

Programming for Evolutionary Biologists

John P. Huelsenbeck

*Department of Integrative Biology
University of California, Berkeley*

July 22, 2018

Contents

1	Introduction to Programming	1
1.1	On becoming a card-carrying computational biologist	1
1.2	What is programming?	2
1.3	Choosing a programming language	2
1.4	Goals	3
2	IDE Misery	5
2.1	What is an IDE?	5
2.2	Getting started with Xcode	6
2.3	Getting started with CLion	8
2.4	Getting started with Eclipse	8
3	Hello World: the World's Most Boring Program	9
3.1	Hello World: Boring, but informative	9
3.2	<code>main</code> is a function	9
3.3	Every C++, and C, program starts with <code>main</code>	10
3.4	Include statements expose the built in functions of the language	10
3.5	Please explain <code>std::cout << "Hello, World!" << std::endl</code>	12
3.6	Functions take arguments and return values	13
4	Variables are Fun!	17
4.1	The basic variable types	17
4.2	Variables take up space	18
4.3	Variables in computers have limits	20
4.4	You can use a variable to hold a memory address	22
4.5	Dereferencing pointers	23
4.6	Arrays	24
5	Conjunction Function	31
5.1	Making your own functions	31
5.2	You can pass variables to a function by value or by reference	32
5.3	Variables have a scope	37

<i>CONTENTS</i>	1
6 Classes	41
6.1 Into the deep end of the pool	41
6.2 Making your first class	41
6.3 Why do we split the class into .hpp and .cpp files?	43
6.4 Instantiating a class	45
6.5 Implementing the uniformRv function	51
7 Markov chain Monte Carlo	55
7.1 Some theory	55

Chapter 1

Introduction to Programming

1.1 On becoming a card-carrying computational biologist

As an evolutionary biologist interested in programming, you are joining a group with a proud history that extends back to the beginnings of the computer era. It is largely unacknowledged that the field of computational biology was founded by biologists, and not just by any flavor of biologist, but by *evolutionary* biologists. In fact, Sir Ronald A. Fisher — yes, *that* R.A. Fisher who was a founder of modern population genetics as well as an architect of modern frequentist statistics — is likely the first person to have used a computer to solve a problem in biology. Fisher (1950) used an EDSAC computer in Cambridge to compute the expected allele frequencies in a cline. Just a little more than a decade later, in 1963, his former postdoc and student, Luca Cavalli-Sforza and Anthony Edwards, respectively, wrote some of the earliest programs to estimate phylogeny, executing the programs on an Italian-made Olivetti Elea 6001 computer (Edwards, 2009). The next generation of evolutionary computational biologists came of age, scientifically at least, in the 1970s and 1980s and included such luminaries as Joe Felsenstein, Elizabeth Thompson, Steve Farris, Masatoshi Nei, David Swofford, and the brothers, Wayne and David Maddison. They addressed all sorts of problems in evolutionary biology, but mainly concentrated in population genetics and phylogeny estimation which posed problems that could not be solved analytically, but were amenable to numerical solution using a computer. Although these researchers did not necessarily all get along, it was at this time that a community consisting of evolutionary biologists who took a computational approach in their research developed. As a rule, this group of researchers was generous in sharing their knowledge with others; they passed on their knowledge, either through formal mentorship or through more informal means (which typically involved some consumption of beer), to the next generation which includes too many biologists to list here. This author, in fact, learned many of the tricks of the trade for dealing with trees in computer memory from David Swofford¹. This book is my attempt to pay it forward by passing some of the tricks of the trade on to you, a member of the next generation of evolutionary computational biologists. My hope is that my means of paying it forward, in the form of this book, will be as instructive to you as David Swofford's tutelage was for me.

¹Swofford is a strong proponent of the informal method for passing on programming knowledge.

1.2 What is programming?

Programming is the act of writing instructions for a computer to perform. The instructions are written in a language, called a programming language of which there are many to choose from. Typically, the instructions are simply written on a plain text file which is then read by a computer program called a ‘compiler’ which compiles the code, turning your instructions into a form that can be read directly by the computer. The coding language is a necessity as it is readable by ordinary humans whereas the compiled code, which is gobbled up by the computer and executed instruction-by-instruction, is much more difficult to comprehend. The coding language also helps insure portability of the ideas expressed in your code to various computer platforms. Machine-readable code — the output of a compiler — is machine specific, so the executable code for one computer and operating system will not necessarily work on another. The code, however, can be ported from computer to computer and compiled for each.

1.3 Choosing a programming language

Programming is difficult. Turning a concept into an algorithm that is implemented in code is not an easy task, for one. But, for the beginner, there are several hurdles that must be cleared before the interesting problem of algorithm development can even be tackled. You need to master, or begin the process of mastering, the programs that allow you to program. I will discuss these programs, called Integrated Developer Environments (IDEs), in the next chapter. The first decision to make, however, is which programming language to use. All programming languages have similarities, and as you become more experienced, you will notice these similarities and realize that learning two different computer languages, such as C++ and Fortran, is much easier than learning two spoken languages such as Spanish and German. But, these similarities won’t become apparent until you have one language, at least, under your belt. Which language should you invest your precious time in learning?

Popular computer languages include Java, Python, JavaScript, C, C++, C#, PHP, Perl, Swift, R, Rust, and Ruby, to name a few. More experienced programmers can be helpful in guiding you in choosing a first language, but you should beware. Programmers often have very strong opinions about the merits (and demerits) of various languages. In this book I chose to use C++ to illustrate ideas. C++ is not a bad choice for a first language to learn. You could do worse. It is widely used and powerful.

Importantly, C++ is a compiled language. “What is a compiled language?,” you ask. A compiled language is one that must be run through a compiler, which produces instructions specific to the target machine. Some languages, such as the popular Python, are interpreted languages. The instructions for an interpreted language are not directly executed by the target machine, but rather are read and executed by another program, cleverly called the ‘Interpreter.’ Interpreted languages are often easier to run and use, but typically do not run as fast as compiled languages.

A C++ programmer can also take advantage of the code written by others to solve various problems, an advantage C++ shares with other languages. This code is referred to as a library. Sometimes, they are precompiled. The important point is this: you can integrate the library into your program and use the functionality of the library to solve your own problems. This saves you a lot of time, allowing you to concentrate on the aspects of your problem that are truly unique. Moreover, libraries can often be of very high quality. The routines you take advantage of in a library

are likely better-written than anything you (or I) could write. An example of a C++ library often used by scientists is the Boost Library.

1.4 Goals

This book is not meant to provide you with an in depth review of C++ and programming. Rather, my goal is to get you started with programming in evolutionary biology. The exercises presented in each chapter build on each other. The idea is that by working through the exercises, you will not only learn a lot about programming, but better understand many of the algorithms that are used by evolutionary biologists.

As I mentioned, above, programming is hard. You should not expect to understand every concept on the first go. Be persistent! Don't give up! If you approach programming with a positive attitude, you'll find yourself enjoying the process.

Good luck and happy programming!

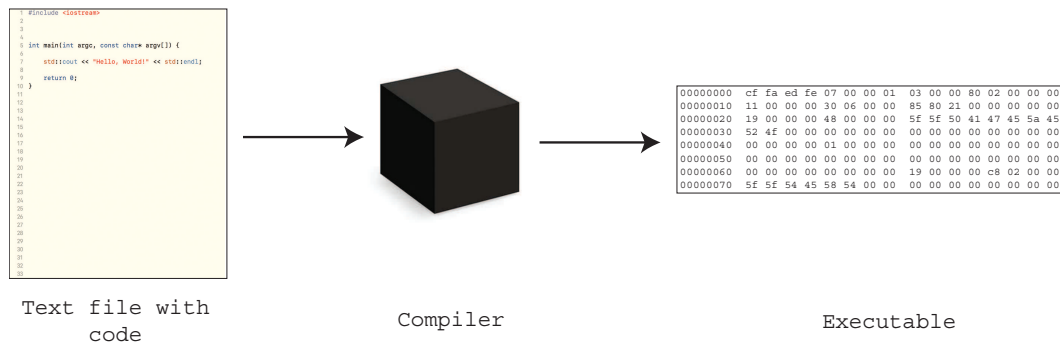
Chapter 2

IDE Misery

2.1 What is an IDE?

When I was a graduate student, my advisor bought me a powerful (for the time) and exceedingly expensive Sun SPARC workstation. It was the first computer I had used that ran a Unix operating system. For the first two weeks, until I figured out how to install and operate the compiler, this workstation was nothing more than an expensive time piece. The process of working out how to install and use the compiler was, for me, a frustrating experience. However, my experience was not unique; the first, and worst, experience for a beginning programmer is learning how to actually compile and run a program for the first time. Fortunately, it is much easier to get started programming today. Why? Integrated Developer Environments (IDEs).

Remember that writing, compiling, and running a program involve the following steps:



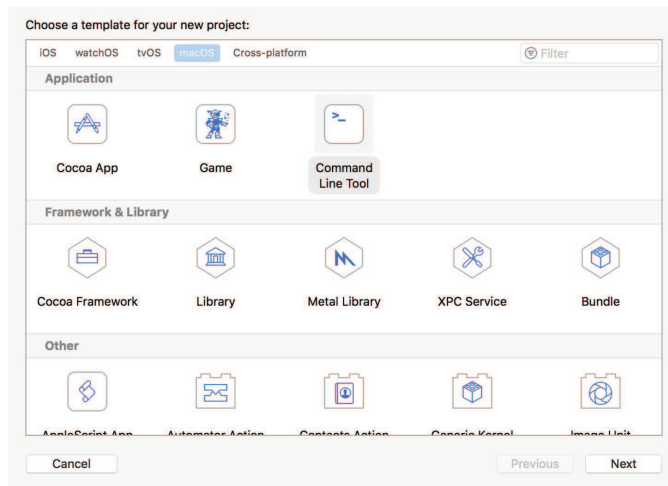
Entering the code into a text file can be accomplished using any text editor, even Microsoft Word as long as the file is saved as a plain text file. And, compiling the code can be performed directly, by running the compiler from the command line. IDEs, however, simplify both tasks by wrapping the code generation and compilation into one program. The text editor for an IDE can be customized to suit your style, or the style of your programming team. Colors can be used to indicate different aspects of the code, such as control statements, variables, and comments. The IDE also manages the source code files, of which there may be many, that constitute the program. The compiler is run from the IDE. Typically, the settings for the compiler can be set from the IDE. Finally, IDEs have built in debuggers, which can help you track down errors in your code.

There are many IDEs that you can use. In this book, I will assume that you are using Xcode, if you are using a computer running the MacOS. If you are using a computer running the Windows operating system, I would recommend installing CLion, Eclipse, or Visual Studio.

