C++ Input/Output Overview

- C++ stream classes
  - What is a C++ I/O stream?
  - Extractors and Inserters (shift operators)
  - Formatting and Manipulators
- Lower level interfaces
  - C FILEs
  - direct system calls

What is a C++ I/O Stream?

- A stream is a logical entity that can produce or consume data.
  - Produce for input
  - Consume from output
- The meaning of produce/consume depends on the actual device that the stream is connected to, e.g.,
  - Read from a keyboard, write to a screen/window
  - Read from/write to a disk file
  - Read from/write to a pipe (between processes)
  - Read from/write to a socket handle
- Operations on a stream are ordered, so input and output are ordered.
- Input/Output operations can be written independently of the actual device, and of the particular stream type
Popular Output Stream (ostream) Subclasses

- ofstream for file output
  ```cpp
class ofstream : public fstreambase, public ostream {
public:
    ofstream () : fstreambase () { }
    ofstream (int fd) : fstreambase (fd) { }
    ofstream (const char *name, int mode=ios::out,
             int prot=0664)
       : fstreambase(name, mode, prot) { }
       // [...]
};
```

- ostrstream for string output
  ```cpp
  ostrstream (char *cp, int n, int mode=ios::out)
     : strstreambase(cp,n,mode){}
  char *str() { return ((strstreambuf*)_strbuf)->str(); }
  ```

Example C++ Stream Usage

- Simple example that echos one line read from standard input to both standard output and standard error:

```cpp
#include <iostream.h> /* or <iostream>, with ANSI C++ */
#include <stdio.h>    /* for BUFSIZ */

int main (int, char *[])  
{  
    char buf[BUFSIZ];    // BUFSIZ is a handy constant.  
    cin >> buf;  
    cout << buf << endl; 
    cerr << buf << endl; 
    return 0; 
}
```

Can Combine Input and Output Streams

- iostream, i.e.,

```cpp
class iostream : public istream, public ostream  
{   
    public:  
    iostream () { }  
    iostream (streambuf* sb, ostream *tied = NULL);  
};
```
C++ Streams: Predefined Streams

- **cin**: standard input istream (from keyboard)
- **cout**: standard output ostream (to screen/window)
  - buffered
- **cerr**: error output ostream (to screen/window)
  - unit buffered (flushed at end of each operation)
- **clog**: error output ostream (to screen/window)
  - buffered
  - very rarely used, and not recommended
- There are also wide-character versions (**win**, **wout**, **werr**, and **wlog**)
  - implemented using templates, instantiated on character type

Extractors

- Also called *right-shift operators and input operators*

  Overloaded for built-in types:
  
  - `istream& operator>>(char& c);`
  - `istream& operator>>(short&);`
  - `istream& operator>>(unsigned short&);`
  - `istream& operator>>(int&);`
  - `istream& operator>>(unsigned int&);`
  - `istream& operator>>(long&);`
  - `istream& operator>>(unsigned long&);`
  - `istream& operator>>(bool&);`
  - `istream& operator>>(char*);
  - `istream& operator>>(float&);`
  - `istream& operator>>(double&);`

Inserters

- Also called *left-shift operators and output operators*

  Overloaded for built-in types:
  
  - `ostream& operator<<(char c);`
  - `ostream& operator<<(short n) { return operator<<((int)n); }`
  - `ostream& operator<<(unsigned short n) {
      return operator<<((unsigned int)n); }
  - `ostream& operator<<(int n);`
  - `ostream& operator<<(unsigned int n);`
  - `ostream& operator<<(long n);`
  - `ostream& operator<<(unsigned long n);`
  - `ostream& operator<<(bool b) { return operator<<((int)b);`
  - `ostream& operator<<(const char *s);`
  - `ostream& operator<<(const void *p);`
  - `ostream& operator<<(float n) { return operator<<((double)n);`
  - `ostream& operator<<(double n);`

A Note on Extensibility

- The extractors and inserters for built-in types are overload as `ostream` and `istream` member functions.
  - Therefore, they only have one argument: the object that is being output or input.

