Output

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

One way of accomplishing this in C++ is with the use of streams. We already saw an example of the use of a stream to do output.

In order to use streams, we must inform the compiler that we desire to use streams. This is done by having in our program the compiler directive:

```
#include <iostream.h>
```

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

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

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

Output Examples

```
cout << "The answer is " << total << endl;
    The answer is 123.45

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

    ____________________

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

    ____________________

float X  =  1234.56;
cout <<  X  << endl;

    1234.56
```

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

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

Input

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

One way of accomplishing this in C++ is with the use of streams.

In order to use streams, we must inform the compiler that we desire to use streams.

This is done by having in our program the compiler directive

```
#include <iostream.h>
```

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

```
cin >> X;
```

where X is the name of some variable or constant that we want to input from the keyboard.

Input Examples

Syntax:

```
cin >> var1 >> var2 >> var3 ...  ;
```

A cin statement ignores extra information in the input stream but does not discard it.

Example: assume the input stream contains the following

```
12  17  19  4
```

```
cin >> X >> Y >> Z;
```

```
cin >> X >> Y >> Z;
cin >> W;
```

In both examples, assuming they're done independently, the result is

X = 12, Y = 17, and Z = 19. Also, in the second, W = 4.

Extraction Operator (Input)

Think of an input stream as an endless sequence of characters into your program from an input device.

```
#include <iostream.h>  //contains definition of two data types:

istream cin;     // associated with the standard input device (keyboard)
ostream cout;    // associated with the standard output device (monitor)
```

The extraction operator (>>) takes two operands:

- the left-hand operand is a stream expression (like `cin`)
- the right-hand operand is a variable of simple type (`char, int, or float`)

Note: The extraction operator skips whitespace characters (blanks, newline characters, etc.)

code	whitespace character
\n	new line
\t	tab
\r	carriage retur
\f	form feed
\v	vertical tab
	blank

C++ Streams are Classes

The input and output streams, cin and cout are actually C++ classes. 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 (class) 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 calling 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.

Whitespace Input

If `ch1` and `ch2` are char variables and your program executes:

```
cin >> ch1 >> ch2;
```

and the input stream contains:

```
A M
```

(there's one space between `A` and `M` here).

Then `ch1` gets the character `A` and `ch2` gets the character `M`, since the extract operator skips whitespace

But, suppose we wanted to read all three characters: `A`, blank, and the `M`?

So how can a program read the blank?

get( ) Member Function

The input stream object `cin` has a member function named `get( )` which returns the next character in the stream, whether it is whitespace or not.

To call a member function of an object, state the name of the object, followed by a period, followed by the function call:

```
cin.get(someChar);    // where someChar is a char variable
```

This call to the `get( )` function will remove the next character from the stream `cin` and place it in the variable `someChar`.

So to read all three characters (from the previous slide), we could have:

```
cin >> ch1;          // read 'A'
cin.get(someChar);   // read the space
cin >> ch2;          // read 'M'
```

See the discussion about line input in the strings section of the array notes.

We could also have used the `get()` function to read all three characters.

ignore( ) Member Function

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.

Insertion Operator (Output)

The insertion operator (<<) takes two operands:

- the left-hand operand is an output stream object (like `cout`)

- the right-hand operand is an expression of simple type or a string

E.g.
```
cout << "Hello world!!" << endl;
```

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

Prompts:

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

```
cout << " Enter your Age: " ;
cin >> age ;
```

The statements above allow the user to enter her/his age on the same line as the prompt (input instructions).

Interactive I/O

More on prompts:

```
cout << " Enter your Age: " << endl;
cin  >> UserAge;
```

Because of buffering of the I/O, 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 << " Enter your Age: " << flush;
cin  >> UserAge;
```

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

Streams for File I/O

What includes are necessary? For basic file stream use:

```
#include <fstream.h>
```

What variables must be declared? Generally you'll need file stream variables, for input and for output, declared something like this:

```
ifstream inStream;     // input file stream object
ofstream outStream;    // output file stream object
```

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

Manipulators and member functions for file stream objects are essentially the same as those for the standard streams, `cin` and `cout`.

open( ) Member Function

How do I tell my program what files to use? By using the `open( )` member function associated with each stream object:

```
inStream.open("readme.dat");
outStream.open("writeme.dat");
```

This sets up the file streams to read data from a file called "readme.dat" and write output to a file called "writeme.dat". If the specified file does not exist, it will be created by the operating system.

You can prevent the creation of the specified file by adding a special parameter to the call to open:

```
inStream.open("readme.dat", ios::nocreate);
```

One, almost always, does not want an empty input file to be created if the input file cannot be found.

Now, calling `inStream.fail( )` (see fail function slide that follows) will indicate whether the file exists.

File Names and Paths

You may use any file name you like, as long as it conforms to the operating system you're using.

The example on the previous slide assumes the files are in the same directory as your program when you run it. If not, you need to specify the path when you open:

```
inStream.open("c:\\vis50\\1044\\readme.dat");
```

Note the doubled backslashes - that is necessary instead of the usual Windows\DOS notation; in C++ the backslash (\) is used to indicate an escape sequence, such as the newline character (\n).

So, to indicate a backslash character we need an escape sequence: \\

close( ) Member Function

Anything special to do when I'm finished with a file?

Yes. You must close the file using the `close( )` member function associated with each file stream variable:

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

(That's right, empty parentheses; 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 section seven for a presentation of scope]).

Formatting Output

First of all you need to include the manipulator header file:

```
#include <iomanip.h>
```

Then to activate the manipulators, (setw and setprecision), include two lines similar to:

```
outStream.setf(ios::fixed, ios::floatfield);
outStream.setf(ios::showpoint);
```

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

Omitting the `fixed`, `floatfield` and `showpoint` flags will cause `setprecision( )` to fail, and will cause integer values to be printed without trailing zeroes regardless of `setprecision( )`.

Numeric Output Manipulators

`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.

Justification and Padding Manipulators

Left Justification

- The default justification in output fields is to the right, with padding occurring first (on the left).
- To reverse the default justification to the left:

```
outStream.setf(ios::left);   //turn on left justification
// insert left justified output statements here
outStream.unsetf(ios::left); //turn off left justification
```

Padding Output

By default the pad character for justified output is the space (blank) character.

This can be changed by calling the fill function:

```
outStream.fill('0');   //pad with zeroes
outStream << setw(10) << setprecision(2) << salary;
outStream.fill(' ');   //reset padding
```

Manipulators Example


```
#include <fstream.h>
#include <iomanip.h>

void main( ) {
   ofstream oFile;
   oFile.open("stuff.out");

   float aFloat = 7.2564F;
   oFile.setf(ios::fixed, ios::floatfield);
   oFile.setf(ios::showpoint);

   oFile << "x =" << setw(7) << setprecision(2) << aFloat;

   oFile.close( );
}
```

produces the output:

```
x =   7.26
   ^^^^^^^
```

Reading to Input Failure

The end of a file is marked by a special character, called the end-of-file or EOF marker.

When you attempt to extract a value from the input stream, the stream variable returns an indication of success (true) or failure (false). You can use that to check for when you've reached the end of the file from which you're reading data, or if the input operation has failed for some other reason.

A while loop is used to extract data from the input stream until it fails.

Note well: a preliminary or priming read is used before the while loop in order to avoid an attempt to process (unread) data when the input stream is initially empty.

```
Now is the¶
time for¶
all good men¶
to come to the¶
aid of their party!¶§
```

¶	represents the return char
§	represents the end of file char

Failure-Controlled Input Example


```
#include <fstream.h>
void main( ) {
   int anInt;
   ifstream inStream;
   ofstream outStream;
   inStream.open("infile.dat");
   outStream.open("outfile.dat");

   inStream >> anInt;                  // priming read before loop

   while (inStream) {                  // check for read failure
      outStream << anInt << endl;      // print value
      inStream >> anInt;               // read next value at end of
   }                                   //    the loop body
   inStream.close( );
   outStream.close( );
}
```

It is important to understand the logic of this program. Reading to input failure is often necessary and alternative logical designs are likely to be incorrect.

Failure-Controlled Input Example

The program given on the previous slide will produce the output file shown below from the input file shown below:

infile.dat

```
171      32       41       17§
```

outfile.dat

```
171¶
32¶
41¶
17¶§
```

. . . and it will produce the output file shown below from the input file shown below:

infile.dat

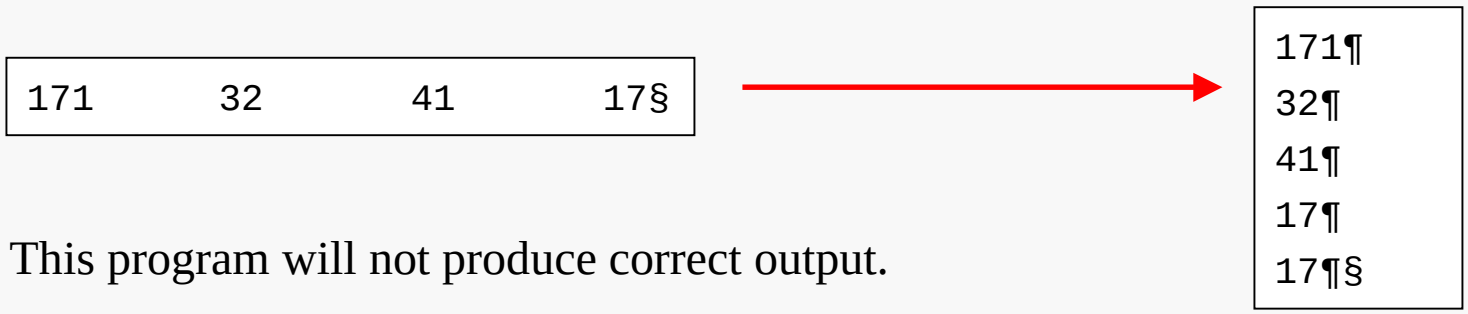
```
171      32       Fred     17§
```

outfile.dat

