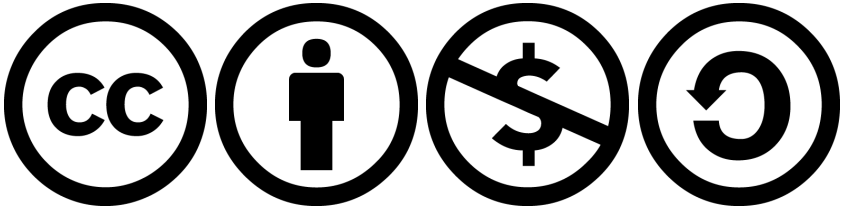
The Rook's Guide to C++

Kickstarter Backer & Contributor Version

August 1, 2013

Contents

License



Dramatis Personæ

Managing Editor:

Jeremy A. Hansen, PhD, CISSP

Technical Editing & Typesetting:

Jeremy A. Hansen

Matt Jadud, PhD

Craig D. Robbins

Theodore M. Rolle, Jr.

Media & Outreach:

Matthew E. Russo

Cover Art & Graphic Design:

Allyson E. LeFebvre

Content Authors:

Tyler Atkinson, Troy M. Dundas, Connor J. Fortune, Jeremy A. Hansen, Scott T. Heimann, Benjamin J. Jones, Michelle Kellerman, Michael E. Kirl, Zachary LeBlanc, Allyson E. LeFebvre, Gerard O. McEleney, Phung P. Pham, Megan Rioux, Alex Robinson, Kyle R. Robinson-O'Brien, Jesse A. Rodimon, Matthew E. Russo, Yosary Silvestre, Dale R. Stevens, Ryan S. Sutherland, James M. Verderico, Christian J. Vergnes, Rebecca Weaver, Richard Z. Wells, and Branden M. Wilson.

Funding & Support:

Peter Stephenson, PhD, VSM, CISSP, CISM, FICAF, LPI at the Norwich University Center for Advanced Computing & Digital Forensics

Andrew Pedley at Depot Square Pizza

Kickstarter contributors:

Nathan Adams, Chris Aldrich, Jay Anderson, Kent Archie, Erik Arvedson, Astrolox, Phoebe Ayers, Papa Joe Barr, Julia Benson-Slaughter, Georgia Perimeter College, Patrick Berthon, Francis Bolduc, Greg Borenstein, Mark Braun, Patrick Breen, Igor Bronshteyn, Valdemar Bučilko, Ross Buckley, Nikita Burtsev, Jakob Bysewski, David Camara, Dave M. Campbell, Brian V. Campbell III, S. Canero, Serge Canizares, Andrew Carlberg, Casey B. Cessnun, Winston Chow, W. Jesse Clements, Greg Crawford, Sean Cristofori, Jordan G Cundiff, Michael David, Joseph Davies, Ashley Davis, David C. Dean, DJS, Carlton Doc Dodd, Phil Dodds, Dominic, Sankar Dorai, dryack, Matt DuHarte, Brandon Duong, Van Van Duong, Daniel Egger, Chris Fabian, Jorge F. Falcon, Tek Francis, Fuchsi, Steve Gannon, Michael Gaskins, Gavlig, Adam Gibson, Russell E. Gibson, Goldenwyrm, James Green, Brian J. Green, Casey Grooms, Vitalik Gruntkovskiy, Vegar Guldal, Felix Gutbrodt, Jeremy Gwinnup, Beau T. Hahn, Paul R. Harms - Norwich 1975, Corey H. Hart, MBA, Aaron A. Haviland, Josh Heffner, Greg Holland, Henry Howard, Mark V Howe, Ivaliy Ivanov, Matt Jadud, Joseph Jaoudi, Tim R. Johnson, Ibi-Wan Kentobi, Mark King, Mitchell Kogut III, Sigmund Kopperud, Michael Korolev, Jamie Kosoy, Aria Kraft, Alexander Týr Kristjánsson, Richard Kutscher, Eric Laberge, John Lagerquist, Philip Lawrence, Mark Brent Lee, John and Nancy LeFebvre, Nevin :-) Liber, Jonathan Lindquist, Thomas Lockney, Stuart A. MacGillivray, Dr. Pedro Maciel, Troels Holstein Madsen, William Marone, Fred Mastropasqua, Miles Mawyer, michael mazzello, Ryan McDonough, Matthew McFadden, John McIntosh II, Sean McNamara, mdsaravanan, Brandon Meskimen, Andrew Mike, G.F. Miller IV, Marcus Millin, Salvador Torres Morales, Danny Morgan, Ken Moulton, Aaron Murray, mvi, Jon Nebenfuhr, Philip K. Nicholson, chris nielsen, Pontus Nilsson, Mike Noble, Aleksander R. Nordgarden-Rødner, Greg O'Hanlon, Doug Otto, Randy Padawer, Ph.D., J Palmer, Tasos Papastylianou, Paul, James Pearson-Kirk, Matthew Peterson, Grigory Petrov, pezmanlou, Joachim Pileborg, Kyle Pinches, pkh, Mary Purdey, Marshall Reeves, Matthew Ringman, Craig D. Robbins, Antonio Rodriguez, Armando Emanuel Roggio, Victor Suarez Rovere, Christian Sandberg, Jaymes Sattler, Paolo Scalia, Patrice Scheidt, Daniel Schmitt, Levi Schuck, Raman Sharma_Himachali, Michael Shashoua, Daniel Shiffman, Clay Shirky, sillygoatgirl, Kevin J. Slonka, Brian Smith, Hazel Smith & Rebecca Twigg, Andrey Soplevenko, Kasper Souren, Derek A. Spangler, Speckman, Kellan St.Louis, Nick Stefanou, Steve, Andrew Stewart, Jeremy Sturdivant, Cyrille Tabary, Adam 8T Tannir, M Taylor, Telecat Productions, Aron Temkin, Mitchell Tilbrook, Nathan Tolbert, Devin M. Tomlinson - Vermont Born, Todd Trimble, Michiel van Slobbe, James A. Velez, Marco Verdecchia, David Walter, Lothar Werzinger, Wayne West, Sean Whaley '05 & M'08, Mark Wheeler, Tommy Widenflycht, Dylan Widis, Tony Williami-

