

## Lecture 12

### The design of the C++ stream library

We know how to use cin and cout to get input and produce output in a C++ program. But hiding within the innocent names cin and cout is a large system of interoperating classes that makes the system very flexible.

We will study the system, with an aim to understand how flexible software modules can be made using OO techniques.

#### References:

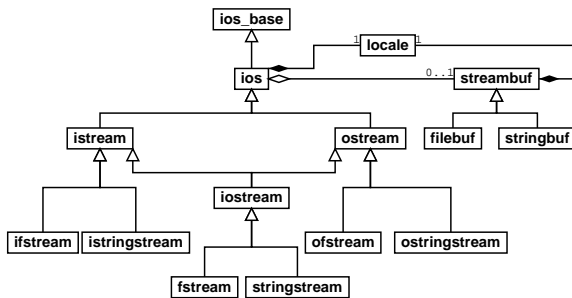
- The C++ Programming Language: Chapter 21.
- G++ 3.0 I/O howto (available in the web page)

PMOOP(0396A)

PMOOP(0396A)-12.1

#### General design

C++ I/O classes are much more complicated than just cin and cout.



Note: this is a bit simplified: all classes other than ios\_base are actually template classes, with 2 common instances, one for 8-bit char, one for 32-bit wchar (for international characters).

PMOOP(0396A)-12.2

PMOOP(0396A)-12.3

#### The sharing of responsibilities

- ios\_base is responsible for **managing the formatting flags**, e.g., whether to output in binary, the width of the output, etc.
- streambuf is responsible for **managing the external "device"** used for input and output. It also manages a buffer to make I/O efficient.
- ios **manages the state** of a stream, e.g., is it good for reading, is EOF seen, etc. It also bridges between the ios and streambuf hierarchy.
- istream and ostream **implements the conversion** between objects and a stream of characters. iostream **combines istream and ostream using MI** (with virtual base ios) to provide functionalities of both.
- stringbuf and filebuf provide **alternative character source and destinations**, to strings of program and to files in disk. Derived classes of stream classes provide convenient access points to these buffer classes.
- locale provides **national conventions** (e.g., numeric format).

#### The ios\_base class: formatting states

- What we call **I/O** is understood by C++ library as "**conversion from an object to a sequence of characters and vice versa**".
- How these conversions are done? E.g., is a number 45.2 to be written like 45.20 or +4.52e1? It is specified by some **formatting flags**.
- The ultimate base of any input and output stream is a **set of format flags**, represented by the ios\_base class.
- Most of these flags use just one or two bits. There are exceptions, though, e.g., the field width or floating point precision are integers.
- The class provide **constants** for these bits, e.g., left, right, skipws, boolalpha, dec, hex, oct, scientific, fixed, etc.  
Most I/O constants, possibly unrelated to formatting, are defined in ios\_base. E.g., ios\_base::failbit (a stream failed), ios\_base::app (open a file for append), etc.
- Functions are provided to set and get these flags: flags get and set flags, while setf and unsetf add and remove a flag.

PMOOP(0396A)-12.4

#### Example use

One can do something like the following:

```
cout << 1234 << ' ' << 1234.0 << endl; // => 1234 1234
cout.setf(ios::hex, ios::basefield); // Hex for integers
cout << 1234 << ' ' << 1234.0 << endl; // => 4d2 1234
cout.setf(ios::showbase); // Show also the base
cout.setf(ios::scientific, ios::floatfield); // Scientific notation for floats
cout.width(10); // Next field is 10
cout << 1234 << ' ' << 1234.0 << endl; // => 0x4d2 1.234000e+03
cout.width(15);
cout.precision(3); // 3 digits after decimal point
cout.setf(ios::showpos); // Show positive sign
cout.setf(ios::uppercase); // Uppercase 'E'
cout << 1234.0 << endl; // => +1.234E+03
cout.setf(ios::fixed, ios::floatfield); // Fixed point format for floats
cout << 1234.0 << endl; // => +1234.000
```

All "ios" here should really be "ios\_base", but we can do that only in g++ version 3.0. Most field manipulations can be done more conveniently by "manipulators".

