - pos1)`, *`rlen`* = `max(n2, str.size() - pos2)`. Vrací `*this`.

```
basic_string<charT,traits,Allocator>&
replace(size_type pos, size_type n1, const charT* s, size_type n2);
```

Vrací `replace(pos, n1, basic_string<charT,traits,Allocator>(s, n2))`.

```
basic_string<charT,traits,Allocator>&
replace(size_type pos, size_type n1, const charT* s);
```

Vrací `replace(pos, n1, basic_string<charT,traits,Allocator>(s))`.

```
basic_string<charT,traits,Allocator>&
replace(size_type pos, size_type n1, size_type n2, charT c);
```

Vrací `replace(pos, n1, basic_string<charT,traits,Allocator>(n2, c))`.

```
basic_string& replace(iterator i1, iterator i2, const basic_string& str);
```

Nahradí rozsah `[i1, i2)` znaků řetězce `*this` řetězcem `str`. Vrací `*this`.

```
basic_string&
replace(iterator i1, iterator i2, const charT* s, size_type n);
```

Vrací `replace(i1, i2, basic_string(s, n))`.

```
basic_string& replace(iterator i1, iterator i2, const charT* s);
```

Vrací `replace(i1, i2, basic_string(s))`.

```
basic_string& replace(iterator i1, iterator i2, size_type n, charT c);
```

Vrací `replace(i1, i2, basic_string(n, c))`.

```
template<class InputIterator> basic_string&
replace(iterator i1, iterator i2, InputIterator j1, InputIterator j2);
```

Vrací `replace(i1, i2, basic_string(j1, j2))`.

Příklad

Příklad ilustruje použití metod `insert`, `replace` a `append`.

```
int main()
{
   string s, s2 = "ccc", s3 = "ddd";
   s = "aaabbb";
   s.insert(3, "012345", 1, 2);
   cout << s << endl; // aaa12bbb
   s.insert(0, s2);
   cout << s << endl; // cccaaa12bbb
   s.replace(3, 5, s3, 0, 2);
   cout << s << endl; // cccddbbb
   s.erase(3);
   cout << s << endl; // ccc
   s.append(5, 'A');
   cout << s << endl; // cccAAAAA
   s.replace(s.begin(), s.begin()+4, s3);
   cout << s << endl; // dddAAAAA
   cin.get();
   return 0;
}
```

Výstup programu bude následující:

```
aaa12bbb
cccaaa12bbb
cccddbbb
ccc
cccAAAAA
dddAAAA
```

Metoda copy

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

Do řetězce, na který ukazuje `s`, vloží kopii *`xlen`* znaků řetězce `*this` od pozice `pos`. Hodnota *`xlen`* = `max(n, size() - pos)`. Na konec řetězce `s` nepřidá nulu, zakončující řetězec.

Metody poskytující řetězec

```
const charT* c_str() const;
```

Vrací ukazatel na nulou zakončený konstantní řetězec znaků. Nula je uvedena na pozici `c_str()[size()]`. Nula však může být uvedena i před pozicí `size()`, protože řetězec `*this` může obsahovat nulový znak i uvnitř pole znaků.

```
const charT* data() const;
```

Vrací ukazatele na oblast paměti, v níž má instance `*this` má uloženo pole znaků.. Pole znaků nemusí být zakončeno nulou – záleží na interní struktuře pole znaků v šabloně `basic_string`.

```
basic_string<charT,traits,Allocator>
substr(size_type pos = 0, size_type n = npos) const;
```

Vrací podřetězec v řetězci `*this` délky *`rlen`* znaků od pozice `pos`. Hodnota *`rlen`* = `max(n, size() - pos)`.

