Vstupno/výstupné prúdy

S obyčajným vstupom a výstupom sa dá robiť omnoho viac než len pretransformovať štandardnú knižnicu vstupno/výstupných funkcií C jazyka do tried.

Nebolo by to praktické, keby sme sa mohli na všetky bežné "schránky" – napríklad štandardný vstup/výstup, súbory a dokonca i na bloky pamäte – pozerať rovnako, aby sme si mohli pamätať iba jedno rozhranie? To je myšlienka, ktorá stojí v pozadí implementácie vstupno-výstupných prúdov (streams). Sú jednoduchšie, bezpečnejšie a občas dokonca i výkonnejšie než rôznorodé funkcie zo štandardnej **stdio** knižnice.

Prúdové triedy sú zvyčajne prvou časťou C++ knižnice, ktorú sa C++ začiatočníci naučia používať. V nasledujúcom si povieme o čo sú prúdy lepšie než nástroje štandardnej C knižnice. Okrem štandardných konzolových prúdov sa pozrieme i na správanie súborových a reťazcových prúdov.

Prečo prúdy?

Možno sa zdá podivné, čo je zlé na dobrej starej C knižnici. Prečo "neobaliť" C knižnicu do tried a hotovo? Samozrejme pre niektoré situácie to stačí. Predpokladajme napríklad, že sa chceme presvedčiť, či súbor, reprezentovaný smerníkom typu **FILE** je vždy bezpečne otvorený a správne zatvorený bez toho, aby sme sa spoliehali na to, že užívateľ nezabudne zavolať funkciu **close()**:

```
#ifndef FILECLASS_H
#define FILECLASS_H
#include <cstdio>
#include <stdexcept>

class cFileClass {
  std::FILE* f;
public:
  struct FileClassError : std::runtime_error {
     public:
       FileClassError(const char* msg)
         : std::runtime_error(msg) {}
  };
  cFileClass(const char* fname, const char* mode = "r");
  ~cFileClass();
  std::FILE* fp();
};

#endif
```

Keď realizujeme vstup alebo výstup v jazyku C, pracujeme so smerníkom na štruktúru typu FILE. Vyššie uvedená trieda obaľuje tento smerník a zabezpečuje, že sa korektne inicializuje a uprace prostredníctvom konštruktora a deštruktora. Druhý argument konštruktora je režim súboru, ktorý je implicitne nastavený na hodnotu "**r**" – čítanie.

Na získavanie smerníka, ktorý sa používa vo vstupno-výstupných funkciách slúži prístupová funkcia **fp()**. Definície členských funkcií vypadajú nasledovne:

```
#include "cFileClass.h"
#include <cstdlib>
#include <cstdio>
using namespace std;

cFileClass::cFileClass(const char* fname, const char* mode) {
  if((f = fopen(fname, mode)) == 0)
    throw FileClassError("Chyba otvorenia suboru");
}

cFileClass::~cFileClass() { fclose(f); }
```

```
FILE* cFileClass::fp() { return f; }
```

Konštruktor volá funkciu **fopen()**, avšak zároveň zabezpečuje, že výsledok nebude nulový, čo indikuje vyslaním výnimky - ak sa súbor neotvorí podľa očakávania, vyšle sa výnimka.

Deštruktor súbor zatvára a prístupová funkcia **fp**() vracia *popisovač súboru* **f**. Triedy **cFileClass** by sme mohli použiť nasledujúcim spôsobom:

```
#include <cstdlib>
#include <iostream>
#include "cFileClass.h"
using namespace std;

int main() {
  try {
    cFileClass f("test.txt");
    const int BSIZE = 100;
    char buf[BSIZE];
    while(fgets(buf, BSIZE, f.fp()))
      fputs(buf, stdout);
  }
  catch(cFileClass::FileClassError& e) {
    cout << e.what() << endl;
    return EXIT_CHYBA;
  }
  return EXIT_USPECH;
}
```

Vytvoríme objekt triedy **cFileClass** a použijeme ho v normálnych vstupno/výstupných funkciách jazyka C volaním členskej funkcie **fp()**. Keď skončíme, jednoducho naň zabudneme, súbor automaticky zatvorí kód deštruktora na konci rozsahu platnosti objektu.

I keď smerník typu **FILE** je **private**, tento smerník nie je bezpečný, pretože sa sprístupňuje členskou funkciou **fp()**. Pretože jediným efektom tejto triedy je inicializácia a upratovanie prečo ho neurobiť **public** alebo namiesto **class** použiť **struct**? Všimnime si, že zatiaľ čo pomocou funkcie **fp()** získavame dátový člen **f**, priraďovať do **f** nemôžeme - toto plne riadi trieda. Samozrejme po získaní smerníka prostredníctvom členskej funkcie **fp()** môže klientský programátor priraďovať do prvkov štruktúry typu **FILE** alebo dokonca môže súbor zatvoriť, takže trieda zabezpečuje jedine platnosť smerníka typu **FILE** a nie správny obsah štruktúry.

Ak by sme chceli zabezpečiť úplnú bezpečnosť, musíme užívateľovi triedy zabrániť, aby mal priamy prístup k smerníku **FILE**. Znamenalo by to, že niektoré vstupno-výstupné funkcie sa musia stať členmi triedy **cFileClass**, a všetko čo môžeme robiť v C jazyku bude dostupné prostredníctvom C++ triedy:

```
// Zapúzdrenie V/V
#ifndef FULLWRAP_H
  #define FULLWRAP_H

class File {
     std::FILE* f;
     std::FILE* F(); // Dáva kontrolovaný smerník na f
   public:
     File(); // Vytvor objekt ale neotváraj súbor
     File(const char* path, const char* mode = "r");
     ~File();
     int open(const char* path, const char* mode = "r");
     int reopen(const char* path, const char* mode);
     int getc();
```

```
        int ungetc(int c);
        int putc(int c);
        int puts(const char* s);
        char* gets(char* s, int n);
        int printf(const char* format, ...);
        size_t read(void* ptr, size_t size, size_t n);
        size_t write(const void* ptr, size_t size, size_t n);
        int eof();
        int close();
        int flush();
        int seek(long offset, int whence);
        int getpos(fpos_t* pos);
        int setpos(const fpos_t* pos);
        long tell();
        void rewind();
        void setbuf(char* buf);
        int setvbuf(char* buf, int type, size_t sz);
        int error();
        void clearErr();
};
#endif // FULLWRAP_H
```

Trieda obsahuje takmer všetky súborové vstupno/výstupné funkcie zo súboru **<cstdio>**.

Trieda **File** má taký istý konštruktor ako trieda v predchádzajúcom príklade, ale tiež má i implicitný konštruktor. Implicitný konštruktor je dôležitý, ak chceme vytvárať pole objektov triedy **File** alebo použiť objekt triedy **File** ako člen inej triedy, kedy sa inicializácia nevykonáva v konštruktore, ale niekedy po vytvorení obaľujúceho objektu.

Implicitný konštruktor nastavuje privátny smerník typu **FILE** na nulu. Avšak teraz pri akomkoľvek odkaze na **f** sa musí jeho hodnota testovať, aby sa zabezpečilo, že nie je nulová. toto je zabezpečené prostredníctvom funkcie **F**, ktorá je privátna, pretože je určená len pre ďalšie členské funkcie (užívateľovi nechceme poskytnúť priamy prístup ku štruktúre **FILE** z triedy.)

Takýto prístup nie je zlým riešením. Je celkom funkčný a vedeli by sme si predstaviť vytvorenie podobných tried pre štandardný (konzolový) vstup/výstup a interné formátovanie (čítanie/zápis do pamäti namiesto do súboru alebo na konzolu).

Veľkým problémom je interpretátor, použitý pre funkcie s variabilným počtom parametrov. Toto je kód, ktorý analyzuje formátovací reťazec za chodu programu a vyberá a interpretuje argumenty z premenlivého zoznamu argumentov. Tento problém má štyri dôvody:

1. I keď použijeme iba zlomok funkčnosti interpretátora, celá vec sa zavedie do vykonateľného kódu. Takže ak napíšeme **printf("%c",'x');**, dostaneme celý balík, vrátane častí, ktoré tlačia čísla v pohyblivej rádovej čiarke a reťazce. Neexistuje štandardná voľba na redukciu veľkosti priestoru, použitého programom.

2. Pretože interpretácia nastáva za chodu programu, nemôžeme sa zbaviť výkonnostnej réžie. To je frustrujúce, pretože všetky informácie sú už počas kompilácie tam vo formátovacom reťazci, ale vyhodnocujú sa až pri vykonávaní programu. Avšak ak by sme analyzovali argumenty vo formátovacom reťazci už počas kompilácie, mohli by sme priamo volať funkcie, ktoré by mohli byť potencionálne omnoho rýchlejšie, než interpretátor za chodu programu (hoci rodina funkcií **printf** je zvyčajne celkom dobre optimalizovaná).

3. Ešte väčším problémom je, že formátovací reťazec sa vyhodnocuje až počas chodu programu: neexistuje kontrola chýb kompilácie. C++ venuje veľkú časť kompilácie kontrolám chýb, aby sa chyby našli včas a tým sa uľahčil život programátora. Bolo by zlé vyhodiť typovú bezpečnosť zo vstupno/výstupnej knižnice, najmä z dôvodu, že vstup/výstup sa používa veľmi často.

4. Kritickým problémom v C++ je, že skupina funkcií **printf** nie je rozširovateľná. Funkcie sú navrhnuté na spracovanie štyroch základných typov jazyka C (**char, int, float, double, wchar_t, char *** a **void ***) a ich variácii. Mohlo by sa zdať, že po každom pridaní novej triedy by sme mohli pridať preťažené funkcie **printf** a **scanf** (a ich varianty pre súbory a reťazce), ale nezabúdajme, preťažené funkcie sa musia líšiť typom svojich argumentov a skupina funkcií **printf** skrýva typové informácie vo formátovacom reťazci a v premenlivom zozname argumentov. Pre jazyk ako je C++, ktorého cieľom je byť schopný jednoducho doplňovať nové dátové typy je toto nepríjemné obmedzenie.

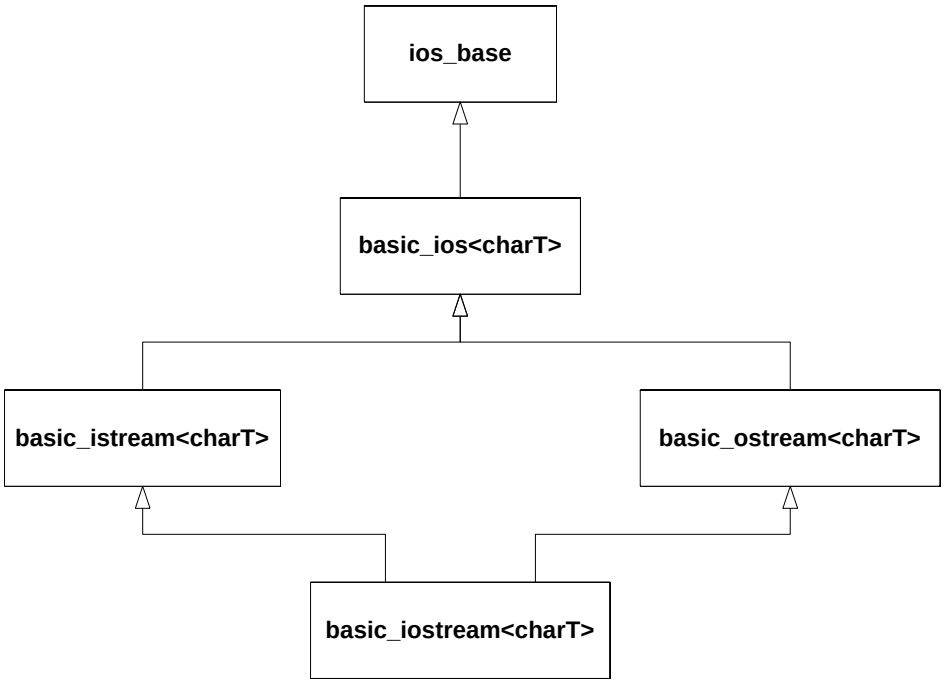
Prúdy ako záchrana

Všetky tieto vyššie uvedené úvahy viedli k tomu, že jednou z prvých priorít štandardných C++ knižníc tried by malo byť spracovanie vstupu/výstupu. Pretože "Ahoj svet" je prvým programom v každom novom jazyku a pretože vstup/výstup je súčasťou prakticky každého programu, vstupno/výstupná knižnica v C++ sa musí sať používať jednoducho. Ešte náročnejšou úlohou je, že musí uspokojiť ľubovoľnú novú triedu.