PMOOP(0396A)-12.5

### The ios class: non-format states

- The *ios* class adds **non-format states**, and **format states** which depends on the character type (8 or 32 bit) or locale information.
- E.g., it adds the format state of “**fill character**” used to fill up the unused space when the field width is too big (set using *fill*).
- It also adds the **failure state** of the stream. It includes several flags: *badbit*, *eofbit* and *failbit*, **indicating why the last stream operation failed**: corrupted, ended, or wrong format.
  - Check them only **after** the stream fails!
- The bits set can be retrieved using *rdstate*. The *bad*, *fail* and *eof* function do the checking for individual bits. The *clear* function sets the bits.
  - Called clear because the primary use it to clear the error state.
- A state is **good if all these flags are not set** (the *good* function return true). This can also be checked by casting the stream to a *bool*. When a stream is not good, all stream operations do nothing.

PMOOP(0396A)-12.6

### Example

```
cout.fill('>');
for (;;) {
    int a;
    cin >> a;
    if (!cin) {
        if (!cin.eof() && cin.fail()) {
            cout << "Format error" << endl;
            cin.clear();
            cin.ignore(1000, '\n');
            continue;
        }
        break;
    }
    cout << "Got " << setw(10) << a << endl;
}
if (cin.bad()) {
    cout << "Bad stream!!" << endl;
}
```

PMOOP(0396A)-12.7

### A wrong example

Many students wrongly checks for end of file like this.

```
while (!cin.eof()) {
    int number;
    cin >> number;
    ... // process number
}
```

This is **completely** wrong: we can know that the cin stream has ended **only after** the stream fail, probably by *cin >> number*. The correct way:

Simpler, stop on any error

```
int number;
while (cin >> number) {
    ... // process number
}
```

Complete error checking

```
for (;;) {
    int number;
    if (!(cin >> number)) {
        if (cin.eof()) break;
        // process error
    }
    ... // process number
}
```

PMOOP(0396A)-12.8

### The istream and ostream classes: between data and characters

- All the **formatting work** is done by *istream* and *ostream*. E.g., when you perform an output operation like “*cout << 10;*”:
  - The number (10) is converted into a character string (“10”).
  - The converted characters are passed to the buffer associated with the stream.
- Similar things occur when you perform **input**. Just that the direction is reversed. (i.e., characters are extracted from the buffer, and converted to requested object types.)
- You should be familiar with most things here, especially the << (put to) and >> (get from) operators.
- One can use **manipulators** to do most formatting tasks, rather than using the *ios\_base* functions directly.
  - This usually results in nicer-looking programs.

PMOOP(0396A)-12.9

### Example

The same program, this time using manipulators.

```
#include <iomanip>
using namespace std;
...
cout << 1234 << ' ' << 1234.0 << endl;
cout << hex;
cout << 1234 << ' ' << 1234.0 << endl;
cout << showbase; // needs g++-3.0
cout << scientific; // needs g++-3.0
cout << setw(10) << 1234 << ' ' << 1234.0 << endl;
cout << setw(15) << setprecision(3);
cout << showpos; // needs g++-3.0
cout << uppercase; // needs g++-3.0
cout << 1234.0 << endl;
cout << fixed; // needs g++-3.0
cout << 1234.0 << endl;
```

Note that quite some manipulators are not implemented in g++ 2.95.

PMOOP(0396A)-12.10

### Note: operator overloading

- We say that the library is **extensible** to new types, right?
- Then how to **extend the capability** of *istream* and *ostream* to handle new object types?
- The answer: **you can define a << and >> operator** for the *ostream* and *istream* classes.
- Typically, we **use the primitive << and >> operator** to construct the object or to turn it into characters.
- Remember that **the characters must be passed to the stream buffer** rather than printed directly. In particular, don't just *printf* it using the C library, because that makes the result unusable for other types of streams.
- Usually, error checking can be done easily.

PMOOP(0396A)-12.11

### Example

Let's see how we can handle the reading of a complex number in 3 formats: 2.5, (2.5), and (2.5 0). Note how all errors are handled in this code.

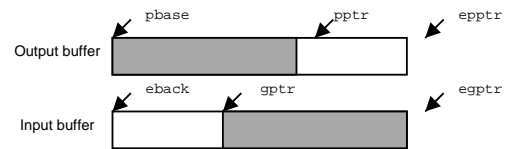
```
istream& operator>>(istream& s, complex& a) { // overload >> operator
    double re = 0, im = 0;
    char c = 0;
    s >> c;
    if (c == '(') {
        s >> re >> c;
        if (c == ',') s >> im >> c;
        if (c != ')') s.clear(ios::badbit);
    } else {
        s.putback(c); // put the character back into the stream buffer
        s >> re;
    }
    if (s) a = complex(re, im);
    return s;
}
```

PMOOP(0396A)-12.12

### The stringstream class: buffering

The stringstream class makes I/O efficient by providing buffering.