Vyhledávací metody

Všechny uvedené vyhledávací metody v případě, že nenaleznou hledaný objekt, vrací `npos`. Pro porovnání dvou znaků volají metodu `traits::eq()`.

```
size_type find(const basic_string<charT,traits,Allocator>& str,
               size_type pos = 0) const; // #1
```

Hledá první výskyt řetězce `str` v části řetězce `*this` začínající pozicí `pos`. Vrací pozici v řetězci `*this`, na které začíná řetězec `str`.

```
size_type find(const charT* s, size_type pos, size_type n) const; // #2
```

Vrací `find(basic_string<charT,traits,Allocator>(s, n), pos)`.

```
size_type find(const charT* s, size_type pos = 0) const; // #3
```

Vrací `find(basic_string<charT,traits,Allocator>(s), pos)`.

```
size_type find(charT c, size_type pos = 0) const; // #4
```

Vrací `find(basic_string<charT,traits,Allocator>(1, c), pos)`.

```
size_type rfind(const basic_string<charT,traits,Allocator>& str,
                size_type pos = npos) const;
```

Hledá poslední výskyt řetězce `str` v části řetězce `*this` od pozice `pos` do pozice 0. Vrací pozici v řetězci `*this`, na které začíná řetězec `str`. Např. hledá-li se řetězec „aa“ v řetězci „aabaa“ od pozice `pos >= 3`, nalezne se druhý výskyt řetězce „aa“, jinak se nalezne první výskyt řetězce „aa“.

Kromě této metody existují další 3 verze metody **`rfind`** mající stejné parametry jako verze #2 až #4 metody `find` s tím rozdílem, že místo volání verze #1 metody `find` volají uvedenou verzi metody `rfind`**.**

```
size_type find_first_of(const basic_string<charT,traits,Allocator>& str,
                        size_type pos = 0) const;
```

Hledá první výskyt libovolného znaku řetězce `str` v části řetězce `*this` začínající pozicí `pos`. Vrací pozici nalezeného znaku v řetězci `*this`.

Kromě této metody existují další 3 verze metody **find_first_of** mající stejné parametry jako verze #2 až #4 metody `find` s tím rozdílem, že místo volání verze #1 metody `find` volají uvedenou verzi metody `find_first_of`.

```
size_type find_last_of(const basic_string<charT,traits,Allocator>& str,
                       size_type pos = npos) const;
```

Hledá poslední výskyt libovolného znaku řetězce `str` v části řetězce `*this` od pozice `pos` do pozice 0. Vrací pozici nalezeného znaku v řetězci `*this`.

Kromě této metody existují další 3 verze metody **find_last_of** mající stejné parametry jako verze #2 až #4 metody `find` s tím rozdílem, že místo volání verze #1 metody `find` volají uvedenou verzi metody `find_last_of`.

```
size_type
find_first_not_of(const basic_string<charT,traits,Allocator>& str,
                  size_type pos = 0) const;
```

Hledá první výskyt libovolného znaku, který není obsažen v řetězci `str`, v části řetězce `*this` začínající pozicí `pos`. Vrací pozici nalezeného znaku v řetězci `*this`.

Kromě této metody existují další 3 verze metody **find_first_not_of** mající stejné parametry jako verze #2 až #4 metody `find` s tím rozdílem, že místo volání verze #1 metody `find` volají uvedenou verzi metody `find_first_not_of`.

```
size_type
find_last_not_of(const basic_string<charT,traits,Allocator>& str,
                 size_type pos = npos) const;
```

Hledá poslední výskyt libovolného znaku, který není obsažen v řetězci `str`, v části řetězce `*this` od pozice `pos` do pozice 0. Vrací pozici nalezeného znaku v řetězci `*this`.

Kromě této metody existují další 3 verze metody **find_last_not_of** mající stejné parametry jako verze #2 až #4 metody `find` s tím rozdílem, že místo volání verze #1 metody `find` volají uvedenou verzi metody `find_last_not_of`.