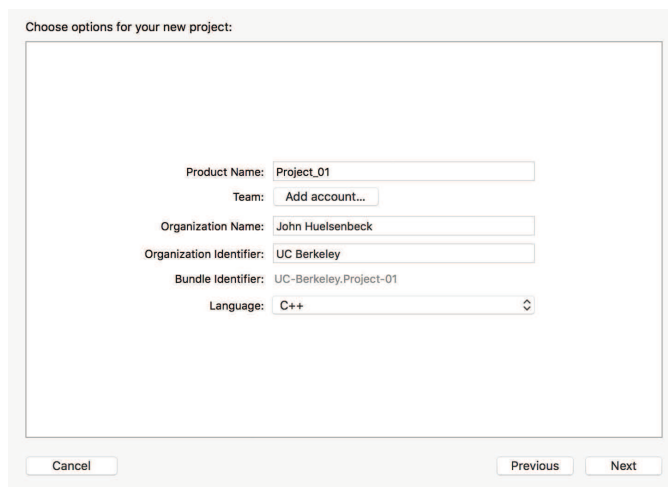
2.2 Getting started with Xcode

XCode is an IDE developed by Apple and given away for free, which is a pretty good deal in my opinion. You can download XCode through the App Store application on your Mac. Search for XCode in the App Store search field, then click on the 'Download' button. XCode is a large program, so be patient while the program downloads to your computer.

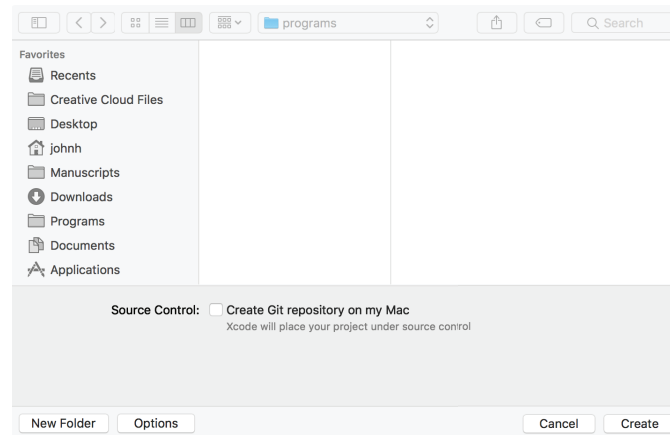
Launch XCode after you download it. Select **New > Project** from the **File** Menu. You will see the following window:



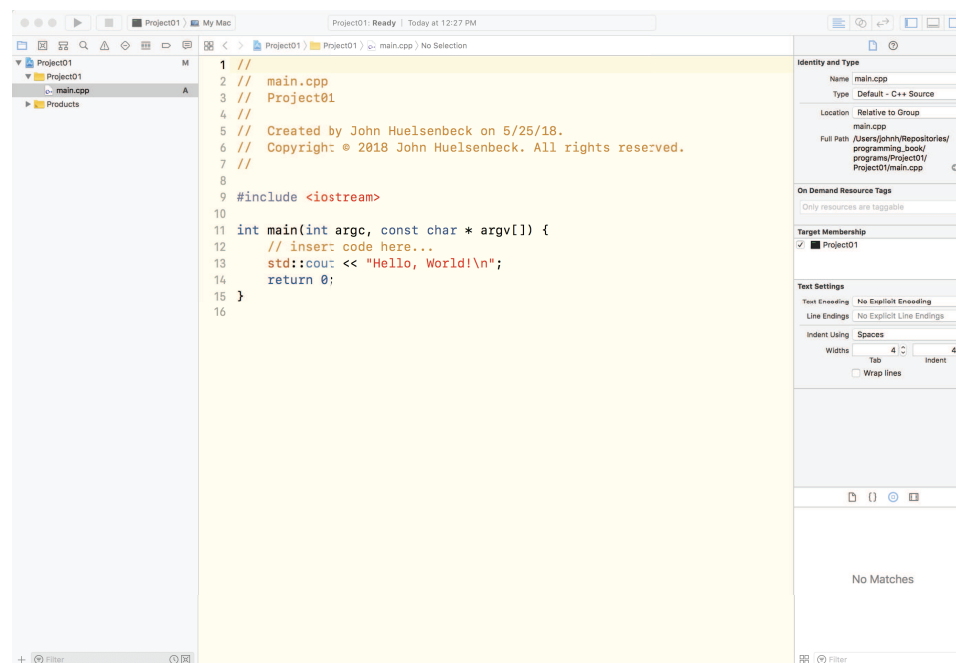
Select **Command Line Tool** and click on **Next**. This leads to the following window:




Name the project. Here, I cleverly named the project `Project01`. You can name your project whatever you like. However, I will be referring to this program as ‘Project01’ throughout this book, so you can simplify your life by following my lead here. You might consider entering your name instead of my name in the **Organization Name** field. Similarly, enter something sensible for **Organization Identifier**. Most importantly, make certain that **C++** is selected as the **Language**. Then select **Next** to continue to the next (and, thankfully, last) window that allows you to save your project to a specific location on your computer:

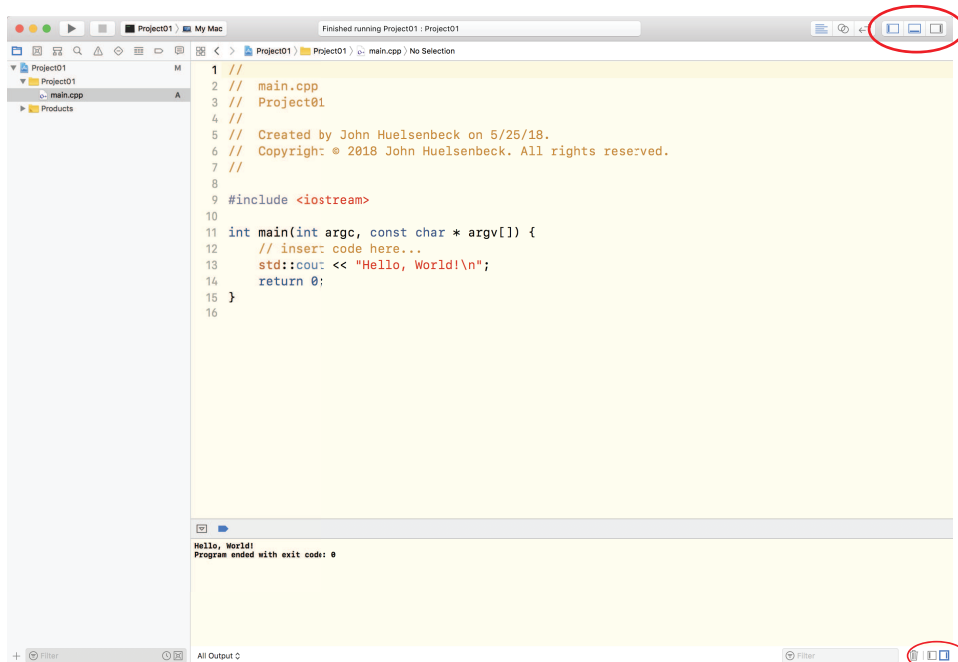


Do not choose to **Create Git repository on my Mac**. I’ll explain source code repositories, such as Git, in a later chapter. You should see the following after you save your project:



Congratulations! You have just ‘written’ your first computer program. How? XCode fills in a

bit of starter code for you. If you want to, you can compile and run this code by clicking on the ‘play’ button () in the top-left corner of the window.



Play with the toggles, indicated by the red ellipses until you get the window to look like the one, above. The window, above, contains just the information you need to write the code (the portion in the top window) and see the results of the program running (the portion in the lower window).

2.3 Getting started with CLion

2.4 Getting started with Eclipse

Chapter 3

Hello World: the World's Most Boring Program

3.1 Hello World: Boring, but informative

If your IDE has not already done so, enter the following text in the `main.cpp` file for the first program you set up in your IDE, called 'Program01':

```
#include <iostream>

int main(int argc, char* argv[]) {

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

The code you have entered implements the well-known 'Hello, World' program. Almost every introductory programming book starts with some variant of this program. What does the program do? It simply prints the phrase, "Hello, World!" to the standard output on your computer. (In Xcode, the output is directed to a window in the lower portion of the viewer. In Eclipse, the standard output is directed to ...) Clearly, this is a very boring program. You would think we could do better. (We can.) That said, the Hello World program illustrates a few points.

3.2 main is a function

Mathematicians use functions all of the time. For example, consider the function called $f(x)$:

$$f(x) = x^2$$

This function takes as input the variable x and outputs the square of x , or x^2 . Programmers also make extensive use of functions. In fact, the Hello World program has one function, called `main`. The outline of the `main` function is:

```
int main(int argc, char* argv[]) {
}

```

The `main` function takes as input arguments two variables which are listed in the parentheses (`int argc`, `char* argv[]`, which will be discussed in more detail later). The `main` function also returns a value, the type of which is an integer (called `int` here). (Note, we will discuss variables extensively in the next chapter.) The body of the function is the space between the two curly brackets.

How would we define a function that squares a value, like the simple example function we gave above in which $f(x) = x^2$? The following would accomplish this:

```
double squareFunction(double x) {
    return x * x;
}

```

The function is named `squareFunction`. The function takes as an argument the value `double x`. What does this mean? The word `double` refers to the variable type. We won't go into great detail about `double` variables here, but for now it suffices that `double` variables can hold real numbers, such as 3.14. The function also returns a value, which is the square of the value that is passed into the function as an argument. (Here, `x * x` is equivalent to $x \times x$).

3.3 Every C++, and C, program starts with main

You now have a basic idea of how functions are defined in C++, and C for that matter. The pattern is

```
<return type> <function name>(<input variable(s)>) {
}

```

with the body of the function being the commands that are typed between the curly brackets.

Every C++ program begins execution at a function called `main`. Every C++ program must contain one, and only one `main` function. If you ever happen to be examining someone else's code, look for the `main` function. Everything starts there.

3.4 Include statements expose the built in functions of the language

Change line 1 of the program by typing two backslashes in front of the statement, `#include <iostream>`. The line should look like

```
//#include <iostream>
```

The double slash denotes comments in your code. Everything after the `//` on that line is not code that is read by the compiler, but only there to help you and others understand the code. So, the `//` in front of the `#include` ‘comments out’ that line of code. What happened when you did this? In Xcode, I get the following error pop up in the IDE, next to that line: ‘Use of undeclared identifier ‘std’.’ What is the include statement doing that is so important that leaving it out breaks the program?

Any command that starts with the pound sign, `#`, is called a ‘compiler directive.’ The compiler directive is read by the preprocessor of the compiler, which reads the code before compiling, resolving any of the directives. The `#include <iostream>` directive tells the compiler to look for a file called `iostream` and include that file in the project.

On my computer, `iostream` is a physical file located in the Xcode program bundle. To understand what is going on here, I’m going to back up a few steps and give you a more general view of a programming language. Imagine the following scenario: two different companies are writing C++ compilers. Company 1 decides that they really don’t like how C++ uses the word `double` when referring to a variable that will hold a real number. Sensibly enough, they decide that in their compiler, whenever you want to declare a variable named `x` that will hold a real number, you will type

