Лекція 26

Потоки

У цій лекції...

Найбільш яскравим утіленням принципів об'єктно-орієнтованого програмування є система вводу-виводу мови C++. У мові C++ співіснують дві системи вводу-виводу: процедурно-орієнтована, заснована на сімействах функцій `printf()` і `scanf()`, і об'єктно-орієнтована, в основі якої лежить ієрархія класів.

В усіх попередніх програмах ми використовували, як правило, систему вводу-виводу мови C, щоб не забігати наперед. Це — застаріла система. Утім, ніяких особливих недоліків вона не має, за винятком декількох обмежень, з якими ми зштовхнулися при описі стандартної бібліотеки шаблонів. Просто вона не є об'єктно-орієнтованою, і цього достатньо, щоб відмовитися від її на користь нової системи.

Опис засобів об'єктно-орієнтованого вводу-виводу розподілено по двох заголовках: `<iostream.h>` і `<iostream>`. Хоча вони майже однакові, заголовок `iostream` містить більш сучасні механізми реалізації вводу-виводу.

26.1. Класи потоків

В основі системи вводу-виводу мови C++ лежить поняття потоку, що ріднить її з процедурно-орієнтованою системою мови C. Нагадаємо, що потік являє собою абстракцію фізичного пристрою вводу-виводу і забезпечує однаковий інтерфейс роботи з ним. Основні операції вводу-виводу визначені в мудрій ієрархії класів, що міститься в заголовку `<iostream>`. Відзначимо одну тонкість: оскільки система вводу-виводу, власне кажучи, є символьною, а в мові C++ існують два види символів — звичайні (8 біт) і розширені (26 біт), класи вводу-виводу повинні враховувати цю обставину. Для того щоб не створювати дві рівносильні ієрархії для окремих видів символів, класи вводу-виводу оголошені шаблонними. Поняття шаблонного класу відноситься до парадигми узагальненого програмування, що ми розглянули в третій частині книги.

Бібліотека вводу-виводу розподілена по наступних заголовках.

`<iosfwd>`	Неповні оголошення класів
`<iostream>`	Основні засоби вводу-виводу
`<ios>`	Базові класи потоків вводу-виводу
`<streambuf>`	Засобу буферизації потоків

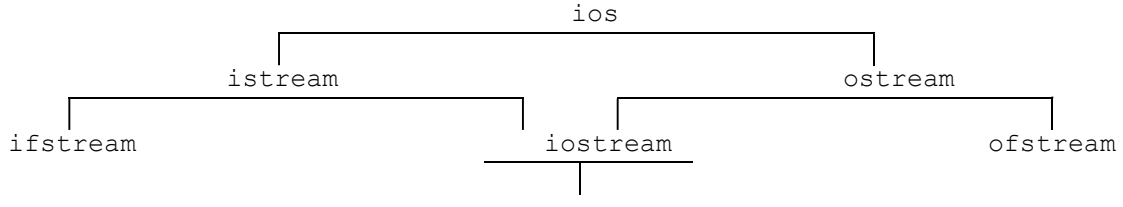
`<istream>`
`<ostream>`
`<iomanip>`
	Засобу форматування і маніпулятори

`<sstream>` `<cstdlib>`	Строкові потоки

`<fstream>`
`<cstdio>`
`<cwchar>`
	Файлові потоки

Система вводу-виводу мови C++ складається з двох ієрархій класів. В основі першої ієрархії лежить клас `ios_base`. Його єдиним спадкоємцем є шаблонний клас `basic_ios`, що містить визначення операцій вводу-виводу високого рівня. Клас `basic_ios` є базовим стосовно класів `basic_iostream`, `basic_ostream` і `basic_iostream`, що являють собою класи потоків вводу, виводу і вводу-виводу відповідно. Шаблонні класи, що входять у цю ієрархію, мають дві спеціалізації: для звичайних і для розширених символів.

Як відомо, бібліотека вводу-виводу створює двох спеціалізацій шаблонних класів, що входять в ієрархію: одну — для восьмибітових символів, а іншу — для розширених. Нижче приводиться список імен шаблонних класів, призначених для вводу-виводу звичайних і розширених символів. Це породжує дві рівносильні ієрархії спеціалізованих класів: 1) `ios` → `istream`, `ostream` → `ifstream`, `iostream`, `ofstream` (для звичайних символів), 2) `wios` → `wistream`, `wostream` → `wifstream`, `wiostream`, `wfstream` (для розширених символів).



Ієрархія шаблонних класів вводу-виводу звичайних класів

У мові C++ передбачено вісім стандартних потоків, що автоматично відкриваються при запуску будь-якої програми. Перші чотири потоки призначені для вводу-виводу звичайних символів: `cin` — потік вводу, `cout` — потік виводу, `cerr` — потік помилок, `clog` — буферизований потік помилок. Друга четвірка потоків дозволяє вводити і виводити розширені символи. Вони називаються `win`, `wout`, `werr` і `wlog`.

За замовчуванням потік `cin` зв'язаний із клавіатурою, а інші потоки — з екраном. При бажанні потоки вводу-виводу можна зв'язати з іншими пристроями (наприклад, чи принтером портом).

У мові C++ існує два способи форматування вводу-виводу: 1) за допомогою членів класу `ios` і 2) за допомогою *маніпуляторів*.

26.2. Прапорці форматування

Форматування потоків вводу-виводу за допомогою прапорців здійснюється за допомогою членів класу `ios` — спеціалізації шаблонного класу `basic_ios` для звичайних символів. Ім'я `ios` визначається в заголовку `<iosfwd>` у такий спосіб.

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

У свою чергу, шаблонний клас `basic_ios` є похідним від класу `ios_base`. Отже, щоб розглянути члени класу `ios`, необхідно звернутися до визначення класу `ios_base`, а потім описати члени класу `basic_ios`.

26.2.1. Клас ios_base

Розглянемо визначення класу `ios_base`, що лежить в основі ієрархії класів вводу-виводу.

```
namespace std {
class ios_base
{
public:
   class failure
       : public exception
   {
   public:
       explicit failure(const string& msg);
       virtual ~failure();
       virtual const char* what() const throw();
   };

   typedef T1 fmtflags;
   static const fmtflags boolalpha;
   static const fmtflags dec;
   static const fmtflags fixed;
   static const fmtflags hex;
   static const fmtflags internal;
   static const fmtflags left;
   static const fmtflags oct;
   static const fmtflags right;
   static const fmtflags scientific;
   static const fmtflags showbase;
   static const fmtflags showpoint;
   static const fmtflags showpos;
   static const fmtflags skipws;
   static const fmtflags unitbuf;
   static const fmtflags uppercase;
   static const fmtflags adjustfield;
   static const fmtflags basefield;
   static const fmtflags floatfield;

   typedef T2 iostate;
   static const iostate badbit;
   static const iostate eofbit;
   static const iostate failbit;
   static const iostate goodbit;
```

```
   typedef T3 openmode;
   static const openmode app;
   static const openmode ate;
   static const openmode binary;
   static const openmode in;
   static const openmode out;
   static const openmode trunc;

   typedef T4 seekdir;
   static const seekdir beg;
   static const seekdir cur;
   static const seekdir end;

   class Init {
    public:
      Init();
     ~Init();
    private:
      static int init_cnt;
    };

   // Прапори формату
   fmtflags flags() const;
   fmtflags flags(fmtflags fmtfl);
   fmtflags setf(fmtflags fmtfl);
   fmtflags setf(fmtflags fmtfl, fmtflags mask);
   void unsetf(fmtflags mask);

   streamsize precision() const;
   streamsize precision(streamsize prec);
   streamsize width() const;
   streamsize width(streamsize wide);

      // Засобу локалізації
   locale imbue(const locale& loc);
   locale getloc() const;

   // Засобу збереження данных
   static int xalloc();
   long&  iword(int index);
   void*& pword(int index);

   virtual ~ios_base();

   enum event { erase_event, imbue_event, copyfmt_event };
   typedef void (*event_callback)(event, ios_base&, int index);
   void register_callback(event_callback fn, int index);

   static bool sync_with_stdio(bool sync = true);

protected:
   ios_base();
private:
   static int index;
   long* iarray;
   void** parray;
};
}
```

Примітка. Типи `T1–T4` залежать від конкретної реалізації компілятора. Звичайно вони являють собою перерахування. Масиви `iarray` і `parray` можуть бути реалізовані у виді розподіленої структури.

