

# Потоци (Rev: 1.1)

Любомир Чорбаджиев<sup>1</sup>  
lchorbadjiev@elsys-bg.org

8 май 2007 г.

## Съдържание

## Съдържание

1	Йерархия на потоците за вход/изход	1
2	Писане в поток	3
3	Четене от поток	4
4	Състояние на потока	5
5	Четене на символни низове	8
6	Изключения генерирани от входно/изходните операции	9
7	Форматиране при изходни операции	10
8	Манипулатори	16
9	Файлови потоци	17

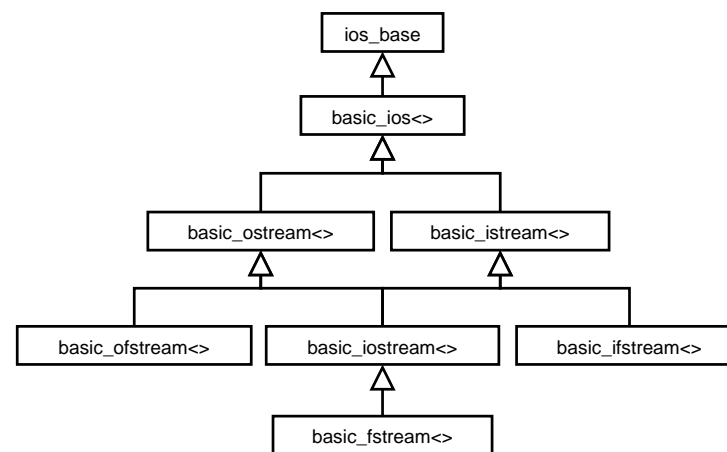
## 1. Йерархия на потоците за вход/изход

### Въведение

- Входно/изходните потоци в C++ са обектно ориентирани.
- Входно/изходните операции в C++ са **строго типизирани**. Това позволява при обработването на данни с различен тип да се използва C++ механизмът за предефиниране на оператори и функции.

- Входно/изходните операции са **разширяеми**. Лесно се реализират входно/изходни операции за типове, дефинирани от потребителя.

### Йерархия на потоците за вход/изход



### Йерархия на потоците за вход/изход

- Потоците за вход/изход са организирани като йерархия от шаблонни класове.
- Потоците, които се използват обичайно са: `ostream`, `istream`, `ofstream`, `ifstream`. Тези потоци са 8 битови – т.е. последователността от символи, които се четат/пишат са от типа `char`.
- Дефинициите на тези потоци са следните:

```
typedef basic_ostream<char> ostream;  
typedef basic_istream<char> istream;  
typedef basic_ofstream<char> ofstream;  
typedef basic_ifstream<char> ifstream;  
typedef basic_fstream<char> fstream;
```

### Йерархия на потоците за вход/изход

- В езика C++ освен типа `char` (8 битов символ) е дефиниран и типа `wchar_t` (16 битов символ);
- Потоците, които работят с типа `wchar_t` са:

```
typedef basic_ostream<wchar_t> wostream;
typedef basic_istream<wchar_t> wistream;
typedef basic_ofstream<wchar_t> wofstream;
typedef basic_ifstream<wchar_t> wifstream;
typedef basic_fstream<wchar_t> wfstream;
```

## Работа с потоците за вход/изход

- Всички потоци за вход/изход са дефинирани в пространството от имена `std`.
- Основните потоци са дефинирани в заглавния файл `<iostream>`. В него са дефинирани обектите:
  - `cin` — обект от класа `istream`, който е свързан със стандартния вход.
  - `cout` — обект от класа `ostream`, който е свързан със стандартния изход.
  - `cerr` — обект от класа `ostream`, който е свързан със стандартната грешка. Този поток е **небуфериран**.

## 2. Писане в поток

### Писане в поток

- За писане в изходен поток се използва операторът `operator<<`. В класа `basic_ostream` този оператор е дефиниран за всички примитивни типове:

```
1 template<class Ch, class Tr=char_traits<Ch> >
2 class basic_ostream:
3     virtual public basic_ios<Ch, Tr> {
4 public:
5     //...
6     basic_ostream& operator<<(short n);
7     basic_ostream& operator<<(int n);
8     basic_ostream& operator<<(long n);
9     basic_ostream& operator<<(unsigned short n);
10    //...
11 };
```

### Писане в поток

- Операторът `operator<<` връща псевдоним, насочен към използвания изходен поток `ostream`. Това дава възможност операторът за изход да се прилага каскадно. Изразът:

```
cout << "x=" << x;
```

