3.3 Math functions

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like $\sin(\pi/2)$ and $\log(1/x)$. First, you evaluate the expression in parentheses, which is called the **argument** of the function. For example, $\pi/2$ is approximately 1.571, and $1/x$ is 0.1 (if x happens to be 10).

Then you can evaluate the function itself, either by looking it up in a table or by performing various computations. The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like $\log(1/\sin(\pi/2))$. First we evaluate the argument of the innermost function, then evaluate the function, and so on.

C++ provides a set of built-in functions that includes most of the mathematical operations you can think of. The math functions are invoked using a syntax that is similar to mathematical notation:

```
double log = log (17.0);
double angle = 1.5;
double height = sin (angle);
```

The first example sets `log` to the logarithm of 17, base e. There is also a function called `log10` that takes logarithms base 10.

The second example finds the sine of the value of the variable `angle`. C++ assumes that the values you use with `sin` and the other trigonometric functions (`cos`, `tan`) are in *radians*. To convert from degrees to radians, you can divide by 360 and multiply by 2π.

If you don't happen to know π to 15 digits, you can calculate it using the `acos` function. The arccosine (or inverse cosine) of -1 is π, because the cosine of π is -1.

```
double pi = acos(-1.0);
double degrees = 90;
double angle = degrees * 2 * pi / 360.0;
```

Before you can use any of the math functions, you have to include the math **header file**. Header files contain information the compiler needs about functions that are defined outside your program. For example, in the "Hello, world!" program we included a header file named `iostream` using an **include** statement:

```
#include <iostream>
using namespace std;
```

`iostream` contains information about input and output (I/O) streams, including the object named `cout`. C++ has a powerful feature called namespaces, that allow you to write your own implementation of `cout`. But in most cases, we would need to use the standard implementation. To convey this to the compiler, we use the line

```
using namespace std;
```

As a rule of the thumb, you should write `using namespace std;` whenever you use `iostream`.

Similarly, the math header file contains information about the math functions. You can include it at the beginning of your program along with `iostream`:

```
#include <cmath>
```

Such header files have an initial 'c' to signify that these header files have been derived from the **C** language.

3.4 Composition

Just as with mathematical functions, C++ functions can be **composed**, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

```
double x = cos (angle + pi/2);
```

This statement takes the value of `pi`, divides it by two and adds the result to the value of `angle`. The sum is then passed as an argument to the `cos` function.

You can also take the result of one function and pass it as an argument to another:

```
double x = exp (log (10.0));
```

This statement finds the log base e of 10 and then raises e to that power. The result gets assigned to `x`; I hope you know what it is.

3.5 Adding new functions

So far we have only been using the functions that are built into C++, but it is also possible to add new functions. Actually, we have already seen one function definition: `main`. The function named `main` is special because it indicates where the execution of the program begins, but the syntax for `main` is the same as for any other function definition:

```
void NAME ( LIST OF PARAMETERS ) {
  STATEMENTS
}
```

You can make up any name you want for your function, except that you can't call it `main` or any other C++ keyword. The list of parameters specifies what information, if any, you have to provide in order to use (or **call**) the new function.

`main` doesn't take any parameters, as indicated by the empty parentheses `()` in it's definition. The first couple of functions we are going to write also have no parameters, so the syntax looks like this:

```
  void newLine () {
    cout << endl;
  }
```

This function is named `newLine`; it contains only a single statement, which outputs a newline character, represented by the special value `endl`.

In `main` we can call this new function using syntax that is similar to the way we call the built-in C++ commands:

```
int main ()
{
  cout << "First Line." << endl;
  newLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

The output of this program is

```
First line.

Second line.
```

Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

```
int main ()
{
  cout << "First Line." << endl;
  newLine ();
  newLine ();
  newLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

Or we could write a new function, named `threeLine`, that prints three new lines:

```
void threeLine ()
{
  newLine ();  newLine ();  newLine ();
}

int main ()
{
  cout << "First Line." << endl;
```

```
  threeLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

You should notice a few things about this program:

- You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
- You can have one function call another function. In this case, `main` calls `threeLine` and `threeLine` calls `newLine`. Again, this is common and useful.
- In `threeLine` I wrote three statements all on the same line, which is syntactically legal (remember that spaces and new lines usually don't change the meaning of a program). On the other hand, it is usually a better idea to put each statement on a line by itself, to make your program easy to read. I sometimes break that rule in this book to save space.

So far, it may not be clear why it is worth the trouble to create all these new functions. Actually, there are a lot of reasons, but this example only demonstrates two:

1. Creating a new function gives you an opportunity to give a name to a group of statements. Functions can simplify a program by hiding a complex computation behind a single command, and by using English words in place of arcane code. Which is clearer, `newLine` or `cout << endl`?
2. Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call `threeLine` three times. How would you print 27 new lines?

3.6 Definitions and uses

Pulling together all the code fragments from the previous section, the whole program looks like this:

```
#include <iostream>
using namespace std;

void newLine ()
{
  cout << endl;
}

void threeLine ()
```

```
{
  newLine ();  newLine ();  newLine ();
}

int main ()
{
  cout << "First Line." << endl;
  threeLine ();
  cout << "Second Line." << endl;
  return 0;
}
```

This program contains three function definitions: `newLine`, `threeLine`, and `main`.

Inside the definition of `main`, there is a statement that uses or calls `threeLine`. Similarly, `threeLine` calls `newLine` three times. Notice that the definition of each function appears above the place where it is used.

This is necessary in C++; the definition of a function must appear before (above) the first use of the function. You should try compiling this program with the functions in a different order and see what error messages you get.

3.7 Programs with multiple functions

When you look at a class definition that contains several functions, it is tempting to read it from top to bottom, but that is likely to be confusing, because that is not the **order of execution** of the program.

Execution always begins at the first statement of `main`, regardless of where it is in the program (often it is at the bottom). Statements are executed one at a time, in order, until you reach a function call. Function calls are like a detour in the flow of execution. Instead of going to the next statement, you go to the first line of the called function, execute all the statements there, and then come back and pick up again where you left off.

That sounds simple enough, except that you have to remember that one function can call another. Thus, while we are in the middle of `main`, we might have to go off and execute the statements in `threeLine`. But while we are executing `threeLine`, we get interrupted three times to go off and execute `newLine`.

Fortunately, C++ is adept at keeping track of where it is, so each time `newLine` completes, the program picks up where it left off in `threeLine`, and eventually gets back to `main` so the program can terminate.

What's the moral of this sordid tale? When you read a program, don't read from top to bottom. Instead, follow the flow of execution.

3.8 Parameters and arguments

Some of the built-in functions we have used have **parameters**, which are values that you provide to let the function do its job. For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, `sin` takes a `double` value as a parameter.

Some functions take more than one parameter, like `pow`, which takes two `doubles`, the base and the exponent.

Notice that in each of these cases we have to specify not only how many parameters there are, but also what type they are. So it shouldn't surprise you that when you write a class definition, the parameter list indicates the type of each parameter. For example:

```
void printTwice (char phil) {
  cout << phil << phil << endl;
}
```

This function takes a single parameter, named `phil`, that has type `char`. Whatever that parameter is (and at this point we have no idea what it is), it gets printed twice, followed by a newline. I chose the name `phil` to suggest that the name you give a parameter is up to you, but in general you want to choose something more illustrative than `phil`.

In order to call this function, we have to provide a `char`. For example, we might have a `main` function like this:

```
int main () {
  printTwice ('a');
  return 0;
}
```

The `char` value you provide is called an **argument**, and we say that the argument is **passed** to the function. In this case the value `'a'` is passed as an argument to `printTwice` where it will get printed twice.

Alternatively, if we had a `char` variable, we could use it as an argument instead:

```
int main () {
  char argument = 'b';
  printTwice (argument);
  return 0;
}
```

Notice something very important here: the name of the variable we pass as an argument (`argument`) has nothing to do with the name of the parameter (`phil`). Let me say that again:

> **The name of the variable we pass as an argument has nothing to do with the name of the parameter.**

They can be the same or they can be different, but it is important to realize that they are not the same thing, except that they happen to have the same value (in this case the character `'b'`).

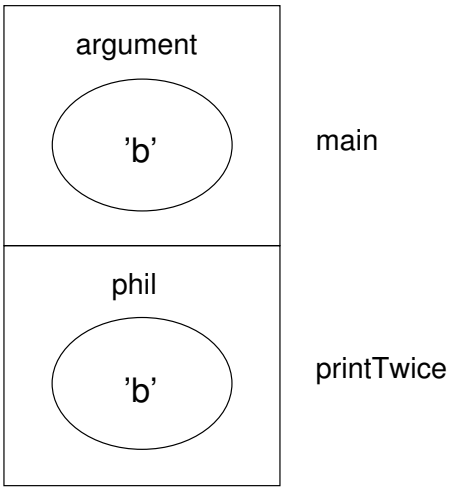
The value you provide as an argument must have the same type as the parameter of the function you call. This rule is important, but it is sometimes confusing because C++ sometimes converts arguments from one type to another automatically. For now you should learn the general rule, and we will deal with exceptions later.

3.9 Parameters and variables are local

Parameters and variables only exist inside their own functions. Within the confines of `main`, there is no such thing as `phil`. If you try to use it, the compiler will complain. Similarly, inside `printTwice` there is no such thing as `argument`.

Variables like this are said to be **local**. In order to keep track of parameters and local variables, it is useful to draw a **stack diagram**. Like state diagrams, stack diagrams show the value of each variable, but the variables are contained in larger boxes that indicate which function they belong to.

For example, the state diagram for `printTwice` looks like this:



Whenever a function is called, it creates a new **instance** of that function. Each instance of a function contains the parameters and local variables for that function. In the diagram an instance of a function is represented by a box with the name of the function on the outside and the variables and parameters inside.

In the example, `main` has one local variable, `argument`, and no parameters. `printTwice` has no local variables and one parameter, named `phil`.

3.10 Functions with multiple parameters

The syntax for declaring and invoking functions with multiple parameters is a common source of errors. First, remember that you have to declare the type of every parameter. For example

```
void printTime (int hour, int minute) {
  cout << hour;
  cout << ":";
  cout << minute;
}
```

It might be tempting to write `(int hour, minute)`, but that format is only legal for variable declarations, not for parameters.

Another common source of confusion is that you do not have to declare the types of arguments. The following is wrong!

```
int hour = 11;
int minute = 59;
printTime (int hour, int minute);   // WRONG!
```

In this case, the compiler can tell the type of `hour` and `minute` by looking at their declarations. It is unnecessary and illegal to include the type when you pass them as arguments. The correct syntax is `printTime (hour, minute)`.

3.11 Functions with results

You might have noticed by now that some of the functions we are using, like the math functions, yield results. Other functions, like `newLine`, perform an action but don't return a value. That raises some questions:

- What happens if you call a function and you don't do anything with the result (i.e. you don't assign it to a variable or use it as part of a larger expression)?
- What happens if you use a function without a result as part of an expression, like `newLine() + 7`?
- Can we write functions that yield results, or are we stuck with things like `newLine` and `printTwice`?

The answer to the third question is "yes, you can write functions that return values," and we'll do it in a couple of chapters. I will leave it up to you to answer the other two questions by trying them out. Any time you have a question about what is legal or illegal in C++, a good way to find out is to ask the compiler.