

## Topic 9

### C++ Review Part II: More on IO Streams and Exception Handling

資料結構與程式設計  
Data Structure and Programming

11.28.2018

### What we have learned...

- ◆ iostream, fstream
  - Header files, classes, objects
- ◆ I/O stream manipulator
  - Most of them are sticky

### Key Concept #1: User-Defined Stream Manipulators

- ◆ Programmers can create their own stream manipulators
  - Interface: an ostream member function is defined as manipulator ---  
ostream& operator << (ostream& (\*p)(ostream&));
- ◆ [e.g.] Output stream manipulators
  - Must have return type and parameter type as ostream &
  - e.g.

```
ostream& myendl(ostream& os) {
    return (os << endl << "Ric> ");
}
=====
int main() {
    cout << myendl;
}
```

### Key Concept #2: Formatted vs. Unformatted I/O

- ◆ Formatted I/O
  - "High-level", bytes are grouped into meaningful units
    - Integers, floating-point numbers, characters, etc.
    - Satisfactory for most I/O other than high-volume file processing
  - I/O operations are sensitive to data types
    - Improper data cannot "sneak" through
  - Using operators "<<" and ">>", I/O manipulators
- ◆ Unformatted I/O
  - Low-level, individual bytes are the items of interest
  - High-speed, high-volume
  - Not particularly convenient for programmers
  - Member functions (e.g. get, getline, put, read, write...)
  - May have portability problem

## Type-Safe I/O (Formatted I/O)

- ◆ << and >> operators are overloaded to accept data of specific types
  - Attempts to input or output a user-defined type that << and >> have NOT been overloaded will result in compiler errors
- ◆ If unexpected data is processed, error bits are set
  - User may test the error bits to determine I/O operation success or failure
- ◆ ostream& operator <<
  - Does not print out until '\n' or 'flush()' is called
- ◆ istream& operator >>
  - Stop at white space, but not process until '\n' is entered

## Recall: Overloading "<<" operator for user-defined types

```
class MyClass {
    // friend to "global domain"
    // so that it can be accessed by "<<(cout, m)"
    friend ostream& operator <<
        (ostream& os, const MyClass& m);
}; // Why friend? friend to whom???

ostream& operator << (ostream& os, const MyClass& m) {
    os << m._dataMember1 << " is " << m._dataMember2...
    return os; // Why return "os"? Return to whom?
}

int main()
{
    MyClass m(100);
    cout << m << endl;
}
```

## "friend" is NOT a must, just a custom

```
class MyClass {
    // ostream& operator << Not a member function!!
    // (ostream& os, const MyClass& m);
};

ostream& operator << (ostream& os, const MyClass& m) {
    os << m.getData1() << " is " << m.getData2()...
    return os;
}

int main()
{
    MyClass m(100);
    cout << m << endl;
}
```

However, it is a good practice to add a friend entry in the class so that users can be aware of that such overloading exists.

## Try this...

```
◆ int i;
while (true) {
    cin >> i;
    // ... do something on i,
    // for example:
    cout << i << endl;
}
```

➔ What will you see if we enter the char 'a'?

### Key Concept #3: I/O Stream State Bits

- ◆ Control the state of the stream (as ios data members)
  - failbit
    - Set if input data is of wrong type (format error)
    - Data still remains in stream buffer
    - Usually can be recovered
  - badbit
    - Set if stream extraction operation fails (more serious)
    - Usually difficult to recover
  - eofbit
    - Set if the end of file is reached during stream input
  - goodbit
    - !(failbit | badbit | eofbit)

### I/O Stream State Bits

- ◆ Functions
  - bool eof() const;
    - Returns true when end-of-file has occurred
    - [What's wrong??] while (!infile.eof()) { infile >> ch; ... }
  - good(), fail(), bad()
  - rdstate() // read state bits
  - clear(iostate state=ios::goodbit)
    - Sets the specified bit for the stream
    - Default argument is goodbit
    - Examples
      - cin.clear();
        - Clears cin and sets goodbit
      - cin.clear( ios::failbit );
        - Sets failbit
  - setstate(iostate state) → clear(rdstate() | state)

### To fix the previous problem...

```
◆ int i;
  while (true) {
    cin >> i;
    while (cin.fail()) {
      cin.clear();
      string str;
      cin >> str;
      cerr << "Error: " << str
            << " is NOT an int!!" << endl;
      cin >> i;
    }
    cout << i << endl;
  }
```

### Or simply...

```
◆ int i;
  while (cin >> i) {
    cout << i << endl;
  }
```

- Shouldn't (cin >> i) return "istream&"?
- Then, what does "while (cin)" mean?
- Which member function does it call?