е еквивалентен на:

```
(cout.operator<<("x=")).operator<<(x);
```

### Операции за изход, дефинирани от потребителя

- Да разгледаме клас `point`, определен от потребителя:

```
1 class point {
2     double x_, y_;
3 public:
4     point(double x, double y)
5         : x_(x), y_(y)
6     {}
7     double get_x(void) {return x_;}
8     double get_y(void) {return y_;}
9     void set_x(double x) {x_=x;}
10    void set_y(double y) {y_=y;}
11    //...
12 };
```

### Операции за изход, дефинирани от потребителя

- Операторът `operator<<` за този тип може да бъде дефиниран по следния начин:

```
1 ostream& operator<<(ostream& out,
2                     const point& p) {
3     out << '('
4         << p.get_x() << ', '
5         << p.get_y() << ')';
6     return out;
7 }
```

## 3. Четене от поток

### Четене от поток

- За четене от входен поток се използва операторът **operator>>**. В класа `basic_istream` този оператор е дефиниран за всички примитивни типове:

```

1 template<class Ch, class Tr=char_traits<Ch> >
2 class basic_istream:
3     virtual public basic_ios<Ch, Tr> {
4 public:
5     //...
6     basic_istream& operator>>(short& n);
7     basic_istream& operator>>(int& n);
8     basic_istream& operator>>(long& n);
9     //...
10 };

```

- Операторът **operator>>** пропуска символите разделители ('`\n`', '`\n`', '`\r`', '`\t`', '`\f`', '`\v`').

## 4. Състояние на потока

### Състояние на потока

- Най-разпространената грешка при използване на потоци за вход `istream`: това, което е в потока се различава от това, което очакваме да бъде в потока. Например: искаме да прочетем променлива от типа `int`, а в потока има букви.
- За да се предпазим от този тип грешки е необходимо преди да се използват прочетените от потока данни да се провери **състоянието** на потока.

### Състояние на потока

- Всеки поток `istream` и `ostream` има свързано с него **състояние**. Състоянието на потока е дефинирано в базовия клас `basic_ios<>`.

```

1 template<class Ch, class Tr=char_traits<Ch> >
2 class basic_ios: public ios_base {
3 public:
4     //...
5     bool good(void) const;
6     bool eof(void) const;
7     bool fail(void) const;
8     bool bad(void) const;
9     //...
10 };

```

### Състояние на потока

- `good()` – предходните операции са изгълнени успешно;
- `eof()` – вижда се края на файла;
- `fail()` – следващата операция няма да се изгълни успешно;
- `bad()` – потокът е повреден.

### Състояние на потока

- Състоянието на потока представлява набор от флагове, които са дефинирани в базовия клас `ios_base`:

```

1 class ios_base {
2 public:
3     //...
4     typedef ... iostate;
5     static const iostate
6         badbit, // потокът е развален
7         eofbit, // вижда се края на файла
8         failbit, // следващата операция няма да се изгълни
9         goodbit; // потокът е наред
10    //...
11 };

```

### Състояние на потока

- В класа `basic_ios<>` са дефинирани следните методи за манипулиране на състоянието на потока:

```

1 template<class Ch, class Tr=char_traits<Ch> >
2 class basic_ios: public ios_base {
3 public:
4     ...
5     // връща флаговете на състоянието на потока
6     iostate rdstate(void) const;
7     // установява флаговете на състоянието
8     void clear(iostate f=goodbit);
9     // добавя f към флаговете на състоянието
10    void setstate(iostate f) {
11        clear(rdstate()|f);
12    }
13 };

```

### Състояние на потока

```
1 template<class Ch, class Tr=char_traits<Ch> >
2 class basic_ios: public ios_base {
3 public:
4     ...
5     operator void* () const;
6 };
```

Операторът, дефиниран в ред 5, е оператор за преобразуване към **void\***. Този оператор връща стойност, различна от NULL, ако потокът е наред. Този оператор за преобразуване позволява потоците да участват в условни оператори и оператори за цикъл. Например:

```
while (cin){
    cin >> i;
}
```

### Операции за вход, дефинирани от потребителя

Нека формата за въвеждане на променливи от типа `point` е `(x,y)`, където `x` и `y` са числа с плаваща точка. Тогава дефиницията на оператор за четене на обекти от класа `point` може да бъде направена по следния начин:

```
1 istream& operator>>(istream& in, point& p) {
2     double x,y;
3     char c;
4     in >> c;
5     if(c!='(') {
6         in.clear(ios_base::badbit);
7         return in;
8     }
9     in >> x >> c;
```