26.2.1. Клас failure

У класі `ios_base` визначений вкладений клас `failure`, що є похідним від класу `exception`. Клас `failure` є базовим для всіх об'єктів, що генеруються функціями вводу-виводу при виникненні помилок під час виконання операцій над буферизованими потоками.

Конструктор `failure(const string& msg)` створює об'єкт класу `failure`, ініціалізуючи його рядком `msg`. Віртуальний деструктор `~failure()` знищує об'єкт класу `failure`, що був згенерований, а віртуальна константна функція `what()` визначає зміст рядка `msg`, що описує виниклу виняткову ситуацію.

26.2.2. Клас Init

Вкладений клас `Init` забезпечує створення восьми об'єктів стандартних потоків. Конструктор `Init()` створює об'єкт класу `Init`, ініціалізуючи об'єкти `cin`, `cout`, `cerr`, `clog`, `wcin`, `wcout`, `wcerr` і `wclog`. Об'єкт `cin` є екземпляром класу `istream`, а об'єкти `cout`, `cerr` і `clog` належать класу `ostream`. Аналогічно об'єкт `wcin` є екземпляром класу `wistream`, а об'єкти `cout`, `cerr` і `clog` належать класу `wostream`. Після створення об'єктів лічильник `init_cnt` збільшується на одиницю.

Деструктор `Init()` знищує об'єкт класу `Init` і віднімає з лічильника одиницю. Якщо в результаті лічильник містить одиницю, деструктор викликає функції `cout.flush()`, `cerr.flush()`, `clog.flush()`, `wcout.flush()`, `wcerr.flush()` і `wclog.flush()`.

26.2.3. Бітова маска ios_base::fmtflags

Тип `fmtflags` являє собою бітову маску, що складається з елементів, що є прапорцями форматування (табл. 26.2).

Таблиця 26.2. Елементи бітової маски fmtflags

Ім'я	Значення	Величина
`boolalpha`	Ввести і вивести булеве значення за абеткою	`1<<0 == 0x0001`
`dec`	Перед вводом чи виводом привести ціле число до десяткового виду	`1<<1 == 0x0002`
`fixed`	Вивести десяткове число з фіксованою крапкою	`1<<2 == 0x0004`
`hex`	Перед вводом чи виводом привести ціле число до шістнадцатерічному вигляді	`1<<3 == 0x0008`
`internal`	Вставити пробіли між знаком числа і його першою цифрою, щоб заповнити усі поле виводу. Якщо прапор не встановлений, зробити вирівнювання по правому краї	`1<<4 == 0x0010`
`left`	Зробити вирівнювання по лівому краї	`1<<5 == 0x0020`
`oct`	Перед вводом чи виводом привести ціле число до восьмеричного виду	`1<<6 == 0x0040`
`rigth`	Зробити вирівнювання по правому краї	`1<<7 == 0x0080`
`scientific`	Вивести десяткове число в науковому форматі	`1<<8 == 0x0100`
`showbase`	Вивести підстава системи числення	`1<<9 == 0x0200`
`showpoint`	Вивести десяткову точку	`1<<10 == 0x0400`
`showpos`	Вивести знак перед ненегативними числами	`1<<11 == 0x0800`
`skipws`	Ігнорувати роздільники	`1<<12 == 0x1000`
`unitbuf`	Очистити буфер після кожної операції виводу	`1<<13 == 0x2000`
`uppercase`	Вивести малі літери як прописні	`1<<14 == 0x4000`

Крім того, бітова маска `fmtflags` містить несколько констант, що являють собою комбінацію декількох елементів (табл. 26.2).

Таблиця 26.2. Константи бітової маски `fmtflags`

Ім'я	Значення
`adjustfield`	`left | right | internal`
`basefield`	`dec | oct | hex`
`floatfield`	`scientific | fixed`

26.2.4. Бітова маска ios_base::iostate

Тип `iostate` являє собою бітову маску, що складається з наступних елементів (табл. 26.3). Цей набір битів описує стан потоку.

Таблиця 26.3. Елементи бітової маски `iostate`

Ім'я	Значення	Величина
`goodbit`	Помилок немає	0x00
`badbit`	Спроба виконати некоректну операцію	0x04
`eofbit`	При уведенні виявлений кінець файлу	0x01
`failbit`	Помилка при виконанні операції чи вводу виводу	0x02

26.2.5. Бітова маска ios_base::openmode

Тип `iostate` являє собою бітову маску, що складається з наступних елементів (табл. 26.4). Цей набір бітових прапорців визначає режим використання потоку.

Таблиця 26.4. Елементи бітової маски openmode

Ім'я	Значення	Величина
`app`	Перед кожним виконанням операції запису переводити курсор у кінець потоку	0x08
`ate`	Відкрити потік і перевести курсор у кінець	0x04
`binary`	Виводити дані в двійковому вигляді	0x80
`in`	Відкрити потік для читання	0x01
`out`	Відкрити потік для запису	0x02
`trunc`	Усікання існуючого потоку	0x10

26.2.6. Бітова маска ios_base::seekdir

Тип `seekdir` являє собою бітову маску, що складається з елементів, що дозволяють установити позицію курсору в потоці (табл. 26.5).

Таблиця 26.5. Елементи бітової маски seekdir

Ім'я	Значення	Величина
`beg`	Відраховувати зсув курсору від початку потоку	0
`cur`	Відраховувати зсув курсору від поточної позиції	1
`end`	Відраховувати зсув курсору від кінця потоку	2

26.2.7. Перерахування event

- `enum event { erase_event, imbue_event, copyfmt_event };`

Визначає типи подій, що можуть відбутися при введенні-виводу.

26.2.8. Вказівник iarray

- `long* iarray;`

Посилається на перший елемент допоміжного масиву типу `long`, що має довільну довжину. Використовується для закритого застосування в програмі.

26.2.9. Вказівник parray

- `void** parray;`

Посилається на перший елемент допоміжного масиву вказівників, що має довільну довжину. Використовується для закритого застосування в програмі.

26.2.10. Функції-члени

Керування прапорцями форматування здійснюється наступними функціями-членами.

- `fmtflags flags() const;`
- `fmtflags flags(fmtflags` *прапор*`) const;`

Перша версія повертає поточний стан бітової маски. Друга версія привласнює параметру *прапор* поточне стан бітової маски.

- `fmtflags setf(fmtflags` *`прапор`*`)`
- `fmtflags setf(fmtflags` *`прапор1`*`, fmtflags` *`маска`*`)`

Установлює новий стан бітової маски *прапор*. Повертає попередній стан бітової маски. Друга версія привласнює бітовій масці форматування значення `fmtf1 & mask`.

- `void unsetf(fmtflags` *`маска`*`);`

Скидає в поточній бітовій масці форматування біти, задані параметром *маска*.

- `streamsize precision() const;`

Задає кількість цифр після десяткової крапки, які варто виводити при записі десяткового числа.

- `streamsize width() const;`
- `streamsize width(streamsize` *`ширина`*`);`

Перша версія визначає поточну мінімальну ширину поля виводу. Друга версія встановлює нову мінімальну ширину поля виводу.

- `locale imbue(const locale` *`локалізація`*`);`

Установлює нові національні особливості, повертаючи попередні.

- `locale getloc() const;`

Повертає поточні національні особливості.

- `static int xalloc();`

Виділяє пам'ять для цілого числа і вказівника, ініціалізуючи їх нулем, а також визначає позицію, до якої можуть звертатися функції `iword()` і `pword()`.

- `long&  iword(int index);`

Якщо вказівник `iarray` є нульовим, функція `iword()` розміщає в пам'яті масив типу `long` невизначеної довжини і зберігає адресу його першого елемента в вказівнику `iarray`. Потім при кожнім виклику функції `iword()` довжина масиву збільшується на одиницю й в останній осередок записується нуль.

- `void*& pword(int index);`

Якщо вказівник `parray` є нульовим, функція `pword()` розміщає в пам'яті масив вказівників типу `void` невизначеної довжини і зберігає адресу його першого елемента в вказівнику `iarray`. Потім при кожнім виклику функції `pword()` довжина масиву збільшується на одиницю й в останній осередок записується нульова константа.

- `virtual ~ios_base();`

Знищує об'єкт класу `ios_base`. Викликає функцію, зареєстровану для цієї події: `(*`*`fn`*`)(erase_event, *this, index)`.