- There are two areas allocated for input and output buffering for each stringstream, one for input, one for output.
- The **output buffer** is from *pbase()* to just before *pptr()*, with the part from the beginning to *pptr()* being filled.
- The **input buffer** is from *eback()* to just before *gptr()*, with the part from *gptr()* to the end of the buffer being filled.



PMOOP(0396A)-12.13

### Buffer and device

- When the output occur while the buffer is **full**, the *overflow()* member function is used to put all characters of the buffer (and the character causing the overflow) to the real device.
- The output buffer can also be emptied by **flushing**, which calls the *sync()* member function. Both *overflow()* and *sync()* puts the characters between *pbase()* and *pptr()* to the device. But *overflow()* puts also the overflowing character.
- When all characters are **read**, the *underflow()* member function is used to retrieve new characters from the real device.
- All these member functions are virtual protected, so that derived classes can **redirect the output and input** to different destinations.
- The actual buffer storage is owned by the derived classes of *stringstream*. After allocation, they use *setg()* and *setp()* to tell stringstream the 3 different pointers of the get and put area respectively.

PMOOP(0396A)-12.14

### Derived classes of stringstream

- The default *stringstream* redirects all calls to a **few protected functions** which does nothing. Derived class define these function.
- **Buffer is not allocated** (so that derived classes can control how this is done), but the above redirection of calls does allow a buffer.
- The cin and cout streams are initialized to a stream buffer that redirect all the input and output to the **standard C library FILE objects**. So the C++ and C I/O can actually interoperate.
- But the whole object-to-string conversion framework can be used for **other "real devices" as well**.
- These include some **standard classes** defined by the library, and also other classes that **we define ourselves**. See how polymorphism allows all the work of *istream* and *ostream* to be reused for a completely different purpose.

PMOOP(0396A)-12.15

### Strings as real device: stringstream and stringstream

- The easier redirection is done by *stringstream*, which use an **internally allocated string as buffer**. It is defined in the `<sstream>` header file.
- The *istringstream*, *ostringstream* and *stringstream* classes can be used to conveniently create such buffer and associate it with a *(i/o)stream*.
- The stream buffer, and the convenience classes, provides a function *str* to **get and set the value** of the string within the buffer.
- E.g., in order to convert between string and number (no error check):

```
// string to double // double to string
double string2double(string s) { string double2string(double d) {
    istringstream ss(s);          ostringstream os;
    double ret = 0;              ss << d;
    ss >> ret;                   return os.str();
    return ret;                  }
}
```

PMOOP(0396A)-12.16

### File as real device: filebuf

- The filebuf class defines buffers based on **disk files**: Upon overflow or underflow, data is written to and read from the underlying disk file.
- The filebuf class provides member functions *open()* and *close()* which can be used to open and close a file.
- The *open* call takes 2 arguments: the filename, and the **open mode**:
  - *ios\_base::in*, *ios\_base::out*: open for **input** or **output**.
  - *ios\_base::ate*, *ios\_base::app*: **seek to file end** at file opening or before each file write operation.
  - *ios\_base::trunc*: **truncate** the file when opening.
- They can be combined together using the bitwise-or (`|`) operator.
- **Amount of buffer space** automatically allocated is system defined. (Linux: 8k.) A **new buffer** can be allocated to it by calling the *pubsetbuf(buf, size)* function, which adopts a *buf* of *size* bytes (requires g++ 3.0).

PMOOP(0396A)-12.17

## filebuf: Continued

- Like string streams, we have **file streams** *ifstream*, *ofstream* and *fstream* that make it easy to create a file buffer and attach it to a stream object. They are defined in header `<fstream>`
- File buffers also supports **seeking** the file pointer to a particular offset (the *streambuf* defines this, but it does nothing). *stringbuf* support seeking as well.
- This is usually done through the **istream and ostream interface** *seekp* and *seekg* (seek the get and put pointers). It accepts 2 arguments:
  - The offset to seek.
  - Where to seek from: **beginning** (*ios\_base::beg*) of file, **current position** (*ios\_base::cur*) or **ending** (*ios\_base::end*) of file. The put pointer and get pointer of a filebuf is tied, so seeking one will move the other. This is not the case for *stringbuf*.
- The **current** file position can be found using *tellp()* and *tellg()*.