```
real x;
```

Meanwhile, the programming team from Company 2, which happens to be German, decides like Company 1 that `double` is not a sensible name for a real number. But, being German, they decide that in their compiler program, that variables that hold real numbers will be declared

```
zahl x;
```

Both companies have made seemingly sensible decisions. What could go wrong?

It turns out that a lot could go wrong if compiler coders made arbitrary decisions like the ones I described. Imagine that you wrote a computer program in C++ that is read by and compiled on Company 1’s compiler. You give your code to your friend, who happens to have bought Company 2’s compiler. Your friend would not be able to compile your code without the hassle of changing all of the ‘reals’ to ‘zahls.’ Yikes! Or, I should say, “Heiliger Strohsack!”

To prevent such chaos, writers of compilers adhere to a standardized version of the C++ language. The language standards, of course, are maintained by a committee, in this case the International Organization for Standardization (ISO). A C++ compiler must closely adhere to the ISO guidelines in order to be ISO compliant. Variables that store real numbers must be called `double`, not `real` or `zahl`. And, certain built-in functions must not only be implemented, but must be implemented in certain files that can be included to expose that functionality. The line that begins `std::cout` outputs text to the console. `std::cout` is implemented in the `iostream` file, which must be included if you use `std::cout` in your code.

Several standards for the C++ language have been released by ISO. The most recent is C++17, which was released in December 2017.

Throughout this book, you will see examples of new functions that require different files to be included (*e.g.*, if you want to take the natural log of a number, using the `log` function, you need to include the `cmath` file). Why didn't the C++ standard just include *all* of the functions automatically? Why include functionality piecemeal? There are several reasons. For one, by including only those files necessary for the bit of code you are writing, compiling that code is faster. More generally, good programmers follow the principle of including only the minimum necessary to compile code. When you garden, you only bring the tool(s) necessary to prune your shrubs, or weed. You don't bring the entire contents of your gardening shed. Similarly, in programming, we only include the tools necessary for the portion of code expressed in the file.

You will see two examples of `#include` directives:

```
#include <file>
#include "file"
```

The first is used to expose the built-in functionality of the C++ language. Files that you will often include are `iostream`, `iomanip`, `cmath`, `time`, and `fstream`, among others. The second include, using the quotation marks, is used to include files from your program project. You will see that you do not write all of the code in a single text file. Rather, the code that constitutes your program is scattered among files. A large program, such as RevBayes (Höhna et al., 2016), can include thousands of files!

3.5 Please explain `std::cout << "Hello, World!" << std::endl`

The main body of the 'Hello, World' program does two things: print the text, "Hello, World!" and then return the number 0 to the operating system. Output is handled by `cout`, which stands for 'console out.' The command, `std::cout` opens an output stream, directed to the console. A stream is an abstraction used for input and output in C++.

C++ uses the abstraction of a 'stream' to control input and output to a device. The input and output can be from the console or to a file or to some object. A stream is nothing but a sequence of characters. One can insert characters into the stream with the `<<` operator. One can also extract characters from the stream with the `>>` operator.

The command, `std::cout` opens a stream to the console. The next command, `<< "Hello, World!"` inserts into the stream the string, 'Hello, World!' A string is a series of characters, that can include spaces. A string is defined by the characters between the quotation marks. The last command, `<< std::endl` inserts into the stream an end-of-line character. (If you are familiar with those ancient devices called typewriters, you can think of the end-of-line character as being a carriage return.)

A lot more could be said about streams here. They provide the C++ language with a flexible way to output data and to take in data that is device independent (*i.e.*, the C++ interface remains the same regardless of where the stream comes from or goes to). We'll see more of streams in later chapters.

3.6 Functions take arguments and return values

Earlier, I mentioned that `main` is a function and that the general pattern of functions is:

```
<return type> <function name>(<input variable(s)>) {  
  
}
```

Let's delve more deeply into what the `main` function is taking in as arguments and what it is returning.

The `main` function takes as arguments two variables, `int argc` and `char* argv[]`. Where do these arguments come from? In this case, `argc` and `argv` are supplied by the operating system when the program is executed. `int argc` is a variable called 'argc' which can hold an integer value. The other variable that is provided by the operating system, 'char* argv[],' is a bit more mysterious. This is an array of pointers to strings. I realize that probably made no sense, but here is another try: `char* argv[]` is a vector of memory addresses, each of which indicates the starting address for a string.

What is contained in `int argc` and `char* argv[]`? `int argc` holds the number of strings contained in `argv`. Rewrite your `main` function to look like this:

```
int main(int argc, char* argv[]) {  
  
    std::cout << "argc = " << argc << std::endl;  
    for (int i=0; i<argc; i++)  
    {  
        std::cout << "argv[" << i << "] = " << argv[i] << std::endl;  
    }  
  
    return 0;  
}
```

When I run the above program, I get the following output:

```
argc = 1  
argv[0] = /Users/johnh/Project01/DerivedData/Project01/Build/Products/Debug/Project01  
Program ended with exit code: 0
```

The operating system is passing to the program, through the `main` function, information on how the program was called. In this case, there is one argument and that argument is the path to the executable.

On my computer, a Macintosh, I opened the `terminal` app and typed the path to the executable, which is everything in the line, above, except the executable name:

```
cd /Users/johnh/Project01/DerivedData/Project01/Build/Products/Debug/
```

If I type the unix command, `ls`, I see the list of files in the `Debug` directory: 'Project01.' The executable can be run using the following command,

```
./Project01
```

which gives the following output:

```
argc = 1
argv[0] = ./Project01
```

Now, let's have some fun. I am going to type the following the next time I execute the program,

```
./Project01 David Swofford had a little lamb which was named, sensibly enough, PAUP
```

This line produces the following output on my computer:

```
argc = 13
argv[0] = ./Project01
argv[1] = David
argv[2] = Swofford
argv[3] = had
argv[4] = a
argv[5] = little
argv[6] = lamb
argv[7] = which
argv[8] = was
argv[9] = named,
argv[10] = sensibly
argv[11] = enough,
argv[12] = PAUP
```

Note that the operating system broke up the sentence by words, separated by blank space(s). Again, the first string that is passed to the `main` function is the path to the executable name. The other strings, however, are the individual words that followed the executable name.

You may have had experience with programs that run from the command line. Genomics analysis often involves the use of many command-line programs, perhaps stitched together using a language such as python. For example, ClustalW is a widely-used program that aligns nucleotide or amino acid sequences. The user manual gives the following example of how to call ClustalW from the command line:

```
clustalw2 -infile=my_data -type=protein -matrix=pam -outfile=my_aln -outorder=input
```

Note that the word `clustalw2` is the executable name. Simply typing this word executes the program. All of the words after `clustalw2`, however, are arguments that are passed into Clustal's main function. The programmers for Clustal read the number of arguments (`argc`) and the strings `argv` to set the program's state. Command line arguments passed into main using `int argc` and `char* argv[]` are a convenient, if not particularly user-friendly, way to specify things such as input and output files, *etc.*

Finally, the program returns to the operating system the number 0. It does this using the return statement, `return 0`.

This concludes our discussion of the 'Hello World' program. Who would have thought that such a simple program would provide so much fodder for discussion?

Chapter 4

Variables are Fun!

4.1 The basic variable types

Modify the main function for your first project, `Project01` so that it reads:

```
int main(int argc, char* argv[]) {  
  
    int x = 3;  
    std::cout << "x = " << x << std::endl;  
  
    return 0;  
}
```

This simple program declares a variable named ‘x’ and initializes its value to `x = 3`. This all occurs on one line, `int x = 3`. The next line prints the value of `x`. The output of the program should look like:

```
x = 3  
Program ended with exit code: 0
```

The single line in which we declared and initialized the variable `x` could have been done in two lines, instead:

```
int x;    // declare variable called x  
x = 3;   // set the value of the variable x to 3
```

Most programmers, when declaring a variable, will also initialize it to some value. Normally, if I were declaring a variable like `x`, I would initialize its value to zero, `int x = 0`, so that I could rely on it being zero. Some compilers, when you declare a variable without initializing it (*e.g.*, `int x`),

initialize the value to zero. Other compilers, however, do not do this and the value of the variable will reflect the pattern of bits that happen to be at that memory address. Declaring and initializing the variable ensures that it has a value that you can rely on. Note that I added comments to the two lines, above; the words after the `//` are the comment and are not read by the compiler.

In C++, when you declare a variable, you also indicate the variable type. Here, we are declaring the variable `x` to be of type `int`. In C++, and other languages too, `int` declares a variable that can hold integers. (Remember, integers are the numbers `...`, `-3`, `-2`, `-1`, `0`, `1`, `2`, `3`, `...`)

What if you wanted to store and manipulate a real-valued number in computer memory? You can do this with the `float` or `double` variable types. Modify the `main` program, again, to read:

```
int main(int argc, char* argv[]) {

    int x = 3;
    std::cout << "x = " << x << std::endl;
    double y = 3.14;
    std::cout << "y = " << y << std::endl;

    return 0;
}
```

When I run this program, I get the following output:

```
x = 3
y = 3.14
Program ended with exit code: 0
```

The other standard variable types are summarized in the following table:

Variable	Type	Example
<code>int</code>	Integers	<code>-1, -100, 0, 314, 10001</code>
<code>unsigned int</code>	Natural Numbers	<code>0, 1, 2, ...</code>
<code>float</code>	Real Numbers	<code>-4.23, 10.01e+4, 10203.0001</code>
<code>double</code>	Real Numbers	Same as with floats
<code>char</code>	Characters	<code>c, a, B, Z</code>
<code>bool</code>	Boolean	<code>true, false</code>

There are a few other variable types that you might come across, but the table summarizes the main ones you will likely ever use.

4.2 Variables take up space

When you declare a variable, such as `int x = 0`, the operating system sets aside enough space on your computer's memory to represent the variable. Interestingly, C++ allows you to see how much

space is set aside and even where the variable resides in memory. Rewrite `main` in `Project01` to read:

```
int main(int argc, char* argv[]) {

    int x = 0;
    std::cout << "x's value   = " << x << std::endl;
    std::cout << "x's address = " << &x << std::endl;
    std::cout << "x's size   = " << sizeof(int) << std::endl;

    return 0;
}
```

When I run this program, I get the following output:

```
x's value   = 0
x's address = 0x7ffeefbfff57c
x's size    = 4
Program ended with exit code: 0
```

The first line of code, `int x = 0`, simply declares and initializes an integer variable called `x`. The next line of code should also make sense to you; it simply prints out the value of `x`, which because you initialized the variable at the same time that you declared it, is predictably zero. It's on the next line of code which prints out the memory address of `x`, `std::cout << "x's address = " << &x << std::endl`, where things get interesting. You can get the memory address of a variable by putting the ampersand symbol, '&,' in front of the variable's name. So, `&x` represents the memory address of the variable `x`. You can see that when I ran the program on my computer, the memory address is reported to be `0x7ffeefbfff57c`.

The memory address deserves explanation. First of all, the memory on your computer is arrayed in bytes, each of which has an address. The address is either a 32 bit or 64 bit number. On my computer, the addresses are 64 bits and output as hexadecimal numbers. The number representing the memory address for `x`, `0x7ffeefbfff57c`, is in hexadecimal format¹ It's unlikely that when you ran the program that your computer put the value `x` in the same place in memory as when I ran the program on my computer.

¹Converting between number bases is not too difficult. Remember that a base-10 number is in the following format: $\dots _ \times 10^4 + _ \times 10^3 + _ \times 10^2 + _ \times 10^1 + _ \times 10^0 + _ \times 10^{-1} \dots$ where the blanks are filled with the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or, 9 depending on the number that is represented. For example, the number 1066 would be $1 \times 10^3 + 0 \times 10^2 + 6 \times 10^1 + 6 \times 10^0$. A binary (base two) number works in a similar fashion, but the format is $\dots _ \times 2^4 + _ \times 2^3 + _ \times 2^2 + _ \times 2^1 + _ \times 2^0 + _ \times 2^{-1} \dots$, with the blanks filled in with the digits 0 or 1. A hexadecimal number is in base 16. The format for the numbers is $\dots _ \times 16^4 + _ \times 16^3 + _ \times 16^2 + _ \times 16^1 + _ \times 16^0 + _ \times 16^{-1} \dots$, with the sixteen digits being 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *C*, *D*, *E*, and *F*, the letters standing for 10, 11, 12, 13, 14, and 15 (in base-10).

The last line of the modified program prints the size of the variable, `x`. You can see that an integer takes up four bytes on my computer. In fact, the first byte will reside at the address that was printed out (`0x7ffeefbff57c`). The next three bytes will be adjacent to the first, at the locations `0x7ffeefbff57d`, `0x7ffeefbff57e`, and `0x7ffeefbff57f`.

Of course, the memory address of a variable can change each time the program is run. The amount of space that is set aside for each variable type, however, is constant. On my computer, the standard variable types take up the following amount of space:

Variable	Number Bytes
<code>int</code>	4
<code>unsigned int</code>	4
<code>float</code>	4
<code>double</code>	8
<code>char</code>	1
<code>bool</code>	1

4.3 Variables in computers have limits

We now know that when we declare a variable to be of type `int`, that the compiler sets aside four bytes of space somewhere on your memory card. What is the largest and smallest value that can be stored in computer memory as an `int`?

Each byte of memory consists of eight ‘bits,’ each of which has two states, on (1) or off (0). The binary representation of the first ten positive integers is

Number	Binary Representation
0	00000000000000000000000000000000
1	00000000000000000000000000000001
2	00000000000000000000000000000010
3	00000000000000000000000000000011
4	00000000000000000000000000000100
5	00000000000000000000000000000101
6	00000000000000000000000000000110
7	00000000000000000000000000000111
8	00000000000000000000000000001000
9	00000000000000000000000000001001
10	00000000000000000000000000001010

Note that most of the bits, the leading ones, are not being used for our `int` variable. In fact, if we knew that the largest number we wanted to hold was 10, we could get away with one byte (8 bits). There is a variable type called ‘`short int`’ which only takes up two bytes of memory. In this case, in which we know that the maximum size of the integer we want to hold is 10 (in base-10), we could use a `short int` instead, thereby saving two bytes of memory. If we were programming in the 1970s, we might go ahead and do just this. In the 1970s, a good computer had only thousands of bytes of memory. Today, we have billions of bytes of memory to play with. Most programmers do not worry about saving a few bytes. Rather, they worry more about places in the code that a lot of memory is allocated (where a lot of memory might be millions of bytes, or megabytes, are allocated).

internal representation of the real number for the `double` type uses twice as many bytes as the `float` type.

You should always be concerned about numerical accuracy when doing computations in evolutionary biology. Some of the more obscure output from many programs, such as reporting the log of a probability, are done to avoid underflow.

4.4 You can use a variable to hold a memory address

As I pointed out earlier, we can get the memory address of a variable by putting an ampersand in front of the variable in computer code. This seems to be a neat trick, but otherwise useless. After all, why should we care about the location of the variable in memory? This is especially true because we, as the programmer, do not even control where the variable is to reside.

It turns out that knowing the memory address is quite important. If we know where a variable resides in memory, we can manipulate that variable. Moreover, other parts of your code, such as the functions you code, also have a memory address when the program is executed. If we know where a function resides in computer memory, we can also manipulate it.

What if we want to remember the memory address of a variable? We can make another variable that will hold the memory address. Such a variable is called a ‘pointer’ in the computer science lingo. Rewrite your little program so that the main function now reads:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x = 0;
    int* xPtr = &x;

    std::cout << "x = " << x << std::endl;
    std::cout << "&x = " << &x << std::endl;
    std::cout << "xPtr = " << xPtr << std::endl;
    std::cout << "&xPtr = " << &xPtr << std::endl;
    std::cout << "int size = " << sizeof(int) << std::endl;
    std::cout << "int* size = " << sizeof(int*) << std::endl;

    return 0;
}
```

When I run this program on my computer, I get the following output:

```

x = 0
&x = 0x7ffeefbfff56c
xPtr = 0x7ffeefbfff56c
&xPtr = 0x7ffeefbfff560
int size = 4
int* size = 8
Program ended with exit code: 0

```

We declare and initialize two variables in the code. The first should look familiar to you by now. We simply declare a variable called `x` to be of type `int` and set its value to zero (`int x = 0`). The second line is new. Here, we declare a pointer variable of type `int*`. The asterisk indicates that the variable is a pointer. In fact, it is a variable that can hold the memory address of an `int`. We also initialize the pointer variable to be equal to the memory address of `x`.

Confusingly, different programmers will put the asterisk, which indicates that the variable will hold a memory address, in different places. Compilers will accept the following as equivalent:

```

int* xPtr = &x;
int * xPtr = &x;
int *xPtr = &x;

```

It seems the asterisk can attach itself to the variable type (here `int`), to the variable name, or even stay in between the two like a baseball player caught in a pickle. I follow the convention of having the asterisk cling to the variable type.

Two of the lines display the same memory address:

```

&x = 0x7ffeefbfff56c
xPtr = 0x7ffeefbfff56c

```

This makes perfect sense because we set the value of `int*` to be the memory address of `x`. The next line simply shows that the variable named `xPtr` also has a memory address. After all, it is a variable! You will note that the pointer variable takes up eight bytes of memory.

What if, for some reason, we wanted to remember the memory address of the variable `xPtr`? We could do this by declaring another variable to hold the memory address. The big question here is what would the variable type be? Clearly it is a pointer, but it is a pointer to a variable that is itself a pointer. The answer is to append another asterisk to the variable type, resulting in:

```

int** anotherDamnPointer = &xPtr;

```

The variable, `anotherDamnPointer`, can also hold a memory address, but only for variables of type `int*`. You should feel confident enough to modify the program to see that a variable of type `int**` also takes up eight bytes of memory. All pointer variables take up the same amount of memory (eight bytes) regardless of the type of variable it holds the memory address of.

4.5 Dereferencing pointers

I mentioned that if you know the address of a variable, that you can manipulate it. You can do this by ‘dereferencing’ the pointer. Here’s an example using a re-written `main` function:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x = 0;
    int* xPtr = &x;
    std::cout << "x = " << x << std::endl;

    *xPtr = 3;
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

When I run this program, I get the following output:

```
x = 0
x = 3
Program ended with exit code: 0
```

Note that I didn't change the value of `x` directly by simply typing `x = 3`. Rather, I changed its value indirectly using `*xPtr = 3`. Essentially, the program changes the value at the memory address stored in the pointer variable, `xPtr`.

4.6 Arrays

Often, a problem can best be solved by using a vector of variables. As an example, in phylogenetics, we use vectors of `double` variables to hold conditional probabilities at different points on a tree.

Vectors, called 'arrays' in C++ and other languages, can be declared by appending square brackets with the number of variables in the array to the end of the variable name:

```
int x[10];          // declares an array of 10 ints
int* xPtr[10];     // declares an array of 10 int pointers
double y[1000];    // declares an array of 1000 double variables
```

The value for each element can be accessed, again, using the square brackets. Implement the following code in your program:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x[100];

    for (int i=0; i<100; i++)
        x[i] = i;

    for (int i=0; i<100; i++)
        std::cout << "x[" << i <<"] = " << x[i] << std::endl;

    return 0;
}
```

This little program does three things. First, we declare a vector of 100 `int` variables, called `x`. Second, we initialize each of the 100 variables to be its position in the vector. We do this using a loop. This is the second time you have seen a loop in this book. Here, we use the `for` loop. (There are also `do` and `while` loops, but `for` loops are probably the most frequently used in C++.) The `for` loop has three conditions, separated by semicolons. The first, `int i=0`, declares a counter variable named `i` and initializes its value to zero. The second, `i<100`, indicates the condition for the variable `i`. As long as the condition is met, we will continue through the loop. Here, we will continue through the loop while the variable `i` is less than 100. The third part of the `for` loop increments the value of `i` each time we pass through the loop. The value is incremented at the end of the loop. The `i++` is short-hand for the statement `i = i + 1`. So, the values of `i` that will be visited in the `for` loop are 0, 1, 2, ..., 99. The `for` loop could have been replaced by the following 100 lines of code:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x[100];

    x[0] = 0;
    x[1] = 1;
    x[2] = 2;
    x[3] = 3;
    x[4] = 4;
```

```
// 90 lines of similar code!
x[95] = 95;
x[96] = 96;
x[97] = 97;
x[98] = 98;
x[99] = 99;

for (int i=0; i<100; i++)
    std::cout << "x[" << i <<"] = " << x[i] << std::endl;

return 0;
}
```

Mercifully, I did not write out the full 100 lines of code that would be necessary to accomplish what the two lines of code in the `for` loop accomplished.

After initialization, the third and final part of the little C++ program prints out each value of the vector, `x`. It does this using another `for` loop.

