C++ Program Design
-- Input and Output

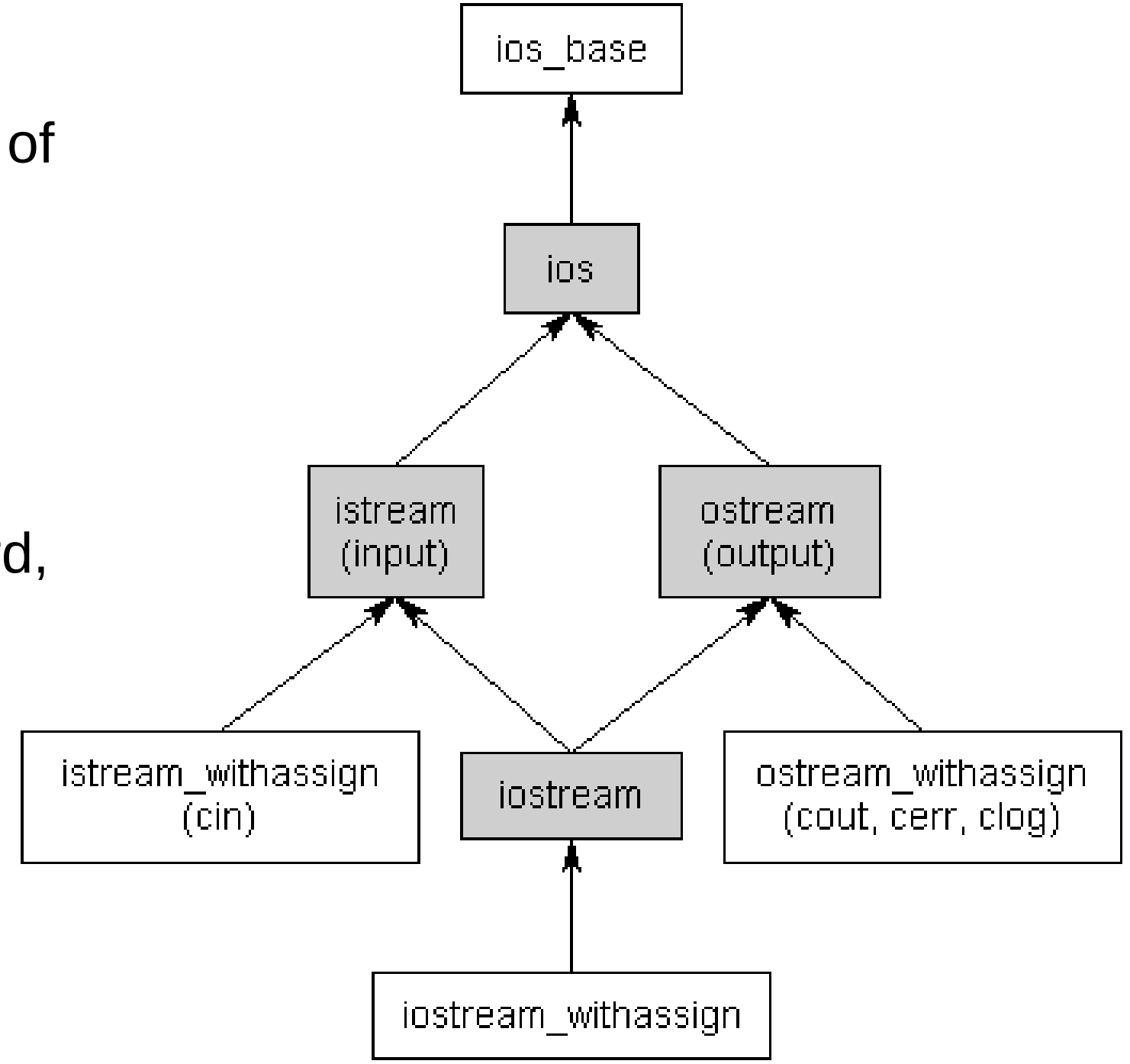
Junjie Cao @ DLUT

Summer 2016

http://jjcao.github.io/cPlusPlus

The iostream library

- a **stream** is just a sequence of characters that can be accessed sequentially.
- **Input streams** are used to hold input from a data producer, such as a keyboard, a file, or a network.
- files and networks, are capable of being both input and output sources.



Input with istream

- **The extraction operator >>**

```
char buf[10];cin >> buf;
```

- what happens if the user enters 18 characters?
- A **manipulator** is an object that is used to modify a stream when applied with the extraction (>>) or insertion (<<) operators
  - endl: prints a newline character and flushes any buffered output.
  - setw: limit the number of characters read in from a stream

```
#include <iomanip.h>
char buf[10];
cin >> setw(10) >> buf;
```

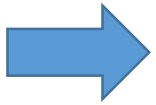
- This program will now only read the first 9 characters out of the stream (leaving room for a terminator). Any remaining characters will be left in the stream until the next extraction.