tis, Adam M. Williams, Stephen D. Williams, Dylan Wilson, Wesley Wiser, wizzy, Sam Wright, Janet Hong Yam, and Jy Yaworski.

Chapter 1

History

Chapter 2

Sample Program

Chapter 3

Variables

Chapter 4

Constants

Chapter 5

Assignments

Assignments are a way for a user or a programmer to assign a value to a variable. The way we assign a value to a variable in C++ is different from how we would might do it in math. In mathematics we are allowed to say that `x = 3` or `3 = x`, but in C++ the only acceptable way to assign the value of 3 to `x` is to type `x = 3`. The = in the expression `x = 3` is known as an *assignment operator*. This allows the program to set a variable's value depending on its type. Here are some examples on setting a value to different types of variables:

int x = 4;

char alpha = 'A';

string word = "Älphä";

float y = 3.14;

We are able to declare variables and assign a value to those variables immediately by using the assignment operator. When we assign literal values to variables of type char, the value must be surrounded by single quotes (example: `'A'`). When we assign values of type string, the literal value must be surrounded by double quotes (example: `"Älphä"`). We do not have to initialize the values of the variables, however. We can set them with a value later in the code like this: int myVal; //some code myVal = 0;

In all of the lines of code in this section where a variable is set using the assignment operator, the "thing that is being given a value" is known as an *lvalue*, and the expression on the right that is being stored in the variable is known as the *rvalue*. Literals such as `'A'` or 3 can never be an lvalue. Aside from literals, the rvalue can consist of other variables as well, like this:

myVal = myVal2;

Even though `myVal2` is a variable, we are only passing the *value stored in the*

variable, not the variable itself. For example if `myVal2` had a value of 6, `myVal` would then have a value of 6 with the above code.

We can also store an arithmetic expression in a variable like this:

myVal = 5 + 6; //assigns myVal a value of 11 because 5 plus (+) 6 is 11

But we can't write

5 + 6 = myVal; // ERROR!

since 5 + 6 doesn't refer to a place where we can store a value. We can also combine arithmetic expressions with other variables to be stored in a variable like this:

myVal2 = 6; myVal = 4 + myVal2;

In this case, the variable `myVal` would be given a value of 10 because the variable `myVal2` was initialized with a value of 6 before the assignment was executed. The value of `myVal2` remains unchanged. Make sure that the variable `myVal`, the variable `myVal2`, and the value 4 are the same type. For example, the following code will result in an error:

int myValue = 4; int yourVal; string myString = ẅord̈;

yourVal = myValue + myString; // Adding an int to a string is // probably not what you meant!

When we try to combine different variable types, the compiler will get very mad at us. Some exceptions to this rule are if we try to combine `floats`, `ints`, and `doubles`. These types have the ability to be combined because they are all numeric values. Both `doubles` and `floats` can hold values with a decimal point such as -3.14, 0.003, or 5.167289 whereas an `int` can only hold round values such as 2, -18, or 100. Refer to Chapter **??** for more information on converting between data types.