### Операции за вход, дефинирани от потребителя

```
10 if(c!=',') {
11     in.clear(ios_base::badbit);
12     return in;
13 }
14 in >> y >> c;
15 if(c!=')') {
16     in.clear(ios_base::badbit);
17     return in;
18 }
```

```
19 if(in.good()){
20     p.set_x(x);
21     p.set_y(y);
22 }
23 return in;
24 };
```

## 5. Четене на символни низове

### Четене на символи

- Операторът `operator>>` е предназначен за форматиран вход — т.е. за четене на обекти от някакъв очакван тип, в някакъв очакван формат.
- Когато предварително не се знае какви типове ще има във входния поток, то от входния поток обикновено се четат символни низове. За тази цел се използва семейството от методи `get()`, дефинирани в `basic_istream<>`.

### Четене на символи

```
char c;
cin.get(c);

char buf[100];
cin.get(buf,100);
cin.get(buf,100,'\n');

cin.getline(buf,100);
cin.getline(buf,100,'\n');
```

### Четене на символи

- Функциите `in.get(buf,n)` и `in.get(buf,n,term)` прочитат не повече от  $n - 1$  символа. Тези методи винаги поставят 0 след прочетените в буфера символи.
- Ако при четене с `get` е срещнат завършващият символ, то той остава в потока. Следващият фрагмент е пример за “хитър” безкраен цикъл:

```
1 char buf[256];
2 while (cin) {
```

```

3  cin.get(buf,256);
4  cout << buf;
5 }

```

- Функцията `getline()` се държи аналогично на `get()`, но прочита от `istream` срещнатия завършващ символ.

### Четене на символи

- `in.ignore(max,term)` – пропуска следващите символи докато не срещне символа `term` или не прочете `max` символа.
- `in.putback(ch)` – превръща `ch` в следващия непрочетен символ от потока `in`.
- `in.peek()` – връща следващия символ от потока `in`, но не го изтрива от потока.

## 6. Изключения генерирани от входно/изходните операции

### Изключения

- Да се проверява за грешки след всяка входно/изходна операция е неудобно. Поради това е предвидена възможност да “помолите” потока да генерира изключения, когато се променя състоянието му.
- В базовия клас `basic_ios<>` са дефинирани две функции:
  - `void exceptions(iostate st)` – установява състоянията, при които трябва да се генерира изключение.
  - `iostate exceptions() const` – връща набора от флагове на състоянието, при които се генерира изключение.
- Основната роля на генерирането на изключения при вход/изход е да се обработват малко вероятни изключителни ситуации. Изключенията могат да се използват и за контролиране на входно-изходните операции.

### Пример: изключения

```

1 #include <iostream>
2 #include <list>
3
4 using namespace std;

```

```

5
6 int
7 main(int argc, char* argv[]) {
8     list<int> result;
9     ios_base::iostate oldstate=cin.exceptions();
10    cin.exceptions(ios_base::eofbit
11                  |ios_base::badbit
12                  |ios_base::failbit);

```

### Пример: изключения

```

14 try {
15     for(;;){
16         int i;
17         cin >> i;
18         result.push_back(i);
19     }
20 } catch(const ios_base::failure& e) {
21     cout<<"ios_base::failure_ caught..."<<endl;
22 }
23
24 return 0;
25 }

```

## 7. Форматиране при изходни операции

### Форматиране

- Форматирането на входно/изходните операции се контролира чрез класовете `basic_ios` и `ios_base`.
- За управление на форматирането на входно/изходните операции се използва набор от флагове, определени в `ios_base`.
- Част от флаговете, определящи състоянието на формата са представени в следния фрагмент:

### Форматиране

```

1 class ios_base {
2 public:
3     typedef implementation_dependent fmtflags;
4     static const fmtflags
5     skipws, // пропуска разделителите при четене

```

```

6 boolalpha, // типа boolean се представят
7           // като true и false
8 // целочислени типове
9 dec,      // десетична система
10 hex,     // шестнадесетична система
11 oct,     // осмично система
12 showbase, // поставя префикс,
13          // обозначаващ системата