The values of the array are adjacent to one another in computer memory. You can see this if you modify the line in the code in which the values are printed to read:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x[100];

    for (int i=0; i<100; i++)
        x[i] = i;

    for (int i=0; i<100; i++)
        std::cout << "x[" << i <<"] = " << x[i] << " (" << &x[i] << ")" << std::endl;

    return 0;
}
```

When I run the program, the first ten lines of output read:

```
x[0] = 0 (0x7ffeefbff420)
x[1] = 1 (0x7ffeefbff424)
x[2] = 2 (0x7ffeefbff428)
```

```
x[3] = 3 (0x7ffeefbff42c)
x[4] = 4 (0x7ffeefbff430)
x[5] = 5 (0x7ffeefbff434)
x[6] = 6 (0x7ffeefbff438)
x[7] = 7 (0x7ffeefbff43c)
x[8] = 8 (0x7ffeefbff440)
x[9] = 9 (0x7ffeefbff444)
x[10] = 10 (0x7ffeefbff448)
```

Note that each `int` variable in the array is four bytes away from the previous variable.

The above example assumes that you know the size of the vector before compiling the program. Here, for whatever reason, we knew we would need 100 integer values in the array. However, what would you do if you didn't know the size of the array before compiling the program? Perhaps for some input to the program, you will need 50 integer values in the vector but for other input you will need 1000. One possible solution is to simply declare the variable to be of a sufficient size. For example, you could write,

```
int x[1000];
```

if you knew that the maximum size of the vector would be 1000 `int` variables. This is an inelegant and inefficient solution to the problem. What if we only need 50 variables? In this case, we would never use 950 of the variables that were set aside when the program is run. The problem is even messier if we cannot predict how many variables we will need when the program is executed, which is the usual case in evolutionary biology in which the size of the problems to be analyzed can vary dramatically.

Fortunately, there is a simple solution. We dynamically allocate the memory that we will need. Rewrite the main function to read:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int numInts = 0;
    std::cout << "How many ints: ";
    std::cin >> numInts;

    int* x = NULL;
    if (numInts > 0)
        x = new int[numInts];
    else
```

```
    {
        std::cout << "Too few ints!" << std::endl;
        exit(1);
    }

    for (int i=0; i<numInts; i++)
        x[i] = i;

    for (int i=0; i<numInts; i++)
        std::cout << "x[" << i <<"] = " << x[i] << " (" << &x[i] << ")" << std::endl;

    delete [] x;

    return 0;
}
```

When I run this program, I am prompted to enter a number. I entered '5' and got the following output:

```
How many ints: 5
x[0] = 0 (0x10292c5a0)
x[1] = 1 (0x10292c5a4)
x[2] = 2 (0x10292c5a8)
x[3] = 3 (0x10292c5ac)
x[4] = 4 (0x10292c5b0)
Program ended with exit code: 0
```

The simple program does several things. First, it declares a variable called `numInts` and initializes its value to zero. Second, the program prints out `How many ints:` to the standard console. At this point, nothing happens until the user enters (hopefully) a number. There is no checking that the user actually enters a number. Ideally, we would check that the entry made by the user is valid. In any case, we proceed to declare another variable, this time an `int*` (`int` pointer) variable and initialize its value at the same time we declare it. We initialize its value to be `NULL`. (In C++, we use `NULL` if you want to initialize a pointer variable so that it does not contain an address.) If the variable `numInts` is greater than zero, then we go ahead and allocate the desired number of integer variables using the `new` function. If `numInts` is not greater than 0, then we print an error and bail out of the program using the `exit` function. Finally, the program initialized the integer variables in the array and prints the value and memory address of each. At the end of the program, we free the memory that was allocated. We do this with the `delete []` function. Every time we dynamically allocate memory using the `new` function, we should remember to free that memory when it is no longer needed using the `delete` function.

For fun, I ran the program again, this time entering '1000000' when prompted to enter how many `int` variables I wanted. The output, minus 990,000 lines, was as follows:

```
How many ints: 1000000
x[0] = 0 (0x103400000)
x[1] = 1 (0x103400004)
x[2] = 2 (0x103400008)
x[3] = 3 (0x10340000c)
x[4] = 4 (0x103400010)
x[5] = 5 (0x103400014)

[many lines not shown]

x[999995] = 999995 (0x1037d08ec)
x[999996] = 999996 (0x1037d08f0)
x[999997] = 999997 (0x1037d08f4)
x[999998] = 999998 (0x1037d08f8)
x[999999] = 999999 (0x1037d08fc)
Program ended with exit code: 0
```


Chapter 5

Conjunction Function

5.1 Making your own functions

Computer programs use functions all the time. Functions promote code reuse (why type the same code repeatedly?) and help clarify the logic of the code. So far, we have seen only one function, called `main`. Remember, every C++ program must have a function called `main`. In this chapter, we will explore functions.

Modify your program to read:

```
#include <iostream>

// prototype the new function
int sumTwoInts(int x, int y);

int main(int argc, char* argv[]) {

    int var1 = 3;
    int var2 = 5;
    int sum = sumTwoInts(var1, var2);
    std::cout << "sum = " << sum << std::endl;

    return 0;
}

int sumTwoInts(int x, int y) {

    return x + y;
}
```

```
}
```

The output should look like this:

```
sum = 8
Program ended with exit code: 0
```

This program defines a new function, called `sumTwoInts`. The function is implemented after the `main` function, though it could have been implemented before it (the order doesn't matter). The function `sumTwoInts` is quite simple; it just returns the sum of the two `int` variables that are passed to it. The only mysterious portion of this new program is the statement before the `main` function:

```
// prototype the new function
int sumTwoInts(int x, int y);
```

This statement provides the compiler a heads up that a user-defined function, called `sumTwoInts` will be used in this file. Besides providing the compiler the name of the function, it informs the compiler that this function will take two `int` variables and return an `int`.

Making new functions is quite easy! Now, let's explore some important details of functions.

5.2 You can pass variables to a function by value or by reference

One of the more confusing aspects of functions is what happens to variables when they are passed into the function. To explore functions, try the following:

```
#include <iostream>

// prototype the new function
void changeMyVariable(int x);

int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
```

```

        std::cout << "var(after) = " << var << std::endl;

        return 0;
    }

    void changeMyVariable(int x) {

        x = 3;
    }

```

Examine this program closely. We define a new function called `changeMyVariable`. This function takes as an argument a `int` variable and returns nothing (hence the `void` as the return statement). The implementation of the `changeMyVariable` function is straight forward. We change the value of `x` to be equal to 3.

The program prints the value of the variable declared in `main` twice: once before and once after the `changeMyVariable` function is called. What do you predict the value of `var` will be after the `changeMyVariable` function is called?

Many new programmers would predict that the value of `var` will be 3 after the `changeMyVariable` function is called. After all, the variable is passed into the function and its value changed there, right? Wrong! Run the program and see for yourself. Your output should look like:

```

var(before) = 0
var(after) = 0
Program ended with exit code: 0

```

Clearly, the value of `var` did not change after the `changeMyVariable` function was called.

The value of `var` does not change after the `changeMyVariable` function is called for the simple reason that the value is copied to a new location when it is passed into the `changeMyVariable` function. This is called passing by value. It is more clear when we print the memory address of `var` in the `main` function and `x` in the `changeMyVariable` function:

```

#include <iostream>

// prototype the new function
void changeMyVariable(int x);

```

```
int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var's address: " << &var << std::endl;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int x) {

    std::cout << "x's address: " << &x << std::endl;
    x = 3;
}
```

When I run the program, I got the following output:

```
var's address: 0x7ffeefbff67c
var(before) = 0
x's address: 0x7ffeefbff63c
var(after) = 0
Program ended with exit code: 0
```

You can see from the output that `var` and `x` are different `int` variables residing at different memory addresses. When the function `changeMyVariable` changes the value of `x`, it is not changing the value of `var`. Rather, `x`'s initial value is whatever `var`'s value was. Check this by changing the code to read:

```
#include <iostream>

// prototype the new function
void changeMyVariable(int x);
```

```

int main(int argc, char* argv[]) {

    int var = 11;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int x) {

    std::cout << "x(before): " << x << std::endl;
    x = 3;
    std::cout << "x(after): " << x << std::endl;
}

```

The output from this modified program clarifies what is going on:

```

var(before) = 11
x(before): 11
x(after): 3
var(after) = 11
Program ended with exit code: 0

```

The variable `var` is initialized to 11. This value is passed to the function `changeMyVariable` where the new variable, `x` within the scope of that function, is initialized to 11. It is then changed to the value 3 and the program exits.

It seems that the `changeMyVariable` function is not acting as we intended. The idea was that we would pass an integer variable into that function and it would change its value to 3. Instead, we get a copy of the variable we want to change and set its value to 3.

How do we change the value of the variable `var` that was instantiated in the `main` function? There are two ways to do this.

First, we could pass in the address of `var` and have the `changeMyVariable` change the value indirectly, by dereferencing the pointer and changing the value at `var`'s address. Let's rewrite the program yet again:

```
#include <iostream>

// prototype the new function
void changeMyVariable(int* x);

int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(&var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int* x) {

    (*x) = 3;
}
```

The output for this program looks like:

```
var(before) = 0
var(after) = 3
Program ended with exit code: 0
```

Finally, we have managed to change the value of `var`, that was declared in `main`, in the `changeMyVariable` function. However, we had to do this indirectly. Keep in mind that the variable that is passed into the `changeMyVariable` function is a new variable, in this case an `int*` pointer variable. Its value is initialized when we pass the memory address of `var` to the `changeMyVariable` function when we call it in `main`.

The second way to change the variable is to pass it by reference. We change the program to read:

```
#include <iostream>
```



```
// prototype the new function
void changeMyVariable(int& x);

int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var's address: " << &var << std::endl;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int& x) {

    std::cout << "x's address: " << &x << std::endl;
    x = 3;
}
```

The program gives the following output:

```
var's address: 0x7ffeefbff67c
var(before) = 0
x's address: 0x7ffeefbff67c
var(after) = 3
Program ended with exit code: 0
```

Note that `var` (in `main`) and `x` (in `changeMyVariable`) are the same variable. After all, they both are `int` variables at the same memory address. Passing by reference is an alternative method for changing the value of a variable in another function.