Vkladač a vyberač

Prúd (*stream*) je objekt, ktorý transportuje a formátuje znaky s pevnou šírkou. Vytvárať môžeme ***vstupný prúd*** (potomkovia triedy **istream**), ***výstupný prúd*** (objekty triedy **ostream**) alebo prúdy, ktoré umožňujú vykonávať obidve činnosti súčasne (objekty, odvodené z triedy **iostream**). Knižnica prúdov **iostream** poskytuje rôzne typy takýchto tried: **ifstream**, **ofstream** a **fstream** pre súbory a **istringstream**, **ostringstream** a **stringstream** ako rozhranie pre štandardnú C++ triedu **string**. Všetky tieto prúdové triedy majú takmer identické rozhrania, takže prúdy môžeme používať jednotným spôsobom, či už pracujeme so súbormi, štandardným vstupom/výstupom, oblasťou pamäti alebo objektom triedy **string**. Jediné rozhranie, ktoré sa naučíme, bude fungovať i pre rozšírenia, doplnené ako podpora nových tried. Niektoré funkcie implementujú formátovacie príkazy a niektoré funkcie čítajú a zapisujú znaky bez formátovania.

Vyššie spomínané prúdové triedy sú v skutočnosti špecializácie šablón, podobne ako je trieda **string** špecializáciou šablóny **basic_string**. Základné triedy **iostream** hierarchie dedičnosti sú uvedené na nasledujúcom obrázku:



Trieda **ios_base** deklaruje všetko spoločné pre všetky prúdy bez ohľadu na typ znaku, ktorý prúd spracováva. Týmito deklaráciami sú väčšinou konštanty a funkcie na ich spracovanie. Zvyšok tried tvoria šablóny, ktorých parametrom je typ. Trieda **istream** je napríklad definovaná takto.

```
typedef basic_istream<char> istream;
```

Všetky vyššie spomínané triedy sú definované podobnými typovými definíciami. Existujú tiež definície typové pre všetky prúdové triedy použitím **w_char** (typ pre široký znak) namiesto **char**. Základná šablóna **basic_ios** definuje funkcie spoločné pre vstup i výstup, ale závisí to od znakového typu. Šablóna **basic_istream** definuje generické funkcie pre vstup a trieda **basic_ostream** robí to isté pre výstup. Triedy pre súborové a reťazcové prúdy doplňujú funkčnosť, špecifickú pre ich typy prúdov.

V iostream knižnici sú kvôli zjednodušeniu použitia prúdov preťažené dva operátory. Operátor **<<** sa v súvislosti s prúdmi často označuje ***vkladač*** a operátor **>>** zasa ***vyberač*** (***extraktor***).

Extraktory analyzujú informácie, ktoré cieľový objekt očakáva na základe ich typu. Ako príklad použijeme objekt **cin**, ktorý je prúdovým ekvivalentom **stdin** v jazyku C, t.j. presmerovateľný štandardný vstup. Objekt je preddefinovaný všade tam, kde včleníme hlavičkový súbor **<iostream>**.

```
int i;
cin >> i;

float f;
cin >> f;

char c;
cin >> c;

char buf[100];
cin >> buf;
```

Preťažený operátor **>>** existuje pre každý zabudovaný typ. Ako uvidíme neskôr tento operátor môžeme preťažiť i vo vlastných typoch.

Ak potrebujeme zistiť obsah rozmanitých premenných, môžeme použiť objekt **cout** (zodpovedajúci štandardnému výstupu; existuje tiež objekt **cerr**, zodpovedajúci štandardnému chybovému výstupu) spolu s vkladacím operátorom:

```
cout << "i = ";
cout << i;
cout << "\n";
cout << "f = ";
cout << f;
cout << "\n";
cout << "c = ";
cout << c;
cout << "\n";
cout << "buf = ";
cout << buf;
cout << "\n";
```

Toto vypadá dosť zdĺhavé a nevypadá to ako vylepšenie funkcie **printf** napriek vylepšenej kontrole typu. Našťastie preťažené vkladače a extraktory sú navrhnuté tak, že sa dajú reťaziť do zložitejších výrazov, ktoré sa zapisujú (a čítajú) ľahšie:

```
cout << "i = " << i << endl;
cout << "f = " << f << endl;
cout << "c = " << c << endl;
cout << "buf = " << buf << endl;
```

Definovanie vkladacích a vyberacích operátorov pre vlastné triedy je iba záležitosťou preťažovania príslušných operátorov, pričom môžeme postupovať podľa nasledovného postupu:

- Prvým parametrom je nekonštantný odkaz na prúd (**istream** pre vstup, **ostream** pre výstup)
- Vykonáme operáciu vložením/vybratím dát do/z prúdu (samozrejme prostredníctvom spracovania prvkov objektu)
- Vrátime odkaz na prúd

Prúd nesmie byť konštantný, pretože spracovanie prúdových dát mení stav prúdu. Návratom prúdu umožníme reťazenie prúdových operácií v jedinom príkaze (pozri vyššie).

Na ilustráciu si rozoberme ako by sme dali na výstup reprezentáciu objektu triedy **Datum** v tvare DD-MM-RRRR. Túto činnosť realizuje nasledovný vkladací operátor:

```
ostream& operator<<(ostream& os, const Datum& d) {
  char fillc = os.fill('0');
  os << setw(2) << d.dajDen() << '-'
     << setw(2) << d.dajMesiac() << '-'
     << setw(4) << setfill(fillc) << d.dajRok();
  return os;
}
```

Táto funkcia nemôže byť členom triedy **Datum**, pretože ľavým operandom operátora **<<** musí byť výstupný prúd. Členská funkcia **fill()** triedy **ostream** mení vyplňujúci znak, ktorý sa použije, keď šírka výstupnej položky, určená ***manipulátorom*** **setw**, je väčšia, než dáta vyžadujú. Použili sme znak '0', takže mesiace pred októbrom budú zobrazované s nulou, napríklad september sa zobrazí ako "09". Funkcia **fill()** vracia predchádzajúci vyplňujúci znak (ktorým je implicitne medzera), takže ho neskôr môžeme obnoviť pomocou manipulátora **setfill** (o manipulátoroch neskôr).

Extraktory si vyžadujú o čosi viac pozornosti, pretože vstupné dáta sa občas pokazia. Spôsobom ako signalizovať chybu prúdu je nastavenie **fail** bitu:

```
istream& operator>>(istream& is, Datum& d) {
  is >> d.den;
  is >> pomlcka;
  if (pomlcka != '-')
    is.setstate(ios::failbit);
  is >> d.mesiac;
  char pomlcka;
  is >> pomlcka;
  if (pomlcka != '-')
    is.setstate(ios::failbit);
  is >> d.rok;
  return is;
}
```

Po nastavení chybového bitu prúdu sa všetky nasledujúce prúdové operácie ignorujú až kým prúd neobnovíme do *dobrého* stavu. Vyššie uvedený kód pokračuje v extrahovaní i keď sa bit **ios::failbit** nastaví. Táto implementácia je tak trochu zhovievavá v tom, že dovoľuje biele medzery medzi číslami a pomlčkami v dátumovom reťazci (pretože operátor >> biele medzery pri čítaní zabudovaných typov implicitne preskakuje). Platné reťazce pre tento extraktor by mohli vypadať nasledovne:

```
"10-08-2002"
"10-8-2002"
"10 - 08 - 2002"
```

ale nie takto:

```
"A-10-2002"    // Abecedné znaky nie sú povolené
"08%10/2002"   // Ako oddeľovač je povolená len pomlčka
```

O stave prúdu však podrobnejšie neskôr.

Všeobecné použitie

Ako ilustruje extraktor triedy **Datum**, pozor si musíme dávať iba na chybný vstup. Ak vstup produkuje neočakávanú hodnotu, proces sa naruší a ťažko sa zotaví. Okrem toho formátovaný vstup zanedbáva ako oddeľovač biele medzery. Pozrime sa, čo sa stane, keď pospájame všetky fragmenty kódu do jedného programu:

```
// Prúdy - príklad
#include <iostream>
using namespace std;

int main()
{
  int i;
  cin >> i;

  float f;
  cin >> f;

  char c;
  cin >> c;

  char buf[100];
  cin >> buf;

  cout << "i = " << i << endl;
  cout << "f = " << f << endl;
  cout << "c = " << c << endl;
  cout << "buf = " << buf << endl;

  cout << flush;
  cout << hex << "0x" << i << endl;
}
```

a zadáme nasledovný vstup:

```
12 1.4 c toto je moj test
```

Očakávame rovnaký výstup, aký sme zadali:

```
12
1.4
c
toto je moj test
```

ale výstup bude tak trochu neočakávaný:

```
i = 12
f = 1.4
c = c
buf = toto
0xc
```

Všimnime si, že do premennej **buf** sa načítalo iba prvé slovo, pretože vstupná rutina považuje medzeru za oddeľovač vstupov, ktorý sa nachádza za slovom **toto**. Okrem toho ak je súvislý vstupný reťazec dlhší, než pamäť, alokovaná pre **buf**, prekročíme kapacitu bufra.

V praxi v interaktívnych programoch zvyčajne berieme vstup po riadkoch ako postupnosť znakov, prezrieme ich a potom sa urobia také konverzie, ktoré budú pre bufer bezpečné. Takýmto spôsobom sa nemusíme obávať, že sa vstupná rutina zanesie neočakávanými dátami.

Ďalšou vecou, ktorú musíme brať do úvahy, je celkový koncept rozhrania príkazového riadku. Toto malo význam v minulosti, kedy konzola bola čosi viac než, písací stroj, avšak svet sa rýchlo mení a v súčasnosti dominuje grafické užívateľské rozhranie. Aký je teda význam konzolového vstupu/výstupu v súčasnosti? Konzolový vstup **cin** sa moc nepoužíva, až na jednoduché príklady alebo testy. Pre vstup sa používa nasledovný prístup:

1. Ak program vyžaduje vstup, čítajme vstup zo súboru – ako uvidíme, práca so súbormi pomocou prúdov je pozoruhodne ľahká. prúdy pre súbory fungujú s GUI[1] dobre.

2. Vstup čítajme bez pokusov konvertovať. Keď je vstup miestom, kde nemôže veci pokaziť počas konverzie, môžeme ho bezpečne snímať.

3. Výstup je iný. Ak používame GUI, **cout** nemusí fungovať a výstup musíme posielať do súboru (čo je identické s vysielaním do **cout**) alebo použiť GUI prostriedky na zobrazovanie dát. V obidvoch prípadoch sú veľmi užitočné formátovacie funkcie prúdov.

Ďalší všeobecný postup šetrí dobu kompilácie veľkých projektov. Zoberme napríklad ako by sme deklarovali prúdový operátor triedy **Datum**. Stačí nám včleniť prototypy funkcií, takže nie je skutočne nevyhnutné včleňovať celý hlavičkový súbor **<iostream>** do hlavičkového súboru **datum.h**. Štandardným postupom je triedy len deklarovať:

```
class ostream;
```

Toto je starý spôsob oddelenia rozhrania od implementácie a často sa nazýva ***predbežná deklarácia*** (trieda **ostream** by v tomto okamihu bola považovaná za ***neúplný typ***, pretože kompilátor ešte nevidel definíciu triedy).

Avšak toto by nefungovalo správne z dvoch dôvodov:

1. Prúdové triedy sú definované v menopriestore **std**.

2. Sú to šablóny.

Správna deklarácia by vypadala nasledovne:

```
namespace std {
  template<class charT, class traits = char_traits<charT> >
    class basic_ostream;
  typedef basic_ostream<char> ostream;
}
```

(Ako vidíme, podobne ako trieda **string**, prúdové triedy používajú triedy vlastností znakov). Pretože by bolo strašne zdĺhavé písať všetko toto pre každú prúdovú triedu, na ktorú sa chceme odkazovať, norma zabezpečuje hlavičkový súbor, ktorý to robí za nás: **<iosfwd>**. Hlavičkový súbor triedy **Datum** by vypadal nasledovne:

```
// Datum.h
#include <iosfwd>
class Datum {
  friend std::ostream& operator<<(std::ostream&,const Datum&);
  friend std::istream& operator>>(std::istream&, Datum&);
  // atď.
```

[1] GUI - Graphic User Interface - grafické užívateľské rozhranie

Riadkovo orientovaný vstup

Na načítanie celého riadku máme k dispozícií tri možnosti:

- Členskú funkciu **get**
- Členskú funkciu **getline**
- Globálnu funkciu **getline**, definovanú v hlavičkovom súbore **<string>**

Prvé dve funkcie majú tri argumenty:

- Smerník na znakový bufer, do ktorého sa ukladá výsledok.
- Veľkosť bufra (aby nedošlo k pretečeniu)
- Ukončovací znak, aby sme vedeli vstup zastaviť

Ukončovací znak má implicitnú hodnotu **'\n'**, čo je znak, ktorý sa bežne používa. Obidve funkcie uchovávajú nulu vo výslednom bufri, keď narazia na ukončovací znak na vstupe.

