Потоци

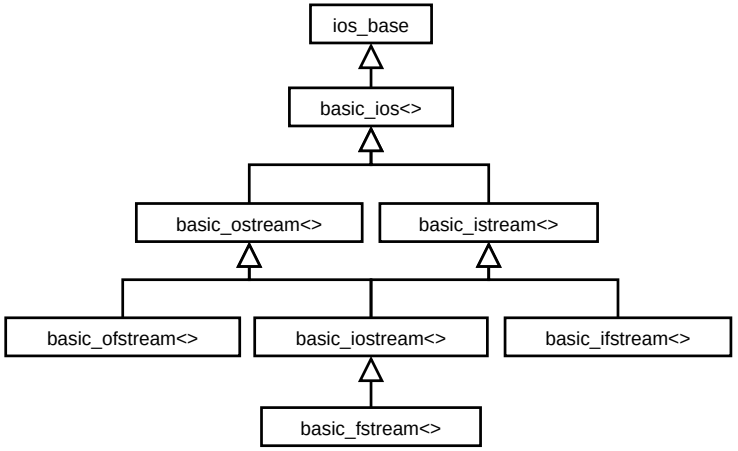
Любомир Чорбаджиев
Технологическо училище "Електронни системи"
Технически университет, София
lchorbadjiev@elsys-bg.org
Revision : 1.6

17 април 2005 г.

Въведение

- Входно/изходните потоци в C++ са обектно ориентирани.
- Входни/изходните операции в C++ са **строго типизирани**. Това позволява при обработването на данни с различен тип да се използва C++ механизма 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 класове-шаблони;
- 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 езикът 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**.

Писане в поток

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

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


Писане в поток

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

```
class point {
  double x_, y_;
public:
  point(double x, double y)
    : x_(x), y_(y)
  {}
  double get_x(void) {return x_;}
  double get_y(void) {return y_;}
  void set_x(double x) {x_=x;}
  void set_y(double y) {y_=y;}
  //...
};
```


Операции за изход, дефинирани от потребителя

- Операторът **operator**<< за този тип може да бъде определена по следния начин:

```
ostream& operator<<(ostream& out, const point& p) {
  out << '(' << p.get_x() << ','
      << p.get_y() << ')';
  return out;
}
```


Четене от поток

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

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_istream: virtual public basic_ios<Ch, Tr
public:
  //...
  basic_istream& operator>>(short& n);
  basic_istream& operator>>(int& n);
  basic_istream& operator>>(long& n);
  basic_istream& operator>>(unsigned short& n);
  //...
};
```

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

Четене от поток

- 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, преди да се използват прочетените от потока данни трябва 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<>`.

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ios: public ios_base {
public:
  //...
  bool good(void) const;
  bool eof(void) const;
  bool fail(void) const;
  bool bad(void) const;
  //...
};
```


Състояние на потока

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

Състояние на потока

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

```
class ios_base {
public:
  //...
  typedef ... iostate;
  static const iostate
      badbit, // потокът е развален
      eofbit, // вижда се краят на файла
      failbit,// следващата операция не може да се изпълни
      goodbit;// потокът е наред
  //...
};
```


Състояние на потока

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

```
template<class Ch, class Tr=char_traits<Ch> >
class basic_ios: public ios_base {
public:
  //...
  // връща флаговете на състоянието на потока
  iostate rdstate(void) const;

  // установява флаговете на състоянието
  void clear(iostate f=goodbit);

  // добавя f към флаговете на състоянието
  void setstate(iostate f) {
    clear(rdstate()|f);
  }
  //...
  operator void* () const;
};
```

Състояние на потока

- Операторът, дефиниран в ред 16, 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

Формата за въвеждане на променливи от типа `point` е: `(x,y)`, където `x` и `y` са числа с плаваща точка.

```
1  istream& operator>>(istream& in, point& p) {
2    double x,y;
3    char c;
4
5    in >> c;
6    if(c!='(') {
7      in.clear(ios_base::badbit);
8      return in;
9    }
10   in >> x >> c;
11   if(c!=',') {
12     in.clear(ios_base::badbit);
13     return in;
14   }
15   in >> y >> c;
16   if(c!=')') {
17     in.clear(ios_base::badbit);
18     return in;
19   }
20   if(in.good()){
21     p.set_x(x);
22     p.set_y(y);
23   }
24   return in;
25 };
```


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 входният поток, трябва символите да се четат като символи, а след това да се проверява какво точно е прочетено. За тази цел се използва семейството от методи `get()`, дефинирани в `basic_istream<>`.

```
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` срещнатият 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(ch)` – връща следващият символ от потока `in`, но не го изтрива от потока.

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 except)` – 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
3  using namespace std;
4
5  int
6  main(int argc,char* argv[]) {
7    ios_base::iostate oldstate=cin.exceptions();
8    cin.exceptions(ios_base::eofbit|
9          ios_base::badbit|ios_base::failbit);
10   try {
11     for(;;){
12       int i;
13       cin >> i;
14       cout << "i=" << i;
15     }
16   } catch(const ios_base::failure& toCatch) {
17     cout << "ios_base::failure␣catched..." << endl;
18   }
19   return 0;
20 }
```