5.3 Variables have a scope

Variables have a scope, which is the portion of the code in which they can be accessed, changed, *etc.*. The scope of a variable is defined between `{` and `}`. Rewrite your code as follows:

```
#include <iostream>

int main(int argc, char* argv[]) {

    {
        int x = 0;
        std::cout << "x = " << x << std::endl;
    }

    return 0;
}
```

This program should run and give the output `x = 0`. Now, let's make a very minor change to the program by moving one of the curly brackets:

```
#include <iostream>

int main(int argc, char* argv[]) {

    {
        int x = 0;
    }
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

When I do this, XCode gives me one warning and one error. The error reads, “Use of undeclared identifier 'x'.” Normally, the scope of `x` would be the entire `main` function (or, at least, the portion of the `main` function after it was declared). However, in this little program, I restricted `x`'s scope to be between another set of curly braces. When I attempted to use `x` outside of its scope, in the `std::cout` statement, the compiler gives me an emphatic “No.”

We could have declared the variable outside of the function:

```
#include <iostream>
```

```
int x; // global x

int main(int argc, char* argv[]) {

    {
    x = 0;
    }
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

Now, the variable `x` is available to any function in that file. It is said to be a ‘global variable.’ Declaring variables to be global in scope is generally not a great idea.

Chapter 6

Classes

6.1 Into the deep end of the pool

In this chapter, we are going to make a random number generator. As it turns out, computers can't generate randomness on their own. All they can do is logical operations and math. So, how do we make randomness from one of the most deterministic tools made by man? The answer is that we don't. Rather, we can generate sequences of numbers that in many respects can behave randomly. To be more correct, I will modify my original statement: In this chapter, we are going to make a pseudorandom number generator.

The sequence of pseudorandom numbers is completely determined by the initial value, called the seed. If we initialize the pseudorandom number generator with the same seed, we will get the same sequence out. If the sequence of random numbers revisits the value for the seed, it will repeat. The period of the pseudorandom number is the length of this repeat. Good pseudorandom number generators will have a very long period. Moreover, the values will be difficult to distinguish from truly random numbers. For example, they shouldn't be correlated with previous values.

You should be wary of using pseudorandom number generators. There is an entire sub-discipline in computer science dedicated to developing better pseudorandom numbers.

We will implement a simple linear congruential generator described by Park and Miller (1988). It is not a state-of-the-art pseudorandom number generator, but it will give you an idea of how apparent randomness can be generated on a computer. It should be good enough for the applications in this book.

A linear congruential generator starts with an initial value of an `int` variable called the seed. The seed value is changed sequentially using the following equation:

$$s_{i+1} = (a \times s_i + c) \pmod m$$

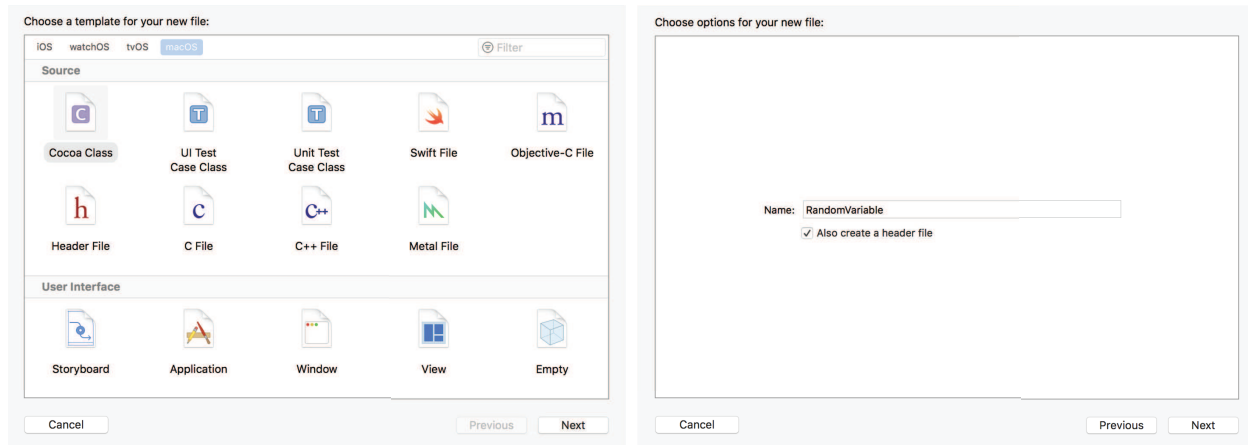
where s_i is the i th value of the seed, a is the multiplier, c is the increment, and m is the modulus. Good pseudorandom number generators that use the linear congruential generator choose a , c , and m wisely. The seed value for a good choice of a , c , and m will bounce between 1 and the maximum value possible in a way that has the appearance of randomness.

6.2 Making your first class

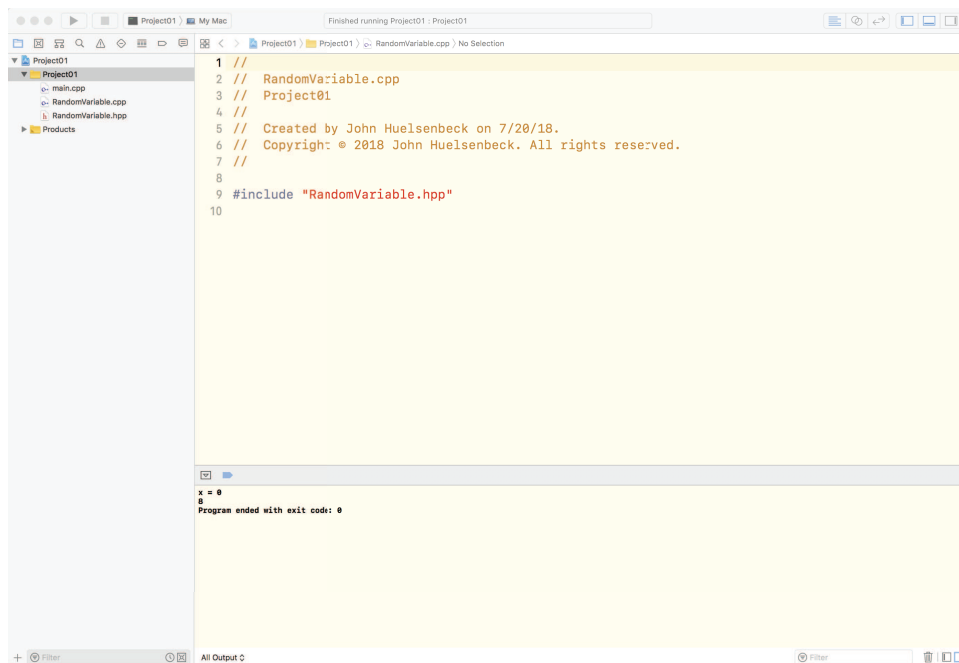
You are now going to make your first C++ class. We will discuss what a class is as we develop it.

Make a new project naming it `Project02`. You can follow the instructions in the second chapter of this book to do this. At the end, you should have a single file, `main.cpp`, that has the `main` function.

Once you have made the new project, you are ready to add a new class. If you are using XCode, select `File > New > File` from the `File` menu item. This will lead to the following windows:



Choose `C++ File` and click on `Next`. In the following window, name the new class `RandomVariable` and click `Next`. This leads to one final window. Simply click `Create` to make your new class files. In the end, your project should look something like this (if you are using XCode):



You created two files, `RandomVariable.hpp` and `RandomVariable.cpp`. You can see them both

listed with the `main.cpp` file. When you compile the program, all the files are compiled to form the executable.

Select the file `RandomVariable.hpp`. Add text to that file so it looks like the following:

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp

class RandomVariable {

    double  uniformRv(void);
    int     seed;
};

#endif
```

Next, select the `RandomVariable.cpp` file. Add text to this file, too:

```
#include "RandomVariable.hpp"

double RandomVariable::uniformRv(void) {

    return 0.0;
}
```

Congratulations! You just made your first class. Eventually, it will be a very cool class, able to generate pseudorandom numbers. We will build to that point, slowly.

What have we done, so far? We defined a class called `RandomVariable`. This class has one `int` variable called `seed` and one function called `uniformRv` associated with it.

6.3 Why do we split the class into .hpp and .cpp files?

One of the more confusing aspects of programming is the fact that a typical program is not implemented in a single file, but rather in many files. As mentioned earlier, the source code for a program like `RevBayes` is distributed among several thousand files! Enjoy your good fortune; so far, this program has only three (`main.cpp`, `RandomVariable.hpp`, and `RandomVariable.cpp`).

The class definition was split into two files. Generally speaking, classes are split into two. The `RandomVariable.hpp` file is called the header file. It contains the outline of the class including the variables that are associated with each instance of the class and the functions that are associated with the class. Again, in this class, we only have one variable (`int seed`) and one function (`uniformRv`). The class definition is simple:

```
class RandomVariable {

};
```

We include the variables and functions that will be associated with each instance of this class between the curly brackets. You will also note that there are several compiler directives associated with the file:

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp

#endif
```

The class definition occurs between the compiler directives. The compiler directives ensure that the class is only defined once. Remember, our source code might be distributed among many files. Any file that uses the capabilities of our `RandomVariable` class will include the file at the top, using the include directive:

```
#include "RandomVariable.hpp"
```

Even though we may include the file in dozens (or more) files, we only want the class to be defined once; hence, the compiler directives that guard against this. Alternatively, we could guard against multiple inclusion using the `once` guard:

```
#pragma once

class RandomVariable {

    double  uniformRv(void);
    int     seed;

};
```

Which you use is a matter of personal preference.

If the header file outlines the data and functions associated with each instance of our `RandomVariable` class, the implementation file (`RandomVariable.cpp`) represents the guts of the class. It's in the implementation file that we actually implement the functions. In our implementation file, we only implement the `uniformRv` function:

```
double RandomVariable::uniformRv(void) {

    return 0.0;

}
```


Eventually, this function will generate a pseudorandom number between 0 and 1. For now, it simply returns the value 0. Note that we include the header file that contains the class definition at the head (or top) of this file.

One last thing about the implementation file. You will have noted that the implementation of the `uniformRv` function had `RandomVariable::` before the function name. This is C++'s way of indicating that this function definition is part of the `RandomVariable` class.

Why do we spread the implementation of the class across two files? One idea prevalent in object oriented programming is that the implementation of a class should be hidden from those who use the class. Imagine I wrote a class with functionality that you wanted in your program. You could include the two files I wrote in your project and, voilà, your program now has the capabilities from my class. In order to use that functionality in some file of your program, you would include the header file from my class. The header file of my class should contain all of the information you need to use the functionality of the class. By examining the functions that are available, for example, you would see how to 'interface' with my code. (Hopefully, I would also include useful comments to guide you.) Although the ideal of separating the interface and the implementation is usually not purely implemented in any program that I am aware of, it's a nice idea.

6.4 Instantiating a class

Rewrite the `main.cpp` file to read:

```
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv;
    double u = rv.uniformRv();
    std::cout << "u(0,1) = " << u << std::endl;

    return 0;
}
```

If you do this, you should come across an error. In XCode, the error reads, “‘`uniformRv`’ is a private member of ‘`RandomVariable`’.” What is going on? Help!