Nuž v čom je rozdiel? Malý, ale dôležitý: **get()** sa zastaví, keď vo výstupnom prúde narazí na oddeľovač, ale zo vstupného prúdu ho nevyberie. Takže ak urobíme ďalší **get()** použijúc ten istý oddeľovač, nevyberie sa žiaden vstup (predpokladajme, že v ďalšom **get()** príkaze použijeme iný oddeľovač alebo inú vstupnú funkciu). Na druhej strane funkcia **getline()**oddeľovač zo vstupného prúdu vyberá, ale neukladá ho do výsledného bufra.

Vhodná je funkcia **getline()**, definovaná v súbore **<string>**. Nie je to členská funkcia, ale iba samostatná funkcia, deklarovaná v menovom priestore **std**. Má dva argumenty bez implicitnej hodnoty – vstupný prúd a objekt triedy **string**, ktorý sa má zaplniť. Podobne ako jej menovec číta znaky, až kým nenarazí na prvý výskyt oddeľovač (implicitne **'\n'**), spotrebuje a zruší oddeľovač. Výhodou tejto funkcie je, že číta do objektu triedy **string**, takže sa nemusíme starať o veľkosť bufra.

Platí teda, že keď spracovávame textový súbor, ktorý čítame po riadkoch, môžeme použiť jednu z funkcií **getline**.

Preťažená verzia funkcie get

Funkcia **get()** má tiež tri preťažené verzie: jednu bez argumentov, ktorá vracia ďalší znak, pričom vracia hodnotu typu **int**, jednu, ktorá napĺňa znak do argumentu typu **char** použijúc odkaz, a jednu, ktorá ukladá priamo do bufrovej štruktúry ďalšieho prúdového objektu (o tejto funkcii neskôr).

Čítanie čistých bytov

Ak vieme presne s čím pracujeme a chceme presúvať byty priamo do premennej, poľa alebo pamäťovej štruktúry, môžeme použiť funkciu **read()**, ktorá je určená pre naformátovaný vstup. Prvým parametrom je smerník na cieľovú pamäť a druhým je počet bytov, ktoré sa majú prečítať. Toto je obzvlášť užitočné ak sme už predtým uložili do súboru informáciu o súbore, napríklad v binárnom tvare použijúc pre výstupný prúd komplementárnu členskú funkciu **write()** (samozrejme v tom istom kompilátore).

Ošetrenie prúdových chýb

Vyššie uvedený extraktor triedy **Datum** nastavuje za určitých podmienok bit prúdu nazvaný **fail**. Ako zistíme, že nastala takáto chyba? Chyby prúdu môžeme detekovať buď volaním určitých členských funkcií prúdu, čím zisťujeme, či nenastal chybový stav alebo ak nás nezaujíma o akú konkrétnu chybu sa jedná, stačí nám vyhodnotiť prúd v boolean kontexte. Obidva spôsoby sú odvodené od stavu chybových bitov prúdu.

Stav prúdu

Trieda **ios_base**, z ktorej je odvodená[2] trieda **ios**, definuje štyri indikátory, ktoré môžeme používať na indikovanie stavu prúdu:

[2] A preto môžeme písať `ios::failbit` namiesto `ios_base::failbit`, aby sme ušetrili písanie.

Indikátor	Význam
badbit	Nastala nejaká fatálna (snáď hardvérová) chyba. Prúd by mal byť považovaný za nepoužiteľný.
eofbit	Nastal koniec vstupu (buď narazením na fyzický koniec súborového prúdu alebo ukončenie užívateľského konzolového prúdu napríklad Ctrl-Z alebo Ctrl-D).
failbit	Zlyhala vstupno/výstupná operácia, najpravdepodobnejšie kvôli neplatným dátam (napríklad narazilo sa na písmena pri pokuse čítať číslice). Prúd je stále použiteľný, tento bit sa nastavuje i v prípade konca vstupu.
goodbit	Všetko je v poriadku, žiadne chyby. Koniec vstupu ešte nenastal.

Či niektorý z týchto stavov nastal môžeme zisťovať volaním zodpovedajúcej členskej funkcie, ktorá vracia boolean hodnotu, indikujúcu, či bol nastavená niektorý z bitov. Členská funkcia **good()** prúdu vracia hodnotu **true**, ak nie je nastavený žiaden z troch bitov. Funkcia **eof()** vracia hodnotu **true** ak je nastavený **eofbit**, ktorý sa nastaví, ak sa pokúsime čítať z prúdu, ktorý už neobsahuje žiadne dáta (zvyčajne súbor). Pretože koniec vstupu nastáva v C++ i keď sa pokúšame čítať za koncom fyzického média, nastaví sa i **failbit**, ktorý indikuje, že "očakávané" dáta sa nenačítali úspešne. Funkcia **fail()** vracia hodnotu **true** ak je nastavený buď **failbit** alebo **badbit**. Funkcia **bad()** vracia hodnotu **true** iba ak je nastavený **badbit**.

Akonáhle je niektorý z chybových bitov prúdu nastavený, zostáva nastavený, čo nie je vždy čo by sme chceli. Napríklad keď čítame súbor, mohli by sme sa chcieť premiestniť na predchádzajúce miesto v súbore pred ktorým nastal koniec súboru. Ale iba presunutie súborového smerníka automaticky neobnoví **eofbit** alebo **failbit**, musíme to urobiť explicitne volaním funkcie **clear()**:

```
mojPrud.clear();     // Obnov všetky chybové bity
```

Po zavolaní funkcie **clear()** funkcia **good()** vráti hodnotu **true**, ak ju ihneď zavoláme. Ako sme už videli pri extraktore triedy **Datum**, funkcia **setstate()** nastavuje bity, ktoré do nej pošleme. Funkcia **setstate()** neovplyvní žiadne iné bity – ak sú úž nastavené, zostanú nastavené. Ak chceme nastaviť určité bity a zároveň vynulovať zvyšok, môžeme zavolať preťaženú verziu funkcie **clear()**, do ktorej pošleme bitový výraz, reprezentujúci bity, ktoré chceme nastaviť.

```
mojPrud.clear(ios::failbit | ios::eofbit);
```

Vo väčšine prípadov nás kontrola jednotlivých stavových bitov prúdu nezaujíma. Zvyčajne chceme iba vedieť, či je všetko v poriadku. Toto je i prípad pri čítaní súboru od začiatku po koniec. Akurát chceme vedieť, kedy sa vstupné dáta vyčerpali. Pre takéto prípady je definovaný konverzný operátor pre typ **void ***, ktorý sa automaticky zavolá, keď sa prúd vyskytne v booleovskom výraze. Na čítanie až do konca vstupu pomocou prúdu môžeme použiť nasledovný fragment kódu:

```
int i;
while (mojPrud >> i)
  cout << i << endl;
```

Nezabúdajme, že operátor >> vracia svoj prúdový argument, takže príkaz **while** testuje prúd ako booleovský výraz. Tento konkrétny príklad predpokladá, že vstupný prúd **mojPrud** obsahuje celé čísla, oddelené bielou medzerou. Funkcia **ios_base::operator void *()** jednoducho volá funkciu **good()** svojho prúdu a vracia výsledok. Pretože väčšina prúdových operácií vracia svoje prúdy[3], použitie tohto fragmentu kódu je výhodné.

Prúdy a výnimky

Prúdy boli súčasťou C++ dlho pred výnimkami, a preto manuálna kontrola stavu prúdu bola jediným spôsobom, ako to urobiť. Kvôli zachovaniu spätnej kompatibility toto stále platí, ale prúdy môžu vysielať i výnimky. Členská funkcia **exceptions()** prúdu má parameter, reprezentujúci stavové bity, pre ktoré chceme

[3] Je obvyklé použiť **operator void*( )** namiesto operator **bool()**, pretože implicitné konverzie z **bool** na **int** môžu spôsobiť prekvapenie v prípade, že prúd náhodne vložíme do kontextu, kde sa aplikuje celočíselná konverzia. Funkcia **operator void*()** sa zavolá implicitne v tele **boolean** výrazu.

výnimky vysielať. Vždy, keď prúd narazí na takýto stav, vyšle výnimku typu **std::ios_base::failure**, ktorá je potomkom triedy **std::exception**.

Hoci môžeme zapínať chybovú výnimku pre ľubovoľný zo štyroch stavov prúdu, nie je zvyčajne dobré povoliť výnimky pre všetky z nich. Výnimky by sa mali vysielať pre skutočne chybové stavy, ale napríklad koniec súboru nie je nič výnimočné – je to dokonca ***očakávaný*** stav. A preto by sme mali povoľovať výnimky len pre chyby, reprezentované bitom **badbit**, čo urobíme nasledovne.

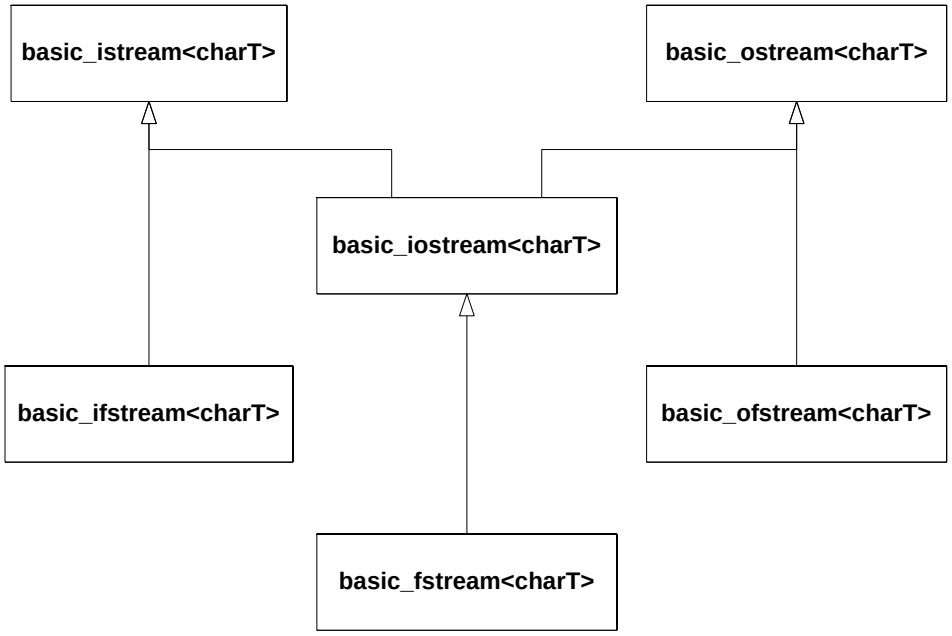
```
mojPrud.exceptions(ios::badbit);
```

Výnimky povoľujeme pre každý prúd zvlášť, pretože **exceptions()** je členská funkcia prúdov. Funkcia **exceptions()** vracia bitovú masku[4] (typu **iostate**, čo je nejaký kompilátorovo-závislý typ, premeniteľný na **int**) indikujúcu, ktoré stavy prúdu spôsobia vyslanie výnimky. Ak tieto stavy už boli nastavené, výnimka sa vysiela okamžite. Samozrejme ak použijeme výnimky v spojení s prúdmi, musíme byť pripravení zachytávať ich, čo znamená, že potrebujeme obaliť všetky prúdové spracovania **try** blokom, ktorý bude obsahovať ošetrenie výnimky **ios::failure**. Mnoho programátorov považuje toto za zdĺhavé a stav iba kontroluje manuálne tam, kde očakávajú výskyt chyby (pretože očakávajú, že funkcia **bad()** vo väčšine prípadov nevráti hodnotu **true**). Toto je i ďalší dôvod, prečo vysielanie výnimiek prúdmi je voliteľné a nie je nastavené implicitne. V každom prípade si môžeme vybrať ako chceme chyby prúdov ošetrovať.

