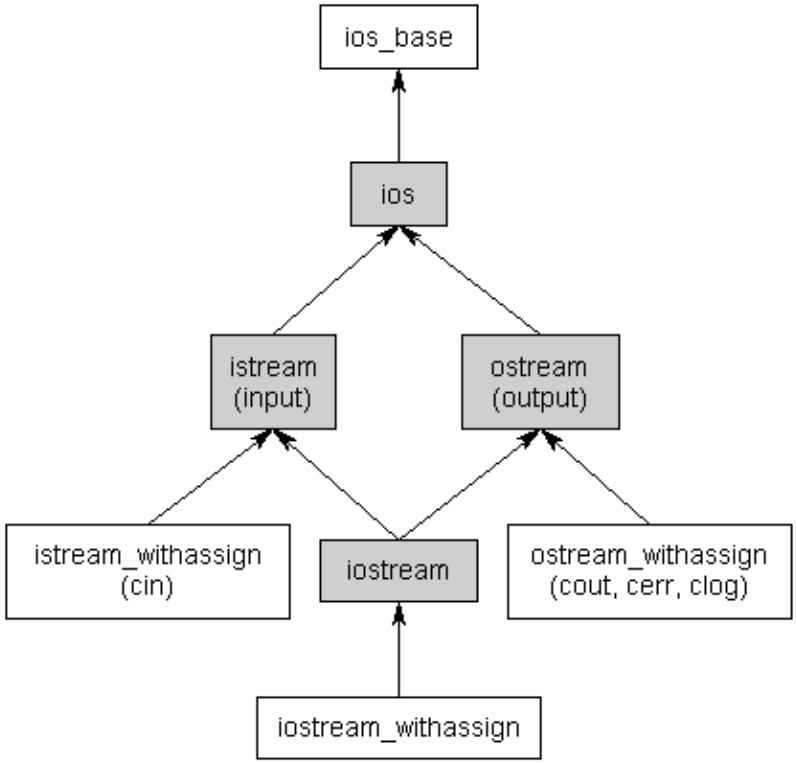
Input and Output

Outline

- Input and output (I/O) streams
- File reading
- File writing

Input and output (I/O) streams

- When you include the iostream header, you gain access to a whole hierarchy of classes responsible for providing I/O functionality (including one class that is actually named iostream). The class hierarchy for the non-file-I/O classes looks like this:



Input and output (I/O) streams

- Abstractly, a **stream** is just a sequence of characters that can be accessed sequentially. Over time, a stream may produce or consume potentially unlimited amounts of data.

- **Input streams** are used to hold input from a data producer, such as a keyboard, a file, or a network.

- **output streams** are used to hold output for a particular data consumer, such as a monitor, a file, or a printer.

- A standard stream is a pre-connected stream provided to a computer program by its environment. C++ comes with four predefined standard stream objects that have already been set up for your use. The most commonly used two are:

1. **cin** -- an istream_withassign class tied to the standard input (typically the keyboard)
2. **cout** -- an ostream_withassign class tied to the standard output (typically the monitor)

Input with Cin

- As seen in many lessons now, we can use the extraction operator (>>) to read information from an input stream. For example:

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

- What happens if the user enters 18 characters? The buffer overflows, and bad stuff happens.

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

- **cin** will treat special characters like space, tab as formatted separators,

```
int main()
{
    int a,b;
    cin>>a>>b;
    cout<<a+b<<endl;
    return 0;
}
```

Input: 1 2
Output: 3

Input with Cin

- However, cin's formatted input has side effect. It will filter out special characters like space, tab.

```
int main()
{
    char arr[20];
    cin>>arr;
    cout<<arr;
    return 0;
}
```

Input: Hello World
Output: Hello

- Oftentimes, you'll want to get user input but not discard whitespace. **get()** function, which simply gets a character from the input stream.

