

The basic data type for I/O in C++ is the stream. C++ incorporates a complex hierarchy of stream types. The most basic stream types are the standard input/output streams:

<code>istream</code>	<code>cin</code>	built-in input stream variable; by default hooked to keyboard
<code>ostream</code>	<code>cout</code>	built-in output stream variable; by default hooked to console

header file: `<iostream>`

C++ also supports all the input/output mechanisms that the C language included. However, C++ streams provide all the input/output capabilities of C, with substantial improvements.

We will exclusively use streams for input and output of data.

The input and output streams, `cin` and `cout` are actually C++ objects. Briefly:

class: a C++ construct that allows a collection of variables, constants, and functions to be grouped together logically under a single name

object: a variable of a type that is a class (also often called an instance of the class)

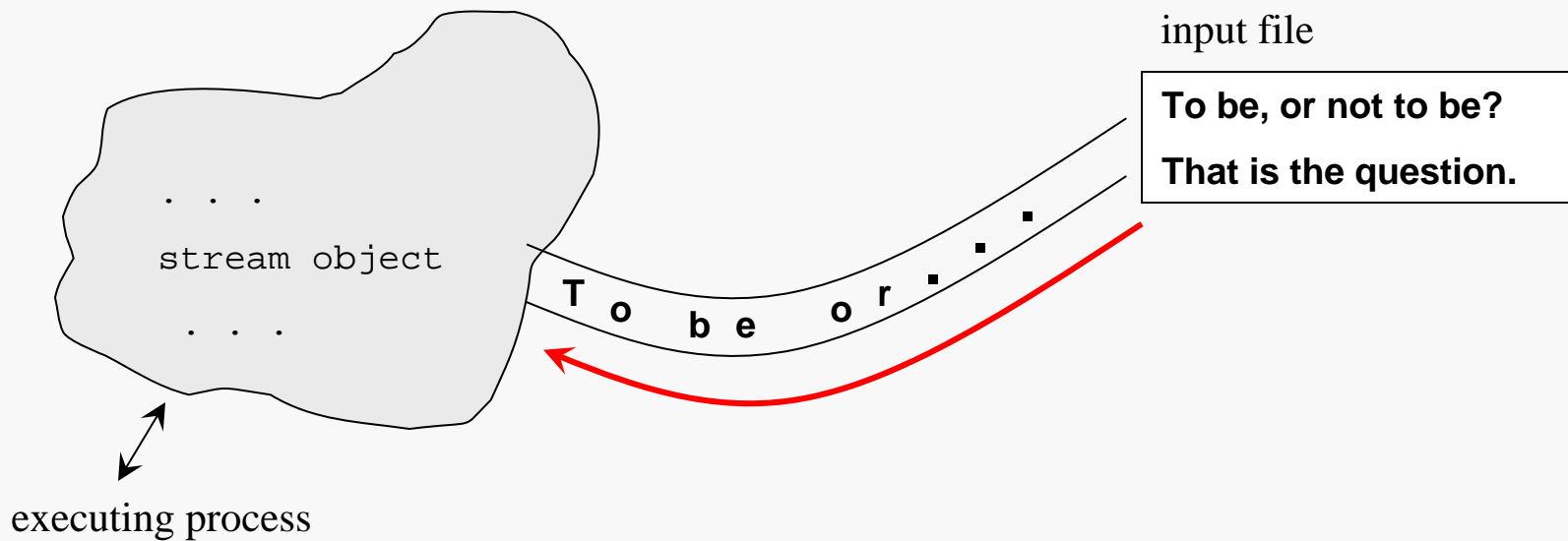
For example, `istream` is actually a type name for a class. `cin` is the name of a variable of type `istream`.

So, we would say that `cin` is an instance or an object of the class `istream`.

An instance of a class will usually have a number of associated functions (called member functions) that you can use to perform operations on that object or to obtain information about it. The following slides will present a few of the basic stream member functions, and show how to go about using member functions.

Classes are one of the fundamental ideas that separate C++ from C. In this course, we will explore the standard stream classes and the standard string class.

A stream provides a connection between the process that initializes it and an object, such as a file, which may be viewed as a sequence of data. In the simplest view, a stream object is simply a serialized view of that other object. For example, for an input stream:



We think of data as flowing in the stream to the process, which can remove data from the stream as desired. The data in the stream cannot be lost by “flowing past” before the program has a chance to remove it.

The stream object provides the process with an “interface” to the data.

To get information out of a file or a program, we need to explicitly instruct the computer to output the desired information.

One way of accomplishing this in C++ is with the use of an output stream.