- `typedef void (*event_callback)(event, ios_base&, int index);`

Посилається на функцію, що зареєстрована для виклику при настанні визначеної події.

- `void register_callback(event_callback fn, int index);`

Реєструє пари `(fn, index)`, щоб при викликах функцій `imbue()`, `copytmt()` чи деструктора `~ios_base` функція `fn()` викликалася з аргументом `index`. Функція `fn()` не повинна генерувати виняткові ситуації. Якщо функція двічі зареєстрована з однаковим аргументом, вона буде викликана двічі.

- `static bool sync_with_stdio(bool sync = true);`

Повертає значення `true`, якщо об'єкти стандартних потоків вводу-виводу синхронізовані. У противному випадку повертає значення `false`. З одного боку, виклик функції `sync_with_stdio()` дозволяє використовувати загальні буфери при введенні-виводу в стандартні потоки мов C и C++. З іншого боку, стандартні потоки мов C и C++ можна роз'єднати, викликавши функцію з параметром `false`.

26.2.11.Клас basic_ios

Клас `basic_ios` є базовим стосовно класів `basic_istream` і `basic_ostream`. Він керує форматним вводом і виводом, задаючи інформацію про систему числення, точності й інших параметрів. Крім того, клас `basic_ios` є шаблонним. Його синопсис виглядає в такий спосіб.

```
  namespace std {
template <class charT, class traits = char_traits<charT> >
class basic_ios :  public ios_base {
...
  };
  }
```

26.2.12. Конструктор

У класі передбачені три конструктори.

- `template <class charT, class traits = char_traits<chatT> > explicit basic_ios(basic_streambuf<charT,traits>* sb);`
- `basic_ios();`
- `basic_ios(const basic_ios& );`

Створює об'єкт класу `basic_ios`. Для ініціалізації членів об'єкта викликається функція `init(sb)`. Друга версія є захищеної і не передбачає ініціалізації об'єктів. Третя версія є закритої і забезпечує ініціалізацію об'єкта зазначеним аргументом.

26.2.13. Функції-члени

Керування параметрами буфера здійснюється наступними функціями-членами.

- `template <class char, class traits = char_traits<chat> > void init(basic_streambuf<char,traits>* sb);`

Виклик функції `init()` приводить до наступного установкам початкових параметрів буфера (табл. 26.6).

Таблиця 26.6. Початкові значення параметрів буфера

Член класу	Значення
`rdbuf()`	`sb`
`tie()`	`0`
`rdstate()`	`goodbit`, якщо `sb` — не нульовий вказівник, у противному випадку `badbit`
`exceptions()`	`goodbit`
`flags()`	`skipws | dec`
`wirth()`	`0`
`precision()`	`6`
`fill()`	`widen(' ');`
`getloc()`	Копія значення, повернутого функцією `locale()`
`iarray`	Нульовий вказівник
`parray`	Нульовий вказівник

- `template <class charT, class traits = char_traits<chatT> > basic_ostream<charT,traits>* tie() const;`
- `template <class charT, class traits = char_traits<chatT> > basic_ostream<charT,traits>* tie(basic_ostream<charT,traits>* tiestr);`

Установлює (синхронізує) і розриває зв'язку між потоками вводу і виводу. Повертає вказівник на зв'язаний потік. При виклику `tie(0)` зв'язок між потоками розривається.

- `template <class charT, class traits = char_traits<chatT> > basic_streambuf<charT,traits>* rdbuf() const;`
- `template <class charT, class traits = char_traits<chatT> > basic_streambuf<charT,traits>* rdbuf(basic_streambuf<charT,traits>* sb);`

Повертає вказівник на буферізований потік `streambuf`, зв'язаний з поточним потоком. Друга версія викликає функцію `clear()`, установлює потік у стан, заданий параметром `sb`, і повертає попередній стан потоку.

- `basic_ios& copyfmt(const basic_ios& rhs);`

Привласнює членам об'єкта `*this` значення відповідних членів об'єкта `rhs`. Буфер потоку `rdbuf()` і стан буфера `rdstate()` залишаються незмінними.

- `char_type fill() const;`
- `char_type fill(char_type ch);`

Перша версія повертає поточний символ-заповнювач. Друга версія, крім цього, установлює новий символ-заповнювач.

- `operator void*() const`

Якщо `fail()==true`, повертає нульовий вказівник, у противному випадку — ненульовий вказівник.

- `bool operator!() const`

Повертає значення `fail()`.

- `iostate rdstate() const;`

Повертає стан потоку у виді набору битів.

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

Якщо `rdbuf()!=0`, то `state==rdstate()`, у противному випадку `rdstate()==state|ios_base::badbit`. Інакше кажучи, якщо потік не містить помилок, аргументу `state` привласнюється поточне стан потоку. Якщо потік містить помилки, генерується виняткова ситуація `fail`, що належить класу `basic_ios::failure`.

- `void setstate(iostate state);`

Викликає функцію `clear()` з аргументом `rdstate()|state`, додаючи в стан потоку біти, встановлені в наборі `state`.

- `bool good() const;`

Повертає значення логічного вираження `rdstate()==0`. Інакше кажучи, перевіряє, ли містить стан потоку ознаки помилок.

- `bool eof() const;`

Повертає значення `true`, якщо в поточному стані потоку `rdstate()` установлений біт `eofbit`.

- `bool fail() const;`

Повертає значення `true`, якщо в поточному стані потоку `rdstate()` установлені біти `eofbit` чи `badbit`.

- `bool bad() const;`

Повертає значення `true`, якщо в поточному стані потоку `rdstate()` установлений біт `badbit`.

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

Повертає стан виняткової ситуації, зв'язаної з потоком. Друга версія встановлює новий стан виняткової ситуації, тобто біти `eofbit`, `bitset` і `failbit`.

- `locale imbue(const locale& loc);`

Викликає функція-член `ios_base::imbue(lok)`. Якщо стан потоку не містить помилок, викликається функція `rdbuf()- >pubimbue(loc)` із класу `basic_streambuf`.

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

Перетворення символу типу `char_type` у звичайний символ типу `char`.

- `char_type widen(char c) const;`

Перетворення звичайного символу типу `char` у символ типу `char_type`.

Отже, ми розглянули два класи, що лежать в основі ієрархії потоків. Однак у програмах ми, як правило, звертаємося до об'єктів стандартних потоків `cin`, `cout` і т.д., що є екземплярами класів `istream` і `ostream`. Ці класи являють собою спеціалізації шаблонних класів `basic_istream` і `basic_ostream` для звичайних символів. Отже, необхідно розглянути визначення класів `basic_istream` і `basic_ostream`.

26.3. Клас basic_istream

Визначення класу `basic_istream` дано в заголовку `<istream>`. Цей клас забезпечує створення двох стандартних потоків вводу: `cin` і `wcin`. Синопсис шаблонного класу `basic_istream` виглядає в такий спосіб.

```
   namespace std {
template <class charT, class traits = char_traits<charT> >
class basic_istream :  virtual public basic_ios<charT, traits> {
...
   };
   }
```

26.3.1. Конструктор

Клас містить тільки один варіант конструктора.

- `template <class char, class traits = char_traits<chat> >`
  `explicit basic_istream(basic_streambuf<char,traits>* sb);`

Створює об'єкт класу `basic_istream`, ініціалізуючи його члени за допомогою функції `basic_ios::init(sb)`.

26.3.2.Деструктор

У класі передбачений віртуальний деструктор.

- `virtual ~basic_istream();`

Знищує об'єкт класу `basic_istream`.

26.3.3. Вкладений клас sentry (“вартовий”)

Для контролю за порядком виконання операцій над потоком призначений клас `sentry`.

```
  // Префікс/суфікс
 template <class char, class traits = char_traits<chat> >
 class sentry
 {
 public:
    explicit sentry(basic_istream<char,traits>& is,
                    bool noskipws = false);
    ~sentry();
    operator bool() const;
 private:
    sentry(const sentry&);
    sentry& operator=(const sentry&);
};
```

Забезпечує правильну послідовність операцій над потоком. Наприклад, операція, що повинна виконуватися першої, реалізується конструктором класу `sentry`, а операція, що повинна виконуватися останньої, — деструктором.

26.3.4. Функції-члени

Операції над потоком виконуються наступними функціями-членами.