Súborové prúdy

Manipulácia so súbormi pomocou prúdov je omnoho jednoduchšia a bezpečnejšia, než použitie knižnice **stdio** v C jazyku. Na otvorenie súboru stačí vytvoriť objekt, konštruktor urobí všetku prácu za nás. Súbor nemusíme explicitne zatvárať (hoci môžeme pomocou členskej funkcie **close**), pretože to urobí deštruktor, pri zániku objektu. Ak chceme vytvoriť súbor, ktorý reprezentuje iba vstup, vytvoríme objekt triedy **ifstream**. Ak chceme vytvoriť výstupný súbor, vytvoríme objekt triedy **ofstream**. Objekt triedy **fstream** je určený pre vstup i výstup.

Triedy súborových prúdov zapadajú medzi **iostream** triedy podľa nasledovného obrázku.



obdobne ako predtým, skutočne použité triedy sú špecializáciami šablón, definované typovými definíciami. Napríklad `ifstream`, ktorý spracováva súbory znakov (`char`), je definovaný nasledovne:

[4] Zabudovaný typ, používaný na uchovávanie jednobitových indikátorov.

```
typedef basic_ifstream<char> ifstream;
```

Príklad spracovania súboru

Nasledovný príklad ilustruje mnoho vlastností, o ktorých sme hovorili. Všimnime si včlenenie hlavičkového súboru **<fstream>,** aby sme deklarovali súborové vstupno/výstupné triedy. I keď v mnohých platformách včlenenie tohto súboru automaticky včleňuje i súbor **<iostream>**, norma to od kompilátorov nepožaduje, a preto to neplatí to vždy. Ak chceme vytvárať prenositeľný kód, mali by sme vždy včleňovať obidva súbory

```
// Vstupno/výstupné súborové prúdy
// Rozdiel medzi get() a getline()
#include <fstream>
#include <iostream>
using namespace std;

int main() {
  const int SZ = 100;           // veľkosť bufra
  char buf[SZ];
  {
    ifstream in("test.txt");    // Vstup
    ofstream out("test.out");   // Výstup
    int i = 1; // Line counter

    // Menej vhodný prístup pre riadkový vstup:
    while(in.get(buf, SZ)) { // Necháva \n vo vstupe
      in.get(); // Zahoď ďalší znak (\n)
      cout << buf << endl; // Pridaj \n
      // Výstup do súboru rovnaký ako na štandardný vstup/výstup
      out << i++ << ": " << buf << endl;
    }
  } // Deštruktory zatvoria vstup i výstup

  ifstream in("test.out");
  // Lepší riadkový vstup:
  while(in.getline(buf, SZ)) { // Odstraňuje \n
    char* cp = buf;
    while(*cp != ':')
      cp++;
    cp += 2; // Za ": "
    cout << cp << endl; // Stále musíme pridať \n
  }
}
```

Tu je opäť objekt použitý v situácií, keď kompilátor očakáva booleovskú hodnotu, ktorá indikuje úspech alebo zlyhanie.

Prvý cyklus **while** demonštruje použitie dvoch foriem funkcie **get()**. Prvá vyberá znaky do bufra a ukladá do bufra nulový znak ukončenia, keď sa načíta **SZ – 1** znakov alebo sa narazí na tretí argument (implicitne **'\n'**). Funkcia **get()** nechá ukončovací znak vo vstupnom prúde, a preto tento ukončovací znak musí byť zahodený prostredníctvom volania funkcie **in.get()**, jej verzie bez argumentov, ktorá vyberá jeden byte a vracia ho ako **int**. Tiež by sme mohli použiť členskú funkciu **ignore()**, ktorá má dva implicitné argumenty. Prvým je počet znakov, ktoré sa majú zahodiť a jeho implicitná hodnota je jedna. Druhým parametrom je znak, pri ktorom funkcia **ignore()** končí (po jeho vybratí), ktorého implicitná hodnota je **EOF**.

Ďalej vidíme dva výstupné príkazy, ktoré vypadajú podobne: jeden do objektu **cout** a jeden do súboru, reprezentovaného objektom **out**. Tu si môžeme všimnúť komfort, že sa nemusíme starať s akým druhom

objektu narábame, pretože formátovacie príkazy fungujú rovnako so všetkými objektmi triedy **ostream**. Prvý vypisuje riadok na štandardný výstup a druhý zapisuje riadok do nového súboru a pridáva číslo riadku:

Na ilustráciu funkcie **getline()** sa otvára súbor, ktorý sme práve vytvorili a orezávajú sa čísla riadkov. Aby sme zabezpečili, že súbor sa správne zatvorí pred otvorením na čítanie, máme dve možnosti. Prvú časť programu môžeme obaliť zloženými zátvorkami, čím si vynútime, že objekt **out** pôjde mimo rozsah, čím sa zavolá deštruktor, ktorý súbor zatvorí (tento spôsob je použitý i v príklade). Tiež by sme mohli zatvorí zavolať obidva súbory funkciu **close()**. Ak to spravíme takto, môžeme objekt **in** znovupoužiť na volanie členskej funkcie **open()**.

Druhý cyklus **while** ilustruje ako funkcia **getline** odstraňuje ukončovací znak (jeho tretí argument, ktorý má implicitnú hodnotu **'\n'**) zo vstupného prúdu, keď naň narazí. Hoci funkcia **getline()**, podobne ako funkcia **get()**, vkladá do bufra nulu, nevkladá ukončovací znak.

Tento príklad, ako i ostatné uvedené príklady, predpokladajú, že každé volanie ktorejkoľvek preťaženej funkcie **getline()** skutočne narazí na znak nového riadku. Ak sa to nestane, nastaví sa stavový bit **eofbit** prúdu a volanie funkcie **getline()** vráti hodnotu **false**, čo spôsobí, že program stratí posledný riadok vstupu.

Režimy otvárania súboru

Spôsob, akým sa bude súbor otvárať, ovládame prekrývaním implicitných argumentov konštruktora. Nasledujúca tabuľka obsahuje indikátory, ktoré ovládajú režim otvorenia súboru:

Indikátor	Funkcia
ios::in	Otvorí vstupný súbor. Tento režim sa používa pre **ofstream**, aby sa zabránilo orezaniu existujúceho súboru.
ios::out	Otvorí výstupný súbor. Ak je použitý pre **ofstream** bez **ios::app, ios::ate** alebo **ios::in**, použije sa **ios::trunc** .
ios::app	Otvorí výstupný súbor iba na pridávanie na koniec.
ios::ate	Otvorí existujúci súbor (pre vstup alebo výstup) a nastaví sa na koniec.
ios::trunc	Oreže starý súbor, ak existuje.
ios::binary	Otvorí súbor v *binárnom režime*. Implicitne sa súbor otvára v *textovom režime*.

Jednotlivé indikátory môžeme zlučovať bitovou operáciou súčtu (or).

Binárny indikátor, pokiaľ je prenositeľný, má význam iba pre niektoré nie UNIX-ové systémy, napríklad pre systémy, odvodené z MS-DOS-u, ktoré používajú špeciálne konvencie na ukončovanie súboru. Napríklad v systéme MS-DOS v textovom režime (ktorý je implicitný) po každom výstupe znaku nového riadku (**'\n'**), systém v skutočnosti produkuje dva znaky, dvojicu návrat vozíka/posun o riadok (CRLF), čo je dvojica ASCII znakov **0x0D** a **0x0A**. Opačne pri čítaní takéhoto súboru späť do pamäti v textovom režime každý výskyt tejto dvojice bytov spôsobí, že do programu sa namiesto nich vyšle znak **'\n'** . Ak chceme obísť takéto špeciálne spracovanie, súbor otvárame v binárnom režime. Ak používame funkcie **read()** alebo **write()**, mali by sme súbor vždy otvárať v binárnom režime, pretože tieto funkcie nemajú parameter, určujúci počet bytov.

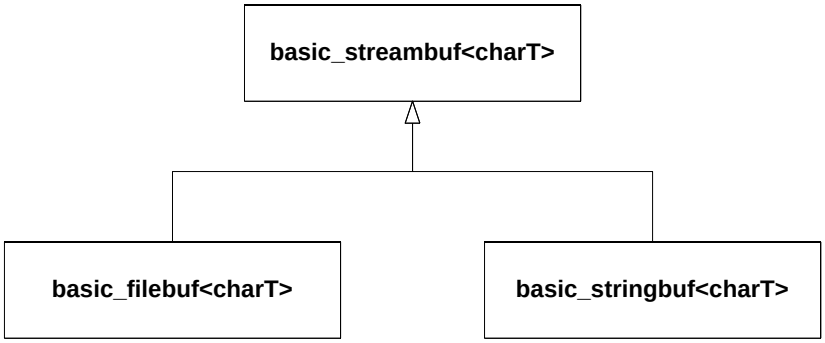
Súbor môžeme otvoriť pre vstup i výstup deklarovaním objektu triedy **fstream**. Keď deklarujeme objekt triedy **fstream**, musíme použiť dostatok indikátorov režimu otvorenia (pozri vyššie), aby sme systému oznámili, či chceme vstup, výstup alebo oboje. Pri prepínaní z výstupu na vstup potrebujeme buď vyprázdniť prúd alebo zmeniť súborovú pozíciu. Pri zmene zo vstupu na výstup musíme zmeniť súborovú pozíciu. Ak chceme vytvoriť súbor prostredníctvom objektu **fstream**, potrebujeme vo volaní konštruktora použiť režim otvorenia **ios::trunc,** ak budeme používať vstup i výstup.

Vyrovnávanie (buffering) prúdov

Dobré skúsenosti diktujú, že vždy, keď vytvárame novú triedu, mali by sme sa snažiť pred užívateľom triedy čo najviac skryť detaily implementácie. Mali by sme im ukázať iba to, čo potrebujú vedieť a zvyšok deklarovať ako **private**, aby sme sa vyhli zmätkom a problémom. Keď používame vkladače a extraktory zvyčajne nepoznáme alebo sa nestaráme, či sa byty produkujú alebo konzumujú, či už narábame so štandardným vstupom/výstupom, súbormi alebo nejakou novovytvorenou triedou alebo zariadením.

Avšak čas pokročil do stavu, kedy je dôležité komunikovať s časťou prúdu, ktorý produkuje a konzumuje byty. Aby sme vytvorili pre túto časť spoločné rozhranie a zároveň skryli implementáciu v pozadí, štandardná knižnica ju abstrahuje do svojej vlastnej triedy, nazývanej **streambuf**. Každý prúdový objekt obsahuje smerník na nejaký druh **streambuf**. (Druh závisí od toho, či pracujeme so štandardným vstupom/výstupom, súbormi, pamäťou, atď.) Ku **streambuf** môžeme pristupovať priamo, napríklad môžeme presúvať surové byty dnu a von zo **streambuf** bez ich formátovania prostredníctvom obaľujúceho prúdu. Toto sa vykonáva prostredníctvom volaním členských funkcií triedy **streambuf**.

Najdôležitejšou vecou, ktorú momentálne potrebujeme poznať je, že každý prúdový objekt obsahuje smerník na objekt triedy **streambuf**, a že trieda **streambuf** má nejaké členské funkcie, ktoré môžeme v prípade potreby volať. Pre súborové a reťazcové prúdy existujú špecializované typy prúdových bufrov:



Na sprístupnenie **streambuf** každý prúdový objekt obsahuje členskú funkciu nazvanú **rdbuf()**, ktorá vracia smerník na objekt triedy **streambuf**. Takýmto spôsobom môžeme volať ľubovoľnú funkciu objektu **streambuf**. Avšak jednou z najzaujímavejších vecí, ktorú môžeme robiť so smerníkom **streambuf** je jeho pripojenie k inému prúdovému objektu pomocou operátora **<<**. Týmto sa všetky znaky objektu odvedú do objektu na ľavej strane operátora **<<**. Ak chceme presunúť všetky znaky z jedného prúdu do druhého, nemusíme ísť cez otravné (a potencionálne chybové) čítanie znak po znaku alebo riadok po riadku. Toto je omnoho elegantnejší postup.

Napríklad nasledujúci jednoduchý program otvára súbor a posiela jeho obsah na štandardný výstup (podobný predchádzajúcemu príkladu):

```
// Výpis súboru na štandardný výstup
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
  ifstream in("test.txt");
  cout << in.rdbuf();   // Výpis celého súboru
}
```

Najskôr sa vytvára sa objekt triedy **ifstream**, ktorý otvára súbor test.txt argument. Celá práca sa v skutočnosti vykonáva v príkaze:

```
cout << in.rdbuf();
```

ktorý posiela celý obsah súboru do objektu **cout**. Toto sa nielen krátko programuje, ale často je to i efektívnejšie, než presun po bytoch.