Chapter 6

Input

Chapter 7

Output

]

Chapter 8

Arithmetic

One of the most important things provided by C++ is the ability to do math. Everything a computer sees is a number. To a computer, its ability to do math and manipulate numbers is as essential to it as breathing is to us. (My apologies to anything not living that may be reading this).

The operators (+, -, *, /) in C++ are slightly different from what you may be used to from your second-grade math class. Addition is still a plus sign (+) and subtraction is still a minus sign (-). On the other hand, multiplication becomes an asterisk (*) and division becomes a forward slash (/). Think of it as *over* as in "5 over 9" is the same as the fraction $5/9$.

To do math in C++, you will either want a variable to store the answer, or output the answer to the user.

The following code directly outputs the answer to the user:

```
cout << 9 + 2;
```

This code shows how to use a variable to store the answer:

```
int sum = 9 + 2;
```

Note that when you use a variable to store an answer, the variable must come first in the equation (before the equal sign) and must be the only thing on the left side of the equation. There are some other things to note. When you use more

complicated equations, you can use parentheses to help. C++ uses a familiar order of operations (Parentheses, Exponents, Multiply, Divide, Add, and Subtract, or PEMDAS), but without the exponent operation (this topic is covered in Chapter ??). However, unlike in normal arithmetic, parentheses do not imply multiplication. `(4)(3)` does not mean the same as `4 * 3`. `(4)(3)` results in a syntax error and does not compile. The compiler returns an error message like this: `'error: '4' cannot be used as a function.'`

In C++, there are several methods of shortening and simplifying the code you're creating. The first is the increment operator (`++`), which is found in the name of the language, C++. This operator increases the value of the variable it's applied to by 1. Conversely, the decrement (`--`) operator decreases the value by 1.

Keep in mind that order does matter with the increment and decrement operators. They can be used as either prefixes or suffixes, but where you put the operator results in slightly different behavior. Starting with similarities, C++ and ++C both increase value of C by one. The difference lies in when another variable is being set to that incremented value, such as `B = C++`. B will be set to C before C is incremented. `B = ++C` will cause B to be set to C+1, in a similar way to `B = 1 + C`.

```
int A;
A = 4;
A++;
//A contains 5
```

```
int A;
A = 9;
A--;
//A contains 8
```

```
int A, B;
B = 7;
A = B++;
//A contains 7, B contains 8
```

```
int A, B;
```

Expression	Equivalent to
A *= 3;	A = A * 3;
B -= 5;	B = B - 5;
C /= 10;	C = C / 10;

```
B = 7;
A = ++B;
//A contains 8, B contains 8
```

```
int A, B;
B = 3;
A = B--;
//A contains 3, B contains 2
```

```
int A, B;
B = 3;
A = --B;
//A contains 2, B contains 2
```

Compound assignment operators can decrease the amount you type and can make your code more readable. These are the operators +=, -=, *=, and /=. What makes these operators special is that they use the value you want to change in the operation. For example, `x += 2` is equivalent to `x = x + 2`.

Keep in mind the order that was used, as this becomes important with subtraction and division. The variable being changed is equivalent to the two leftmost variables in the longhand expression. Let's say we have X and Y, and want to set X equal to the value of Y divided by the value of X. This is impossible with this method, as `X /= Y` is equivalent to `X = X / Y`, and `Y /= X` is equivalent to `Y = Y / X`.

Here is some sample code using the concepts we presented in this chapter:

```
#include <iostream>

using namespace std;
```

```
int main()
{
  int a = 5, b = 10, c = 15, d = 20;

  cout << "a + b = " << a + b << endl;
  cout << "d - c = " << d - c << endl;
  cout << "a * b = " << a * b << endl;
  cout << "d / a = " << d / a << endl;
}
```

The output of this code is:

```
a + b = 15
d - c = 5
a * b = 50
d / a = 4
```

8.1 Review Questions

1. Write a statement declaring two integer variables a and b and initialize them to 6 and 3, respectively.
2. Without changing the last line, fix the following code so there will be an output of 12.

```
int a = 4, b= 2;
a = a + 2 * b;
cout << a;
```

3. What is the output of the following code?

```
int a=2, b=5, c=6;
a++;
b = b*a;
c = (c-a) + 3;
```

```
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

4. What is the output of the following code?

```
int a, b, c;
a = 2;
b = 8;
c = 1;
c = b - b;
c = a + a;
c = b * 8;
b = b + b;
c = a + c;
b = a + b;
a = a * c;
b = a - c;
c = b + a;
cout << a << endl;
cout << b << endl;
cout << c << endl;
```

8.2 Homework Exercises

1. What is the output of the following code?