- `basic_istream& operator>>(basic_istream& (*pf)(basic_istream&));`
-
```
template <class charT, class traits = char_traits<chatT> >
basic_istream& operator>>(basic_ios<charT,traits>&
                          (*pf)(basic_ios<charT,traits>&));
```
- `basic_istream& operator>>(ios_base& (*pf)(ios_base&));`
- `basic_istream& operator>>(bool& n);`
- `basic_istream& operator>>(short& n);`
- `basic_istream& operator>>(unsigned short& n);`
- `basic_istream& operator>>(int& n);`
- `basic_istream& operator>>(unsigned int& n);`
- `basic_istream& operator>>(long& n);`
- `basic_istream& operator>>(unsigned long& n);`
- `basic_istream& operator>>(float& f);`
- `basic_istream& operator>>(double& f);`
- `basic_istream& operator>>(long double& f);`
- `basic_istream& operator>>(void*& p);`
- `basic_istream& operator>>(basic_streambuf<char_type,traits>* sb);`

Ці функції виконують операцію виводу даних убудованих типів з потоку вводу. Роздільники ігноруються. Формат вводу для кожного типу визначається станом потоку, заданим чи користувачем передбаченим за замовчуванням.

- `streamsize gcount() const;`

Повертає кількість символів, лічених при виконанні останньої операції бесформатного вводу.

- `int_type get();`

Вивідає з потоку вводу символ `c`. У випадку невдачі викликає функцію `setstate(failbit)`, здатну генерувати виняткову ситуацію `ios_base::failure`.

- `basic_istream& get(char_type& c);`

Вивідає з потоку вводу символ `c`. У випадку невдачі викликає функцію `setstate(failbit)`, здатну генерувати виняткову ситуацію `ios_base::failure`.

- `basic_istream& get(char_type* s, streamsize n);`

Послідовно вивідає символи з потоку вводу, розміщаючи їх у масиві, на якому посилається вказівник `s`. Символи зчитуються, поки не відбудеться одне з двох подій.

1. Уведені `n-1` символів.
2. Виявлено кінець файлу (викликається функція `setstate(eofbit)`).

- `basic_istream& get(char_type* s, streamsize n, char_type delim);`

Послідовно вивідає символи з потоку вводу, розміщаючи їх у масиві, на якому посилається вказівник `s`. Символи зчитуються, поки не відбудеться одне з трьох подій.

1. Уведені `n-1` символів.
2. Виявлено кінець файлу (викликається функція `setstate(eofbit)`).
3. Виявлено роздільник.

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

Вивідає символи з потоку вводу і поміщає їх у потік виводу `sb`. Символи зчитуються, поки не відбудеться одне з чотирьох подій.

1. Уведені `n-1` символів.
2. Виявлено кінець файлу (викликається функція `setstate(eofbit)`).
3. Виявлено роздільник.
4. Виникла виняткова ситуація.

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

Вивідає символи з потоку вводу і поміщає їх у потік виводу `sb`. Символи зчитуються, поки не відбудеться одне з чотирьох подій.

1. Уведені `n-1` символів.
2. Виявлено кінець файлу (викликається функція `setstate(eofbit)`).
3. Виявлено роздільник.
4. Виникла виняткова ситуація.

- `basic_istream& getline(char_type* s, streamsize n);`

Вивідає символи з потоку вводу і поміщає їх у масив, на який посилається вказівник `s`. Символи зчитуються, поки не відбудеться одне з чотирьох подій.

1. Уведені `n-1` символів.
2. Виявлено кінець файлу (викликається функція `setstate(eofbit)`).
3. Виявлено роздільник.
4. Виникла виняткова ситуація.

- `basic_istream& getline(char_type* s, streamsize n, char_type delim);`

Вивідає символи з потоку вводу і поміщає їх у масив, на який посилається вказівник `s`. Символи зчитуються, поки не відбудеться одне з чотирьох подій.

1. Уведені `n-1` символів.
2. Виявлено кінець файлу (викликається функція `setstate(eofbit)`).
3. Виявлено роздільник.
4. Виникла виняткова ситуація.

- `basic_istream& ignore(streamsize n = 1, int_type delim = traits::eof());`

Вивідає символ з потоку і відкидає його, поки не відбудеться одне з наступних подій.

1. Кількість символів перевищує максимально припустиме значення `numeric_limits<int>::max()`.
2. Виявлено кінець файлу.
3. Виявлено роздільник.

- `int_type peek();`

Переглядає наступний символ, що підлягає введенню.

- `basic_istream& read(char_type* s, streamsize n);`

Якщо стан потоку містить ознаки помилок, викликає функцію `setstate(failbit)` і повертає керування. У противному випадку вивідає символи з вхідного потоку і розміщає їх у масиві, на якому посилається вказівник `s`. Вводу не припиняється, поки не відбудеться одне з наступних подій.

1. Лічено `n` символів.
2. Виявлено кінець файлу (викликається функція `setstate(failbit|eofbit)`).

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

Якщо стан потоку містить ознаки помилок, викликає функцію `setstate(failbit)` і повертає керування. У противному випадку вивідає символи з вхідного потоку і розміщає їх у масиві, на якому посилається вказівник `s`. Визначає, скільки символів доступно в потоці вводу, і зчитує не більш `n` символів.

- `basic_istream& putback(char_type c);`

Повертає символ `c` у буфер.

- `basic_istream& unget();`

Повертає в буфер останній лічений символ.

- `int sync();`

Синхронізує потік вводу і буфер. У випадку невдачі викликає функцію `setstate(badbit)`.

- `pos_type tellg();`

Повертає поточну позицію курсору в потоці вводу.

- `basic_istream& seekg(pos_type);`
- `basic_istream& seekg(off_type, ios_base::seekdir);`

Установлює курсор потоку в задану позицію. Друга версія дозволяє вказати крапку для відліку зсуву.

- ```
  template<class charT, class traits>
  basic_istream<charT,traits>&
  operator>>(basic_istream<charT,traits>& in, charT& c);
  ```

- ```
  template<class traits>
  basic_istream<char,traits>&
  operator>>(basic_istream<char,traits>& in, unsigned char& c);
  ```

- ```
  template<class traits>
  basic_istream<char,traits>&
  operator>>(basic_istream<char,traits>& in, signed char& c);
  ```

Виконує вивід символу з потоку вводу `in` після створення об'єкта класу `sentry`. Символ записується в перемінну `c`. У випадку невдачі виконується виклик функції `in.setstate(failbit)`.

- ```
  template<class charT, class traits>
  basic_istream<charT,traits>&
  operator>>(basic_istream<charT,traits>& in, charT* s);
  ```

- ```
  template<class traits>
  basic_istream<char,traits>&
  operator>>(basic_istream<char,traits>& in, unsigned char* s);
  ```

- ```
  template<class traits>
  basic_istream<char,traits>&
  operator>>(basic_istream<char,traits>& in, signed char* s);
  ```

Виконує вивід символів з потоку вводу `in` після створення об'єкта класу `sentry`. Символ записується в масив, на який посилається вказівник `s`. У випадку невдачі виконується виклик функції `in.setstate(failbit)`. Якщо значення `width()` не равно нулю, то кількість символів, що подлежащих уведенню, равно `n=width()`. У противному випадку `n` равно максимально можливій кількості елементів, що можуть зберігатися в найбільшому масиві типу `char_type`. Вводу символів припиняється, поки не відбудеться одне з наступних подій.

1. Уведені `n-1` символів.
2. Виявлено кінець потоку вводу.
3. Виявлено пробіл.
4. Виявлено нульовий байт.

- ```
  template <class char, class traits>
  basic_istream<char,traits>&
  ws(basic_istream<char,traits>& is);
  ```

Вивідає символи з вхідного потоку, поки не знайде чи роздільник кінець запису. В останньому випадку встановлюється біт `eofbit`, а не `failbit`.

26.4. Клас basic_ostream

Визначення класу `basic_ostream` дано в заголовку `<ostream>`. Цей клас забезпечує створення двох стандартних потоків виводу: `cout` і `wcout`. Синопсис шаблонного класу `basic_ostream` виглядає в такий спосіб.

```
   namespace std {
template <class charT, class traits = char_traits<charT> >
class basic_ostream :  virtual public basic_ios<charT, traits> {
...
   };
   }
```

26.4.1. Конструктор

У класі передбачений только один варіант конструктора.