Forma funkcie **get()** dovoľuje písať priamo do **streambuf** iného objektu. Prvým argumentom je odkaz na cieľový **streambuf** a druhým je ukončovací znak (implicitne **'\n'**), ktorý funkciu **get()** zastavuje. Tu je iný spôsob vytlačenia súboru na štandardný výstup:

```
// Výpis súboru na štandardný výstup
#include <fstream>
#include <iostream>
using namespace std;

int main()
{
  ifstream in("test.txt");
  streambuf &sb = *cout.rdbuf();
  while (!in.get(sb).eof()) {
    if(in.fail())          // Nájdený prázdny riadok
      in.clear();
    cout << char(in.get()); // Spracuj '\n'
  }
}
```

Funkcia **rdbuf()** vracia smerník, takže musíme použiť dereferenciu, aby sme vyhoveli požiadavkám funkcie, ktorá požaduje objekt. Prúdové bufre nie sú určené na kopírovanie (nemajú kopírovací konštruktor), a preto objekt **sb** je deklarovaný ako *odkaz* na bufer prúdu **cout**. Pre prípad, že vstupný súbor obsahuje prázdny riadok potrebujeme volať funkcie **fail()** a **clear()**. Keď táto konkrétna preťažená verzia funkcie **get()** vidí dva znaky nového riadku (dôkaz prázdneho riadku), nastaví **fail** bit vstupného prúdu, a preto musíme volať funkciu **clear()** aby sme ho vymazali a prúd mohol pokračovať v čítaní. Druhé volanie funkcie **get()** vyberie a zobrazí znak nového riadku. (Nezabúdajme, že funkcia **get()** nevyberá oddeľovač ako to robí funkcia **getline()**).

Uvedený spôsob sa často nepoužíva, ale je dobré o ňom vedieť.

Vyhľadávanie v prúdoch

Každý typ prúdu pozná pojem, odkiaľ pochádza jeho "ďalší" znak (v prípade **istream**) alebo kam pôjde (v prípade **ostream**). V niektorých situáciách potrebujeme presunúť pozíciu prúdu. Môžeme to urobiť dvomi spôsobmi: jeden využíva absolútnu polohu v prúde, nazývanú **streampos** a druhý funguje rovnako ako štandardná C knižničná funkcia **fseek()** pre súbor a presúva o daný počet bytov od začiatku, konca a aktuálnej pozície v súbore.

Použitie **streampos** vyžaduje, že najskôr zavoláme "oznamovaciu" funkciu **tellp()** pre **ostream** alebo **tellg()** pre **istream**. ("p" označuje "put ukazovateľ" a "g" zasa "get" ukazovateľ.) Táto funkcia vracia **streampos**, ktorú môžeme neskôr použiť vo volaniach funkcií **seekp()** pre **ostream** a **seekg()** pre **istream**, keď sa chceme vrátiť na túto pozíciu v prúde.

Druhý prístup je relatívne vyhľadávanie a používa preťažené verzie funkcií **seekp()** a **seekg()**. Prvým argumentom je počet znakov, o ktorý sa má presúvať: môže to byť kladné alebo záporné číslo. Druhým parametrom je smer prehľadávania:

ios::beg	Od začiatku prúdu
ios::cur	Od aktuálnej pozície v prúde
ios::end	Od konca prúdu

Nasledujúci príklad ilustruje pohyby v súbore, avšak nezabúdajme, že nie sme ohraničení len na pohyb v rámci súborov ako je tomu v jazyku C. V C++ sa môžeme presúvať v ľubovoľnom type prúdu (aj keď štandardné prúdové objekty ako sú **cin** a **cout** toto explicitne nedovoľujú:

```
// Pohybovanie sa v prúdoch
#include <cassert>
#include <cstddef>
#include <cstring>
#include <fstream>
using namespace std;
```

```
int main() {
  const int STR_NUM = 5, STR_LEN = 30;
  char origData[STR_NUM][STR_LEN] = {
    "Vazeni studenti. . .",
    "Bavi vas C++?",
    "Ak nie,",
    "To nie je prilis dobre,",
    "Ale je vas vela!"
  };
  char readData[STR_NUM][STR_LEN] = { 0 };
  ofstream out("oznam.bin", ios::out | ios::binary);
  for(size_t i = 0; i < STR_NUM; i++)
    out.write(origData[i], STR_LEN);
  out.close();
  ifstream in("oznam.bin", ios::in | ios::binary);
  in.read(readData[0], STR_LEN);
  assert(strcmp(readData[0], "Vazeni studenti. . .") == 0);
  // Pohyb -STR_LEN bytov od konca súboru
  in.seekg(-STR_LEN, ios::end);
  in.read(readData[1], STR_LEN);
  assert(strcmp(readData[1], "Ale je vas vela!") == 0);
  // Absolútny pohyb (niečo ako použitie operátora [] pre súbor)
  in.seekg(3 * STR_LEN);
  in.read(readData[2], STR_LEN);
  assert(strcmp(readData[2], "To nie je prilis dobre,") == 0);
  // Pohyb späť od akyuálnej pozície
  in.seekg(-STR_LEN * 2, ios::cur);
  in.read(readData[3], STR_LEN);
  assert(strcmp(readData[3], "Ak nie,") == 0);
  // Pohyb od začiatku súboru
  in.seekg(1 * STR_LEN, ios::beg);
  in.read(readData[4], STR_LEN);
  assert(strcmp(readData[4], "Bavi vas C++?")
    == 0);
}
```

Tento program zapisuje a (veľmi šikovne) oznam do súboru pomocou binárneho výstupného prúdu. Pretože ho znova otvárame ako objekt triedy **ifstream**, na umiestnenie "get ukazovátka" používame funkciu **seekg()**. Ako vidíme, môžeme sa presúvať od začiatku alebo konca súboru alebo od aktuálnej pozície v súbore. Samozrejme pre pohyb od začiatku súboru musíme zadať kladné číslo a pre pohyb späť od konca súboru zasa záporné číslo.

Teraz, keď už vieme všetko o **streambuf** a ako sa v ňom pohybovať, dokážeme pochopiť alternatívnu metódu (okrem použitia objektu triedy **fstream**) vytvorenia prúdového objektu, ktorým bude súbor pre zápis i čítanie. Nasledujúci kód najskôr vytvorí objekt triedy **ifstream** s indikátormi, ktoré hovoria, že sa jedná vstupný i výstupný súbor. Samozrejme do objektu triedy **ifstream** nemôžeme zapisovať, takže potrebujeme vytvoriť objekt triedy **ostream** s prúdovým bufrom:

```
ifstream in("menosuboru", ios::in | ios::out);
ostream out(in.rdbuf());
```

Možne sme zvedaví, čo sa stane, keď zapíšeme do jedného z týchto objektov. Tu je príklad:

```
// Čítanie a zápis do toho istého súboru
#include <fstream>
#include <iostream>
using namespace std;

int main() {
```

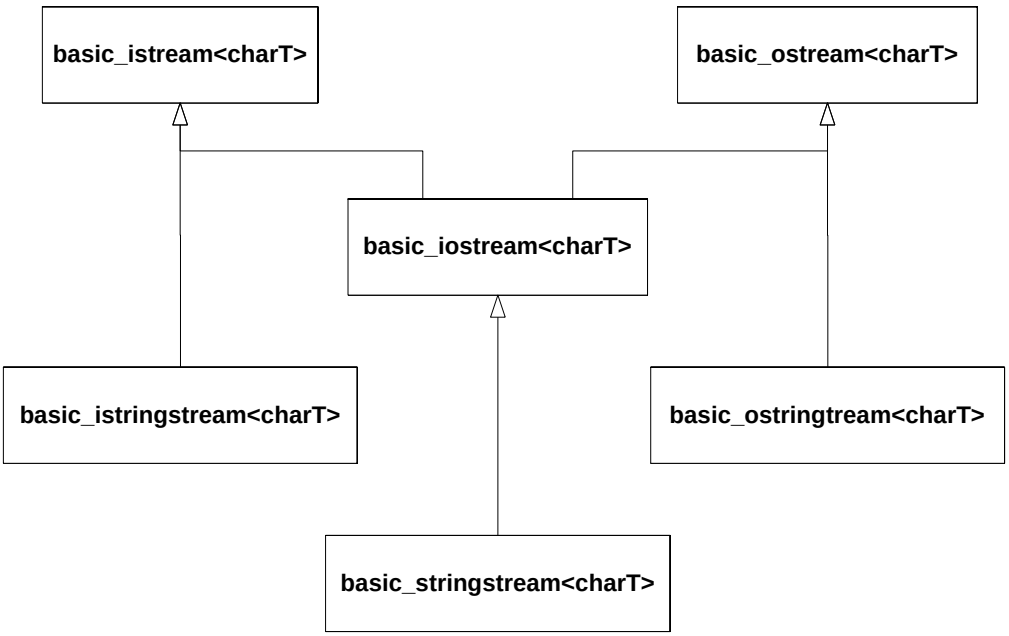
```
    ifstream in("test.txt");
    assure(in, "test.txt");
    ofstream out("test.out");
    assure(out, "test.out");
    out << in.rdbuf(); // Kopíruj súbor
    in.close();
    out.close();
    // Otvor pre čítanie i zápis:
    ifstream in2("test.out", ios::in);
    assure(in2, "test.out");
    ostream out2(in2.rdbuf());
    cout << in2.rdbuf();  // Vytlač celý súbor
    out2 << "Kde toto skonci?";
    out2.seekp(0, ios::end);
    out2 << "A co toto?";
    in2.seekg(0, ios::beg);
    cout << in2.rdbuf();
}
```

Prvých päť riadkov kopíruje zdrojový kód tohto programu do súboru, nazvaného **test.out** a potom sa súbory zatvoria. Toto nám poskytne bezpečný textový súbor na hranie. Potom je použitý vyššie uvedený spôsob na vytvorenie dvoch objektov, ktoré čítajú a zapisujú do toho istého súboru. V **cout << in2.rdbuf()** vidíme, že "get" ukazovátko je inicializované na začiatok súboru. "put" ukazovátko" je nastavené na koniec súboru, pretože veta “Kde toto skonci?” sa objaví na konci súboru. Avšak ak "put" ukazovátko presunieme na začiatok pomocou funkcie **seekp()**, všetok vložený text *prepíše* už existujúci text. Obidva zápisy sú viditeľné, keď "get" ukazovátko presunieme naspäť na začiatok funkciou **seekg()** a potom súbor zobrazíme. Samozrejme súbor sa automaticky uloží a zatvorí, keď objekt **out2** zanikne a zavolá sa deštruktor.

Reťazcové prúdy

Reťazcový prúd pracuje namiesto so súborom priamo s pamäťou alebo štandardným výstupom. Dovoľuje používať rovnaké čítacie a formátovacie funkcie, ktoré používame s objektmi **cin** a **cout** na manipulovanie s bytami v pamäti.

Mená tried reťazcových prúdov kopírujú súborové prúdy. Ak chceme vytvoriť reťazcový prúd, z ktorého chceme vyberať znaky, vytvoríme objekt triedy **istringstream**. Ak chceme vkladať znaky do reťazcového prúdu, vytvoríme objekt triedy **ostringstream**. Všetky deklarácie reťazcových prúdov sú v štandardnom hlavičkovom súbore **<sstream>**. Ako zvyčajne, sú to šablónové triedy, ktoré zapadajú do hierarchie prúdov nasledovne:



Vstupné reťazcové prúdy

Ak chceme čítať pomocou reťazcových prúdových operácií, vytvárame objekt triedy **istringstream**, inicializovaný reťazcom. Použitie takéhoto objektu ilustruje nasledujúci príklad:

```
// Vstupné reťazcové prúdy
#include <cassert>
#include <cmath>  // pre fabs()
#include <iostream>
#include <limits> // pre epsilon()
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream s("47 1.414 Toto je test");
  int i;
  double f;
  s >> i >> f; // Bielymi medzerami oddelený vstup
  assert(i == 47);
  double relerr = (fabs(f) - 1.414) / 1.414;
  assert(relerr <= numeric_limits<double>::epsilon());
  string buf2;
  s >> buf2;
  assert(buf2 == "Toto");
  cout << s.rdbuf(); // " je test"
}
```

Vidíme, že toto je pružnejší a všeobecnejší prístup ako transformovať znakové reťazce na typové hodnoty, než je použitie štandardných C knižničných funkcií **atof()** a **atoi()**, dokonca i keď tieto môžu byť pre jednotlivé konverzie o čosi efektívnejšie.