```
int a=4, b=2, c, d;
a = b + 3;
b++;
c = (b + 4) * 2;
c = c + 2;
d = a + b - 3;
a++;
a = a + 2 - b;
b = b * 2;
cout << "a=" << a << endl;
```

```
cout << "b=" << b << endl;
cout << "c=" << c << endl;
cout << "d=" << d << endl;
```

2. What is the output of the following code?

```
int m=3, n=2, x, y;
x = m + 5;
m--;
y = (m+4) / 3;
n = n + 2;
m = m + n / 2;
m++;
x = x * 2 - 3;
y = y * 2;
n = n + y * 3
cout << "m=" << m << endl;
cout << "n=" << n << endl;
cout << "x=" << x << endl;
cout << "y=" << y << endl;
```

8.3 Review Answers

1. `int a = 6, b = 3;`

2.
```
int a = 4, b = 2;
a = (a + 2) * b;
cout << a;
```

3.
```
3
15
5
```

```
4
  132
  66
  198
```

8.4 Homework Answers

```
1
  a=5
  b=6
  c=16
  d=5
```

```
2
  m=5
  n=16
  x=13
  y=4
```

8.5 Further Reading

- http://www.cplusplus.com/doc/tutorial/operators/
- http://www.sparknotes.com/cs/c-plus-plus-fundamentals/basiccommands/section1.rhtml

Chapter 9

Comments

Chapter 10

Data types, conversion

Chapter 11

Conditionals

Chapter 12

Strings

Chapter 13

Loops

Chapter 14

Arrays

An array is a series of variables that are the same of the same type (`int`, `float`, `double`, `char`, and so on). Arrays are held in a computer's memory in a strict linear sequence. An array does not hold anything other than the elements of the specified type, so there is no assigning an array of type `float` and hoping to store a `string` there. Doing so would cause a "type mismatch error" and the program wouldn't compile. To create an array, the programmer types something like this:

```
char Scott[5];
```

The `char` is the data type for all elements in the array, `Scott` is the name of the array (you can be as creative as you want with the name), and the 5 inside the square brackets represents the size of the array. So `char Scott[5]` can hold 5 pieces of data that are of type `char`. Look at the diagram below for assistance.

When trying to visualize an array, think of a rectangle split up into as many pieces as the array has places to hold data. In the above example, think of a rectangle with 5 open slots, each of type `char` that are waiting for some form of input.

In order to refer to the individual elements in an array, we start with the number 0 and count upwards. We use `[0]` to access the first element in the array, `[1]` for the second, `[2]` for the third, and so on. In order to read or write certain locations of the array, we state the name of the array and the element we want to access. It should look like this:

```
Scott[3] = 'Q';
cout << Scott[3];
```

The diagram below depicts how the computer interprets this.

You can also store values inside the array ahead of time when you declare the array. To do so, you need to enclose the values of the appropriate type in brackets and separate the values with a comma. Below are two examples, one an array where each element is of type `char` and another where each element is of type `int`.

```
char Scott[5] = {'S', 'c', 'o', 't', 't'};
```

```
int John[5] = {99, 5, 1, 22, 7};
```

Note that, in the C and C++ language, arrays of characters intended to be treated as a string must contain a special character called the **null character**[1] or **null terminator**. The NUL character marks the end of the string. In C++ the null character is `'\0'`. Because the null character takes up one "slot" of the array, any character array that is intended to be used as a string must be declared having a size one more than the longest string that you expect to store. Initializing the above character array should really be done as the following (notice that we make the array one slot larger!):

```
char Scott[6] = {'S', 'c', 'o', 't', 't', '\0'};
```

Alternatively, you can initialize a character array with a string literal, as below. We discuss string literals in more detail in Chapter **??**.

```
char Scott[6] = "Scott";
```

It is also possible to let the computer figure out the appropriate length for an array when the array is being initialized at the same time as when it is declared. The below code produces an identical array as the previous example:

[1] Abbreviated NUL; Note that this is not the same as the NULL pointer described in Chapter **??**

```
char Scott[] = "Scott";
```

14.1 Multi-dimensional Arrays

A two-dimensional array (some might call it a "matrix") is the same thing as an array, but is an "array of arrays". Here's a two-dimensional three-by-three array:

```
int Rich[3][3]; // 2D
```

Declaring arrays with more dimensions are possible with similar syntax. Here's a three-dimensional 10x10x10 example:

```
int Sam[10][10][10]; // 3D
```

And here is a four-dimensional 10x10x10x10 array. This is possible, even though it's hard to visualize.

```
int Travis[10][10][10][10]; // 4D
```

A user can input values into a multi-dimensional array in a similar way as a single-dimensional array.