- `explicit basic_ostream(basic_streambuf<char_type,traits>* sb);`

Створює об'єкт класу `basic_ostream`, ініціалізуючи члени базового класу за допомогою виклику функції `basic_ios<char,traits>::init(sb)`.
```

26.4.2.Деструктор

Клас містить оголошення віртуального деструктора.

- `virtual ~basic_ostream();`

Знищує об'єкт класу `basic_ostream`.

26.4.3. Вкладений клас sentry (“вартовий”)

Послідовність операцій над потоком контролюється класом `sentry`.

```
// Префікс/суфікс
template <class char, class traits = char_traits<chat> >
 class basic_ostream<chatT,traits>::sentry{
bool ok_;
public:
  explicit sentry(basic_ostream<charT,traits>& os);
  ~sentry();
  operator bool() const {return ok_};
private:
  sentry(const sentry&);
  sentry& operator=(const sentry&);
};
```

Забезпечує правильну послідовність операцій над потоком. Якщо значення `os.good()` не равно нулю, конструктор класу `sentry` підготовляє потік до виводу даних. Якщо значення `os.tie()` не є нульовим вказівником, викликається функція `os.tie()- >flush()`. Якщо в процесі підготовки потоку виникають проблеми, конструктор може викликати функцію `setstate(failbit)`, що може генерувати виняткову ситуацію `ios_base::failure`.

26.4.4. Функції-члени

Операції над потоком виконуються наступними функціями-членами.

- `operator bool() const;`

Повертає значення `ok_`.

- `pos_type tellp();`

Повертає поточну позицію курсору в потоці виводу.

- `basic_ostream& seekp(pos_type);`
- `basic_ostream& seekp(off_type, ios_base::seekdir);`

Установлює курсор потоку в задану позицію. Друга версія дозволяє вказати крапку для відліку зсуву.

- `basic_ostream& operator<<(bool n);`
- `basic_ostream& operator<<(short n);`
- `basic_ostream& operator<<(unsigned short n);`
- `basic_ostream& operator<<(int n);`
- `basic_ostream& operator<<(unsigned int n);`
- `basic_ostream& operator<<(long n);`
- `basic_ostream& operator<<(unsigned long n);`
- `basic_ostream& operator<<(float f);`
- `basic_ostream& operator<<(double f);`
- `basic_ostream& operator<<(long double f);`

Виконує операцію вставки даних у потік вводу. Типи даних можуть бути убудованими і визначатися користувачем. Роздільники ігноруються. Формат вводу для кожного типу визначається станом потоку, заданим чи користувачем передбаченим за замовчуванням.

-
```
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);`

Шаблонні функції виконують операції вставки символу `c` у потік `out`, після того як буде створений об'єкт класу `sentry`. Якщо потік містить розширені символи, аргумент `c` перетвориться в тип `wchar_t`. Поля виводу доповнюються символом-заповнювачем.

- `template<class charT, class traits>`
  `basic_ostream<charT,traits>&`
  `operator<<(basic_ostream<charT,traits>& out, 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);`

Шаблонні функції виконують операції вставки рядка `s` у потік `out`, після того як буде створений об'єкт класу `sentry`. Якщо потік містить розширені символи, аргумент `c` перетвориться в тип `wchar_t`. Поля виводу доповнюються символом-заповнювачем.

- `basic_ostream& put(char_type c);`

Вставляє в потік виводу символ `c`. У випадку невдачі виконує виклик функції `setstate(badbit)`, що може генерувати виняткову ситуацію `ios_base::failure`.

- `basic_ostream& write(const char_type* s, streamsize n);`

Вставляє в потік виводу символи, вивіднуті з рядка `s`. Запис припиняється, якщо відбулося одне з наступних подій. Повертає вказівник на потік виводу.

1. Виведені `n` символів.
2. Виникла виняткова ситуація (викликана функція `setstate(badbit)`).

- `basic_ostream& flush();`

- `template <class char, class traits>`
  `basic_ostream<char,traits>&`
  `flush(basic_ostream<char,traits>& os);`

Очищає буфер, вивантажуючи дані на фізичний пристрій.

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

Маніпулятор `endl()` виконує операцію `os.put(os.widen('\n'))`, а потім очищає буфер, викликаючи функцію `os.flush()`.

- `template <class char, class traits>`
  `basic_ostream<char,traits>&`
  `ends(basic_ostream<char,traits>& os);`

Очищає потік, викликаючи функцію `os.flush()`.

26.4.5. Клас basic_iostream

Клас `basic_iostream` є похідним від класів `basic_istream` і `basic_ostream`. Його опис виглядає так.

```
  template <class char, class traits = char_traits<char> >
class basic_iostream
   : public basic_istream<char,traits>,
     public basic_ostream<char,traits>
{
public:
   explicit basic_iostream(basic_streambuf<char,traits>* sb);
   virtual ~basic_iostream();
};
```

Класи `iostream` являють собою спеціалізацію шаблонного класу `basic_iostream<char>`. Аналогично клас `wiostream` є спеціалізацією шаблонного класу `basic_iostream<wchar_t>`. Обидві спеціалізації з'являються в заголовку `<iostream>`.

```
  typedef basic_iostream<char>    iostream;
  typedef basic_iostream<wchar_t> wiostream;
```

26.4.6. Застосування прапорців форматування

Розглянемо несколько прикладів, що ілюструють застосування прапорців форматування. Їхній зміст не вимагає додаткових коментарів. Нагадаємо, що до всіх полів і функцій-членів класів `basic_ios`, `basic_istream` і `basic_stream` можна звернутися через об'єкт стандартного потоку `cin`, `cout` чи інший об'єкт. Крім того, можливий прямий доступ до членів класу.

Установка прапорців форматування

```
  #include <iostream>

using namespace std;

int main()
{
   // Одержання поточного стану формату
   ios_base::fmtflags first = cout.flags();

   // Установка окремих битів за допомогою ЧИ операції
   first = ios::hex | ios::showbase;

   // Зміна поточного стану формату
   cout.flags(first);

   // Вивід цілого числа в шістнадцатерічному виді
   // із указівкою підстави системи числення

   cout << 1 << endl;

   // Установка окремих битів за допомогою функції setf()
   cout.setf(ios::dec);                                  // Один біт
   cout.setf(ios::showpos, ios::unitbuf);                // Два бита
   cout.setf(ios::dec | ios::showpos | ios::unitbuf); // Несколько битів
     cout << 2 << endl;

    // Вивід десяткових чисел з фіксованою крапкою,
    //  знаком і незначними нулями

   cout.setf(ios::showpoint | ios::showpos | ios::internal);

   cout.precision(10);                          // Десять цифр після коми

   cout << 3.0 << endl;

   // Вивід чисел із крапкою, що плаває, у науковому форматі з прописною буквою

   cout.setf(ios::scientific | ios::uppercase);
```

```
   cout << 4.0 << endl;

   // Поля виводу

   cout.setf(ios::fixed);      // Число з фіксованою крапкою
   cout.width(10);             // Ширина поля виводу дорівнює 10 символів
   cout.precision(3);          // Кількість знаків після коми дорівнює 3
   cout.fill('*');             // Замість ведучих нулів уставити зірочки

   cout << 5.0 << endl;

   return 0;
}
```

Результат

```
 0x1
 +2
 +3.000000000
 +4.0000000000E+000
 +*****5.00
```

Скидання прапорців форматування

```
 #include <iostream>

using namespace std;

int main()
{

   int x = 50;

   ios::fmtflags format;
   format = ios::hex | ios::showbase;
   cout.flags(format);                  // Установлюємо прапорці hex і showbase
   cout << "Прапорці hex і showbase установлені     " << x << endl;

   cout.unsetf(cout.flags());           // Скидаємо всі прапорці

   cout << "Усі прапорці скинуті                   " << x << endl;

   format = ios::hex | ios::showbase;   // Установлюємо прапорці hex і showbase
   cout.flags(format);

   cout << "Прапорці hex і showbase установлені     " << x << endl;

   cout.unsetf(ios::hex | ios::showbase); // Скидаємо прапорці hex і showbase

   cout << "Прапорці hex і showbase скинуті        " << x << endl;

   cout.setf(ios::hex | ios::showbase);  // Установлюємо hex і showbase
   cout << "Прапорці hex і showbase установлені     " << x << endl;

   cout.unsetf(ios::showbase);          // Скидаємо прапорець showbase
   cout << "Прапор showbase скинутий                " << x << endl;

   return 0;
}
```