Vo výraze **s >> i >> f** sa prvé číslo vyberá do premennej **i** a druhé do premennej **f**. Toto nie je "prvá množina znakov, oddelená bielou medzerou" pretože to závisí od dátového typu, do ktorého ideme extrahovať. Napríklad ak by reťazec obsahoval "**1.414 47 Toto je test**," potom by premenná **i** obsahovala hodnotu **1** pretože vstupná rutina by sa zastavila na desatinnej bodke. Premenná **f** by potom obsahovala hodnotu **0.414**. Toto sa hodí, iba ak chceme rozdeliť reálne číslo na celé číslo a zlomkovú časť. Inak by to bola chyba. Druhé volanie funkcie **assert()** počíta relatívnu odchýlku medzi tým čo sme načítali a čo očakávame. Toto je vždy lepšie než porovnávať dve reálne čísla na rovnosť. Konštanta, vrátená funkciou **epsilon()**, a definovaná v hlavičkovom súbore **<limits>** reprezentuje ***strojový epsilon*** pre **double** čísla, čo je najlepšia tolerancia, ktorú môžeme očakávať od porovnania dvoch čísel typu **double**.

Ako je už asi zrejmé, premenná **buf2** nedostane zvyšok reťazca, iba ďalšou bielou medzerou oddelené slovo. Všeobecne platí, že je lepšie použiť prúdový extraktor vtedy, keď poznáme presnú postupnosť dát vo vstupnom prúde a konvertujeme ich na nejaký iný typ, než je reťazec. Avšak ak chceme extrahovať zvyšok reťazca naraz a odoslať ho do iného prúdu, môžeme zavolať funkciu **rdbuf()** (pozri ďalej).

Výstupné reťazové prúdy

Na vytvorenie výstupného reťazcového prúdu, do ktorého chceme ukladať dáta, musíme vytvoriť objekt **ostringstream**, ktorý spravuje a dynamicky mení znakový bufer, ktorý bude uchovávať čokoľvek, čo do neho vložíme. Ak chceme získať formátovaný výsledok ako objekt triedy **string**, zavoláme členskú funkciu **str**:

```
// Ilustrácia ostringstream
#include <iostream>
#include <sstream>
#include <string>
```

```
using namespace std;

int main() {
  cout << "zadaj int, float a string: ";
  int i;
  float f;
  cin >> i >> f;
  cin >> ws; // Zahoď biele medzery
  string stuff;
  getline(cin, stuff); // Zober zvyšok riadku
  ostringstream os;
  os << "integer = " << i << endl;
  os << "float = " << f << endl;
  os << "string = " << stuff << endl;
  string result = os.str();
  cout << result << endl;
}
```

Príklad vykonávania by mohol byť nasledovný (vstup z klávesnice je tučným písmom).

```
zadaj int, float a string: 10 20.5 a koniec
integer = 10
float = 20.5
string = a koniec
```

Vidíme, že podobne ako ostatné výstupné prúdy môžeme na ukladanie bytov do objektu triedy **ostringstream** použiť bežné formátovacie nástroje, ako je operátor << a **endl**. Funkcia **str()** po každom svojom zavolaní vracia nový objekt triedy **string**, takže objekt **stringbuf** v pozadí, patriaci reťazcovému prúdu zostane neporušený.

A tu je elegantná verzia programu **HTMLStripper.cpp**, ktorý z textového súboru odstraňuje všetky HTML značky a špeciálne kódy, využívajúca reťazcové prúdy.

```
// Filter na odstraňovanie html tagov a značiek
#include <cstddef>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>
using namespace std;

string& replaceAll(string& context, const string& from,const string& to)
{
  size_t lookHere = 0;
  size_t foundHere;
  while ((foundHere = context.find(from, lookHere))!=string::npos) {
    context.replace(foundHere, from.size(), to);
    lookHere = foundHere + to.size();
  }
  return context;
}

string& stripHTMLTags(string& s) throw(runtime_error) {
  size_t leftPos;
  while ((leftPos = s.find('<')) != string::npos) {
    size_t rightPos = s.find('>', leftPos+1);
```

```
    if (rightPos == string::npos) {
      ostringstream msg;
      msg << "Neuplny HTML tag zacinajuci na pozicii " << leftPos;
      throw runtime_error(msg.str());
    }
    s.erase(leftPos, rightPos - leftPos + 1);
  }
  // Odstran vsetky specialne HTML znaky
  replaceAll(s, "&lt;", "<");
  replaceAll(s, "&gt;", ">");
  replaceAll(s, "&amp;", "&");
  replaceAll(s, " ", " ");
  // atd.
  return s;
}

int main(int argc, char* argv[])
{
  requireArgs(argc, 1, "usage: HTMLStripper2 InputFile");
  ifstream in(argv[1]);
  assure(in, argv[1]);
  // Čítaj celý súbor do reťazca a potom orež
  ostringstream ss;
  ss << in.rdbuf();
  try {
    string s = ss.str();
    cout << stripHTMLTags(s) << endl;
    return EXIT_SUCCESS;
  }
  catch (runtime_error& x) {
    cout << x.what() << endl;
    return EXIT_FAILURE;
  }
}
```

V tomto programe prečítame celý súbor do reťazca vložením volania funkcie **rdbuf()** súborového prúdu do **ostringstream**. Teraz je už jednoduché vyhľadávať dvojice HTML oddeľovačov a mazať ich bez toho, aby sme sa starali o prekročenie hraníc riadkov ako sme to museli robiť v predchádzajúcej verzii.

Nasledujúci príklad ilustruje, ako používať obojsmerný (čítanie/zápis) reťazcový prúd:

```
// Čítanie a zápis do reťazcového prúdu
#include <cassert>
#include <sstream>
#include <string>
using namespace std;

int main() {
  string text = "My sme nepredali ziadne hrozno";
  stringstream ss(text);
  ss.seekp(0, ios::end);
  ss << " skor ako dozrelo.";
  assert(ss.str() ==
                "My sme nepredali ziadne hrozno skor ako dozrelo.");
  // Zmena "nepredali" to "neukradli"
  ss.seekg(9, ios::beg);
  string word;
  ss >> word;
```

```
  assert(word == "preda");
  ss.seekp(9, ios::beg);
  ss << "ukrad";
  // Zmena "hrozno" na "pecivo"
  ss.seekg(24, ios::beg);
  ss >> word;
  assert(word == "hrozno");
  ss.seekp(24, ios::beg);
  ss << "pecivo";
  assert(ss.str() ==
    "My sme neukradli ziadne pecivo skor ako dozrelo.");
  ss.str("Kon odlisnej farby.");
  assert(ss.str() == " Kon odlisnej farby.");
}
```

Ako vždy, na posun "put" ukazovátka voláme funkciu **seekp()** a na premiestnenie "get" ukazovátka zasa funkciu **seekg()**. Dokonca i keď sme to v tomto príklade neukázali, reťazcové prúdy sú trochu zhovievavejšie, než súborové prúdy v tom, že sa môžeme kedykoľvek prepínať medzi čítaním a zápisom. Nepotrebujeme premiestňovať get a put ukazovátka alebo vyprázdňovať prúd. Tento program zároveň ilustruje preťaženie funkcie **str()**, ktorá nahrádza **stringbuf** prúdu novým reťazcom.

Formátovanie výstupného prúdu

Cieľom návrhu prúdov bolo umožniť ľahko presúvať a/alebo formátovať znaky. Určite by boli zbytočné, keby sme nemohli formátovať prinajmenšom tak, ako to umožňuje **printf** skupina funkcii knižnice jazyka C.
Formátovacie funkcie prúdov sú na prvý pohľad mätúce, pretože často existuje viac ako jeden spôsob ovládania formátovania: prostredníctvom členských funkcií a prostredníctvom manipulátorov. Aby to bolo ešte komplikovanejšie, stavové indikátory, ktoré riadia formátovanie (napríklad zarovnanie sprava alebo zľava, použitie veľkých písmen pre hexa notáciu, používanie desatinnej bodky pre reálne čísla, atď.) nastavuje všeúčelová členská funkcia,. Na druhej strane samostatné členské funkcie nastavujú a čítajú hodnoty vyplňovacieho znaku, šírku položky a presnosť.

Aby sme všetko toto objasnili, najskôr si pozrieme interné formátovacie dáta prúdu spolu s členskými funkciami, ktoré tieto dáta modifikujú. (všetko sa dá ovládať členskými funkciami). O manipulátoroch si povieme neskôr.

Formatovácie indikátory

Trieda **ios** obsahuje dátové členy, ktoré uchovávajú všetky formátovacie informácie, prináležiace prúdu. Niektoré z týchto dát majú interval hodnôt uložený v premenných: presnosť reálneho čísla, šírka výstupnej položky a znak, používaný na doplňovanie výstupu (normálne medzera). Zvyšok formátovania je určovaný indikátormi, ktoré sú zvyčajne spojené aby sa šetrilo priestorom a spoločne sa nazývajú ***formátovacie indikátory***. Hodnotu týchto formátovacích indikátorov zisťujeme členskou funkciou **ios::flags()**, ktorá nemá argumenty a vracia objekt typu **fmtflags** (zvyčajne synonymum pre **long**), ktorý obsahuje aktuálne hodnoty formátovacích indikátorov. Všetky ostatné funkcie robia zmeny formátovacích indikátorov a vracajú predchádzajúcu hodnotu formátovacích indikátorov.

```
fmtflags ios::flags(fmtflags newflags);
fmtflags ios::setf(fmtflags ored_flag);
fmtflags ios::unsetf(fmtflags clear_flag);
fmtflags ios::setf(fmtflags bits, fmtflags field);
```

Prvá funkcia mení ***všetky*** indikátory, čo sa občas robí. Častejšie však meníme jeden indikátor pomocou zvyšných troch funkcií.

Použitie funkcie **setf()** je trochu mätúce. Aby sme vedeli, ktorú preťaženú verziu použiť, musíme poznať typ indikátora, ktorý chceme meniť. Existujú dva typy indikátorov:

- ktoré sa iba jednoducho zapínajú a vypínajú, a

- ktoré pracujú v skupine s inými indikátormi.

Zapni/vypni indikátory pochopíme ľahko, pretože ich zapneme funkciou **setf(fmtflags)** a vypneme funkciou **unsetf(fmtflags)**. Tieto indikátory sú uvedené v nasledujúcej tabuľke:

Zapni/vypni indikátor	Dôsledok
ios::skipws	Preskoč bielu medzeru. (pre vstup; implicitne nastavený)
ios::showbase	Indikuj základ čísla (aký je nastavený, napríklad **dec**, **oct** alebo **hex**) pri tlači zabudovanej hodnoty. Vstupné prúdy tiež rozpoznávajú prefix základu ak je **showbase** zapnutý.
ios::showpoint	Zobrazuj desatinnú bodku a koncové nuly pre hodnoty v pohyblivej rádovej čiarke.
ios::uppercase	Zobrazuj veľké písmena A-F pre hexadecimálne hodnoty a E pre vedecké hodnoty.
ios::showpos	Zobrazuj znamienko plus (+) pre kladné hodnoty.
ios::unitbuf	"Jednotkové bufrovanie." Prúd sa vyprázdňuje po každom vložení.

Napríklad ak chceme zobrazovať znamienko plus na **cout**, zavoláme **cout.setf(ios::showpos).** Zobrazovanie zrušíme zavolaním **cout.unsetf(ios::showpos)**.

Indikátor **unitbuf** riadi tzv. ***jednotkové bufrovanie***, čo znamená, že každé vloženie sa okamžite posiela do výstupného prúdu. Toto je vhodné pri vyhľadávaní chýb, pretože keď program zhavaruje, dáta sú zapísané do protokolovacieho súboru. Takéto bufrovanie ilustruje nasledovný príklad:

```
#include <cstdlib>  // pre abort()
#include <fstream>
using namespace std;

int main() {
  ofstream out("log.txt");
  out.setf(ios::unitbuf);
  out << "jeden\n";
  out << "dva\n";
  abort();
}
```

Je nevyhnutné zapnúť jednotkové bufrovanie pred vkladaním do prúdu. Ak zakomentujeme volanie funkcie **setf()**, konkrétny kompilátor zapísal iba písmeno do súboru **log.txt**. S jednotkovým bufrovaním ku strate dát nedochádza.