```
int main()
{
    int neo[3][3] = { {1,2,3}, {4,5,6}, {7,8,9} }; //
        filling matrix
    cout << neo[0][0] << endl << endl; // first number
        , 1
    cout << "  " << neo[2][2]; // last number, 9
    return 0;
}
```

The same logic is applied for 3-dimensional and 4-dimensional arrays, but when filling them be mindful of the order of the input so that when you want to view certain elements in the array you are able to correctly access them.

14.2 Review Questions

14.3 Homework Questions

14.4 Review Answers

14.5 Homework Answers

14.6 Further Reading

- http://www.cplusplus.com/forum/beginner/43663/
- https://www.youtube.com/watch?v=SFGOAKYXfOo
- http://visualcplus.blogspot.com/2006/03/lesson-15-matrixe
  html

Chapter 15

Blocks, Scope, and Functions

15.1 Blocks

Since we've covered `if` statements and loops, let's go into more detail about the code that's contained within them. When you need to contain multiple lines of code, we've shown how to use braces. These braces will create a new layer in the code, and the lines within would be grouped into what is known as a compound statement, sometimes called a block.

```
int x;
cin << x;

if(x < 5)
{
  int y;
  cin << y;
  x += y; //Declares Y, asks user to define Y, then
      sets Y to X + Y
}

if(x > 5)
{
  int z;
```

```
    cin << z;
    x -= z; //Declares z, asks user to define z, then
        sets x to x - z
}

cout >> x;
//outputs either x-z, x-y, or 5
```

Take a look at the example above. There are two blocks here: the one for if `x` is less than 5, and the one for if `x` is greater than 5. Notice the variables declared in each, `y` and `z`. When these are declared, they are only usable within the blocks that they were declared. When that block reaches its end, they are lost to the rest of the program. This is because the scope of the variables within the blocks is limited to those blocks. We discuss scope further at the end of this chapter.

15.2 Basic Functions in C++

15.2.1 What are Functions and why do we use them?

Functions are an important part of C++ programming. Without them, programs would be confusing and difficult to troubleshoot. When programs are written, they tend to be written in logical chunks which we call subprograms. These subprograms are known as functions in C++ which, when called in a program, may execute whatever the programmer wants. Simply put, functions are like miniature programs that when pieced together form the actual program that you are trying to write.

15.2.2 The parts of a basic function

A **function declaration** (sometimes known as the **prototype**) is normally placed before the `main()` function in your code. This lets the compiler know that there is a function that will be defined in more detail further on in your program. With basic functions, your declarations should start with a **return type** such as `double`, `int`, and so on; this is the data type your function will return.

After the return type, the next item that needs to be written is the function's name, which can be almost anything you want. Remember that you will be using it again later in your code, so it makes sense to make it something short and logical that you can remember! Now that you have your data type and your function name, it's time for zero or more function **parameters**. These will be written inside parentheses immediately following your function's name. Each parameter is in turn made

up of a data type and a name like a variable declaration. A comma separates function parameters and your declaration must end with a semicolon after the closing right parenthesis.

Here is an example of a function declaration:

```
double profit (int cost, double price); //cost and
    price are parameters.
```

Using function looks much like an abbreviated version of the function declaration. A **function call** is responsible for telling the compiler when and how to execute a function. Function calls are found in another function like `main()`. Often the user is prompted to enter necessary data with `cout` statements and his or her response is collected with `cin`. Once this data is collected, the program holds it until a function call is made somewhere in the code. Once the function call is made, the compiler takes the entered data and then uses the code in the **function definition** (which we will go over shortly) to operate on the parameters and return a value. For your function call, write your function name followed by the variables or values you want to pass in. In a functiton call, it is not necessary to specifty the data types, as they are already understood.

Here is an example of a function call:

```
#include <iostream>

using namespace std;

double profit (int cost, double price); // function
    declaration (prototype)
int main ()
{

  double a, b;
  int c;
  cout << "Enter the manufacturing cost of the item: "
      ;
  cin >> c;
  cout << "Enter the retail price of the item: ";
  cin >> b;

  a = profit (c, b); // function call profit with cost
       = c and price = b
```

```
    cout << a << endl;
    return 0;
}
```

You have a declaration and a function call now. The only thing left is the code inside the function definition—the **function body** is the most important part because it contains the code needed by the compiler to execute the function.

The function definition will usually have a lot more code than both the declaration and the function call. As a result, the definition and body are also more difficult to write than the declaration or call. The function definition and body is often placed after your `main()` function. Multiple function definitions and bodies can be placed after your `main()` in no particular order, though it makes it less confusing if you use the same order as when they were declared. Start your function definition with your *function heading*, which looks exactly like your function declaration but without a semicolon. Following your heading, you need your function body. Start your function body by placing an opening left brace (
Code{) on the line following your heading. The code that makes up the function body follows the brace. After the code in the body is finished, you end the body with a closing right brace (
Code}). Notice that the semicolon is not necessary either after your heading or after your closing brace. The standard rules for semicolons apply within the body of the function, though. What goes inside the function body depends completely on what you want the function to do. You may declare variables to be used just in your function and can leave the function using `return` statements at any time. Below is an example of a function definition:

```
double profit (int cost, double price) // function
    definition
{
  double p; // temporary variable
  p = price  cost; // calculate the profit
  return p; // return the result to the calling
      function
}
```

Great, now that you have a grasp of the three major parts of basic functions we can move on to other related material!

The functions we just described are known as **programmer defined functions** since the programmer defines these functions. There are also **predefined**

functions which are available for your convenience. Predefined functions are functions that are already written and defined. In order to use predefined functions, the programmer needs to include the necessary library and then call the function wherever they need it.

In the following example we will use the `sqrt()` function to calculate the square root of the user's input. The `sqrt()` function is described in more detail in Chapter ??.

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
  double num;
  cout << "Please enter a number: ";
  cin >> num;
  cout << sqrt(num) << endl;
  return 0;
}
```

15.3 `void` Functions

`void` functions are functions that do not return a value. Notice that other function declarations that do return a value start with their return type such as `double`, `int`, or the like. `void` functions behave the same except no value is returned. A common application where a `void` function is used is printing the result of calculations to the screen. The calculations might be performed elsewhere, but the results would be printed using the `void` function. Syntax for `void` functions works in the same way as normal functions, but the keyword `void` is written where the return data type would normally go. The declaration, function call and definition for `void` functions will follow the same format as other functions. Note that, like other functions, there does not necessarily need to be parameters in a `void` function.

Here is an example of a simple void function declaration:

```
void displayMessage();
```

Remember the definition and calling of `displayMessage()` would be the same as any other function with the exception of the `void` return type and that no value is returned! Here is an example of a definition, declaration, and how this function would be called:

```
#include <iostream>

using namespace std;

void displayMessage();

int main()
{
  int x = 2, y;
  y = x + 1;
  displayMessage(); // This doesn't return anything

  return 0;
}

void displayMessage()
{
  cout << "Calculations are done!" << endl;
}
```

15.4 Overloading Function Names

Overloading function names allows the same name to be used in multiple function definitions but with different parameter listings. Function names can be reused using this feature. Function name overloading eliminates problems associated with having multiple names for functions with similar purposes and can make the code both more understandable and more convenient for the programmer to write.

Below is an example of an overloaded function name. Notice that both functions have the same name, but different parameter types.

```
int plus(int num, int numr);
float plus(float num, float numr);
```

Here is an example of improper function overloading. Simply changing the return type does not work, the parameters must be different!

```
int plus(int num, int numr);
float plus(int num, int numr);
```

15.5 Scope

As we dive into more complex programs there is a need for a wide variety of variables in different locations in the code. Some of these variables are declared within individual blocks of code, such as within loops or conditionals. Others are declared completely outside functions. The two primary types of variables we are going to look at here are local and global. The location of the declaration of a variable within the code changes how that variable may be used.

Local variables are declared within a block of code. A local variable is available to code from the point of its declaration through the end of that block of code. A simple example is a variable declared in `main()`:

```
int main()
{
  int games;
  // Other code here
  return 0;
}
```

The variable `games` is a **local variable** because it exists only within the local function, `main()`. It cannot be used anywhere outside `main()` without some additional work (such as passing it by reference to a function). Similarly, variables declared in other functions are not available to code in `main()`. Here is an example:

```
#include <iostream>

using namespace std;

void my_games();
```

```
int main()
{
  my_games();
  // More code here
  cout << games; // ERROR! No such variable here!
  return 0;
}

void my_games()
{
  int games = 10;
  cout << games;
}
```

In the previous example function, `my_games()` is called by `main()` and outputs `10`. The variable `games` is local to that function. If `games` is referenced anywhere else outside that function, the program will not compile.

An easy way to understand local variables is to compare them to your neighbors. Everyone that lives on your street and around you are variables, and since you all share the same street, they are local. The neighbors on an adjacent street might be close to where you live, but since they do not share the same street, they might not be considered neighbors. You can think of these neighbors on the adjacent street as other functions. While they might be close by, they do not share the same street.

Global variables are quite different from local variables. Global variables can be used by code anywhere within the program. A global variable is declared outside of any function. Using similar code as in the example above, we make the `games` variable global:

```
#include <iostream>

using namespace std;

void my_games();
void their_games();

int main()
{
  games = 5;
  my_games();
```

```
    their_games();
    return 0;
}

void my_games()
{
    cout << games << endl;
}

void their_games()
{
    cout << games << endl;
}
```

Both functions print the same variable, causing the program to produce the following output:

```
5
5
```

To sum it up, local variables work only within the block of code that it is declared. Global variables are declared outside functions, and can be used at any point in the program.