In C++, variables and functions in a class can be private or public. By default, variables and functions are private. Because we did not indicate otherwise, the compiler assumed that both `seed` and `uniformRv` were both private. To fix the problem, we need to go back to our class definition in `RandomVariable.hpp` and insert two lines:

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp

class RandomVariable {

public:
    double  uniformRv(void);

protected:
    int     seed;
};

#endif
```

Public variables and functions are available to other parts of the code to use. The private or protected elements of the class, however, are only available to other members of the class. We want other parts of the code to have the ability to get a uniform random variable from this class. Therefore, we make the `uniformRv` function public. On the other hand, the variable `seed` is part of the mechanism that will generate the pseudorandom numbers. We don't want other parts of the program to have the ability to even touch this mechanism. Hence, we hide this variable away. This sort of compartmentalization is a useful feature of C++. In general, you should attempt to keep functions and variables `private` or `protected`.

With the aside on public versus private/protected completed, let's revisit our `main` function. The first line of that function,

```
RandomVariable rv;
```

declares an instance of the `RandomVariable` class called `rv`. This line of code is analogous to others you have seen, such as `int x`, which makes an instance of an integer variable called `x`. The terminology we use when declaring instances of classes is a bit different than when we declare a variable to be of type `int`. We would say that we are instantiating the `RandomVariable` class or that we made an instance of the `RandomVariable` class. Just like other variables, our new variable `rv` has a scope (the function `main`, in this case). You might have heard that C++ is an 'object-oriented' programming language. Objects are just instances of classes. If it is important to you, you can tell your loved ones that you just made your first object.

If you run the program, the output should look like

```
u(0,1) = 0
Program ended with exit code: 0
```

The code works. Often, I build up portions of the code just as we are doing here: incrementally.

When you make an instance of a class, as we have in our `main` function, several things happened behind the scenes that you are unaware of. First of all, when the variable was created, a function was called that you didn't even know about! This function is called the default constructor function. Again, this function was created even though we never wrote a line of code. Interestingly, we can make our own default constructor that will be called instead. Add to the implementation file, `RandomVariable.cpp`, several new functions, so that the file now looks like:

```
#include <ctime>
#include "RandomVariable.hpp"

RandomVariable::RandomVariable(void) {
    seed = (int)time(NULL);
}

RandomVariable::RandomVariable(int x) {
    seed = x;
}

double RandomVariable::uniformRv(void) {
    return 0.0;
}

double RandomVariable::uniformRv(double lower, double upper) {
    return 0.0;
}
```

Modify the header file to reflect the changes you made in the implementation file:

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp
```

```
class RandomVariable {  
  
    public:  
        RandomVariable(void);  
        RandomVariable(int x);  
        double uniformRv(void);  
        double uniformRv(double lower, double upper);  
  
    protected:  
        int seed;  
};  
  
#endif
```

You can test that you did everything correctly by attempting to run the program.

The default constructor has the same name as the class and does not return a variable (in fact, it doesn't even have the keyword `void` to indicate this, like normal functions that don't return a value). If we wanted to replicate, exactly, the default constructor that was created by the compiler, we would simply include the following code:

```
RandomVariable::RandomVariable(void) {  
  
}
```

Our implementation of the default constructor, however,

```
RandomVariable::RandomVariable(void) {  
  
    seed = (int)time(NULL);  
}
```

sets the `seed` variable to be equal to the current time, obtained using the `time` function. The variable that is returned by the `time` function is not an `int` variable, so we cannot directly equate it to `seed`. The `time` function returns a variable of type `time_t`. However, we can force the `time_t` variable to act as an `int` using the casting argument, which was the `(int)` before the `time(NULL)`. Casting can be dangerous. You can't always sensibly cast one variable type into another. In this case, however, you can sensibly cast a `time_t` variable into an `int` variable.

Add a line of code to the default constructor, so that it now reads:

```
RandomVariable::RandomVariable(void) {  
  
    seed = (int)time(NULL);  
    std::cout << "Default constructor. The seed equals " << seed << std::endl;  
}
```

In order to get the code to compile, you will need to add `iostream` using the `include` compiler directive to the `RandomVariable.cpp` file. When I ran the program, I obtained the following output:

```
Default constructor. The seed equals 1532124958  
u(0,1) = 0  
Program ended with exit code: 0
```

The important point is this: We never explicitly called the default constructor. It was called for us when `rv` was created.

Constructors are obvious places to initialize variables. For our `RandomVariable` class, it makes sense to initialize the `seed` variable when an instance of the class is created. Moreover, because we want a different sequence of pseudorandom numbers every time we run the program, we should initialize the `seed` variable to something that changes, like time. Many implementations of pseudorandom number generators use the computer's internal clock to initialize the seed.

What if we want to initialize the `seed` variable to some other value? We have given our program the ability to do just this. In the code, above, we implemented a second constructor! This one,

```
RandomVariable::RandomVariable(int x) {  
  
    seed = x;  
}
```

Takes an `int` argument and sets the `seed` variable to be equal to it. Add a line to this second constructor, so it reads:

```
RandomVariable::RandomVariable(int x) {  
  
    seed = x;  
    std::cout << "Alternate constructor. The seed equals " << seed << std::endl;  
}
```

Now, go to the main function and rewrite it so it reads:

```
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv1;
    RandomVariable rv2(3);
    RandomVariable rv3(1929);
    RandomVariable rv4;

    return 0;
}
```

When I ran the program, I got the following output:

```
Default constructor. The seed equals 1532125424
Alternate constructor. The seed equals 3
Alternate constructor. The seed equals 1929
Default constructor. The seed equals 1532125424
Program ended with exit code: 0
```

We made four different instances of the `RandomVariable` class. This means that we have four instances of seed that were created, one for each. `rv1` and `rv4` were created using the default constructor which sets the seed using the current time. Both of those objects have the same value for the `seed` variable. They will produce the same sequences of pseudorandom numbers. The computer created the four instances of `RandomVariable` so quickly that the time didn't have a chance to turn over, even by one! The other two instances of `RandomVariable`, `rv2` and `rv3` were created using the alternative constructor that sets the seed to some value that is passed in.

One nice feature of C++ is function overloading. Different functions can have the same name, as long as the arguments that are passed to the functions are different. In our `RandomVariable` class, we see function overloading in two places: for the constructors

```
RandomVariable(void);
RandomVariable(int x);
```

and in the functions `uniformRv`

```
double uniformRv(void);
double uniformRv(double lower, double upper);
```

Even though the functions have the same name, the compiler can tell them apart because the arguments that are passed to the functions are different. The compiler can tell which function is being called, implicitly, by examining the list of arguments that are passed to it. We were able to indicate to the compiler which constructor we wanted to be called when the objects were instantiated: `RandomVariable rv` calls the default constructor, which doesn't take any arguments, whereas `RandomVariable rv(3)` calls the alternative constructor which takes a single `int` as an argument.

6.5 Implementing the `uniformRv` function

We will implement the pseudorandom number generator described by Park and Miller (1988). Change the code for the `uniformRv(void)` function in the `RandomVariable.cpp` file to read:

```
double RandomVariable::uniformRv(void) {  
  
    int hi = seed / 127773;  
    int lo = seed % 127773;  
    int test = 16807 * lo - 2836 * hi;  
    if (test > 0)  
        seed = test;  
    else  
        seed = test + 2147483647;  
    return (double)(seed) / (double)2147483647;  
}
```

The code for pseudorandom number generators is always disturbing to look at. The non-randomness of a pseudorandom number generator sort of smacks you in the face when you see the code. You should be feeling a little queasy at this point.

Clearly, the `seed` variable is changed every time the `uniformRv` function is called. The number that is returned is the new value for the seed divided by its maximum possible value. We use casting, again, to force the compiler to treat `seed` as a `double` (real-valued number) when the division occurs in the last line. If we didn't do this, the function would return 0 every time.

We have implemented the `uniformRv(void)` function, but have the `uniformRv(double lower, double upper)` remaining to be implemented. Go to that function, and add the following code:

```
double RandomVariable::uniformRv(double lower, double upper) {  
    return ( lower + uniformRv() * (upper - lower) );  
}
```

This function uses the function that returns a uniformly-distributed random number on the interval (0,1) to generate a random number uniformly-distributed on the interval (Lower, Upper).

Now for some fun! Let's test our random number generator. Modify your main function so it reads:

```
#include <iostream>  
#include "RandomVariable.hpp"  
  
int main(int argc, char* argv[]) {  
  
    RandomVariable rv;  
    for (int i=0; i<100; i++)  
    {  
        double u = rv.uniformRv();  
        std::cout << i << " -- " << u << std::endl;  
    }  
  
    return 0;  
}
```

When you run this program, the output should be 100 pseudorandom numbers, uniformly distributed on the interval (0,1).

Modify the program so that it starts with the same seed every time. Confirm that the sequence of random numbers is the same every time the program is run.

Besides being a nifty example of function overloading, our function `uniformRv(double lower, double upper)` demonstrates a commonly-used method for generating random numbers that are not uniform(0,1) random variables. That function transforms a uniform(0,1) random number into a uniform(lower,upper) random number. We can use the same idea to generate different types of random variables. Here, we will make one more random variable before moving on from our discussion of classes.

The exponential probability distribution accurately models the waiting time until an event occurs when that event occurs at a constant rate, λ . The exponential probability distribution is a continuous distribution with density function, $f(t) = \lambda e^{-\lambda t}$, for $t > 0, \lambda > 0$. How can we generate an exponentially-distributed random number?

The idea is to transform the uniform(0,1) random variable into an exponential(λ) random variable. We know that if we integrate over all possible values of t , that the overall probability is one (the exponential is a probability distribution). So,

$$\int_0^{\infty} \lambda e^{-\lambda x} dx = 1$$

The key to transforming our uniform into an exponential is to generate a uniform(0,1) random variable called u and set the integral equal to that value:

$$\int_0^t \lambda e^{-\lambda x} dx = u$$

We can solve for t as follows:

$$\begin{aligned} \int_0^t \lambda e^{-\lambda x} dx &= u \\ -e^{-\lambda x} \Big|_0^t &= u \\ -e^{-\lambda t} - -e^0 &= u \\ 1 - e^{-\lambda t} &= u \\ e^{-\lambda t} &= 1 - u \\ -\lambda t &= \ln(1 - u) \\ t &= -\frac{1}{\lambda} \ln(1 - u) \end{aligned}$$

Here, t is an exponential(λ) random variable. It was obtained by transforming our uniform(0,1) random variable, u . Equivalently, we could use the equation $t = -\frac{1}{\lambda} \ln(u)$.