PMOOP(0396A)-12.18

## Example

Suppose we want to open a file, write 10 lines of 10 '0's to file test.txt:

```
ofstream ofile("test.txt", ios::out | ios::trunc);
for (int i = 0; i < 10; ++i) {
    for (int j = 0; j < 10; ++j)
        ofile << '0';
    ofile << endl;
}
```

If we change our mind and want the 5-th line to contain 0 to 9 instead, and move the put pointer back to the end:

```
ofile.seekp(44, ios::beg);
for (int j = 0; j < 10; ++j)
    ofile << j;
ofile.seekp(0, ios::end);
```

PMOOP(0396A)-12.19

## Writing your own stream buffer

You can write **your own stream buffer**. For output, override the *sync* and *overflow* function, and for input, override the *underflow* function.

This is for unbuffered I/O. Buffered case is more complicated, since one needs to handle the buffer.

E.g., an output buffer which print everything in uppercase...

```
#include <streambuf>
class outbuf: public std::streambuf {
protected:
    virtual int overflow(int c) { // No buffer, so c is the only thing to print
        std::cout << char(std::toupper(c));
        return c;
    }
};
int main() {
    outbuf buf; // Make our buffer
    std::ostream ost(&buf); // And make an ostream out of it
    ost << "Hello, World!!" << endl; // Will print in upper case
}
```

PMOOP(0396A)-12.20

## Conclusion

- The C++ standard I/O library is carefully designed to **break up the task into 3 parts**: input, output and buffering.
- Each part can be **independently modified and extended**. The flexibility system is provided by multiple inheritance and aggregation. The buffering part also performs device interaction. Actually there is another part, the management of format flags. It can also be extended, although we didn't learn the extension interface.
- Note that the **break-up of responsibility** is very important: otherwise it will be impossible for independent improvements.
- In order to actually perform input and output, implementations of these 3 parts must be **separately created and combined**.
- This makes it a bit inconvenient to use the classes. The problem is **reduced using convenience classes** like *fstream* and *stringstream*.

PMOOP(0396A)-12.21

## More about overloading assignment operator

- We see that we can **overload an operator** (like `>>`). In general, operators are defined just like normal functions—except that the name contains a word **operator** and the operator symbol. We cannot change the number of argument of operators, and we cannot change the precedence. For `++` and `-`, the postfix operator will receive an extra int argument to differentiate from the prefix operator.
- One particular operator we usually want to override is the assignment operator (`=`).
- There is a **default assignment operator** defined. Similar to the default copy constructor, it performs **member-wise copy**.
- So if the default copy constructor works, the default assignment operator also work. And vice versa. In general, if we overload destructor, we also overload copy constructors and assignment operator.

PMOOP(0396A)-12.22

## Precautions of assignment operator overloading

- In our case of `TIntStack`, we really should **redefine the assignment operator** so that it does the right thing.
- The copy constructor can be used for the assignment operator, right?
- Wrong. Copy constructor makes a new object, but assignment operator substitutes the old object by a new one.
- So two complications:
  1. The assignment operator must take care to **destroy and deallocate** the old object before making a new object.
  2. In the **special case** in which a self assignment is done, the assignment operator **must not fail**. This can be achieved by checking for self assignment, or by not deallocating the old object until the new object is created (better).
- It should return **\*this** to allow chained assignments like `a=b=c`.

PMOOP(0396A)-12.23

### Example

Let's compare the correct copy constructor and assignment operator of the *TIntStack* class (with deep copy).

```
TIntStack::TIntStack
(const TIntStack &other) {
    if (other._count == 0) {
        _count = 0; _capacity = 5;
        _stackData = new int[5];
    } else {
        _count = _capacity
        = other._count;
        _stackData = new int [_count];
        for (int i = 0; i < _count; ++i)
            _stackData[i]
            = other._stackData[i];
    }
}

TIntStack& TIntStack::operator=
(const TIntStack &other) {
    int* oldData = _stackData;
    if (other._count == 0) {
        _count = 0;
    } else {
        _count = _capacity
        = other._count;
        _stackData = new int [_count];
        for (int i = 0; i < _count; ++i)
            _stackData[i]
            = other._stackData[i];
        delete oldData; // do this last
    }
    return *this;
}
```