15.6 Review Questions

15.7 Homework Questions

15.8 Review Answers

15.9 Homework Answers

15.10 Further Reading

-
-

•

Chapter 16

Problem Solving

Chapter 17

Testing

Chapter 18

Preprocessor

Chapter 19

Advanced Arithmetic

Advanced arithmetic in C++ includes math that can't be used in code without the use of the <cmath> library. This is math that goes above and beyond the standard operations: addition (+), subtraction (-), multiplication (*), and division (/). As we have seen before, some simple arithmetic might look like:

```
int x;
x = 1;
x += 5;
return x;
```

The variable x is declared as an integer. The next line sets it to one. The += operator adds five to x, which makes x contain six.

Doing simple operations like these does not require any special libraries or unusual commands. Any compiler can look at a +, -, *, or / in a line of code and know exactly what the programmer is expecting to happen. Some math requires a little extra help, though. In this case, help is the <cmath> library.

<cmath> is a library that is needed for trigonometric, hyperbolic, exponential, logarithmic, rounding, and absolute value functions. The <cmath> library is designed to make your life simple and to make complicated math easier in C++. Using the <cmath> library in code is as simple as including it at the top of your source code file with the rest of your libraries:

```
#include <iostream>
#include <cmath>
```

After the inclusion of the <cmath> library, you can use certain mathematical functions in your code such as pow(x,y), which raises parameter x to the power of parameter y, and sqrt(z), which returns the square root of parameter z. In your first few C++ programs, you will probably not use the more advanced mathematical functions included in the <cmath> library, but for a full list of the mathematical functions, refer to "Further Reading" at the end of this chapter.

19.1 Examples

19.1.1 pow()

pow is the function that is called when a value or variable needs to be raised to a certain power. Take a look at this code and we'll break it down line by line:

```
int x, y;
x = 4;
y = pow (x+1, 3) + 6;
```

First, we are declaring two variables: x and y. After that, we take x and set it to 4. Now we get to a more interesting section of code. We are asking the compiler to raise the value of x plus 1 to the power of 3, add 6, and then place the result in y. To use the pow function, you must understand its syntax. Here is the breakdown:

```
pow (starting value, power being raised)
```

So for pow (x+1, 3) + 6, we are raising the starting value x + 1 to the power of 3. Before the power of 3 is applied to x+1, 1 is added to x. In this case, it is the simple operation of 4+1, which nets us 5. After we get 5, we raise it to the 3rd power to get a value of 125. After we reach the value of 125, we are finished with the pow function and return to using normal math when we add 6 to 125 and get a value of 131.

Undoubtedly there are more complicated uses of the pow function, such as multiple uses of pow in the same line of code. You might use multiple pow operations in code that calculates the length of one side of a triangle using the Pythagorean Theorem. Look at the following code and see if you can figure out what the output value would be:

```
int x, y, z;
x = 3;
y = x + 1;
z = pow (x, 2) + pow (y, 2);
cout << z;
```

If you got 25, then you have the right answer! A breakdown reveals that after initializing the variables `x` and `y` and setting their values (3 for `x` and `x+1` for `y`), we raise each value to the power of 2. For visual reference,

```
z = pow (3, 2) + pow (x+1, 2);
```

gets changed to

```
z = 9 + 16;
```

`z`'s value is set to 25. The `pow` operation is simple to use, and can make operations used in a program simpler from both a computing and visual standpoint.

19.1.2 `sqrt()`

Square roots are calculated using the `sqrt` function. Take a look at the example below to see how it is called in a program:

```
int a, b;
a = 25;
b = sqrt(a);
```

`sqrt` is simpler than `pow` in that it only requires one parameter. Since `sqrt` returns a `float` or `double`, you should usually assign the result to a `float` or `double` variable, but in this example, `sqrt` returns exactly 5, so it can be converted to an `int` without any issues.

There are cases where both `sqrt` and `pow` are used in the same formula, such as when calculating the distance between two points. When writing such code, it is very important to keep track of the parentheses and to use correct syntax. One

such syntax mistake is made when programmers think that C++ syntax is the same as algebraic syntax. This is *not* the case in C++!

```
int x = (5)(pow (3, 3) );
```

is NOT correct syntax. When the compiler sees this, it doesn't view it as multiplication, but instead as (according to a professional), "function shenanigans." It is important to be explicit with symbols relating to math in C++. So instead of the inaccurate code above, use:

```
int x = 5 * (pow (3, 3) );
```

Now, as an example, we will use code to compute the distance between two points on a plane. Refer to the code and the diagrams if you do not understand or get lost.

```
int x1, x2, y1, y2;
float dist;
x1 = 4;
y1 = 4;
x2 = 6;
y2 = 10;

dist = sqrt (pow (x2 - x1, 2) + pow (y2 - y1, 2));
cout << dist;
```

So your final answer after the calculation is executed is roughly 6.342555. Without the help of the advanced arithmetic operations, getting to this result would be a difficult, long, drawn-out process. `pow` and `sqrt` are handy little functions that make life easier, all with the help of the `<cmath>` library.

