# User Input Using the `iostream` Library

- The *extraction* operator (`>>`) is a built-in operator
  - It retrieves characters from an *input stream* and stores their value in a variable
  - Like insertion, this requires using the `iostream` library
- The `iostream` library defines the type `istream` (input stream)
  - Input streams move characters from an output device (the keyboard, a file, etc.) to the program
- The `iostream` library also declares the variable `cin`
  - `cin` is of type `istream` (i.e. `istream cin;`)
  - `cin` reads characters typed into the black box on the screen

# Stream Input

- A stream handles characters in *sequential order*
  - E.g. Characters output to the screen in order
- A program gets characters from an input stream
  - In the order they are typed by the user
  - The program can only get one character at a time
    - It can get remove it from the stream or not
- The cin iostream *only* sends characters when the user presses the return key
- Working at the level of individual characters is tedious and error-prone
  - The extraction operator (<<) provides a higher level of abstraction for you to work with

# Standard Libraries

- Using built-in types and operators we can do a number of basic, important operations
  - Declare variables
  - Store and retrieve data
  - Arithmetic
- To do more, we use *libraries*
  - A library is a set of functions, operators and types written and packaged for other programs to use
  - Libraries allow programs to do important things like print out to the screen, accept user input and advanced math
  - Other libraries allow your program to send email, talk over a network or show images and play sounds

# Printing Using the `iostream` Library

- The insertion operator (`<<`) is a built-in operator
  - However, it needs to know where to print to
  - Those printable "locations" are not built-in
- The `iostream` library defines the type `ostream` (output stream)
  - Output streams move characters from the program to an output device (the screen, a file, etc.)
  - Characters are sent one-by-one in fixed order
- The `iostream` library also declares the variable `cout`
  - `cout` is of type `ostream` (i.e. `ostream cout;`)
  - `cout` puts characters on the screen

# The `#include` Directive

- In order to use a library, you must tell the program to *include* its *header file*
- The syntax for including `iostream` is:
  - `#include <iostream>`
  - Notice that it does not have a semi-colon at the end
    - This is because it's not a C++ statement
    - It is a *preprocessor directive* that makes the preprocessor insert the contents of the specified header file at that point
  - The angle braces `<>` indicate this is a standard library
- Once the header file is included, you can use the types, variables, functions and operators in the library

# using namespace

- `cout` is declared in the standard (`std`) namespace
  - Its full name is `std::cout`
  - This helps prevent programs and libraries trying to use the same names for things
- `using namespace std;`
  - This statement tells the program to always look for things in the std namespace
  - Keeps you from having to specific `std::` all the time

# Newlines

- The `iostream` library defines the constant `endl`
- This constant is equivalent to a newline or return
- Thus, you tell the computer to print a newline with:

```
cout << endl;
```

- You can also use character escape notation:

```
cout << '\n';
```

# Additional Input Functions

- The extraction operator (`>>`) is one way to read characters from an input stream like `cin`
  - It provides a powerful way to get individual pieces of data (integers, reals, chars, strings) separated by spaces
  - However, user input doesn't always look like that
- The `iostream` library defines other ways to do input
  - The function `getline`()
  - The function `cin.ignore`()

# Functions and Operators

- Operator syntax

```
operand1 operator operand2
```

  - Operator is a special symbol
  - All operators are binary (two operands)
  - An operator performs some task and *evaluates* to a value
    - Sometimes you care about the value (e.g. addition)

```
5 + 2
```

    - Sometimes you care about the side-effect (e.g. printing)

```
cout << "Hello"
```

- Function syntax

```
function( parameter2, parameter2 … )
```

  - Function name is an identifier
  - Any number of *parameters* allowed (including none)
  - Also performs some task (set of instructions) and evaluates to a value

```
add( 5, 2 )
insertion( cout, "Hello" )
```

# Some Example Functions

- These functions are in the `cmath` library
  - `#include <cmath>` to use them

- To compute the square root of a number:
  - 1 input (float), 1 output (float)

```
answer = sqrt( 16.0 );
```

- To compute the power function ($x^y$)
  - 2 inputs (float, int), 1 output (float)

```
cout << pow( 2.0, 3 );
```

# Back to Input

- Because the extraction operator reads data separated by whitespace, it cannot read a string with whitespace in it
  - Given the code:

```
string s;
cin >> s;
```

  - If the user types “University of Texas”

# Back to Input

- Because the extraction operator reads data separated by whitespace, it cannot read a string with whitespace in it
  - Given the code:

```
string s;
cin >> s;
```

  - If the user types “University of Texas”
  - `s` will contain the string “University”

# `getline` Function

- To read strings with spaces in them, we use the function `getline()`
  - Takes two arguments (just like extraction):
    - The input stream to read from
    - The (string) variable to store in

    ```
    getline( istreamVar, strVar );
    ```
  - Reads all characters until the end of the line
    - Stores the resulting string in the string variable
  - Evaluates to the stream that was read from
    - To support chaining, but the task (reading) is the main point

# `getline` Function

- `getline()` can also take three arguments
  - The input stream to read from
  - The (string) variable to store in
  - A *delimiting* character

```
getline( istreamVar, strVar, delim );
```

- This version reads until it reaches the specified delimiting character
  - If the delimiter is '\n', it reads to the end of the line

```
getline( istreamVar, strVar, '\n' );
```

# `cin.ignore` Function

- The function `cin.ignore`
  - The "." is class syntax, which we'll discuss later in the course
    - Just memorize for now
  - Takes two arguments:
    - The number of characters to ignore
    - A delimiting character
  - Reads and discards the specified number of characters
    - Unless it reaches the delimiter first
  - Evaluates to the stream that was read from
    - To support chaining, but the task (reading) is the main point

# Input Failure

- Things can go wrong during execution
- If input data does not match corresponding variables, program may run into problems
- Trying to read a letter into an `int` or `double` variable will result in an input failure
- If an error occurs when reading data
  - Input stream enters the fail state

# The `clear` Function

- Once in a fail state, all further I/O statements using that stream are ignored
- The program continues to execute with whatever values are stored in variables
  - This causes incorrect results
- The `clear` function restores input stream to a working state

```
cin.clear();
```

  - However, it does not remove the characters that caused the error from the input stream

# Exercise

```
int x, y;
string line;
```

For the input:

```
13 28 D
14 E 98
A B 56
```

What are the values of `x`, `y` and `line` after:

```
f.  getline( cin, line, '8' );
    cin.ignore( 50, '\n' );
    cin >> x;
    cin.ignore( 50, 'E' );
    cin >> y;
```