#### Key Concept #4: Use “while (fstream)” to check EOF or badbit

◆ What does this do?

```
int main()
{
    ifstream inf("aaa.txt");
    char ch;
    while (inf >> ch) cout << ch;
}
```

◆ ios::operator void\* ( ) const

- Converted to void\*; return NULL if failbit or badbit is set
- (ref) User-defined type conversion

#### operator void\* ()?? Return type?

```
class A {
public: A(int i = 0): _d(i) {}
    operator void* () const {
        return (_d != 0)? (void*)this: NULL; }
private: int _d;
};

int main() {
    A a(10);
    A b(0);
    if (a) { cout << "Yes" << endl; }
    else cout << "No" << endl;
    if (b) { cout << "Yes" << endl; }
    else cout << "No" << endl;
}
```

#### operator void\*() vs. void\* operator () ??

◆ Try this:

```
class A {
public: A(const string& s = 0): _s(s) {}
    operator int () const { return _s.size(); }
private: string _s;
};
```

◆ operator void\*() is type-casting

- A a;  
void \*p = a;

◆ void\* operator() () is operator overloading

- A a;  
void \*p = a();

#### Key Concept #5: Flags for I/O Stream Printing Format

◆ Member function flags()

- With no argument
  - Returns a value of type `fmtflags`
    - Represents the current format settings
- With a `fmtflags` as an argument
  - Sets the format settings as specified
  - Returns the prior state settings as a `fmtflags`
- Initial return value may differ across platforms
- Type `fmtflags` is of class `ios_base`

## What will be the output?

```
int main() {
    int i = 100;
    ofstream outf("ttt");
    outf << showbase << hex << i << endl;
    ios_base::fmtflags origFlags = outf.flags();
    outf.close();

    ifstream inf("ttt");
    inf.flags(origFlags);
    inf >> i;
    cout << setw(10) << right << i << endl;
}
```

◆ What's the content in "ttt"?

## Key Concept #6: Unformatted I/O

- ◆ Think: sometimes you just want to read/write a file as a "stream of bytes"
  - To have better performance in I/O processing
  - You don't care/know about the type of each piece of data
    - ➔ Read data in first. Process it later in program.
    - ➔ Unformatted I/O
- ◆ Use member functions to do file accesses

## istream::get

1. With no arguments
  - int get();
  - Returns one character input from the stream
    - Any character, including white-space and non-graphic characters
  - Returns EOF when end-of-file is encountered
2. With a character-reference argument
  - istream& get ( char& c );
  - Stores input character in the character-reference argument
  - Returns a reference to the `istream` object

## istream::get

3. With three arguments: a character array, a size limit and a delimiter (default delimiter is '`\n`')
  - istream& get ( char\* s, streamsize n );
  - istream& get ( char\* s, streamsize n, char delim );
  - istream& get ( streambuf& sb );
  - istream& get ( streambuf& sb, char delim );
  - Reads and stores characters in the character array
  - Terminates at one fewer characters than the size limit or upon reading the delimiter
    - Delimiter is left in the stream, NOT placed in array
  - Null character is inserted after end of input in array
  - [note] "streamsize" may be platform dependent, usually signed int or signed long.

### Try this...

```
int main() {
    char str[5];
    while (true) {
        cin.get(str, 5, ' ');
        cout << str << endl;
    }
}
```

◆ Try to enter:

- 12 345 67

Anything wrong?

### Key Concept #7: Be aware of “failbit”

- ◆ When “cin.get()” fails to read in any character, the `failbit` is set.  
[Revised]

```
int main() {
    char str[5];
    while (true) {
        cin.get(str, 5, ' ');
        if (cin.fail()) {
            cin.clear(); char ch = cin.get();
            cout << "Clearing... " << ch << "\n";
        }
        else cout << str << endl;
    }
}
```

◆ Try again...

- 12 345 67
- 89 100

### What about...

```
int main() {
    char str[5];
    while (cin.get(str, 5, ' ')) {
        cout << str << endl;
    }
}
```

◆ Try to enter:

- 12 345 67

Anything wrong?

### Key Concept #8: `istream::getline`

- ◆ `istream& getline (char* s, streamsize n);`  
`istream& getline (char* s, streamsize n, char delim);`
- Similar to the three-argument version of `get`
    - Except the delimiter is removed from the stream
  - Three arguments: a character array, a size limit and a delimiter (default delimiter is `'\n'`)
    - Reads and stores characters in the character array
    - Terminates at one fewer characters than the size limit or upon reading the delimiter
      - Delimiter is removed from the stream, but not placed in the array
    - Null character is inserted after end of input in array

## Key Concept #9: ostream::put

- ◆ ostream& put ( char c ); // unformatted
  - Outputs a character
  - Returns a reference to the same ostream object
    - Can be cascaded
  - Can be called with a numeric expression that represents an ASCII value
  - Examples
    - cout.put( 'A' );
    - cout.put( 'A' ).put( '\n' );
    - cout.put( 65 ); // What's the output?