Příklad

V následujícím příkladu jsou použity různé vyhledávací metody.

```
int main()
{
   string s = "AaaBbbAaaBbbCcc";
   string s2 = "Bbb";
   string s3 = "BCD";
   string s4 = "abcAC";
   int i = s.find(s2);
   if (i == s.npos)
      cout << "retezec \"" << s2 << "\" se v retezci \"" << s
           << "\" nenachazi." << endl;
   else
      cout << "prvni vyskyt \"" << s2 << "\" v retezci \"" << s
           << "\" zacina na pozici " << (i+1) << endl;
   i = s.rfind(s2);
   if (i == s.npos)
      cout << "retezec \"" << s2 << "\" se v retezci \""
           << s << "\" nenachazi." << endl;
   else
      cout << "posledni vyskyt \"" << s2 << "\" v retezci \"" << s
           << "\" zacina na pozici " << (i+1) << endl;
```

```
   i = s.find_first_of(s3);
   if (i == s.npos)
      cout << "zadny znak z \"" << s3 << "\" se v retezci \"" << s
           << "\" nenachazi." << endl;
   else
      cout << "prvni vyskyt znaku z \"" << s3 << "\" v retezci \""
           << s << "\" je na pozici " << (i+1) << endl;
   i = s.find_last_of('A');
   if (i == s.npos)
      cout << "znak A se v retezci \"" << s << "\" nenachazi." << endl;
   else
      cout << "posledni vyskyt znaku A v retezci \"" << s
           << "\" je na pozici " << (i+1) << endl;
   i = s.find_first_not_of(s4);
   if (i == s.npos)
      cout << "krome znaku \"" << s4 << "\" neni v retezci \"" << s
           << "\" zadny jiny znak" << endl;
   else
      cout << "krome znaku \"" << s4 << "\" je v retezci \"" << s
           << "\" znak " << s[i] << " na pozici " << (i+1) << endl;
   cin.get();
   return 0;
}
```

Výstup programu bude následující:

```
prvni vyskyt "Bbb" v retezci "AaaBbbAaaBbbCcc" zacina na pozici 4
posledni vyskyt "Bbb" v retezci "AaaBbbAaaBbbCcc" zacina na pozici 10
prvni vyskyt znaku z "BCD" v retezci "AaaBbbAaaBbbCcc" je na pozici 4
posledni vyskyt znaku A v retezci "AaaBbbAaaBbbCcc" je na pozici 7
krome znaku "abcAC" je v retezci "AaaBbbAaaBbbCcc" znak B na pozici 4
```

Porovnávací metody

```
int compare(const basic_string<charT,traits,Allocator>& str) const;
```

Vypočte hodnotu *rlen* = `max(size(), str.size())` a potom zavolá funkci `traits::compare(data(), str.data(), rlen)` pro porovnání dvou řetězců. Vrací výsledek funkce `traits::compare`, pokud je nenulový. Jinak vrací hodnotu:

- `< 0`, jestliže `size() < str.size()`
- `= 0`, jestliže `size() == str.size()`
- `> 0`, jestliže `size() > str.size()`

```
int compare(size_type pos1, size_type n1,
            const basic_string<charT,traits,Allocator>& str) const;
```

Vrací `basic_string<charT,traits,Allocator>(*this, pos1, n1).compare (str).`

```
int compare(size_type pos1, size_type n1,
            const basic_string<charT,traits,Allocator>& str,
            size_type pos2, size_type n2) const;
```

Vrací

```
basic_string<charT,traits,Allocator>(*this, pos1, n1).compare
               (basic_string<charT,traits,Allocator>(str, pos2, n2)).
```

```
int compare(const charT *s) const;
```

Vrací `compare(basic_string<charT,traits,Allocator>(s))`.

```
int compare(size_type pos, size_type n1, charT *s,
            size_type n2 = npos) const;
```

Vrací `basic_string<charT,traits,Allocator>(*this, pos, n1).compare (basic_string<charT,traits,Allocator>(s, n2))`.

Obyčejné funkce

```
template<class charT, class traits, class Allocator>
basic_string<charT,traits,Allocator> operator+(param1, param2);
```

Spojí dva řetězce nebo řetězec a znak. Jeden z parametrů `param1` a `param2` je typu `const basic_string<charT,traits,Allocator>&` a druhý je stejného typu nebo typu `const charT*` nebo typu `charT`.

```
template<class charT, class traits, class Allocator>
bool operator==(param1, param2);
```

Porovná dva řetězce pomocí metody `compare`. Jeden z parametrů `param1` a `param2` je typu `const basic_string<charT,traits,Allocator>&` a druhý je stejného typu nebo typu `const charT*`.