Štandardný chybový výstupný prúd **cerr** ma jednotkové bufrovanie zapnuté implicitne. Samozrejme jednotkové bufrovanie nie je zadarmo, takže ak výstupné prúdy používame veľmi často, nepoužívajme jednotkové bufrovanie, pokiaľ nás zaujíma výkon.

Formátovacie položky

Druhý typ formátovacích indikátorov pracuje v skupine. Jedine jeden z týchto indikátorov môže byť, podobne ako rádiové tlačítko, zapnutý, ostatné musia byť vypnuté. Nanešťastie toto sa nerobí automaticky a preto si musíme dávať pozor, ktoré indikátory nastavujeme, aby sme neúmyselne nezavolali nesprávnu funkciu **setf()**. Napríklad existuje indikátor pre každý číselný základ: hexadecimálny, dekadický a oktálový. Spoločne sa tieto indikátory nazývajú **ios::basefield**. Ak je nastavený indikátor **ios::dec** a zavoláme **setf(ios::hex)**, nastavíme indikátor **ios::hex** ale nezmažeme bit **ios::dec**, výsledkom čoho je nedefinované správanie. Lepšie je zavolať druhú formu funkcie **setf()**, napríklad **setf(ios::hex, ios::basefield)**. Táto funkcia najskôr vymaže bity **ios::basefield** a potom nastaví bit **ios::hex**. Táto forma funkcie **setf()** zabezpečuje, že ostatné indikátory skupiny sa pri zapnutí jedného z nich vypnú. Samozrejme manipulátor **ios::hex** urobí toto všetko za nás automaticky, takže nás nemusia zaujímať implementačné detaily tejto triedy alebo dokonca sa starať o to, že existuje množina binárnych indikátorov. Neskôr si ukážeme, že manipulátory poskytujú ekvivalentnú funkčnosť na všetkých miestach, kde by sa dala použiť funkcia **setf()**.

A tu sú skupiny indikátorov a ich dôsledky:

ios::basefield	Dôsledok
ios::dec	Formátuje zabudované hodnoty v základe 10 (decimálne) (implicitný základ—nie je viditeľný žiaden prefix).
ios::hex	Formátuje zabudované hodnoty v základe 16 (hexadecimálne).
ios::oct	Formátuje zabudované hodnoty v základe 8 (oktálovo).

ios::floatfield	Dôsledok
ios::scientific	Zobrazuj čísla v pohyblivej rádovej čiarke vo vedeckom formáte. Položka presnosti určuje počet číslic za desatinnou bodkou.
ios::fixed	Zobrazuj čísla v pohyblivej rádovej čiarke v pevnom formáte. Položka presnosti určuje počet číslic za desatinnou bodkou.
"automatická" (Ani jeden bit nie je nastavený.)	Položka presnosti určuje celkový počet platných číslic.

ios::adjustfield	Dôsledok
ios::left	Hodnoty zarovnané vľavo. Vyplňujúci znak doplnený sprava.
ios::right	Hodnoty zarovnané vpravo. Vyplňujúci znak doplnený zľava. Toto je implicitné zarovnanie.
ios::internal	Vlož vyplňovacie znaky za znamienko alebo indikátor základu, ale pred hodnotu. (Inými slovami znamienko, ak sa tlačí, je zarovnané vľavo a čísla vpravo).

Šírka, výplň a presnosť

Interné premenné, ktoré riadia šírku výstupnej položky, vyplňovací znak, používaný na doplňovanie výstupnej položky a presnosť pre tlač čísel v pohyblivej rádovej čiarke sa čítajú a zapisujú členskými funkciami s rovnakým názvom (anglickým).

Funkcia	Dôsledok
int ios::width()	Vracia aktuálnu šírku. (Implicitne je 0.) Používa sa pre vkladanie i výber.
int ios::width(int n)	Nastavuje šírku, vracia predchádzajúcu šírku
int ios::fill()	Vracia aktuálny vyplňovací znak. (Implicitne je to medzera.)
int ios::fill(int n)	Nastavuje vyplňovací znak, vracia predchádzajúci vyplňovací znak.
int ios::precision()	Vracia aktuálnu presnosť čísla v pohyblivej rádovej čiarke. (Implicitne je 6.)
int ios::precision(int n)	Nastavuje presnosť čísla v pohyblivej rádovej čiarke, Vracia predchádzajúcu presnosť.

Hodnoty **fill** a **precision** sú dostatočne zrozumiteľné, ale **width** si vyžaduje krátke vysvetlenie. Ak je šírka nulová, vloženie hodnoty dáva minimálny počet znakov, potrebných na reprezentáciu tejto hodnoty. Kladná hodnota znamená, že vloženie hodnoty bude dávať najmenej toľko znakov, koľko určuje šírka, ak je hodnota menšia, než šírka, na doplnenie položky sa použijú vyplňovacie znaky. Avšak hodnota sa nikdy neoreže, takže ak vytlačíme 123 so šírkou dva, stále dostaneme 123. Položka šírky udáva ***minimálny*** počet znakov. Maximálny počet sa špecifikovať nedá.

Šírka je odlišná, pretože sa nuluje po každom vložení alebo výbere, ktorý môže byť ovplyvnený jej hodnotou. Nie je to skutočná stavová premenná, ale iba implicitný argument pre vkladanie a výber. Ak chceme konštantnú šírku, musíme po každom vložení alebo výbere volať funkciu **width()**.

Vyčerpávajúci príklad

Aby sme sa uistili, že vieme volať vyššie uvedené funkcie, tu je príklad, ktorý ich všetky volá.

```
// Formátovacie funkcie
#include <fstream>
#include <iostream>
```

```
using namespace std;
#define D(A) T << #A << endl; A

int main() {
  ofstream T("format.out");
  assure(T);
  D(int i = 47;)
  D(float f = 2300114.414159;)
  char* s = "Je tam este nieco?";

  D(T.setf(ios::unitbuf);)
  D(T.setf(ios::showbase);)
  D(T.setf(ios::uppercase | ios::showpos);)
  D(T << i << endl;) // Implicitne dec
  D(T.setf(ios::hex, ios::basefield);)
  D(T << i << endl;)
  D(T.setf(ios::oct, ios::basefield);)
  D(T << i << endl;)
  D(T.unsetf(ios::showbase);)
  D(T.setf(ios::dec, ios::basefield);)
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.fill('0');)
  D(T << "Vyplnovaci znak: " << T.fill() << endl;)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::right, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T.setf(ios::internal, ios::adjustfield);)
  D(T.width(10);)
  T << i << endl;
  D(T << i << endl;) // Bez width(10)

  D(T.unsetf(ios::showpos);)
  D(T.setf(ios::showpoint);)
  D(T << "Presnost = " << T.precision() << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.unsetf(ios::uppercase);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)
  D(T.precision(20);)
  D(T << "Presnost = " << T.precision() << endl;)
  D(T << endl << f << endl;)
  D(T.setf(ios::scientific, ios::floatfield);)
  D(T << endl << f << endl;)
  D(T.setf(ios::fixed, ios::floatfield);)
  D(T << f << endl;)

  D(T.width(10);)
  T << s << endl;
  D(T.width(40);)
  T << s << endl;
  D(T.setf(ios::left, ios::adjustfield);)
  D(T.width(40);)
  T << s << endl;
}
```

Tento príklad na vytvorenie trasovacieho súboru využíva trik, takže môžeme sledovať čo sa deje. Makro **D(a)** používa "reťazenie" preprocesora na prevod **a** na zobrazovaný reťazec. Potom sa znova zopakuje, takže sa vykoná príkaz. Makro posiela všetky informácie do súboru nazvaného **T**, ktorý je trasovacím súborom. Výstup je nasledovný:

```
int i = 47;
float f = 2300114.414159;
T.setf(ios::unitbuf);
T.setf(ios::showbase);
T.setf(ios::uppercase | ios::showpos);
T << i << endl;
+47
T.setf(ios::hex, ios::basefield);
T << i << endl;
0X2F
T.setf(ios::oct, ios::basefield);
T << i << endl;
057
T.unsetf(ios::showbase);
T.setf(ios::dec, ios::basefield);
T.setf(ios::left, ios::adjustfield);
T.fill('0');
T << "Vyplnovaci znak: " << T.fill() << endl;
Vyplnovaci znak: 0
T.width(10);
+470000000
T.setf(ios::right, ios::adjustfield);
T.width(10);
0000000+47
T.setf(ios::internal, ios::adjustfield);
T.width(10);
+000000047
T << i << endl;
+47
T.unsetf(ios::showpos);
T.setf(ios::showpoint);
T << "Presnost = " << T.precision() << endl;
Presnost = 6
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;

2.300114E+06
T.unsetf(ios::uppercase);
T << endl << f << endl;

2.300114e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.500000
T.precision(20);
T << "Presnost = " << T.precision() << endl;
Presnost = 20
T << endl << f << endl;

2300114.50000000000000000000
T.setf(ios::scientific, ios::floatfield);
T << endl << f << endl;
```

```
2.3001145000000000000e+06
T.setf(ios::fixed, ios::floatfield);
T << f << endl;
2300114.50000000000000000000
T.width(10);
Je tam este nieco?
T.width(40);
0000000000000000000000 Je tam este nieco?
T.setf(ios::left, ios::adjustfield);
T.width(40);
Je tam este nieco?0000000000000000000000
```

Štúdium tohto výstupu ozrejmí pochopenie prúdových formátovacích členských funkcií.

Manipulátory

Ako sme videli v predchádzajúcom programe volanie členských funkcií pre prúdové formátovacie operácie je trochu únavné. Aby sa veci pri čítaní a zápise zjednodušili, existuje množina tzv. ***manipulátorov***, ktoré duplikujú akcie, poskytované členskými funkciami. Manipulátory sú pohodlné, pretože ich môžeme vkladať priamo do výrazu, nemusíme vytvárať oddelené príkazy volania členských funkcií.

Manipulátory namiesto (alebo naviac) spracovania dát **menia** stav prúdu. Keď napríklad do výstupného výrazu vložíme **endl**, nielen že sa vloží znak nového riadku, ale zároveň sa vyprázdni i prúd (t.j. zapíšu sa všetky čakajúce znaky, ktoré boli uložené v internom prúdovom bufri ale neboli ešte odoslané). Prúd môžeme vyprázdniť i nasledovne:

```
cout << flush;
```

čo spôsobí zavolanie členskej funkcie **flush()** ako keby sme použili:

```
cout.flush();
```

Ďalšie základné manipulátory menia základ na **oct** (oktálový), **dec** (decimálny) alebo **hex** (hexadecimálny):

```
cout << hex << "0x" << i << endl;
```

V tomto prípade bude číselný výstup pokračovať v hexadecimálnom režime, až kým ho nezmeníme vložením **dec** alebo **oct** do výstupného prúdu.

Existuje tiež manipulátor pre výber, ktorý "*zje*" biele medzery:

```
cin >> ws;
```

Manipulátory bez argumentov sa nachádzajú v súbore **<iostream>**. Patria sem **dec**, **oct** a **hex**, ktoré vykonávajú rovnaké činnosti ako funkcie **setf(ios::dec, ios::basefield)**, **setf(ios::oct, ios::basefield)** a **setf(ios::hex, ios::basefield)**. Manipulátory sú však stručnejšie. Hlavičkový súbor **<iostream>** obsahuje tiež manipulátory **ws**, **endl**, **flush** a ďalšiu množinu, uvedenú v tabuľke:

Manipulátor	Dôsledok
showbase **noshowbase**	Indikuje číselný základ (**dec**, **oct** alebo **hex**) pri tlači celočíselných hodnôt. Použitý formát môže čítať C++ kompilátor.
showpos **noshowpos**	Zobrazuje znamienko plus (+) pre kladné hodnoty.
uppercase **nouppercase**	Zobrazuje A-F pre hexadecimálne hodnoty a zobrazuje E pre vedecké hodnoty.
showpoint **noshowpoint**	Zobrazuje desatinnú bodku a koncové nuly pre čísla v pohyblivej rádovej bodke.

Manipulátor	Dôsledok
skipws **noskipws**	Preskakuje biele medzery na vstupe.
left **right** **internal**	Zarovnanie vľavo, doplnenie vpravo. Zarovnanie vpravo, doplnenie vľavo. Vyplňovanie medzi znamienkom alebo indikátorom základu a hodnotou.
scientific **fixed**	Indikuje prednostné zobrazovanie výstupu čísel v pohyblivej rádovej čiarke. (vedecká notácia vs. pevná desatinná bodka).