Результат

```
 Прапорці hex і showbase установлені        0x32
 Усі прапорці скинуті                      50
 Прапорці hex і showbase установлені        0x32
 Прапорці hex і showbase установлені        50
 Прапорці hex і showbase установлені        0x32
 Прапорець showbase скинутий                  32
```

Перевірка прапорців форматування

```
 #include <iostream>

using namespace std;
void checkflags();

int main()
{
   ios::fmtflags format;
   format = ios::hex | ios::showbase;
   cout.flags(format);

   checkflags();

   return 0;
}

void checkflags()
{
 ios::fmtflags format;
 format = (long) cout.flags(); // Стан прапорів
 char* names[] = {"boolalpha ", "dec       ", "fixed     ", "hex       ",
                  "internal  ", "left      ", "oct       ", "right     ",
                  "scientific", "showbase  ", "showpoint ", "showpos   ",
                  "skipws    ", "unitbuf   ", "uppercase "};

 int j = 0;
 for(long i=0x0001; i<=0x4000; i = i << 1)
 {
  if(i & format) cout << names[j] << " = 1 " << i << " " << endl;
  else           cout << names[j] << " = 0 " << i << " " << endl;
  j++;
 }

 cout << endl;
}
```

Результат

```
 boolalpha  = 0 0x1
 dec        = 0 0x2
 fixed      = 0 0x4
 hex        = 1 0x8
 internal   = 0 0x10
 left       = 0 0x20
 oct        = 0 0x40
 right      = 0 0x80
 scientific = 0 0x100
 showbase   = 0 0x200
 showpoint  = 0 0x400
 showpos    = 1 0x800
 skipws     = 0 0x1000
 unitbuf    = 0 0x2000
 uppercase  = 0 0x4000
```

Бесформатний ввід

```
#include <iostream>

using namespace std;

int main()
{
   const int SIZE = 10;
   char buffer[SIZE]; // Буфер для збереження size символів
   int num = 1, ind;
   while (ind = cin.getline(buffer, SIZE,'\n') && ind!=EOF || cin.gcount())
   {
       int count = cin.gcount();
       if(cin.fail())
```

```
        {
          // Перевищений розмір буфера
          cout << "Перевищений розмір буфера" << endl;
          cout << "[" << num << "] " << "(" << count << " символів): "
               << buffer << endl;

          // Сбрасываем стан потоку
          cin.clear(cin.rdstate() & ~ios::failbit);
        }
        else
        {
          cout << "[" << num <<"] "
               << "(" << count-1 << " символів): " << buffer << endl;
          num++;
        }
    }
    if(cin.eof()) cout << "Final " << endl;

    return 0;
}
```

Результат

```
abcdf
[1] (5 символів): abcdf
abcdefghijklmn
Перевищено розмір буфера
[2] (9 символів): abcdefghi
[2] (5 символів): jklmn
```

26.4. Маніпулятори

Керування форматуванням за допомогою прапорців довольно громоздко. У мові C++ існує більш елегантний спосіб — застосування маніпуляторів. Маніпулятори являють собою спеціальні функції, що керують станом потоку при виконанні операції чи вводу виводу.

Маніпулятори розділяються на три категорії: вводу, виводу і вводу-виводу. Крім того, одні маніпулятори мають параметри, а інші — немає. При виклику маніпуляторів, що не мають параметрів, дужки ставити не слід.

Підкреслимо також, що маніпулятори не реалізують окремий механізм форматування. Таким чином, вони являють собою зручний інтерфейс для зв'язку зі стандартними засобами форматування з класу `ios_base`.

26.4.1. Стандартні маніпулятори

Маніпулятори визначені в просторі імен `std`. Їхні описи розподілені по різних заголовках: `<ios>`, `<istream>`, `<ostream>`. У заголовку `<iomanip>` окремо визначені маніпулятори, що мають параметри. Розглянемо визначення стандартних маніпуляторів.

26.4.1.1. Маніпулятор вводу-виводу boolaplha

```
inline ios_base& boolalpha(ios_base& str)
{
   str.setf(ios_base::boolalpha);
   return str;
}
```

Установлює прапор `boolalpha`.

26.4.1.2.Маніпулятор вводу-виводу noboolalpha

```
inline ios_base& noboolalpha(ios_base& str)
{
   str.unsetf(ios_base::boolalpha);
   return str;
}
```

Скидає прапор `boolalpha`.

26.4.1.3. Маніпулятор виводу showbase

```
inline ios_base& showbase(ios_base& str)
{
   str.setf(ios_base::showbase);
   return str;
}
```

Установлює прапор showbase.

26.4.1.4. Маніпулятор виводу noshowbase

```
inline ios_base& noshowbase(ios_base& str)
{
   str.unsetf(ios_base::showbase);
   return str;
}
```

Скидає прапор showbase.

26.4.1.5. Маніпулятор виводу showpoint

```
inline ios_base& showpoint(ios_base& str)
{
   str.setf(ios_base::showpoint);
   return str;
}
```

Установлює прапор showpoint.

26.4.1.6. Маніпулятор виводу noshowpoint

```
inline ios_base& noshowpoint(ios_base& str)
{
   str.unsetf(ios_base::showpoint);
   return str;
}
```

Скидає прапор showpoint.

26.4.1.7. Маніпулятор виводу showpos

```
inline ios_base& showpos(ios_base& str)
{
   str.setf(ios_base::showpos);
   return str;
}
```

Установлює прапор showpos.

26.4.1.8. Маніпулятор виводу noshowpos

```
inline ios_base& noshowpos(ios_base& str)
{
   str.unsetf(ios_base::showpos);
   return str;
}
```

Скидає прапор showpos.

26.4.1.9. Маніпулятор вводу skipws

```
inline ios_base& skipws(ios_base& str)
{
   str.setf(ios_base::skipws);
   return str;
}
```

Установлює прапор skipws.

26.4.1.10. Маніпулятор вводу noskipws

```
inline ios_base& noskipws(ios_base& str)
{
   str.unsetf(ios_base::skipws);
   return str;
}
```

Скидає прапор skipws.

26.4.1.11. Маніпулятор виводу uppercase

```
inline ios_base& uppercase(ios_base& str)
{
   str.setf(ios_base::uppercase);
   return str;
}
```

Установлює прапор uppercase.

26.4.1.12. Маніпулятор виводу nouppercase

```
inline ios_base& nouppercase(ios_base& str)
{
  str.unsetf(ios_base::uppercase);
  return str;
}
```

Скидає прапор `uppercase`.

26.4.1.13. Маніпулятор виводу unitbuf

```
inline ios_base& unitbuf(ios_base& str)
{
  str.setf(ios_base::unitbuf);
  return str;
}
```

Установлює прапор `unitbuf`.

26.4.1.14. Маніпулятор виводу nounitbuf

```
inline ios_base& nounitbuf(ios_base& str)
{
  str.unsetf(ios_base::unitbuf);
  return str;
}
```

Скидає прапор `nounitbuf`.

26.4.1.15. Маніпулятор виводу adjustfield

```
inline ios_base& internal(ios_base& str)
{
  str.setf(ios_base::internal, ios_base::adjustfield);
  return str;
}
```

Установлює прапорці `internal` і `adjustfield`.

26.4.1.26. Маніпулятор виводу left

```
inline ios_base& left(ios_base& str)
{
  str.setf(ios_base::left, ios_base::adjustfield);
  return str;
}
```

Установлює прапорці `left` і `adjustfield`.

26.4.1.17. Маніпулятор вводу right

```
inline ios_base& right(ios_base& str)
{
  str.setf(ios_base::right, ios_base::adjustfield);
  return str;
}
```

Установлює прапорці `right` і `adjustfield`.

26.4.1.18. Маніпулятор вводу-виводу dec

```
inline ios_base& dec(ios_base& str)
{
  str.setf(ios_base::dec, ios_base::basefield);
  return str;
}
```

Установлює прапор `dec`.

26.4.1.19. Маніпулятор вводу hex

```
inline ios_base& hex(ios_base& str)
{
  str.setf(ios_base::hex, ios_base::basefield);
  return str;
}
```

Установлює прапор `hex`.

26.4.1.20. Маніпулятор вводу-виводу oct

```
inline ios_base& oct(ios_base& str)
{
  str.setf(ios_base::oct, ios_base::basefield);
  return str;
}
```

Установлює прапор `oct`.

26.4.1.21. Маніпулятор виводу fixed

```
inline ios_base& fixed(ios_base& str)
{
  str.setf(ios_base::fixed, ios_base::floatfield);
  return str;
}
```

Установлює прапор `fixed`.