## Key Concept #10: More “aggressive” unformatted I/O functions

- ◆ istream& read ( char\* s, streamsize n );
  - Inputs some number of bytes to a character array
  - If fewer characters are read than the designated number, failbit is set
  - Null character is NOT inserted after end of input in array
- ◆ [Example]:

```
int main() {
    char str[10];
    while (true) {
        cin.read(str, 10);
        cout << "str is: " << str << endl;
    }
}
```
- ◆ Try this:
  - 12345
  - 67890 Anything potential problem??

## Again, be aware of “failbit”...

- ◆ [Revised example]:

```
int main() {
    ifstream fin("fff");
    char str[10];
    while (true) {
        fin.read(str, 10);
        str[fin.gcount()] = 0;
        cout << "str is: " << str << endl;
        if (fin.fail())
            break;
    }
}
```

Content in “fff”  
123456789012345

## Key Concept #11: ostream::write()

- ◆ ostream& write ( const char\* s , streamsize n );
  - Outputs some number of bytes from a character array
- ◆ Examples:
  - cout.write("1234567890", 5) << endl;
  - cout.write("12345", 10) << endl;
  - cout.write("12345\n7890", 10) << endl;
  - cout.write("12345\07890", 10) << endl;
- ◆ Be aware of the “size” you write!!
- ◆ Take care of NULL, EOF,... etc.
- ◆ Similar for “ofstream::write()”

## Key Concept #12: More istream member functions

- ◆ `istream& ignore`  
( `streamsize n = 1, int delim = EOF` );
  - Reads and discards a designated number of characters or terminates upon encountering a designated delimiter
- ◆ `istream& putback ( char c )`;
  - Places previous character obtained by a `get` from the input stream back into the stream
- ◆ `int peek ( )`;
  - Returns the next character in the input stream, but does not remove it from the stream

## Key Concept #13: Tying an Input Stream to an Output Stream

- ◆ `istream` member function `tie`
  - `ostream* tie ( ) const`;
    - Returns a pointer to the tied output stream
  - `ostream* tie ( ostream* tiestr )`;
    - Ties the `istream` object to *tiestr* and returns a pointer to the `ostream` object previously tied
- ◆ Synchronizes an `istream` and an `ostream`
  - Ensures outputs appear before their subsequent inputs
- ◆ By default, the standard objects `cin`, `cerr` and `clog` are tied to `cout` (Why?)
  - Examples
    - `cin.tie( &cout )`;
      - Ties standard input to standard output
      - C++ performs this operation automatically
    - `inputStream.tie( 0 )`;
      - Unties `inputStream` from the `ostream` it is tied to

## `istream::tie` Example

```
int main () {
    ostream *prevstr;
    ofstream ofs;
    ofs.open ("test.txt");
    cout << "tie example:" << endl;
    *(cin.tie()) << "This is inserted into cout";
    prevstr = cin.tie(&ofs);
    *(cin.tie()) << "This is inserted into the file";
    cin.tie (prevstr);
    ofs.close(); return 0;
}
```

## Key Concept #14: `ostream` or `ostream*`

- ◆ Why do the argument and return type of “`istream::tie()`” have the type “`ostream*`”, not “`ostream`”?
- ◆ Why not:
  - `cin.tie(cout)`;
  - `cin.tie() << “blah, blah...” << endl`;
- ◆ You cannot “copy” a stream object!!
  - Use “pointer” or “reference” instead



## Key Concept #15: File-position pointer

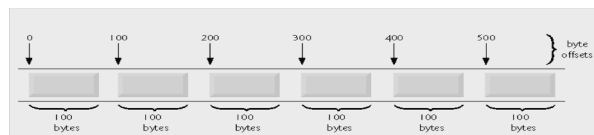
- ◆ The **byte number** of the next byte to be read or written
- ◆ `seekg()` for `ifstream` and `seekp()` for `ofstream`
  - Repositions the file-position pointer to the specified location
  - Two prototypes
    - `seekg(pos)` or `seekg(offset, direction)`
- ◆ `tellg()` for `ifstream` and `tellp()` for `ofstream`
  - Returns current position of the file-position pointer as type `long`

## Seek direction

- ◆ `ios::beg` – default, position relative to the beginning
- ◆ `ios::cur` – relative to current position
- ◆ `ios::end` – relative to the end
- ◆ Examples
  - `fileObject.seekg( n );`
    - Position to the  $n$ th byte of `fileObject`
  - `fileObject.seekg( n, ios::cur );`
    - Position  $n$  bytes forward in `fileObject`
  - `fileObject.seekg( n, ios::end );`
    - Position  $n$  bytes back from end of `fileObject`
  - `fileObject.seekg( 0, ios::end );`
    - Position at end of `fileObject`