Extraction and whitespace

- extraction operator works with “formatted” data -- that is, it skips whites pace (blanks, tabs, and newlines).

```
int main(){
    char ch;
    while (cin >> ch)
        cout << ch;

    return 0;
}
```



```
int main(){
    char ch;
    while (cin.get(ch))
        cout << ch;

    return 0;
}
```

- Hello my name is Alex
- the output is:
- HellomynameisAlex

• get() also can take a maximum number of characters to read:

```
#include <iomanip.h>
char buf[10];
cin >> setw(10) >> buf;

int main(){
    char strBuf[11];
    cin.get(strBuf, 11);
    cout << strBuf << endl;

    return 0;
}
```

If we input:
Hello my name is Alex

The output is:
Hello my n

The remaining characters were left in the input stream.

get() does not read in a newline character!

```
int main(){
    char strBuf[11];
    // Read up to 10 characters
    cin.get(strBuf, 11);
    cout << strBuf << endl;

    // Read up to 10 more characters
    cin.get(strBuf, 11);
    cout << strBuf << endl;
    return 0;
}
```

If the user enters:
Hello Enter

The program prints:
Hello
and then terminate

- first get() read up to the newline and then stopped.
- The second get() saw there was still input in the cin stream and tried to read it.

- But the first character was the newline, so it stopped immediately.
- Then 1st character of strBuf is the newline '\0', so print nothing.
- The newline always stay in the cin stream unless ….

getline()

- **getline()** works exactly like get() but reads the newline as well.
- need to know how many character were extracted by getline()?

```
int main()
{
    char strBuf[100];
    cin.getline(strBuf, 100);
    cout << strBuf << endl;
    cout << cin.gcount() << " characters were read" << endl;

    return 0;
}
```

A special version of getline() for std::string

- There is a special version of getline() included in the string header

```
int main()
{
    using namespace std;
    string strBuf;
    getline(cin, strBuf);
    cout << strBuf << endl;

    return 0;
}
```

A few more useful istream functions

- **ignore(int nCount)** discards the first nCount characters.
- **peek()** allows you to read a character from the stream without removing it from the stream.
- **unget()** returns the last character read back into the stream so it can be read again by the next call.
- **putback(char ch)** allows you to put a character of your choice back into the stream to be read by the next call.

"The C++ Standard Template Library" by Nicolai M. Josuttis

Output with ostream and ios

Formatting

- two ways to change the formatting options: flags, and manipulators.
- You can think of **flags** as boolean variables that can be turned on and off.
- **Manipulators** are objects placed in a stream that affect the way things are input and output.

```
cout.setf(ios::showpos); // turn on the ios::showpos flag
cout << 27 << endl;
```

This results in the following output:
+27

```
cout.setf(ios::showpos | ios::uppercase); // turn on the
ios::showpos and ios::uppercase flag
cout.unsetf(ios::showpos); // turn off the ios::showpos flag
```

format group

- a group of flags that perform similar (sometimes mutually exclusive) formatting options.
- For example, a format group named “basefield” contains the flags “oct”, “dec”, and “hex”, which controls the base of integral values.
- `cout.setf(ios::hex); // try to turn on hex output`
- `cout << 27 << endl;`

We get the following output:
27

- setf() only turns flags on -- it isn’t smart enough to turn mutually exclusi ve flags off.
- Consequently, when we turned ios::hex on, ios::dec was still on, and io s::dec apparently takes precedence.
- There are two ways to get around this problem.

- The 1st way
- `cout.unsetf(ios::dec); // turn off decimal output`
- `cout.setf(ios::hex); // turn on hexadecimal output`
- `cout << 27 << endl;`

Now we get output as expected:

1b

- The 2nd way
- `// Turn on ios::hex as the only ios::basefield flag`
- `cout.setf(ios::hex, ios::basefield);`
- `cout << 27 << endl;`
- two parameters: the first parameter is the flag to set, and the second is the formatting group it belongs to.
- When using this form of setf(), all of the flags belonging to the group are turned off, and only the flag passed in is turned on.

Set formatting by manipulators

- Using setf() and unsetf() tends to be awkward, so C++ provides a second way to change the formatting options: manipulators

```
cout << hex << 27 << endl; // print 27 in hex
cout << 28 << endl; // we're still in hex
cout << dec << 29 << endl; // back to decimal
```

Precision, notation, and decimal points

```
cout << fixed << endl;
cout << setprecision(3) << 123.456 << endl;
cout << setprecision(4) << 123.456 << endl;
cout << setprecision(5) << 123.456 << endl;
cout << setprecision(6) << 123.456 << endl;
cout << setprecision(7) << 123.456 << endl;

cout << scientific << endl;
cout << setprecision(3) << 123.456 << endl;
cout << setprecision(4) << 123.456 << endl;
cout << setprecision(5) << 123.456 << endl;
cout << setprecision(6) << 123.456 << endl;
cout << setprecision(7) << 123.456 << endl;
```

```
123.456
123.4560
123.45600
123.456000
123.4560000

1.235e+002
1.2346e+002
1.23456e+002
1.234560e+002
1.2345600e+002
```