- Overloads for user-defined classes must be global functions, not member functions.
  - Because you can’t add member functions to the `ostream` and `istream` classes.

- The global functions must have two arguments
  - The stream
  - The object, e.g.,
    - `ostream &operator<<(ostream &os, const Foo &foo);`
    - `istream &operator>>(istream &is, Foo &foo);`
Shift Operators for User-Defined Classes

class Foo {
  private:
    int i_;  char c_;  float f_; 
  friend ostream &operator<<(ostream &, const Foo &);
  friend istream &operator>>(istream &, Foo &);
};

ostream &operator<<(ostream &os, const Foo &foo) {
  // Insert white space to separate each // data member.
  os << foo.i_ << ', '  << foo.c_ << ', '
      << foo.f_ << ', '; 
  return os;
}

istream &operator>>(istream &is, Foo &foo) {
  is >> foo.i_ >> foo.c_ >> foo.f_; 
  return is;
}

Shift Operators for Derived Classes

class Bar : Foo {
  private:
    Foo foo_; 
  friend ostream &operator<<(ostream &, const Bar &);
  friend istream &operator>>(istream &, Bar &);
};

ostream &operator<<(ostream &os, const Bar &bar) {
  return os << static_cast<const Foo &> (bar) << bar.foo_; 
}

istream &operator>>(istream &is, Bar &bar) {
  return is >> static_cast <Foo &> (bar) >> bar.foo_; 
}

Shift Operators for Enums

The Simple, but Unsafe, Way:

class Sort_Request_Base {
  public:
    enum Status { UNSORTED, SORTED }; 
};

ostream &operator<<(ostream &os, const Sort_Request_Base::Status status) {
  switch (status) {
    case Sort_Request_Base::UNSORTED :
      os << "UNSORTED "; break;
    case Sort_Request_Base::SORTED :
      os << "SORTED "; break;
  }
  return os;
}

istream &operator>>(istream &is, Sort_Request_Base::Status &status) {
  char buf[32];
  is >> buf; 
  if (! ::strcmp (buf, "UNSORTED"))
    status = Sort_Request_Base::UNSORTED;
  else if (! ::strcmp (buf, "SORTED"))
    status = Sort_Request_Base::SORTED;
  else 
    status = static_cast<
      Sort_Request_Base::Status> (-1); 
  return is;
}

The Better Way to Shift Enums:

ostream &operator<<(ostream &os, const Sort_Request_Base::Status status) {
  switch (status) {
    case Sort_Request_Base::UNSORTED :
      os << "UNSORTED "; break;
    case Sort_Request_Base::SORTED :
      os << "SORTED "; break;
  }
  return os;
}

istream &operator>>(istream &is, Sort_Request_Base::Status &status) {
  char buf[32];
  is >> buf; 
  if (! ::strcmp (buf, "UNSORTED"))
    status = Sort_Request_Base::UNSORTED;
  else if (! ::strcmp (buf, "SORTED"))
    status = Sort_Request_Base::SORTED;
  else 
    status = static_cast<
      Sort_Request_Base::Status> (-1); 
  return is;
}
Enum Shifts

- The cast approach is simple, but not perfectly safe.
  - An enum isn't really an int: it may actually be smaller (and
    promotable to an int). But, most compilers use an int.
  - The shift operators don't need to be changed when enum
    values are added.
- The switch approach has much nicer output (strings instead of ints),
  but is harder to maintain.
  - Must update whenever enum values are added or removed.
  - Must size the temporary buffer for the largest possible value in
    the right-shift (input) operator.
  - Requires more space in the stream, unless very short enum value
    strings are used.