In order to use the standard I/O streams, we must have in our program the pre-compiler directive:

```
#include <iostream>
```

In order to do output to the screen, we merely use a statement like:

```
cout << " X = " << X;
```

Hint: the insertion operator (<<) points in the direction the data is flowing.

where *X* is the name of some variable or constant that we want to write to the screen.

Insertions to an output stream can be "chained" together as shown here. The left-most side must be the name of an output stream variable, such as `cout`.

Inserting the name of a variable or constant to a stream causes the value of that object to be written to the stream:

```
const string Label = "Pings echoed: ";  
int totalPings = 127;  
cout << Label << totalPings << endl;
```

```
Pings echoed: 127
```

No special formatting is supplied by default.

Alignment, line breaks, etc., must all be controlled by the programmer:

```
cout << "CANDLE" << endl;  
cout << "STICK" << endl;
```

```
cout << "CANDLE";  
cout << "STICK" << endl;
```

`endl` is a manipulator.

A manipulator is a C++ construct that is used to control the formatting of output and/or input values.

Manipulators can only be present in Input/Output statements. The `endl` manipulator causes a newline character to be output.

`endl` is defined in the `<iostream>` header file and can be used as long as the header file has been included.

To get information into a file or a program, we need to explicitly instruct the computer to acquire the desired information.

One way of accomplishing this in C++ is with the use of an input stream.

As with the standard input stream, `cout`, the program must use the pre-compiler directive:

```
#include <iostream>
```

In order to do output, we merely use a statement like:

```
cin >> X;
```

Hint: the extraction operator (>>) points in the direction the data is flowing.

where `X` is the name of some variable that we want to store the value that will be read from the keyboard.

As with the insertion operator, extractions from an input stream can also be "chained". The left-most side must be the name of an input stream variable.

Assume the input stream `cin` contains the data:

```
12 17.3 -19
```

Then:

```
int    A, B;
double X;
cin >> A; // A <--- 12
cin >> X; // X <--- 17.3
cin >> B; // B <--- -19
```

If we start each time with the same initial values in the stream:

```
int  A, B;
char C;
cin >> A; // A <--- 12
cin >> B; // B <--- 17
cin >> C; // C <--- '.'
cin >> A; // A <--- 3
```

```
int  A;
char B, C, D;
cin >> A; // A <--- 12
cin >> B; // B <--- '1'
cin >> C; // C <--- '7'
```

The extraction operator is "smart enough" to consider the type of the target variable when it determines how much to read from the input stream.

The extraction operator may be used to read characters into a string variable.

The extraction statement reads a whitespace-terminated string into the target string, ignoring any leading whitespace and not including the terminating whitespace character in the target string.

Assume the input stream `cin` contains the data:

Flintstone, Fred	718.23
------------------	--------

Then:

```
string L, F;
double X;
cin >> L; // L <--- "Flintstone,"
cin >> F; // F <--- "Fred"
cin >> X; // X <--- 718.23
```

The amount of storage allocated for the string variables will be adjusted as necessary to hold the number of characters read. (There is a limit on the number of characters a string variable can hold, but that limit is so large it is of no practical concern.)

Of course, it is often desirable to have more control over where the extraction stops.

In programming, common characters that do not produce a visible image on a page or in a file are referred to as whitespace.

The most common whitespace characters are:

Name	Code
newline	\n
tab	\t
blank	(space)
carriage return	\r
vertical tab	\v

By default, the extraction operator in C++ will ignore leading whitespace characters.

That is, the extraction operator will remove leading whitespace characters from the input stream and discard them.

What if we need to read and store whitespace characters? See the `get ()` function later in the notes.

Assume the input stream `cin` contains:

```
12  17.3  -19
```

The numbers are separated by some sort of whitespace, say by tabs.

Suppose that `X` is declared as an `int`, and the following statement is executed:

```
cin >> X;
```

The type of the targeted variable, `X` in this case, determines how the extraction is performed.

First, any leading whitespace characters are discarded.

Since an integer value is being read, the extraction will stop if a character that couldn't be part of an integer is found.

So, the digits '1' and '2' are extracted, and the next character is a tab, so the extraction stops and `X` gets the value 12.

The tab after the '2' is left in the input stream.

There is also a way to remove and discard characters from an input stream:

```
cin.ignore(N, ch);
```

means to skip (read and discard) up to N characters in the input stream, or until the character ch has been read and discarded, whichever comes first. So:

```
cin.ignore(80, '\n');
```

says to skip the next 80 input characters or to skip characters until a newline character is read, whichever comes first.

The ignore function can be used to skip a specific number of characters or halt whenever a given character occurs:

```
cin.ignore(100, '\t');
```

means to skip the next 100 input characters, or until a tab character is read, or whichever comes first.