## Key Concept #16: Random-Access Files

- ◆ Necessary for instant-access applications
  - Such as transaction-processing systems
    - cf: use ">>", "<<" for *sequential file access*
  - A record can be inserted, deleted or modified without affecting other records
- ◆ Various techniques can be used
  - Require that all records be of the same length, arranged in the order of the record keys
    - Program can calculate the exact location of any record
      - Base on the record size and record key



## Key Concept #17: Use “read” and “write” for Random-Access Files

- ◆ `istream& read(char *str, streamsize nBytes)`
  - Read a number of bytes from the current file position in the stream into an object
- ◆ `ostream& write(const char *str, streamsize nBytes)`
  - Writes a number of bytes from a location in memory to the stream

## Random-Access Files

```
int main() {
    StudentRecord rec;
    ofstream outf("studentRecord.dat");
    for (int i = 0; i < numRecords; ++i) {
        cin >> rec;
        outf.write(reinterpret_cast<const char *>(&rec),
                    sizeof(StudentRecord));
    }
    outf.close();
    ifstream inf("studentRecord.txt");
    for (int i = 0; i < numRecords; ++i)
        inf.read(reinterpret_cast<char *>(&rec),
                  sizeof(StudentRecord));
}
```

## Operator reinterpret\_cast

```
StudentRecord rec;
outf.write(reinterpret_cast<const char *>(&rec),
            sizeof(StudentRecord));
```



- ◆ Casts a pointer of one type to an unrelated type
  - Also converts between pointer and integer types
- ◆ Is performed at compile time
  - Does not change the value of the object pointed to
- ◆ May lead to serious execution-time errors

## Key Concept #18: String Stream (Stream of string)

- ◆ Ref: "sprintf()" in C
    - print something to a string
  - ◆ Sometimes we would like to compose a string from different sources
    - E.g. Compose displayed names with IDs

```
ostringstream st;
for (int i = 0; i < n; i++) {
    st << "Member" << i;
    _name = st.str();
}
```
- ==> Actually, this is not correct...

## More examples on String Stream

```
// #include <sstream>
int main()
{
    int i;
    cin >> i;

    ostringstream st;
    st << i << " square is " << i * i;

    string str = st.str();

    cout << str << endl;
}
// What's the output??
```

## String Stream

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

    ostreamstream st;
    st << i << " square is " << i * i;
    string str = st.str();
    cout << str << endl;

    st << i << " is " << i;
    str = st.str();
    cout << str << endl;
}
// What's the output??
// How to clear the previous string? clear()?
```

## The Solution is... ^^|||

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

    ostreamstream st;
    st << i << " square is " << i * i;
    string str = st.str();
    cout << str << endl;

    st.str("");
    st << i << " is " << i;
    str = st.str();
    cout << str << endl;
}
```

## Key Concept #19: class streambuf

- ◆ A *stream buffer* is an object in charge of performing the reading and writing operations of the *stream* object it is associated with.
  - The stream delegates all such operations to its associated *stream buffer* object, which is an intermediary between the *stream* and its *controlled input and output sequences*.
- ◆ All *stream* objects, no matter whether buffered or unbuffered, have an associated *stream buffer*. Some *stream buffer* types may then be set to either use an intermediate *buffer* or not.

\*ref: [www.cplusplus.com](http://www.cplusplus.com)

## Key Concept #20: ios::rdbuf()

- ◆ There are many member functions for class "streambuf"... to many to cover in this class.
- ◆ "ios::rdbuf()": to get or set streambuf for a stream object
  - streambuf\* rdbuf() const;
    - To get the streambuf
  - streambuf\* rdbuf (streambuf\* sb);
    - To set the streambuf

### “ios::rdbuf” example

```
int main () {
    ofstream filestr("test.txt");
    streambuf *backup = cout.rdbuf();
    streambuf *psbuf
    = filestr.rdbuf();
    cout.rdbuf(psbuf);
    cout <<
        "This is written to the file\n";
    cout.rdbuf(backup);
    cout << "This is written to stdout\n";
}
```

### More “streambuf” example

```
int main () {
    stringstream ss;
    streambuf *backup = cout.rdbuf();
    streambuf *psbuf = ss.rdbuf();

    ss << "Let me write something beforehand..."
        << endl;
    cout.rdbuf(psbuf);
    cout << "This is written to stringstream\n";
    cout.rdbuf(backup);

    cout << "Hello!!" << endl;
    cout << ss.str() << endl;
}
```