File I/O Using Shift Operators

```cpp
#include <fstream.h>

int main (int argc, char *argv[]) {
    ofstream args("args.txt");
    for (int i = 0; i < argc; ++i)
        args << argv[i] << ' ';
    args.close();
    return 0;
}
```

$ ./args a b c 1 2 3 100
$ cat args.txt
 args a b c 1 2 3 100

File I/O Using Shift Operators, (cont'd)

- Only prints out the integers:
  
```cpp
#include <fstream.h>
#include <strstream.h>

int main (int argc, char *argv[]) {
    ifstream args("args.txt");
    char buf[1024];
    while (args >> buf) {
        // Read to next whitespace.
        istringstream temp(buf);
        int i;
        if (temp >> i) // only print ints
            cout << i << ' ';
        cout << endl;
    }
    args.close();
    return 0;
}
```

$ ./args a b c 1 2 3 100
1 2 3 100
### C++ Stream Formatting Operations

- Formatting options
  - field width, *e.g.*, `cout.width(10);`
  - fill character, *e.g.*, `cout.fill('#');`
  - integer base
    ```cpp
    cout << hex << 0xFFF << dec;
    ```
  - show leading + and decimal point
    ```cpp
    #include <iomanip.h>
    cout << setiosflags(ios::showpos)
        << setiosflags(ios::showpoint);
    ```
  - scientific notation

### C++ Streams: Common Predefined Manipulators

- Output stream management
  - `flush`: move buffer contents to the stream
  - `endl`: newline plus flush
- Integer format
  - `hex`: hexadecimal
  - `dec`: decimal
  - `oct`: octal
  - `boolalpha`: to use “true” and “false” for booleans
- and others, *e.g.*,
  - field widths
  - fill characters

### C++ Stream Status Operations

- `operator!` returns true on `ifstream` false a read operation fails, such as at end-of-file
- `eof()` function tests for end-of-file
- `bad()` returns true for invalid operations, such a reading past the end of an input file
- `fail()` returns true for unsuccessful operations, such as opening an input file that is supposed to exist
- `good()` returns true if the `eof()`, `bad()`, and `fail()` would all return false

### C-style I/O

- Lower level than C++ streams
  - Requires explicit type specification for every datum
  - Can’t be extended: only uses built-in types
  - Especially clumsy for input
    - Input must match expected format exactly.
    - It’s often safer to read in strings, then parse them.
- But, still used
  - Convenient for network programming, *e.g.*, sockets
  - Familiar function-call interface
  - Supported on all platforms
C-style I/O Example

Simple example that echos one line read from standard input to both standard output and standard error:

```c
#include <stdio.h>

int main (int, char *[]) {
    char buf[BUFSIZ]; // BUFSIZ is a handy constant.
    ::scanf ("%s", buf);
    ::printf ("%s\n", buf);
    ::fprintf (stderr, "%s\n", buf);
    return 0;
}
```

Direct System Calls

- Very low level
  - Therefore, not generally recommended
- Supported by most Unix (POSIX) systems, Windows NT, etc.
- Direct output system calls are useful for fatal error messages, such as in a `new_handler` (which can be called when `new` fails)
  - Because you don't need to trust that the C++ streams and/or C libraries are in coherent states
  - Doesn't require memory allocation by C++ streams classes or C library

C++ I/O Mechanism Comparison

<table>
<thead>
<tr>
<th>I/O Approach</th>
<th>Output Mechanism</th>
<th>Input Identifier</th>
<th>Standard Output</th>
<th>Standard Input</th>
<th>Error Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ streams</td>
<td><code>&lt;&lt;</code></td>
<td><code>&gt;&gt;</code></td>
<td><code>cin</code></td>
<td><code>cout</code></td>
<td><code>cerr</code></td>
</tr>
<tr>
<td>FILES</td>
<td><code>printf</code></td>
<td><code>scanf</code></td>
<td><code>stdin</code></td>
<td><code>stdout</code></td>
<td><code>stderr</code></td>
</tr>
<tr>
<td>Unix system</td>
<td><code>write</code></td>
<td><code>read</code></td>
<td><code>0</code></td>
<td><code>1</code></td>
<td><code>2</code></td>
</tr>
</tbody>
</table>
For More Information

- On Suns, *ios.intro* and *manip* man pages