Prompts: users must be given a cue when and what they need to input:

```
const string AgePrompt = "Enter your Age: ";  
cout << AgePrompt;  
cin  >> UserAge;
```

The statements above allow the user to enter her/his age in response to the prompt.

Because of buffering of the I/O by the computer, it is possible that the prompt may not appear on a monitor before the program expects input to be entered.

To ensure output is sent to its destination immediately:

```
cout << AgePrompt << flush;  
cin  >> UserAge;
```

The manipulator `flush` ensures that the prompt will appear on the display before the input is required.

The manipulator `endl` includes a implicit `flush`.

C++ also provides stream types for reading from and writing to files stored on disk. For the most part, these operate in exactly the same way as the standard I/O streams, `cin` and `cout`.

For basic file I/O: `#include <fstream>`

There are no pre-defined file stream variables, so a programmer who needs to use file streams must declare file stream variables:

```
ifstream inFile;    // input file stream object
ofstream outFile;  // output file stream object
```

The types `ifstream` and `ofstream` are C++ stream classes designed to be connected to input or output files.

File stream objects have all the member functions and manipulators possessed by the standard streams, `cin` and `cout`.

By default, a file stream is not connected to anything. In order to use a file stream the programmer must establish a connection between it and some file. This can be done in two ways.

You may use the `open()` member function associated with each stream object:

```
infile.open("readme.data");  
outfile.open("writeme.data");
```

This sets up the file streams to read data from a file called "readme.data" and write output to a file called "writeme.data".

For an input stream, if the specified file does not exist, it will not be created by the operating system, and the input stream variable will contain an error flag. This can be checked using the member function `fail()` discussed on a later slide.

For an output stream, if the specified file does not exist, it will be created by the operating system.

You may also connect a file stream variable to a file when the stream variable is declared:

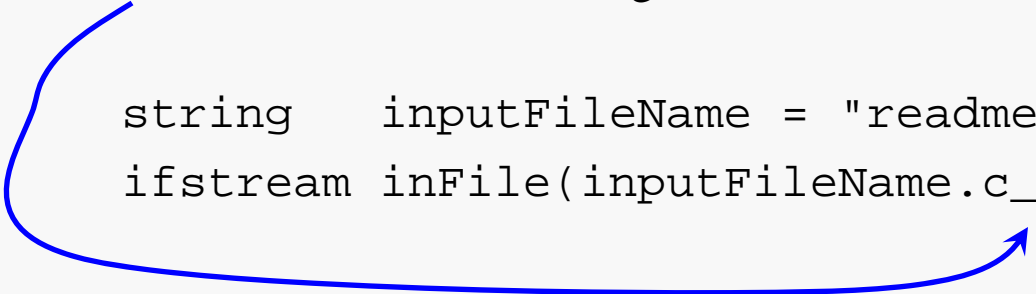
```
ifstream inFile("readme.data");  
ofstream outFile("writeme.data");
```

This also sets up the file streams to read data from a file called "readme.data" and write output to a file called "writeme.data".

The only difference between this approach and using the `open()` function is compactness.

Warning: if you use a `string` constant (or variable) to store the file name, you must add a special conversion when connecting the stream:

```
string  inputFileName = "readme.data";  
ifstream inFile(inputFileName.c_str());
```



When a program is finished with a file, it must close the file using the `close()` member function associated with each file stream variable:

```
inStream.close( );  
outStream.close( );
```

(Including the file name is an error.)

Calling `close()` notifies the operating system that your program is done with the file and that the system should flush any related buffers, update file security information, etc.

It is always best to close files explicitly, (even though by the C++ standard, files are closed automatically whenever the associated file stream variable goes out of scope [see the chapter on functions for a presentation of scope]).

First of all you need to include the manipulator header file: `<iomanip>`

`setw()`:

sets the field width (number of spaces in which the value is displayed).

`setw()` takes one parameter, which must be an integer.

The `setw()` setting applies to the next single value output only.

`setprecision()`:

sets the precision, the number of digits shown after the decimal point.

`setprecision()` also takes one parameter, which must be an integer.

The `setprecision()` setting applies to all subsequent floating point values, until another `setprecision()` is applied.

In addition, to activate the manipulator `setprecision()` for your output stream, insert the following two manipulators once:

```
outStream << fixed << showpoint;
```

(Just use the name of your output stream variable.)

Omitting these manipulators will cause `setprecision()` to fail, and will cause real values whose decimal part is zero to be printed without trailing zeroes regardless of `setprecision()`.

Other useful manipulators:

`bin`

`hex`

`octal`

`dec`

`scientific`