```
171¶
32¶§
```

Incorrect Failure-Controlled Input

```
#include <fstream.h>
void main( ) {
   int anInt;
   ifstream inStream;
   ofstream outStream;
   inStream.open("infile.dat");
   outStream.open("outfile.dat");
   // no priming read before loop
   while (inStream) {                  // check for read failure
      inStream >> anInt;               // read next value at start
                                       //    of the loop body
      outStream << anInt << endl;      // print value
   }
   inStream.close( );
   outStream.close( );
}
```



This program will not produce correct output.

eof( ) Member Function

`eof( )` is a boolean function that returns true if the last input operation attempted to read the end-of-file mark, and returns false otherwise.

The loop test in the program on slide 4.21 could be modified as follows to use `eof( )`:

```
inStream >> anInt;

while (!inStream.eof()) {              // ! means "not"
       outStream << anInt;             // print value
       inStream >> anInt;              // read next value
}
```

This while loop will terminate when `eof( )` returns false.

In general, reading until input failure is safer than the technique illustrated here. The code shown above will not terminate gracefully if an input failure occurs in the middle of the input file.

peek( ) Member Function

`peek( )` provides a way to examine the next character in the input stream, without removing it from the stream.

For example, the following code skips whitespace characters in the input stream:

```
char ch;
ch = inFile.peek();    // peek at first character

// while the first character is a space, tab or newline
while (ch == ' '  || ch == '\t' || ch == '\n') {  // || means "or"

       inFile.get(ch);        // remove it from the stream

       ch = inFile.peek();    // peek at the (new) first char
}
```

putback( ) Member Function

`putback( )` provides a way to return the last character read to the input stream.

For example, the following code also skips whitespace characters in the input stream:

```
char ch;
infile.get(ch);         // remove first character from stream

// while you just got a space, tab or newline
while (ch == ' ' || ch == '\t' || ch == '\n') {

  infile.get(ch);       // remove next character from stream
}

infile.putback(ch);     // last character read was
                        // not whitespace, so put it back
```

fail( ) Member Function

`fail( )` provides a way to check the status of the last operation on the input stream.

`fail( )` returns true if the last operation failed and returns false if the operation was successful.

```
#include <fstream.h>
void main( ) {
   ifstream inStream;
   inStream.open("infile.dat", ios::nocreate);

   if  (inStream.fail() )
       cout << "File Not Found" ;
   else
       inStream.close();
}
```

Operating System I/O Redirection

- UNIX - the gcc compiler allows redirection of standard C++ input and output at the command line (UNIX operating system) level.
- DOS - the MS C/C++ compiler allows redirection of standard C++ input and output at the command line (DOS operating system) level.
- Assumption: The C++ program prog1 has been compiled to disk and a machine code version of prog1 exists on disk. Note: by default the MS C compiler stores the DOS executable in the directory

  `C:\MSDEVSTD\Projects\`*`projectname`*`\Debug`

- If a text file named infile exists, then to make the program read standard input from the file infile, type the following command at the DOS or UNIX prompt:

```
prog1  <   infile
```

- Note that the output of prog1 still goes to the standard output device (screen).

Operating System I/O Redirection (cont)

To write standard output to a file outfile (for later printing):

```
prog1  >  outfile
```

Note that here input is read from the standard input device (keyboard).

The file `outfile` can now be printed at the operating system level.

To redirect both input and output at the same time, choose any valid file names:

```
prog1  <  p1in  >  p1out
```

File Output Redirection

Problem:

How to write output to both the screen and a file?

Solution 1:

Duplicate all of your output code:

```
cout << “The average and standard deviation are ”
     << average << stddev << “ respectively ” ;

outfile << “The average and standard deviation are ”
        << average << stddev << “ respectively ” ;
```

A ***gross*** but simple and effective method.

File Output Redirection (cont)

Solution 2:

Localize output in functions and call twice directing the output to different locations:

```
#include <iostream.h>
#include <fstream.h>

void redirecto(ostream&);
int main( ) {
   ofstream outfile ;

   outfile.open("outfile.txt");
   redirecto(outfile); // send output to the file
   redirecto(cout) ;   // send output to the display
   outfile.close( );
   return 0;
}


void redirecto(ostream &ofile) {
   ofile << "This is an output redirection test" << endl ;
}
```

This code shows how one can use the same output statement to produce output on both the screen and in a file.

Wait to utilize this technique until the function notes have been covered.