Width, fill characters, and justification

```
cout << -12345 << endl;
cout << setw(10) << -12345 << endl;
cout << setw(10) << left << -12345 << endl;
cout << setw(10) << right << -12345 << endl;

cout.fill('*');
cout << -12345 << endl;
cout << setw(10) << -12345 << endl;
```

```
-12345
    -12345
-12345
    -12345

-12345
****-12345
```

Stream classes for strings

#include <sstream>

- `stringstream os;`
- `os << "12345 67.89" << endl;`
- `cout << os.str();`

- `string strValue;  os >> strValue;`
- `string strValue2; os >> strValue2;`
- `// print the numbers separated by a dash`
- `cout << strValue << " - " << strValue2 << endl;`

This program prints:
12345 - 67.89

- >> iterates through the string -- each successive use of >> returns the next extractable value in the stream.
- str() returns the whole value of the stream

Conversion between strings and numbers

```
stringstream os;
int nValue = 12345;    double dValue = 67.89;
os << nValue << " " << dValue;

string strValue1, strValue2;
os >> strValue1 >> strValue2;

cout << strValue1 << " " << strValue2 << endl;
```

```
stringstream os;
os << "12345 67.89";
int nValue; double dValue;

os >> nValue >> dValue;
```

- This snippet prints:

12345 67.89

Clearing a stringstream for reuse

- The first way
- `os.str("");` // erase the buffer
- `os.clear();` // reset error flags

- The second way
- `os.str(std::string());` // erase the buffer
- `os.clear();` // reset error flags

Stream states

Flag	Meaning
goodbit	Everything is okay
badbit	Some kind of fatal error occurred (e.g. the program tried to read past the end of a file)
eofbit	The stream has reached the end of a file
failbit	A non-fatal error occurred (eg. the user entered letters when the program was expecting an integer)

- `cout << "Enter your age: ";`
- `int nAge;`
- `cin >> nAge;`
- this program is expecting the user to enter an integer.
- if the user enters non-numeric data, such as "Alex", the failbit will be set.
- If an error occurs and a stream is set to anything other than goodbit,
- further stream operations on that stream will be ignored.
- This condition can be cleared by calling the clear() function.

Basic file I/O

File output, #include <fstream>

```
ofstream outf("Sample.dat");

// If we couldn't open the output file stream for writing
if (!outf){// Print an error and exit
    cerr << "Uh oh, Sample.dat could not be opened for writing!" << endl;
    exit(1);
}

// We'll write two lines into this file
outf << "This is line 1" << endl;
outf << "This is line 2" << endl;
```

File input

take the file we wrote in the last example and read it back in from disk.

```
ifstream inf("Sample.dat");
if (!inf){
   cerr << "Uh oh, Sample.dat could not be opened for reading!" << endl;
   exit(1);
}

// While there's still stuff left to read
while (inf){
   // read stuff from the file into a string and print it
   std::string strInput;
   inf >> strInput;     cout << strInput << endl;
}
```

This is line 1
This is line 2

This
is
line
1
This
is
line
2

```
while (inf)
{// read stuff from the file into a string and print it
    std::string strInput;
    getline(inf, strInput);
    cout << strInput << endl;
}
```

This is line 1
This is line 2

File modes

Ios file mode	Meaning
app	Opens the file in append mode
ate	Seeks to the end of the file before reading/writing
binary	Opens the file in binary mode (instead of text mode)
in	Opens the file in read mode (default for ifstream)
nocreate	Opens the file only if it already exists
noreplace	Opens the file only if it does not already exist
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it already exists

```
ofstream outf("Sample.dat", ios::app);
if (!outf) {
   cerr << "Uh oh, Sample.dat could not be opened for writing!" << endl;
    exit(1);
}

outf << "This is line 3" << endl;
outf << "This is line 4" << endl;
```

This is line 1
This is line 2
This is line 3
This is line 4

Explicitly opening files using open()

```
ofstream outf("Sample.dat");
outf << "This is line 1" << endl;
outf << "This is line 2" << endl;
outf.close(); // explicitly close the file

// Oops, we forgot something
outf.open("Sample.dat", ios::app);
outf << "This is line 3" << endl;
outf.close();
```

Random file access with seekg() and seekp()

- seekg() function (for input) and seekp() function (for output).
- the g stands for “get” and the p for “put”.

Ios seek flag	Meaning
beg	**The offset is relative to the beginning of the file (default)**
cur	**The offset is relative to the current location of the file pointer**
end	**The offset is relative to the end of the file**

```
inf.seekg(14, ios::cur); // move forward 14 bytes
inf.seekg(-18, ios::cur); // move backwards 18 bytes
inf.seekg(0, ios::beg); // move to beginning of file
inf.seekg(0, ios::end); // move to end of file
```

tellg() and tellp()

- return the absolute position of the file pointer
- This can be used to determine the size of a file:

```
ifstream inf("Sample.dat");
inf.seekg(0, ios::end); // move to end of file
cout << inf.tellg();
```