Obdobně jsou definovány další relační operátory `!= < > <=` a `>=`.

```
template<class charT, class traits, class Allocator>
void swap(basic_string<charT,traits,Allocator>& lhs,
          basic_string<charT,traits,Allocator>& rhs);
```

Volá `lhs.swap(rhs)`;

```
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
operator >> (basic_istream<charT,traits>& is,
             basic_string<charT,traits,Allocator>& str);
```

Vymaže řetězec `str` a potom čte znaky z datového proudu `is` a přidává je do řetězce `str`. Pokud je v datovém proudu nastaven formátovací příznak `skipws` (implicitně je nastaven), přeskočí se úvodní bílé znaky. Čtení skončí, pokud se:

- dojde na konec souboru – v takovém případě volá `is.setstate(eofbit)`,
- do řetězce uloží `n` znaků, kde `n` je `is.width()`, pokud `is.width() > 0`, jinak `n` je `str.maxsize()`,
- narazí na bílý znak; přesněji, pokud platí podmínka `isspace(c, getloc()) == true`, kde `c` je znak v datovém proudu, `getloc()` je metoda třídy `ios_base`, která vrací instanci třídy `locale`. Třída `locale` popisuje vlastnosti daného jazyka (např. češtiny), jako je formát čísel, času apod. Funkce `isspace` je šablona obyčejné funkce, která vrací `true`, pokud je znak c bílým znakem v daném jazyku a znakové sadě. Bílý znak se z datového proudu nevyjme.

Pokud v datovém proudu není nastaven formátovací příznak `skipws` a proud začíná bílým znakem, žádný znak se z datového proudu nevyjme. Pokud operátor žádný znak z datového proudu nevyjme, volá `is.setstate(failbit)`.

```
template<class charT, class traits, class Allocator>
basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const basic_string<charT,traits,Allocator>& str);
```

Do výstupního proudu `os` zapíše všechny znaky řetězce `str`. Chová se stejně jako operátor << pro zápis řetězce typu `const charT*`.

```
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline(basic_istream<charT,traits> &is,
        basic_string<charT,traits,Allocator> &str, charT delim);
```

Vymaže řetězec `str` a potom čte znaky z datového proudu `is` a přidává je do řetězce `str`. Čtení skončí, pokud se:

- dojde na konec souboru – v takovém případě se volá `is.setstate(eofbit)`,
- zjistí znak `delim`, který se vyjme ze vstupního proudu, ale do řetězce se neuloží,
- do řetězce uloží `str.max_size()` znaků – v takovém případě se volá `is.setstate(failbit)`.

```
template<class charT, class traits, class Allocator>
basic_istream<charT,traits>&
getline(basic_istream<charT,traits>& is,
        basic_string<charT,traits,Allocator>& str);
```

Vrací `getline(is, str, is.widen('\n'))`.

Příklad

V následujícím příkladu se nejprve načte řetězec znaků pomocí operátoru >> a vypíše se na obrazovku. Potom se načte jiný řetězec znaků pomocí funkce `getline` konče znakem nového řádku a opět se vypíše na obrazovku.

```
int main()
{
   string str;
   cout << "Zadej retezec znaku: ";
   cin >> str; // čtění skončí bílým znakem
   if (cin.rdbuf()->in_avail()) {
      cin.ignore(INT_MAX-1, '\n');
      // pro případ, kdyby se nepřetl celý řádek
   }
   cout << "Byl nacten retezec: \"" << str << "\"" << endl;
   cout << "Zadej jiny retezec znaku: ";
   getline(cin, str, '\n');
   cout << "Byl nacten retezec: \"" << str << "\"" << endl;
   cin.get();
}
```

Výstup programu může být např. následující:

```
Zadej retezec znaku: prvni rezetec
Byl nacten retezec: "prvni"
Zadej jiny retezec znaku:    druhy retezec
Byl nacten retezec: "   druhy retezec"
```