6. Beginning with C++ program

6.1 Input/output using extraction (>>) and insertion (<<) operators and Writing simple C++ programs

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

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).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;               // prints number 120 on screen
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:

```
cout << "Hello";  // prints Hello
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:

```
cout << "First sentence.\n";
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:

```
cout << "First sentence." << endl;
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:

```
int age;
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.

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

#include <iostream.h>
#include <conio.h>

void main ()
{
  int i;
  cout << "Please enter an integer
value: ";
  cin >> i;
  cout << "The value you entered is
" << i;
  cout << " and its double is " <<
i*2 << ".\n";
  getch();
}
```

```
Please enter an integer value: 702
The value you entered is 702 and its
double is 1404.
```

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.

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.

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

This is equivalent to:

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

6.2 Comments in C++

Program comments are explanatory statements that you can include in the C++ code that you write and helps anyone reading it's source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /* and end with */. For example:

```
/* This is a comment */

/* C++ comments can  also
 * span multiple lines
 */
```

A comment can also start with //, extending to the end of the line. For example:

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

void main()
{
   cout << "Hello World"; // prints Hello World

   getch();
}
```

When the above code is compiled, it will ignore **// prints Hello World** and final executable will produce the following result:

```
Hello World
```

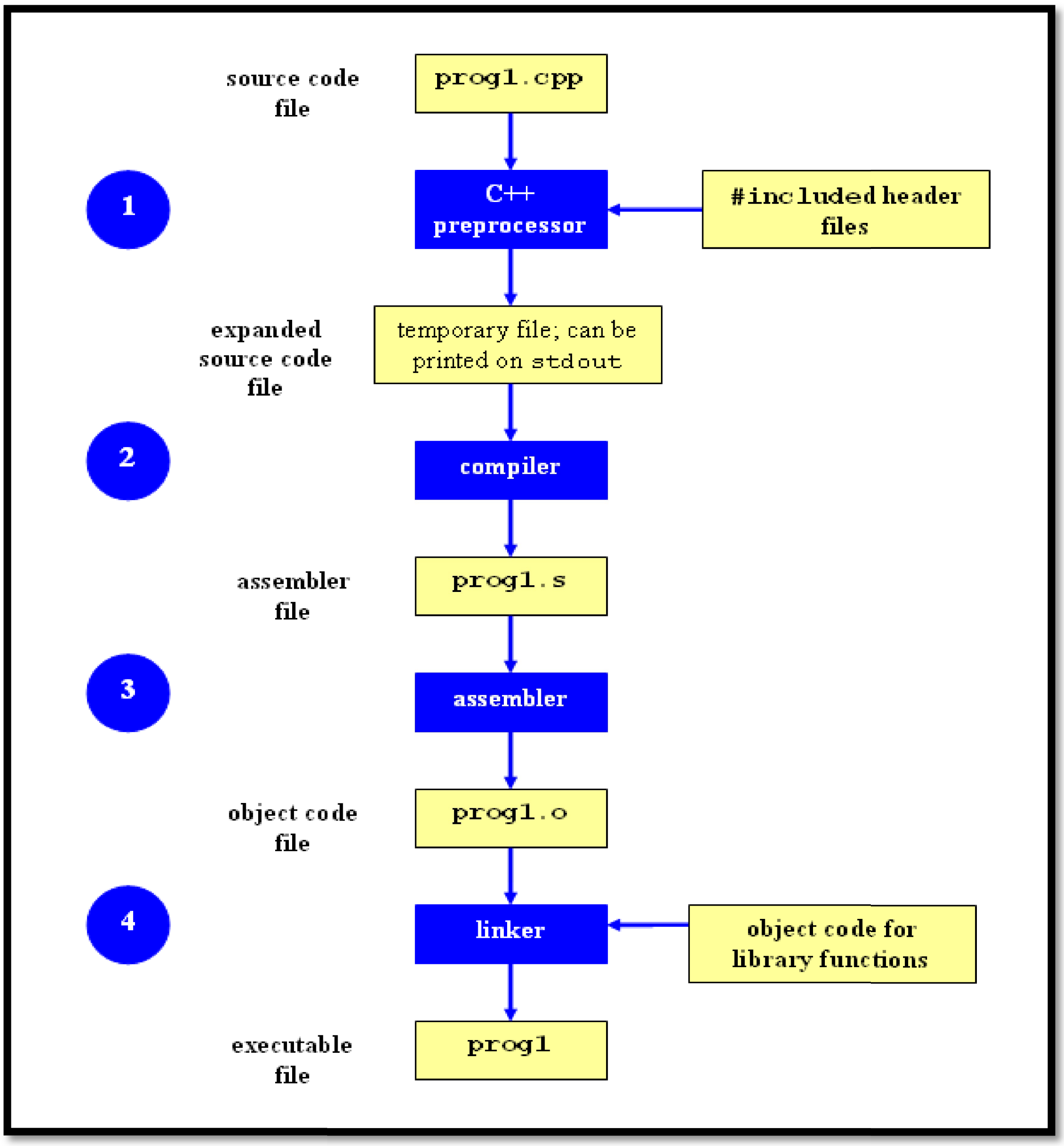
Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example:

```
/* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

*/
```

6.3 Stages of program execution



Compiling a source code file in C++ is a four-step process. For example, if you have a C++ source code file named prog1.cpp the compilation process looks like this:

1. The C++ preprocessor copies the contents of the included header files into the source code file, generates macro code, and replaces symbolic constants defined using #define with their values.
2. The expanded source code file produced by the C++ preprocessor is compiled into the assembly language for the platform.
3. The assembler code generated by the compiler is assembled into the object code for the platform.
4. The object code file generated by the assembler is linked together with the object code files for any library functions used to produce an executable file.