```

### Форматиране

```

1 // числа с плаваща запетая
2 scientific, // представяне във вида: d.dddddeddd
3 fixed,     // представяне във вида: dddd.dd
4 showpoint, // незначеща нула пред десетичната точка
5 showpos,   // явен знак '+' пред положителните числа
6 ...;
7 };

```

### Състояние на формата

skipws	пропускане на разделителните символи
boolalpha	представяне на типа boolean
dec	целочислени типове – в каква бройна
hex	система да се извеждат
oct	
showbase	представяне на бройната система
scientific	как се извеждат типовете с плаваща
fixed	точка
showpoint	
showpos	

### Форматиране

- За манипулиране на състоянието на формата, в класа `ios_base` са дефинирани следните методи:

```

1 class ios_base {
2 public:
3 ...
4 fmtflags flags() const;
5 fmtflags flags(fmtflags f);
6 fmtflags setf(fmtflags f){

```

```

7     return flags(flags()|f);
8 }

```

### Форматиране

```

1 fmtflags setf(fmtflags f, fmtflags mask) {
2     return flags((flags()&~mask)|(f&mask));
3 }
4 void unset(fmtflags mask) {
5     flags(flags()&~mask);
6 }
7 };

```

### Форматиране

- Стандартната схема за работа с флаговете за форматиране е следната:
  - запомняме състоянието на формата;
  - променяме състоянието на формата и използваме потока;
  - възстановяваме предишното състояние на потока.

```

1 void foo(void) {
2     ios_base::fmtflags old_flags=cout.flags();
3     cout.setf(ios_base::oct);
4     ...
5     cout.flags(old_flags);
6 };

```

### Извеждане на цели числа

- Добавянето на флагове чрез метода `setf()` или чрез побитово “ИЛИ” (`|`) е удобно, само когато дадена характеристика на потока се управлява от един бит.
- Тази схема е неудобна в случаи като определяне на бройната система. В такива случаи състоянието на формата не се определя от един бит.
- Решението на този проблем, което се използва в `<iostream>`, е да се предостави версия на `setf()` с втори “псевдо-аргумент”:

```

cout.setf(ios_base::oct, ios_base::basefield);
cout.setf(ios_base::dec, ios_base::basefield);
cout.setf(ios_base::hex, ios_base::basefield);

```

### Извеждане на цели числа: пример

```
1 #include <iostream>
2 using namespace std;
3
4 int main(int argc, char* argv[]) {
5     cout.setf(ios_base::showbase);
6     cout.setf(ios_base::oct, ios_base::basefield);
7     cout << 1234 << ' ' << 1234 << endl;
8     cout.setf(ios_base::dec, ios_base::basefield);
9     cout << 1234 << ' ' << 1234 << endl;
10    cout.setf(ios_base::hex, ios_base::basefield);
11    cout << 1234 << ' ' << 1234 << endl;
12
13    return 0;
14 };
```

### Извеждане на цели числа: пример

```
02322 02322
1234 1234
0x4d2 0x4d2
```

### Извеждане на числа с плаваща точка

- Извеждането на числа с плаваща запетая се определя от **формата** и **точността**.
- Форматите, които се използват за извеждане на числа с плаваща запетая са:
  - *Универсален* — позволява на потока сам да реши в какъв вид да се представи извежданото число. По подразбиране потоците използват този формат.
  - *Научен* — представя числото като десетична дроб с една цифра преди десетичната точка и показател на степента.
  - *Фиксиран* — точността определя максималният брой цифри след десетичната точка.
- По подразбиране точността е 6 цифри.

### Извеждане на числа с плаваща точка: пример

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char* argv[]) {
4     cout << 1234.56789 << ' ' << 1234.5678901 << endl;
5     cout.setf(ios_base::scientific,
6             ios_base::floatfield);
7     cout << 1234.56789 << ' ' << 1234.5678901 << endl;
8     cout.setf(ios_base::fixed,
9             ios_base::floatfield);
10    cout << 1234.56789 << ' ' << 1234.5678901 << endl;
11    cout.setf(static_cast<ios_base::fmtflags>(0),
12            ios_base::floatfield);
13    cout << 1234.56789 << ' ' << 1234.5678901 << endl;
14    return 0;
15 }
```

### Извеждане на числа с плаваща точка: пример

```
1234.57 1234.57
1.234568e+03 1.234568e+03
1234.567890 1234.567890
1234.57 1234.57
```

### Извеждане на числа с плаваща точка

- За промяна на точността на работа с числа с плаваща запетая се използват следните методи:

```
class ios_base {
public:
    ...
    unsigned precision() const;
    unsigned precision(unsigned n);
    ...
};
```

- Използването на `precision()` влияе на всички входно/изходни операции с потока и действа до следващото използване на метода.

### Извеждане на числа с плаваща точка: пример

```
1 #include <iostream>
2 using namespace std;
3 int main(int argc, char* argv[]) {
```

```

4  cout.precision(12);
5  cout<<123456789<<'␣'<<1234.12345<<'␣'
6    <<1234.123456789<<endl;
7  cout.precision(9);
8  cout<<123456789<<'␣'<<1234.12345<<'␣'
9    <<1234.123456789<<endl;
10 cout.precision(4);
11 cout<<123456789<<'␣'<<1234.12345<<'␣'
12   <<1234.123456789<<endl;
13 return 0;
14 }

```

### Извеждане на числа с плаваща точка: пример

```

123456789 1234.12345 1234.12345679
123456789 1234.12345 1234.12346
123456789 1234 1234

```

### Полета за изход

```

1 #include <iostream>
2 using namespace std;
3 int main(int argc, char* argv[]) {
4   cout << '(';
5   cout.width(5);
6   cout << 12 << ')' << endl;;
7   cout << '(';
8   cout.width(5); cout.fill('#');
9   cout << 12 << ')' << endl;;
10  cout << '(';
11  cout.width(5); cout.fill('#');
12  cout << "a" << ')' << endl;;
13  cout << '(';

```

### Полета за изход

```

14  cout.width(0); cout.fill('#');
15  cout << "a" << ')' << endl;;
16  return 0;
17 }

```

```

( 12)
(###12)

```

```

(###a)
(a)

```

## 8. Манипулатори

### Манипулатори

- Да се променя състоянието на потока посредством флаговете на формата е неудобно.
- Стандартната библиотека предоставя набор от функции и обекти за манипулиране на състоянието на потока — **манипулатори**.

### Манипулатори

- Основният начин за използване на манипулатори може да се види от следния пример:

```

cout<<boolalpha<<true<<'␣'
  <<noboolalpha<<true;

```

което извежда: true 1.

- Използват се и манипулатори с аргументи:

```

cout << setprecision(10)
  << 1234.12345678 << endl;

```

което извежда: 1234.123457. Манипулаторите с аргументи са дефинирани в <iomanip>.

boolalpha	noboolalpha	представяне на типа boolean
showbase	noshowbase	представяне на бройната система
showpoint	noshowpoint	
showpos	noshowpos	
skipws	noskipws	пропускане на разделителните символи
dec		целочислени типове – в каква
hex		бройна система да се извеждат
oct		
fixed		как се извеждат типовете
scientific		с плаваща точка
setprecision(n)		точност на извеждането
setw(int n)		ширина на полета за извеждане
setfill(int c)		символ за запълване на полето



## 9. Файлови потоци

### Файлови потоци

- Потоците за работа с файлове са дефинирани в <fstream>.
- Потокът за писане във файл е basic\_ofstream.

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ofstream: public basic_ostream<Ch,Tr> {
public:
    explicit basic_ofstream(const char* p,
                           openmode m=out|trunc);
    bool is_open() const;
    void open(const char* p,
              openmode m=out|trunc);
    void close();
    ...
};
```

### Файлови потоци

```
class ios_base {
public:
    typedef implementation_dependent1 openmode;
    static const openmode app,
                 ate, binary, in, out, trunc;
    ...
};
```

in	отваряне за четене
out	отваряне за запис
app	запис на данните в края на файла; предизвиква отваряне на файла в режим out
ate	запис на данните в края на файла
trunc	унищожава предишното съдържание на файла

### Файлови потоци

```
1 #include <iostream>
2 #include <fstream>
3 #include <cstdlib>
4 void error(const char* p, const char* p2="") {
```

```
5     std::cerr << p << '\n' << p2 << std::endl;
6     std::exit(1);
7 }
```

```
8 int main(int argc, char* argv[]) {
9     if(argc!=3)
10        error("bad_number_of_arguments...");
11    std::ifstream from(argv[1]);
12    if(!from)
13        error("bad_input_file", argv[1]);
14    std::ofstream to(argv[2]);
15    if(!to)
16        error("bad_output_file:", argv[2]);
17    char ch;
18    while(from.get(ch))
19        to.put(ch);
20    if(!from.eof() || !to)
21        error("something_strange...");
22    return 0;
23 }
```