Manipulátory s argumentmi

Existuje šesť štandardných manipulátorov, ktoré akceptujú argumenty. Tieto sú definované v hlavičkovom súbore **<iomanip>** a sú to:

Manipulátor	Dôsledok
setiosflags (fmtflags n)	Ekvivalent volania **setf(n)**. Nastavenie zostáva v platnosti až po ďalšiu zmenu, napríklad **ios::setf()**.
resetiosflags(fmtflags n)	Maže iba formátovacie indikátory, dané hodnotou **n**. Nastavenie zostáva v platnosti až po ďalšiu zmenu, napríklad **ios::unsetf()**.
setbase(base n)	Mení základ na **n**, kde **n** je 10, 8 alebo 16. (Hocičo iné dáva 0.) Ak je **n** nulové, výstup bude so základom 10, ale vstup použije C konvencie: 10 je 10, 010 je 8 a 0xf je 15. Pre výstup by sme mohli tiež použiť **dec**, **oct** a **hex**.
setfill(char n)	Mení vyplňovací znak na **n**, ako to robí funkcia **ios::fill()**.
setprecision(int n)	Mení presnosť na **n**, rovnako ako `funkcia` **ios::precision()**.
setw(int n)	Mení šírku položky na **n**, rovnako ako **ios::width()**.

Ak robíme veľa formátovania, uvidíme ako použitie manipulátorov namiesto volania členských funkcií prúdu prečistí kód. Ako príklad je uvedený predchádzajúci príklad, prepísaný s použitím manipulátorov. Makro **D** bolo kvôli čitateľnosti vynechané.

```
// Použitie manipulátorov
#include <fstream>
#include <iomanip>
#include <iostream>
using namespace std;

int main() {
  ofstream trc("trace.out");
  int i = 47;
  float f = 2300114.414159;
  char* s = " Je tam este nieco?";

  trc << setiosflags(ios::unitbuf
           | ios::showbase | ios::uppercase
           | ios::showpos);
  trc << i << endl;
  trc << hex << i << endl
      << oct << i << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << resetiosflags(ios::showbase)
      << dec << setfill('0');
  trc << "Vyplnovaci znak: " << trc.fill() << endl;
  trc << setw(10) << i << endl;
  trc.setf(ios::right, ios::adjustfield);
  trc << setw(10) << i << endl;
  trc.setf(ios::internal, ios::adjustfield);
```

```
  trc << setw(10) << i << endl;
  trc << i << endl; // Bez setw(10)

  trc << resetiosflags(ios::showpos)
      << setiosflags(ios::showpoint)
      << "Presnost = " << trc.precision() << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << resetiosflags(ios::uppercase) << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;
  trc << setprecision(20);
  trc << "Presnost = " << trc.precision() << endl;
  trc << f << endl;
  trc.setf(ios::scientific, ios::floatfield);
  trc << f << endl;
  trc.setf(ios::fixed, ios::floatfield);
  trc << f << endl;
  trc << f << endl;

  trc << setw(10) << s << endl;
  trc << setw(40) << s << endl;
  trc.setf(ios::left, ios::adjustfield);
  trc << setw(40) << s << endl;
}
```

Vidíme, že mnoho viacnásobných príkazov sa zhustilo do jednej reťaze vkladaní. Všimnime si, volanie **setiosflags**, kde sa posielajú indikátory, sčítané bitovým OR. Toto by sme mohli urobiť i pomocou funkcie **setf** a **unsetf()** ako v predchádzajúcom príklade.

Keď použijeme **setw** vo výstupnom prúde, výstupný výraz sa formátuje do dočasného reťazca, ktorý sa v prípade potreby doplní vyplňujúcim znakom, čo sa určí porovnaním dĺžky formátovaného výsledku s argumentom manipulátora **setw**. Inými slovami, manipulátor **setw** ovplyvní *výsledný reťazec* formátovanej výstupnej operácie. Podobne použitie **setw** so vstupnými prúdmi má zmysel iba pri čítaní *reťazcov*, čo ozrejmuje nasledujúci príklad.

```
// Obmedzenia setw pre vstup
#include <cassert>
#include <cmath>
#include <iomanip>
#include <limits>
#include <sstream>
#include <string>
using namespace std;

int main() {
  istringstream is("jeden 2.34 sedem");
  string temp;
  is >> setw(2) >> temp;
  assert(temp == "na");
  is >> setw(2) >> temp;
  assert(temp == "e");
  double x;
  is >> setw(2) >> x;
  double relerr = fabs(x - 2.34) / x;
  assert(relerr <= numeric_limits<double>::epsilon());
}
```

Ak sa pokúšame čítať reťazec, **setw** bude riadiť počet vybraných znakov celkom pekne po bodku. Prvý výber zoberie dva znaky, ale druhý dostane len jeden, i keď žiadame dva. Je to spôsobené tým, že operátor >> používa biele medzery ako oddeľovač (pokiaľ nevypneme indikátor **skipws**). Avšak keď sa pokúsime čítať číslo, napríklad **x**, nemôžeme použiť **setw** na ohraničenie prečítaných znakov. Pre vstupné prúdy sa **setw** používa iba na výber reťazcov.

Tvorba manipulátorov

Občas by sme si radi vytvorili svoje vlastné manipulátory. Manipulátor bez argumentov, ako napríklad **endl**, je jednoduchá funkcia, ktorej argumentom je odkaz na **ostream** a vracia odkaz na **ostream**. Deklarácia **endl** je nasledovná:

```
ostream& endl(ostream&);
```

Teraz môžeme zapísať:

```
cout << "cau" << endl;
```

endl dáva ***adresu*** tejto funkcie. Takže kompilátor sa pýta, "Je tam funkcia, ktorú môžem volať a ktorá akceptuje adresu funkcie ako svoj argument?". Preddefinované funkcie v **<iostream>** to robia; nazývajú sa ***aplikátory*** (pretože aplikujú funkciu do prúdu). Aplikátor volá svoj funkčný argument, prenášajúc ho do objektu **ostream** ako svoj argument. Pri tvorbe vlastných manipulátorov nepotrebujeme vedieť ako aplikátory fungujú, potrebujeme len vedieť, že existujú. Sú však jednoduché. A tu je kód (zjednodušený) pre **ostream** aplikátor:

```
ostream &ostream::operator<<(ostream& (*pf)(ostream&))
{
  return pf(*this);
}
```

Skutočná definícia je o čosi komplikovanejšia, pretože zahrňuje šablóny, ale uvedený kód ilustruje spôsob. Keď funkciu ako je ***pf** (ktorá akceptuje prúdový parameter a vracia odkaz na prúd) vložíme do prúdu, zavolá sa tento aplikátor funkcie, ktorý vykoná funkciu, na ktorú ukazuje **pf**. Aplikátory pre **ios_base**, **basic_ios**, **basic_ostream** a **basic_istream** sú preddefinované v štandardnej C++ knižnici.

Na ilustráciu celého procesu je tu triviálny príklad, ktorý vytvára manipulátor **nl**, ktorý je ekvivalentný s vložením nového riadku do prúdu (t.j. nerobí sa vyprázdňovanie prúdu, ako to robí **endl**):

```
// Tvorba manipulátora
#include <iostream>
using namespace std;

ostream& nl(ostream& os) {
  return os << '\n';
}

int main() {
  cout << "nove riadky" << nl << "medzi" << nl
       << "kazdym" << nl << "slovom" << nl;
}
```

keď vložíme manipulátor **nl** do výstupného prúdu, akým je napríklad **cout**, vykoná sa nasledovná postupnosť volaní:

```
cout.operator<<(nl) ➔ nl(cout)
```

Výraz

```
os << '\n';
```

vo vnútri manipulátora **nl** zavolá funkciu **ostream::operator(char)**, ktorá samozrejme vracia prúd, ktorý sa nakoniec vracia z **nl**.

Efektory

Ako sme videli, manipulátory bez argumentov sa vytvárajú ľahko. Avšak čo keď potrebujeme vytvoriť manipulátor, ktorý má argumenty? Ak si pozrieme hlavičkový súbor **<iomanip>**, uvidíme typ, nazvaný **smanip**, čo je typ, ktorý vracajú manipulátory s argumentami. Mohlo by nás pokúšať použiť tento typ pre vlastné manipulátory, avšak nepodľahnime pokušeniu. Typ **smanip** je implementačne závislý, takže jeho použitie by nebolo prenositeľné. Našťastie vlastné manipulátory si môžeme definovať zrozumiteľným spôsobom bez nejakého špeciálneho aparátu, založeným na metóde, ktorá sa nazýva ***efektor*** a ktorú zaviedol Jerry Schwarz.[5] Efektor je jednoduchá trieda, ktorej konštruktor formátuje reťazec, reprezentujúci požadovanú operáciu spolu s preťaženým operátorom << na vkladanie reťazca do prúdu. A tu je príklad s dvomi efektormi. Prvý vysiela orezaný znakový reťazec do prúdu a druhý tlačí číslo v binárnom tvare.

```
// Efektory
#include <cassert>
#include <limits>  // pre max()
#include <sstream>
#include <string>
using namespace std;

// Vypis prefix retazca:
class Fixw {
     string str;
   public:
     Fixw(const string &s, int width) : str(s,0,width) {}
     friend ostream &operator<<(ostream& os, const Fixw& fw) {
       return os << fw.str;
     }
};

// Vytlac cislo v binarnom tvare:
typedef unsigned long ulong;
class Bin {
     ulong n;
   public:
     Bin(ulong nn) :n(nn) {};
     friend ostream& operator<<(ostream &os, const Bin &b) {
       const ulong ULMAX = numeric_limits<ulong>::max();
       ulong bit = ~(ULMAX >> 1); // Nastavenie najvyssieho bitu
       while(bit) {
         os << (b.n & bit ? '1' : '0');
         bit >>= 1;
       }
       return os;
    }
};

int main() {
  string words =
    "Things that make us happy, make us wise";
  for(int i = words.size(); --i >= 0;) {
    ostringstream s;
    s << Fixw(words, i);
    assert(s.str() == words.substr(0, i));
```

[5] Jerry Schwarz je konštruktér prúdov - iostreams.

```
  }
  ostringstream xs, ys;
  xs << Bin(0xCAFEBABEUL);
  assert(xs.str() ==
    "1100""1010""1111""1110""1011""1010""1011""1110");
  ys << Bin(0x76543210UL);
  assert(ys.str() ==
    "0111""0110""0101""0100""0011""0010""0001""0000");
}
```

Konštruktor triedy **Fixw** vytvára skrátenú kópiu argumentu typu **char *** a deštruktor uvoľňuje pamäť, alokovanú touto kópiou. Preťažený operátor << zoberie obsah svojho druhého argumentu, objektu triedy **Fixw**, vloží ho do prvého argumentu, objektu triedy **ostream**, a potom vráti **ostream**, takže sa dá používať v zreťazených výrazoch. Keď použijeme **Fixw** vo výraze ako je tento:

```
cout << Fixw(string, i) << endl;
```

vytvorí sa *dočasný objekt* volaním konštruktora triedy **Fixw** a tento dočasný objekt sa prenesie do operátora <<. Výsledkom je manipulátor s argumentami. Dočasný objekt triedy **Fixw** prežíva až do konca príkazu.

Efektor **Bin** sa spolieha na skutočnosť, že posun čísla bez znamienka doprava vsúva nuly do najvyšších bitov. Na vytvorenie hodnoty s nastaveným najvyšším bitom je použitý **numeric_limits<unsigned long>::max()** (najvyššia hodnota typu **unsigned long** zo štandardného hlavičkového súboru **<limits>**), a táto hodnota sa posúva cez číslo, ktorého sa to týka (jeho posunom doprava), maskujúc každý bit. Položili sme do kódu vedľa seba dva reťazcové literály kvôli čitateľnosti, oddelené reťazce sú kompilátorom samozrejme spojené do jedného.

Historicky problém tejto metódy spočíval v tom, že po vytvorení triedy nazvanej **Fixw** pre **char *** alebo **Bin** pre **unsigned long** nikto už nemohol vytvoriť inú triedu **Fixw** alebo **Bin** pre svoj typ. Avšak menopriestory (namespace) tento problém eliminovali.

Zhrnutie

Táto prednáška poskytuje dostatočne dôkladný úvod do iostream knižnice tried. Je to všetko, čo potrebujete na tvorbu programov s využitím prúdov. Niektoré vlastnosti prúdov sa nepoužívajú často, avšak zistíme ich štúdiom hlavičkových súborov alebo dokumentácie.