26.4.1.22. Маніпулятор виводу scientific

```
inline ios_base& scientific(ios_base& str)
{
  str.setf(ios_base::scientific, ios_base::floatfield);
  return str;
}
```

Установлює прапор `scientific`.

26.4.1.23.Маніпулятор виводу endl

```
template <class char, class traits>
basic_ostream<char,traits>& endl(basic_ostream<char,traits>& os)
{
os.put(os.widen('\n');
os.flush();
return 0;
}
```

Перехід на новий рядок з очищенням буфера.

26.4.1.24. Маніпулятор виводу ends

```
template <class char, class traits>
basic_ostream<char,traits>& endl(basic_ostream<char,traits>& os)
{
os.put(char());
return 0;
}
```

Записує ознака кінця рядка.

26.4.1.26. Маніпулятор вводу-виводу setiosflags

- `T& setiosflags(ios_base::fmtflags mask)`

Установлює прапорці, задані бітовою маскою `mask`. Тип `T` залежить від конкретного компілятора.

Якщо маніпулятор використовується у вираженні `out<<value`, де `out` є об'єктом класу `basic_ostream`, або у вираженні `in>>value`, де `in` є об'єктом класу `basic_istream`, маніпулятор поводиться як функція, визначена нижче.

```
ios_base& setiosflags(ios_base& str, ios_base::fmtflags mask)
{
  str.setf(mask);
  return str;
}
```

26.4.1.26. Маніпулятор вводу-виводу resetiosflags

- `T& resetiosflags(ios_base::fmtflags mask)`

Скидає прапорці, задані бітовою маскою `mask`. Тип `T` залежить від конкретного компілятора.

Якщо маніпулятор використовується у вираженні `out<<value`, де `out` є об'єктом класу `basic_ostream`, або у вираженні `in>>value`, де `in` є об'єктом класу `basic_istream`, маніпулятор поводиться як функція, визначена нижче.

```
ios_base& resetiosflags(ios_base& str, ios_base::fmtflags mask)
{
  str.setf(ios_base::fmtflags(0), mask);
```

```
   return str;
}
```

26.4.1.27. Маніпулятор вводу-виводу setbase

- `T& setbase(int base);`

Установлює прапор, що визначає підставу системи числення, — `dec`, `oct` чи `hex`; у залежності від значення параметра `base` — 10, 8 чи 26 відповідно.

Якщо маніпулятор використовується у вираженні `out<<value`, де `out` є об'єктом класу `basic_ostream`, або у вираженні `in>>value`, де `in` є об'єктом класу `basic_istream`, маніпулятор поводиться як функція, визначена нижче.

```
   ios_base& setiosflags(ios_base& str, int base)
{
   str.setf(n == 8  ? ios_base::oct :
            n == 10 ? ios_base::dec :
            n == 26 ? ios_base::hex :
            ios_base::fmtflags(0), ios_base::base_field);
      return str;
}
```

26.4.1.28. Маніпулятор виводу setfill

- `T setfill(char_type c);`

Задає символ-заповнювач.

Якщо маніпулятор використовується у вираженні `out<<value`, де `out` є об'єктом класу `basic_ostream`, або у вираженні `in>>value`, де `in` є об'єктом класу `basic_istream`, маніпулятор поводиться як функція, визначена нижче.

```
   template<class char, class traits>
basic_ios&<char,traits>& setfill(basic_ios<char,traits>& str, char c)
{
   str.fill(c);
   return str;
}
```

26.4.1.29. Маніпулятор виводу setprecision

- `T setprecision(int n);`

Задає кількість цифр після десяткової крапки.

Якщо маніпулятор використовується у вираженні `out<<value`, де `out` є об'єктом класу `basic_ostream`, або у вираженні `in>>value`, де `in` є об'єктом класу `basic_istream`, маніпулятор поводиться як функція, визначена нижче.

```
   ios_base& setprecision(ios_base str, int n)
{
   str.precision(n);
   return str;
}
```

26.4.1.30. Маніпулятор виводу setw

- `T setw(int n);`

Задає ширину поля виводу.

Якщо маніпулятор використовується у вираженні `out<<value`, де `out` є об'єктом класу `basic_ostream`, або у вираженні `in>>value`, де `in` є об'єктом класу `basic_istream`, маніпулятор поводиться як функція, визначена нижче.

```
   ios_base& setw(ios_base str, int n)
{
   str.width(n);
   return str;
}
```

26.4.1.31. Маніпулятор вводу ws

- `template <class char, class traits>`
  `basic_istream<char,traits>& ws(basic_istream<char,traits>& is)`

Ігнорує усі ведучі роздільники.

Розглянемо приклад, еквівалентний одному з попередніх.

Застосування маніпуляторів: перший приклад

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

using namespace std;

int main()
{
   // Вивід цілого числа в шістнадцатерічному виді
   // із указівкою підстави системи числення

   cout << hex << showbase << 1 << endl;

   // Установка окремих битів
   cout << dec << showpos << unitbuf << 2 << endl;

   // Вивід цілого числа з десятковою крапкою
   // і зазначеним кількістю цифр після її
   cout << showpoint << showpos << internal << setprecision(10)
        << 3.0 << endl;

   // Вивід чисел із крапкою, що плаває, у науковому
   форматі // із прописною буквою

   cout << scientific << uppercase << 4.0 << endl;

   // Поля виводу

   cout << fixed << setw(10) << setprecision(3) << setfill('*')
        << 5.0 << endl;

   return 0;
}
```

Результат

```
0x1
+2
+3.000000000
+4.0000000000E+000
+*****5.00
```

Как видим, ми досягли того ж результату, але програма стала в чотири рази коротше!

Приведемо ще один приклад, що ілюструє застосування маніпуляторів.

Застосування маніпуляторів: другий приклад

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

using namespace std;

int main()
{

   int x = 50;

   ios::fmtflags format = cout.flags();
   format = ios::showbase;

   cout << "Прапор showbase установлений     "
        << setiosflags(format);
   cout << hex << x << endl;

   // Скидаємо прапорець showbase

   cout << "Прапор showbase скинутий                "
        << noshowbase << x << endl;

   // Установка окремих битів за допомогою маніпулятора setiosflags
```

```
    format = ios::showbase;
    cout << "Прапор showbase установлений     "
         << setiosflags(format) << x << endl;

   // Скидання окремих битів за допомогою маніпулятора setiosflags
    cout << "Прапор showbase скинутий                "
         << resetiosflags(format) << x << endl;

    return 0;
}
```

Результат

```
Прапорець showbase установлений      0x32
Прапорець showbase скинутий                32
Прапорець showbase установлений      0x32
Прапорець showbase скинутий                32
```

26.4.2. Програмування маніпуляторів без параметрів

Існують способи створити власні маніпулятори, що більш повно задовольняють потреби програміста. Ці способи залежать від виду маніпуляторів.

Схема маніпулятора виводу, що не має параметрів, виглядає так.

```
ostream &ім'я(ostream &stream)
{
  // Код
  return stream;
}
```

Схема маніпулятора вводу, що не має параметрів, відрізняється лише типом потоку.

```
istream &ім'я(istream &stream)
{
  // Код
  return stream;
}
```

Варто мати на увазі, що посилання на потік передаються маніпулятору неявно, тому при виклику маніпулятора цей аргумент не вказується. Крім того, маніпулятор *неодмінно повинний* повертати посилання на потік. Найчастіше власні маніпулятори використовують для об'єднання властивостей різних маніпуляторів. Наприклад, у розглянутому вище прикладі можна було б не викликати несколько маніпуляторів.

```
cout << fixed << setw(10) << setprecision(3) << setfill('*')
       << 5.0 << endl;
```

Об'єднаємо їх в окремому маніпуляторі, додавши нові властивості.

Маніпулятор без параметрів

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

using namespace std;

ostream& setfixed(ostream& stream)
{
   stream.width(10);
   stream.precision(3);
   stream.setf(ios::showpos);
   stream.setf(ios::showpoint);
   stream.fill('*');

   return stream;
}

int main()
{
   // Вивід дійсного числа у фіксованому форматі

   cout << setfixed << 5.0 << endl;

   return 0;
}
```

Результат

```
*****+5.00
```

Схема маніпуляторів вводу, що мають параметри, виглядає в такий спосіб.

