Basic Input/Output

The example programs of the previous sections provided little interaction with the user, if any at all. They simply printed simple values on screen, but the standard library provides many additional ways to interact with the user via its input/output features. This section will present a short introduction to some of the most useful.

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:

stream	**description**
`cin`	standard input stream
`cout`	standard output stream
`cerr`	standard error (output) stream
`clog`	standard logging (output) stream

We are going to see in more detail only `cout` and `cin` (the standard output and input streams); `cerr` and `clog` are also output streams, so they essentially work like `cout`, with the only difference being that they identify streams for specific purposes: error messages and logging; which, in many cases, in most environment setups, they actually do the exact same thing: they print on screen, although they can also be individually redirected.

Standard output (cout)

On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is `cout`.

For formatted output operations, `cout` is used together with the *insertion operator*, which is written as `<<` (i.e., two "less than" signs).

```
1 cout << "Output sentence"; // prints Output sentence on screen
2 cout << 120;               // prints number 120 on screen
3 cout << x;                 // prints the value of x on screen
```

The << operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string `Output sentence`, the number `120`, and the value of variable `x` into the standard output stream `cout`. Notice that the sentence in the first statement is enclosed in double quotes (") because it is a string literal, while in the last one, `x` is not. The double quoting is what makes the difference; when the text is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the identifier of a variable, and its value is printed instead. For example, these two sentences have very different results:

```
1 cout << "Hello";  // prints Hello
2 cout << Hello;    // prints the content of variable Hello
```

Multiple insertion operations (<<) may be chained in a single statement:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text `This is a single C++ statement`. Chaining insertions is especially useful to mix literals and variables in a single statement:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Assuming the *age* variable contains the value 24 and the *zipcode* variable contains 90064, the output of the previous statement would be:

```
I am 24 years old and my zipcode is 90064
```

What cout does not do automatically is add line breaks at the end, unless instructed to do so. For example, take the following two statements inserting into `cout`:
cout << "This is a sentence.";
cout << "This is another sentence.";

The output would be in a single line, without any line breaks in between. Something like:

```
This is a sentence.This is another sentence.
```

To insert a line break, a new-line character shall be inserted at the exact position the line should be broken. In C++, a new-line character can be specified as `\n` (i.e., a backslash character followed by a lowercase `n`). For example:

```
1 cout << "First sentence.\n";
2 cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

```
First sentence.
Second sentence.
Third sentence.
```

Alternatively, the `endl` manipulator can also be used to break lines. For example:

```
1 cout << "First sentence." << endl;
2 cout << "Second sentence." << endl;
```

This would print:

```
First sentence.
Second sentence.
```

The `endl` manipulator produces a newline character, exactly as the insertion of `'\n'` does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly *fully buffered* streams, and `cout` is (generally) not a *fully buffered* stream. Still, it is generally a good idea to use `endl` only when flushing the stream would be a feature and `'\n'` when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.

Standard input (cin)

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is `cin`.

For formatted input operations, `cin` is used together with the extraction operator, which is written as `>>` (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

```
1 int age;
2 cin >> age;
```

The first statement declares a variable of type `int` called `age`, and the second extracts from `cin` a value to be stored in it. This operation makes the program wait for input from `cin`; generally, this means that the program will wait for the user to enter some sequence with the keyboard. In this case, note that the characters introduced using the keyboard are only transmitted to the program when the `ENTER` (or `RETURN`) key is pressed. Once the statement with the extraction operation on `cin` is reached, the program will wait for as long as needed until some input is introduced.

The extraction operation on `cin` uses the type of the variable after the `>>` operator to determine

how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

```
   // i/o example

1
2  #include <iostream>
3  using namespace std;
4
5  int main ()
6  {
7    int i;                                Please enter an integer value: 702
8    cout << "Please enter an integer     The value you entered is 702 and its
9  value: ";                               double is 1404.
10   cin >> i;
11   cout << "The value you entered is
12 " << i;
13   cout << " and its double is " <<
14 i*2 << ".\n";
     return 0;
   }
```

As you can see, extracting from `cin` seems to make the task of getting input from the standard input pretty simple and straightforward. But this method also has a big drawback. What happens in the example above if the user enters something else that cannot be interpreted as an integer? Well, in this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable `i`, producing undetermined results if the value of `i` is used later.

This is very poor program behavior. Most programs are expected to behave in an expected manner no matter what the user types, handling invalid values appropriately. Only very simple programs should rely on values extracted directly from `cin` without further checking. A little later we will see how *stringstreams* can be used to have better control over user input. Extractions on `cin` can also be chained to request more than one datum in a single statement:

```
cin >> a >> b;
```

This is equivalent to:

```
1 cin >> a;
2 cin >> b;
```

In both cases, the user is expected to introduce two values, one for variable `a`, and another for variable `b`. Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

cin and strings

The extraction operator can be used on `cin` to get strings of characters in the same way as with fundamental data types:

```
1 string mystring;
2 cin >> mystring;
```

However, `cin` extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

To get an entire line from `cin`, there exists a function, called `getline`, that takes the stream (`cin`) as first argument, and the string variable as second. For example:

```
// cin with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << " too!\n";
  return 0;
}
```

```
What's your name? Homer Simpson
Hello Homer Simpson.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

Edit & Run

Notice how in both calls to `getline`, we used the same string identifier (`mystr`). What the program does in the second call is simply replace the previous content with the new one that is introduced.

The standard behavior that most users expect from a console program is that each time the program queries the user for input, the user introduces the field, and then presses `ENTER` (or `RETURN`). That is to say, input is generally expected to happen in terms of lines on console programs, and this can be achieved by using `getline` to obtain input from the user. Therefore, unless you have a strong reason not to, you should always use `getline` to get input in your console programs instead of extracting from `cin`.

stringstream

The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on cin and cout. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
1 string mystr ("1204");
2 int myint;
3 stringstream(mystr) >> myint;
```

This declares a string with initialized to a value of "1204", and a variable of type int. Then, the third line uses this variable to extract from a stringstream constructed from the string. This piece of code stores the numerical value 1204 in the variable called myint.

```
1  // stringstreams
2  #include <iostream>
3  #include <string>
4  #include <sstream>
5  using namespace std;
6
7  int main ()
8  {
9    string mystr;
10   float price=0;
11   int quantity=0;
12
13   cout << "Enter price: ";
14   getline (cin,mystr);
15   stringstream(mystr) >> price;
16   cout << "Enter quantity: ";
17   getline (cin,mystr);
18   stringstream(mystr) >> quantity;
19   cout << "Total price: " << price*quantity <<
20 endl;
21   return 0;
   }
```

```
Enter price:
22.25
Enter quantity: 7
Total price:
155.75
```

Edit & Run

In this example, we acquire numeric values from the *standard input* indirectly: Instead of extracting numeric values directly from cin, we get lines from it into a string object (mystr), and then we extract the values from this string into the variables price and quantity. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user

expects, and at the same time gaining more control over the transformation of its content into useful data by the program.