19.1.3 Modulo

The modulo operator (the percent sign: `%`) finds the remainder, or what was left over from division. This program uses the modulo operator to find all prime numbers (all the numbers that never have a remainder of 0 when divided by every number except 1 and itself) that can be held by an `int`.

```
#include <iostream>
using namespace std;
int main()
{
  int testprime = 0, divby = 0, remainder = 0;
  bool isprime;
  cout << "Prime Number Finder" << endl;
  while(testprime < 2147483647)//The Maximum for int
  {
      isprime=true;
      testprime++;
      for(divby=2; divby < testprime; divby++)
      {
          remainder = testprime % divby; // store the
              remainder of the current number when
              divided by divby
          if (remainder == 0)//If the number is not
              prime
          {
              isprime = false;
    break;
          }
      }
      if (isprime)//If it passes the test, it is prime
          .
      {
          cout << " " << testprime; // tell us what
              the prime number is.
      }
  }
  return 0;
}
```

19.2 Review Questions

1. Which `#include` library is needed to use advance arithmetic operators?
2. Write C++ code to calculate 2^9.

3. Write a statement to set the value of a variable of type `double` to the square root of 10001.

19.3 Homework Questions

Complete the code below to find the length of the hypotenuse of a right triangle (remember that $a^2 + b^2 = c^2$) given the lengths of the other two sides. What will be the final output of the code?

```
#include <iostream>
// Add necessary libraries here

using namespace std;

int main()
{
  float a = 3.0, b = 4.0;
  float c;

  //
  // Finish the program...
  //
  cout << "The hypotenuse of the right triangle is "
       << c << endl;
}
```

19.4 Review Answers

- `#include <cmath>` must be included to include advanced operators.
- `pow(2, 9)`
- `double b = sqrt(10001);`

19.5 Homework Answers

```
#include <iostream>
#include <cmath>
```

```
using namespace std;

int main()
{
  float a=3.0, b=4.0;
  double c;

  a=pow(a,2);
  b=pow(b,2);
  c=sqrt(a+b);

  cout << "The hypotenuse of the right triangle is "
       << c << endl;
}
```

The final output of the code is:

```
The hypotenuse of the right triangle is 5
```

19.6 Further Reading

- http://pages.cpsc.ucalgary.ca/~jacob/Courses/Fall00/CPSC231/Slides/08-Arithmetic.pdf
- http://www.cplusplus.com/reference/cmath/

Chapter 20

File I/O

Chapter 21

Pointers

Chapter 22

Dynamic Data

Chapter 23

Separate compilation

Chapter 24

Classes and Abstraction

Chapter 25

STL

The Standard Template Library (STL) provides a set of tools beyond those that are provided by the "base" C++ language. In fact, you have already become familiar with some of them [back in Chapter X], such as input and output streams, Y, and Z. While a comprehensive discussion of the features of the STL is far beyond the scope of this text, there are several libraries that offer extremely important features with which you should become comfortable. [footnote: A reasonable place to start for a wider view of the STL is here: Wikipedia Standard Template Library

#include <cstdlib>
#include <vector>
#include <map>

This library provides one of the STL's associative container object classes. An associative container differs from an array in that items in an array are referenced with a number which indicates the item's position in memory:

```
int myArray[10]; // An array of ten integers
myArray[0] = -5; // Set the first integer in the array
    to -5
```

An associative container, on the other hand, can use any data type to reference the items in the container. For example, you might choose to use a string to reference a collection of int items.

```
std::map<std::string,int> myMap;}
```

Perhaps you want to create the object with some initial values:

```
std::map<std::string,int> myMa=p;

// map::at

int main ()
{
 std::map<std::string,int> mymap = {
              { "alpha", 0 },
              { "beta", 0 },
              { "gamma", 0 } };

mymap.at("alpha") = 10;
mymap.at("beta") = 20;
mymap.at("gamma") = 30;

for (auto\& x: mymap) {
  std::cout << x.first << ": " << x.second << '\\n'
      ;
}

return 0;
}
```

The container can only hold one instance of each key, and attempts to add a duplicate item will not overwrite the existing item. The multimap object is a similar structure to map that allows for duplicate items to be added.

Listings