```
ostream &ім'я(ostream &stream, список параметрів)
{
  // Код
  return stream;
}
```

26.4.3. Функції виводу і вставки

Оскільки изначально оператори << і >> є операторами побітового зрушення, зовсім очевидно, що в мові C++ вони перевантажені для вводу і виводу даних, що мають убудовані типи. Отже, немає ніяких причин, що можуть перешкодити їхньому повторному перевантаженню для вводу і виводу даних, що відносяться до нестандартних типів.

Виходячи з концепції потоків, прийнятої в мові C++, оператори вводу і виводу більш коректно називати *операторами виводу з потоку* і *вставки в потік*. Для того щоб відрізняти стандартні і нестандартні варіанти, перевантажені оператори << і >> прийнятий називати *функціями вставки* і *функціями виводу*.

Схема функції вставки дуже проста.

```
ostream &operator<<(ostream &os, клас arg)
{
  // Код
}
```

Головні вимоги складаються в наступному: 1) функція вставки повинна повертати посилання на потік виводу (тобто `ostream&`); 2) першим аргументом функції вставки повинний бути об'єкт класу `ostream`, переданий по посиланню (цей об'єкт передається неявно і при виклику не вказується); 3) другим аргументом функції вставки повинний бути об'єкт класу, для якого перевантажений оператор виводу.

Розглянемо конкретний приклад.

Перевантаження оператора >> для виводу комплексних чисел

```
#include <iostream>
using namespace std;

class TComplex
{
   double Re, Im;
public:
   TComplex(double x, double y):Re(x),Im(y){}
   friend ostream &operator<<(ostream &os, TComplex z);
};

ostream &operator<<(ostream &os, TComplex z)
{
   os.precision(3);
   os.setf(ios_base::showpos | ios::showpoint);
   os << z.Re << "+i*" <<z.Im;
   return os;
}

int main()
{
   TComplex u(1,2),v(3,4),w(5,6);
   cout << "Комплексні числа:\n";
   cout << " u = " << u << " v = " << v << " v = " << v;

   return 0;
}
```

Результат

```
Комплексні числа:
u = +1.00+i*+2.00 v = +3.00+i*+4.00 v = +3.00+i*+4.00
```

Как видим, для перевантаження використаний механізм дружніх функцій. Це зв'язано з тим, що *функція вставки не може бути членом класу*. Причина криється в тім, що при виводу данных об'єкт потоку *завжди* є лівим операндом оператора вставки. Отже, при виводу комплексного числа `z` вираження `cout << z` эквивалентно вираженню `operator<<(cout,z)`. Якби функція вставки була членом класу, ее лівим

операндом був би об'єкт класу, якому вона належить (неявний аргумент `this`). Це суперечить порядку операндів, прийнятому при виводу даних.

Схема функції виводу відрізняється від схеми функції вставки лише типом потоку.

```
istream &operator<<(istream &is, клас arg)
{
  // Код
}
```

Головні вимоги залишаються колишніми: 1) функція виводу повинна повертати посилання на потік виводу (тобто `istream&`); 2) першим аргументом функції вставки повинний бути об'єкт класу `istream`, переданий по посиланню (цей об'єкт передається неявно і при виклику не вказується); 3) другим аргументом функції вставки повинний бути об'єкт класу, для якого перевантажений оператор вводу.

Розглянемо приклад, що ілюструє операцію вводу.

Перевантаження оператора << для виводу комплексних чисел

```
#include <iostream>
using namespace std;
#define SIZE 10

class TComplex
{
   double Re, Im;
public:
   TComplex():Re(0),Im(0){}
   TComplex(double x, double y):Re(x),Im(y){}
   friend ostream& operator<<(ostream& os, TComplex& z);
   friend istream& operator>>(istream& os, TComplex& z);
};

ostream& operator<<(ostream& os, TComplex& z)
{
   os.setf(ios::showpoint);
   os.precision(3);
   os << z.Re << "+i*" <<z.Im;
   return os;
}

istream& operator>>(istream& is, TComplex& z)
{
   char buffer[SIZE], number[SIZE];
   cin.getline(buffer, SIZE, '\n');
   cout << endl;
   int i=0;
   while(buffer[i]!='+') { number[i]=buffer[i]; i++; }
   number[i+1]='\n';
   z.Re = atof(number);

   while( buffer[i]=='+' ||
          buffer[i]=='i' ||
          buffer[i]=='*')
          i++;
   int j=0;
   while(i!=SIZE) { number[j]=buffer[i]; i++; j++; }
   number[j+1]='\n';
   z.Im = atof(number);

   return is;
}

int main()
{
   TComplex u;
   cout << "Уведення : ";
   cin  >> u;
   cout << "Вивід: " << u << endl;

   return 0;
}
```

Результат

```
Уведення:  1+i*2
Вивід: 1.00+i*2.00
```

26.4.4. Програмування маніпуляторів, що мають параметри

З функціями виводу і вставки тісно зв'язані питання розробки власних маніпуляторів. Допустимо, що стандартні засоби нас не влаштовують і ми хочемо установлювати формат за допомогою власного маніпулятора, що має параметри. Вставити в ланцюжок операторів виводу такий маніпулятор не можна. Отже, необхідно створити окремий клас, инкапсулирующий у собі виклик функції-маніпулятора. Оскільки оператори вводу і виводу перевантажені для убудованих типів, прийдеться створити окремі функції виводу і вставки, перевантажені для нестандартного типу.

Визначимо клас, инкапсулирующий виклик функції маніпулятора, що встановлює ширину поля виводу, кількість цифр після десяткової крапки, а також зовнішній вигляд дійсного числа.

```
  class setfixed
{
friend ostream& operator <<(ostream&, setfixed);
public:
   typedef void (* FManip)(ostream&, int, int);
   setfixed(FManip p, const int& n, const int& m)
            : p(p), width(n), precision(m){}

private:
   FManip p;
   const int width;
   const int precision;
};
```

Клас `setfixed` складається з закритих членів: вказівника на функцію-маніпулятор `p()`, а також константних цілих чисел `width` і `precision`. Крім того, у ньому визначений тип вказівника на функцію-маніпулятор і зазначена дружня функція вставки, що повертає посилання на потік типу `ostream`. Об'єкт класу `setfixed` є другим операндом функції вставки, а ее першим операндом буде об'єкт `cout`, автоматично створюваний при запуску кожної програми.

Дружня функція вставки виконує виклик функції-маніпулятора, передає їй фактичні аргументи і повертає посилання на об'єкт потоку `ostream`. Таким чином, ее визначення повинне мати наступний вид.

```
  ostream& operator <<(ostream& os, setfixed s)
{
   (*(s.p))(os, s.width, s.precision);
   return os;
}
```

Функція-маніпулятор `SetFormat` виконує установку прапорців форматування і задає параметри виводу — ширину поля і кількість цифр після десяткової крапки. Її першим аргументом повинна бути посилання на потік виводу класу `ostream`. Зверніть увагу на те, що функція `SetFormat` *повинна* мати тип `void`.

```
  void SetFormat(ostream& os, int n, int m)
{
   os.precision(m);
   os.width(n);
   os.setf(ios::showpos);
   os.setf(ios::showpoint);
}
```

Збираючи ці фрагменти разом, одержуємо наступну програму.

Вивід дійсних чисел у заданому форматі

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

using namespace std;

void SetFormat(ostream&, int, int);

class setfixed
{
friend ostream& operator <<(ostream&, setfixed);
public:
   typedef void (* FManip)(ostream&, int, int);
   setfixed(FManip p, const int& n, const int& m)
```

```
            : p(p), width(n), precision(m){}

private:
   FManip p;
   const int width;
   const int precision;
};

ostream& operator <<(ostream& os, setfixed s)
{
   (*(s.p))(os, s.width, s.precision);
   return os;
}

void SetFormat(ostream& os, int n, int m)
{
   os.precision(m);
   os.width(n);
   os.setf(ios::showpos);
   os.setf(ios::showpoint);
}

int main()
{
   // Вивід числа

   cout << "Число в заданому форматі: "
        << setfixed(SetFormat,7,3) << 5.7 << endl;

   return 0;
}
```

Результат

```
Число в заданому форматі:   +5.70
```

За рамками цієї книги залишилася ієрархія класів вводу-виводу низького рівня, в основі якої лежить клас `basic_streambuf`, а також ієрархія шаблонних класів, спеціалізованих для вводу-виводу розширених символів. Справедливості заради слід зазначити, що потреба в цих класах виникає досить рідко.