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

    return 0;
}
```

Input: Hello my name is Alex
Output: Hello my n

Cin.get

- The important thing to note about get() is that it does not read in a newline character! This can cause some unexpected results:

```
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;
}
```

Input: Hello!
Output: Hello!

Why didn't it ask for 10 more characters? The answer is because the 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.

Cin.getline

- **Cin.getline** works exactly like get() but reads the newline as well. works exactly like get() but reads the newline as well.

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

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

This code will perform as you expect, even if the user enters a string with a newline in it.

Cin.getline

- **Cin.getline** works exactly like get() but reads the newline as well. works exactly like get() but reads the newline as well.

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

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

This code will perform as you expect, even if the user enters a string with a newline in it.

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

- There is a special version of getline() that lives outside the istream class that is used for reading in variables of type std::string. This special version is not a member of either ostream nor istream, and is included in the string header.

```
#include <string>
#include <iostream>

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

    return 0;
}
```

Basic file I/O

Basic file I/O

- File I/O in C++ works very similarly to normal I/O. There are 3 basic file I/O classes in C++: ifstream, ofstream, and fstream.

- Unlike the cout, cin, cerr, and clog streams, which are already ready for use, file streams have to be explicitly set up by the programmer.

File output

```
#include <fstream>
#include <iostream>
#include <cstdlib> // for exit()

int main()
{
    using namespace std;

    // ofstream is used for writing files
    // We'll make a file called Sample.dat
    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;

    return 0;

    // When outf goes out of scope, the ofstream
    // destructor will close the file
}
```

If you look in your project directory, you should see a file called Sample.dat. If you open it with a text editor, you will see that it indeed contains two lines we wrote to the file.

File input

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib> // for exit()

int main()
{
    using namespace std;

    // ifstream is used for reading files
    // We'll read from a file called Sample.dat
    ifstream inf("Sample.dat");

    // If we couldn't open the output file stream for reading
    if (!inf)
    {
        // Print an error and exit
        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;
    }

    return 0;

    // When inf goes out of scope, the ifstream
    // destructor will close the file
}
```

This produces the result:

```
This
is
line
1
This
is
line
2
```

File input

- That wasn't quite what we wanted. Remember that the extraction operator deals with "formatted output", and breaks on whitespace. In order to read in entire lines, we'll have to use the getline() function.

```
#include <fstream>
#include <iostream>
#include <string>
#include <cstdlib> // for exit()

int main()
{
    using namespace std;

    // ifstream is used for reading files
    // We'll read from a file called Sample.dat
    ifstream inf("Sample.dat");

    // If we couldn't open the input file stream for reading
    if (!inf)
    {
        // Print an error and exit
        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;
        getline(inf, strInput);
        cout << strInput << endl;
    }

    return 0;

    // When inf goes out of scope, the ifstream
    // destructor will close the file
}
```

This produces the result:

```
This is line 1
This is line 2
```

File Modes

- Running the output example again shows that the original file is completely overwritten each time the program is run.
- What if, instead, we wanted to append some more data to the end of the file? It turns out that the file stream constructors take an optional second parameter that allows you to specify information about how the file should be opened.

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)
out	Opens the file in write mode (default for ofstream)
trunc	Erases the file if it already exists

File Modes

```cpp
#include <cstdlib> // for exit()
#include <iostream>
#include <fstream>

int main()
{
    using namespace std;

    // We'll pass the ios:app flag to tell the ofstream to append
    // rather than rewrite the file.  We do not need to pass in ios::out
    // because ofstream defaults to ios::out
    ofstream outf("Sample.dat", ios::app);

    // 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);
    }

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

    return 0;

    // When outf goes out of scope, the ofstream
    // destructor will close the file
}
```

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

Check End of File

```
ifstream in_stream;
string word;

in_stream.open("s1.txt");
if (in_stream.fail()){
   cout << "the input file cannot be opened \n";
   return 0;
}
int count = 0;
while (in_stream >> word )
       count++;

cout << "the total words in this file is " << count;
cout << endl;
in_stream.close();
return 0;
```

Check End of File

```
ifstream in_stream;
string word;
in_stream.open("s1.txt");

if (in_stream.fail()){
   cout << "the input file cannot be opened \n";  return 0;
}
int count = 0;
while (!in_stream.eof()){ //end of file
        in_stream  >> word;
        count++;
}
cout << "the total words in this file is " << count;
cout << endl;

in_stream.close();
return 0;
```