Now that we have the math out of the way, let's implement our exponential random number generator. Add to the `RandomVariable.cpp` file the following function:

```
double RandomVariable::exponentialRv(double lambda) {
    return -log(uniformRv()) / lambda;
}
```

You will have to do two things to get this to compile. First, include the function profile in the header file for the class. Second, you will need to include the `cmath` header file in the implementation file. (The natural log function, `log`, is implemented in `cmath`.)

Once you have successfully implemented the `exponentialRv` function, play around with it. Generate a bunch of exponentially-distributed random numbers. What is the mean of the numbers you generated? They should be near to the expectation for the exponential, $E(X) = 1/\lambda$. Confirm this.

Chapter 7

Markov chain Monte Carlo

7.1 Some theory

Markov chain Monte Carlo (MCMC) is a numerical method for approximating high-dimensional integrals and/or summations. The method was first described by Metropolis et al. (1953) and later modified by (Hastings, 1970)¹. The algorithm that we will implement in this chapter is often referred to as the Metropolis-Hastings algorithm.

Use of the Metropolis-Hastings algorithm was mostly restricted to the field of statistical physics, until it was discovered by statisticians in the mid 1980s (Geman and Geman, 1984). Statisticians, as a group, were largely unaware of the numerical method until Gelfand and Smith (1990) provided a description of the method as a way to approximate intractable probability densities. It is fair to say that MCMC has transformed the field of statistics, and in particular, Bayesian statistics.

Bayesian statisticians are interested in the posterior probability distribution of a parameter, θ , which can be calculated using Bayes's theorem as

$$\mathbb{P}(\theta|D) = \frac{\mathbb{P}(D|\theta)\mathbb{P}(\theta)}{\mathbb{P}(D)}$$

Here, $\mathbb{P}(\theta|D)$ is the posterior probability distribution of the parameter. The posterior probability is a conditional probability: the probability of the parameter conditioned on the observations, D . The other factors in the equation include the likelihood $[\mathbb{P}(D|\theta)]$, prior probability distribution of the parameter $[\mathbb{P}(\theta)]$, and marginal likelihood $[\mathbb{P}(D)]$.

How does Bayesian analysis work in practice? Consider an experiment in which a coin is repeatedly tossed with the objective to estimate the probability that heads appears on a single toss of the coin, a parameter we call θ . We observe x heads on n tosses of the coin. In a Bayesian analysis, the objective is to calculate the posterior probability of the parameter, which for coin tossing is

$$f(\theta|x) = \frac{f(x|\theta)f(\theta)}{\int_0^1 f(x|\theta)f(\theta) d\theta}$$

¹Look at the full list of authors for the Metropolis et al. (1953) paper. If you know your U.S. history, you will recognize E. Teller as the father of the American hydrogen bomb.

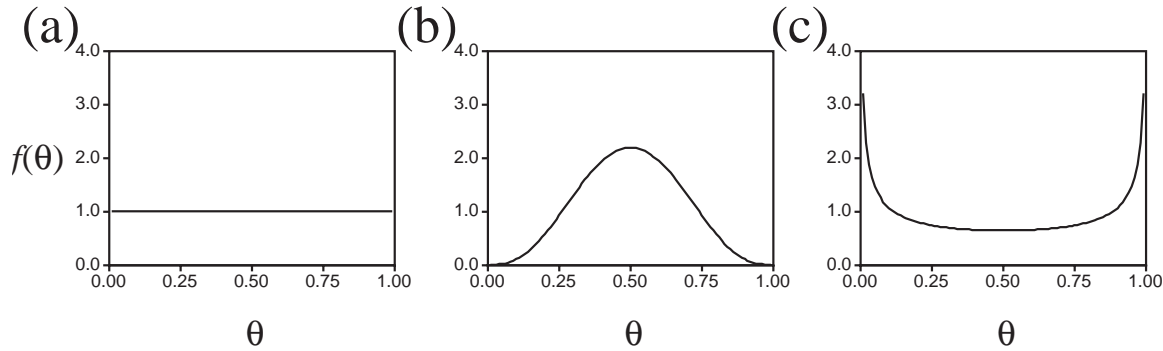


Figure 7.1: The Beta distribution can take a variety of shapes depending on the values of the parameters α and β . Here, (a) $\alpha = \beta = 1$, (b) $\alpha = \beta = 4$, and (c) $\alpha = \beta = 1/2$.

The likelihood, $f(x|\theta)$, is given by the binomial probability distribution,

$$f(x|\theta) = \binom{n}{x} \theta^x (1-\theta)^{n-x}$$

where the binomial coefficient is $\binom{n}{x} = \frac{n!}{x!(n-x)!}$. In addition to the likelihood function, however, we now must also specify a prior probability distribution for the parameter, $f(\theta)$. This prior distribution should describe the investigator's beliefs about the hypothesis before the experiment was performed. The problem, of course, is that different people may have different prior beliefs. In this case, it makes sense to use a prior distribution that is flexible, allowing different people to specify different prior probability distributions and also allowing for an easy investigation of the sensitivity of the results to the prior assumptions. A Beta distribution is often used as a prior probability distribution for the binomial parameter. The Beta distribution has two parameters, α and β . Depending upon the specific values chosen for α and β , one can generate a large number of different prior probability distributions for the parameter θ . Figure 7.1 shows several possible prior distributions for coin tossing. Figure 7.1a shows a uniform prior distribution for the probability of heads appearing on a single toss of a coin. In effect, a person who adopts this prior distribution is claiming total ignorance of the dynamics of coin tossing. Figure 7.1b shows a prior distribution for a person who has some experience tossing coins; anyone who has tossed a coin realizes that it is impossible to predict which side will face up when tossed, but that heads appears about as frequently as tails, suggesting more prior weight on values around $\theta = 0.5$ than on values near $\theta = 0$ or $\theta = 1$. Lastly, Figure 7.1c shows a prior distribution for a person who suspects he is being tricked. Perhaps the coin that is being tossed is from a friend with a long history of practical jokes, or perhaps this friend has tricked the investigator with a two-headed coin in the past. Figure 7.1c, then, might represent the 'trick-coin' prior distribution.

Besides being flexible, the Beta prior probability distribution has one other admirable property; when combined with a binomial likelihood, the posterior distribution also has a Beta probability distribution (but with the parameters changed). Prior distributions that have this property—that is, the posterior probability distribution has the same functional form as the prior distribution—are called conjugate priors in the Bayesian literature. The Bayesian treatment of the coin tossing

experiment can be summarized as follows:

Prior [$f(\theta)$, Beta]	Likelihood [$f(x \theta)$, Binomial]	Posterior [$f(\theta x)$, Beta]
$\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$	$\binom{n}{x}\theta^x(1-\theta)^{n-x}$	$\frac{\Gamma(\alpha+\beta+n)}{\Gamma(\alpha+x)\Gamma(\beta+n-x)}\theta^{\alpha+x-1}(1-\theta)^{\beta+n-x-1}$

[The gamma function, not to be confused with the gamma probability distribution which will be discussed in later chapters, is defined as $\Gamma(y) = \int_0^\infty u^{y-1}e^{-u} du$. $\Gamma(n) = (n-1)!$ for integer $n = 1, 2, 3, \dots$ and $\Gamma(\frac{1}{2}) = \pi$.] We started with a Beta prior distribution with parameters α and β . We used a binomial likelihood, and after a modest amount of calculus we calculated the posterior probability distribution, which is also a Beta distribution but with parameters $\alpha + x$ and $\beta + n - x$.

Figure 7.2 shows the relationship between the prior probability distribution, likelihood, and posterior probability distribution of θ for two different cases, differing only in the number of coin tosses. The posterior probability of a parameter is a compromise between the prior probability and the likelihood. When the number of observations is small, as is the case for Figure 7.2a, the posterior probability distribution is similar to the prior distribution. However, when the number of observations is large, as is the case for Figure 7.2b, the posterior probability distribution is dominated by the likelihood.

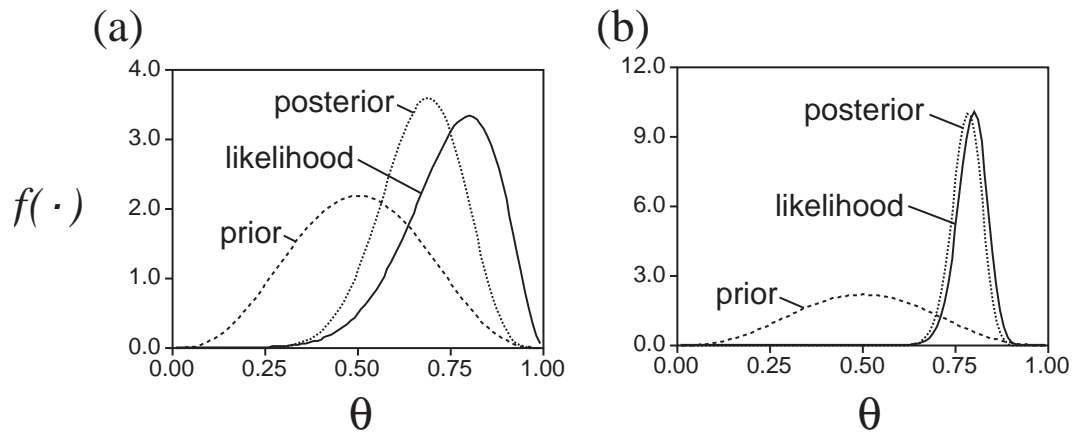


Figure 7.2: As more data are collected for the coin-tossing example, the investigator's prior opinions play a smaller role in the conclusions. (a) The prior distribution, likelihood function, and posterior probability density when $\alpha = \beta = 4$, $n = 10$ and $x = 8$. (b) The prior distribution, likelihood function, and posterior probability density when $\alpha = \beta = 4$, $n = 100$ and $x = 80$

Bibliography

- Edwards, A. W. F. 2009. Statistical methods for evolutionary trees. *Genetics* 183:5–12.
- Fisher, R. A. 1950. Gene frequencies in a cline determined by selection and diffusion. *Biometrics* 6:353–361.
- Gelfand, A. E. and A. F. M. Smith. 1990. Sampling-based approaches to calculating marginal densities. *J. Amer. Statist. Assoc.* 85:398–409.
- Geman, S. and D. Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6:721–741.
- Hastings, W. K. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57:97–109.
- Höhna, S., M. J. Landis, T. A. Heath, B. Boussau, N. Lartillot, B. R. Moore, J. P. Huelsenbeck, and F. Ronquist. 2016. Revbayes: Bayesian phylogenetic inference using graphical models and an interactive model-specification language. *Systematic Biology* 65:726–736.
- Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 21:1087–1092.
- Park, S. K. and K. W. Miller. 1988. Random number generators: good ones are hard to find. *Communications of the ACM* 31:1192–1201.