C++ for Biologists: Evolutionary Models

by Adam G. Jones
C++ for Biologists:

Evolutionary Models

Adam G. Jones
CONTENTS

Chapter 1. Getting Started, 1
Chapter 2. A Simulation Template, 4
Chapter 3. Getting Organized with Classes, 13
Chapter 4. Iterating an Equation, 19
Chapter 5. Moving Toward Individual-Based Simulations, 31
Chapter 6. Setting Up the Simulation, 44
Chapter 7. Setting Up the Life Cycle, 54
Chapter 8. More on the Life Cycle, 69
Chapter 9. Mutation and Selection, 85
Chapter 10. Completing the Life Cycle, 99
Chapter 11. Non-random Mating, 113
Chapter 12. Small but Important Modifications, 126
Chapter 13. Validating the Simulation, 164
Chapter 14. Running on Linux with Shell Scripting, 195
Chapter 15. Taking Mate Choice for a Spin, 215
Appendix 1. Random Number Generator Code, 226
Appendix 2. Setting Up a Linux Virtual Machine, 229
Appendix 3. Useful C++ Commands, 236
Chapter 1. Getting Started

This book is aimed primarily at scientists, especially graduate students, who would like to learn to use computer programming in their research but have essentially no programming skills. The goal is to provide the basics necessary to develop custom programs for modeling in scientific disciplines, especially in evolutionary biology. This book will not dwell too much on the most sophisticated programming techniques. Rather the goal is to get from point A (no program) to point B (a program that performs the desired function) in the shortest amount of time. This book is focused on the Windows operating system, especially Windows 10, but will also show you how to compile code on a Linux operating system. The Mac operating system is similar in many ways to a Linux operating system, so the techniques in this book can also be used to develop programs for Macs.

Choosing a Programming Language

The choice of programming language is an important one. It should not be taken too lightly, but the decision is not one that should be agonized over, either. Some programming languages are better suited to certain tasks than others. However, the trick in learning any programming language is to understand how to structure programs, dealing particularly with the logic and flow of the software under development. Any language requires the same basic set of programming skills, so once the basics are known, the main differences among languages, as far as writing programs, are mostly syntax. Different programming languages also differ from one another in various other ways, some of which are important to a budding programmer.

For individual-based simulations, which are very computationally intensive, the single best programming language is probably C++. This language is object-oriented, like many other languages, and this feature facilitates modularity and straightforward organization of the code. Another great feature of C++ is that it is a compiled language, so it runs very fast, often orders of magnitude faster than R (the statistical programming language), PERL or Python. These latter two languages are often used by biologists, because they are relatively easy to learn (although to be good at them, you still need to have real programming skills). Another important feature of both PERL and Python is that many extensions and libraries already exist to perform a variety of tasks, especially in the biological sciences, and these extensions are easy to include in custom programs. However, PERL and Python are also quite slow, so a programmer needs to weigh development time versus software speed when choosing between a fast-running, “difficult” language such as C++ and a slow-running, easy-to-program language, such as Python.

One huge benefit to learning C++ is that those features that make it difficult to learn also enforce good programming practices. A programmer learning C++ will be forced to think about how the code should be organized and to keep track of how the program uses system resources. The syntax is very specific and rigid, which means coding mistakes will be more obvious than in less rigid languages. An oft-used criticism of C++ is that its syntax is clunky and hard to write, but it is not actually all that different from other languages. Learning C++ might be a bit more difficult than learning Python or PERL, but once you have learned C++, programming in Python and PERL will seem trivial.

Structure of the Book

This book has grown out of my interest in using computer simulations to model evolutionary processes. Over the years, my colleagues and I have encountered many challenges, and this book
CHAPTER 1

describes how we overcame some of these hurdles. The approaches described in this book are by no means the only way to address these issues, as one rule of computer programming is that every problem has many solutions.

In this book, I tackle a variety of issues in evolutionary theory as a starting place to develop simulations of biological processes. Many basic ideas related to microevolutionary mechanisms are treated in one way or another in this document, but I will not attempt to explain the back story of each concept very deeply. For readers unfamiliar with evolutionary biology, a basic evolutionary textbook might be a welcome supplement to this document. While evolutionary simulations are the main focus of this book, C++ programming is powerful and flexible, so the techniques covered in this book can be adapted to address a wide array of questions in bioinformatics and computational biology.

As a companion to this book, it will be helpful to pick up a book on the basics of C++ programming. Many of the topics we will only touch upon briefly are covered much more deeply in programming books. This book will tell you how to get a program to do what you want it to, but some of the reasons for doing things will not be entirely clear without a deeper understanding of how C++ works. In addition, we will not cover debugging at all, even though debugging is often a critical step in the programming endeavor. Good C++ programming books include Sans Teach Yourself C++ in 21 Days by Jesse Liberty and Bradley L. Jones, and Ivor Horton’s Beginning Visual C++.

This book has a companion GitHub page, where you can download the source code for each chapter. The GitHub page is embedded within my University of Idaho Jones Lab GitHub page, and the web address is: https://github.com/JonesLabIdaho/C_for_Biologists. Note that in the web address, “C_for_Biologists” has underscores in the place of spaces. The GitHub page includes source code for each chapter, and the source code for a given chapter is the product that is produced at the completion of the corresponding chapter. In other words, a person wishing to skip to Chapter 13 would want to download the source code for Chapter 12 as a starting place.

A free pdf version of the book, which should allow cutting and pasting of the source code, is available from the Jones Lab website: https://pipefishguysite.wordpress.com/. The pdf version also has the advantage that the code text is in color, corresponding to the color scheme automatically applied by Visual Studio.

The text of this book uses several font conventions that bear mentioning. The main text will be in the font you are reading right now, Times New Roman. However, the proportional nature of this font is not ideal for writing code, because the characters do not line up exactly from one line to the next. Code, then, will be in the font used by Visual Studio, Consolas. The code snippets will be in 9-point font, and a given line may sometimes spill onto two lines in this book. In these cases, the code will work fine with or without the line break when you type it into Visual Studio, because C++ compilers ignore line breaks. If you find these snippets too small to read, download the pdf version of the book or the code from the above-mentioned GitHub page to get a better view. Any code that appears within paragraphs of the main text will also be in boldface Consolas to distinguish it from the rest of the text. Occasional important terms will be in boldface italics when they first appear to make them easy to find in a scan of the page.

The best way to use this book is to start at the beginning and work through each example in order. Ideally, you would type every single line of code yourself rather than cut and paste from the sample code (with the exception of the random-number generator in Appendix I, which you should just cut and paste into a header file – more on that later). By typing the code yourself, you will get used to the syntax and will be forced to look at the details of each line. As you type the code, try to understand it. If you find yourself typing code that you do not understand, reread the portions of the text corresponding to the code.

Another piece of advice is to work with this book for a short period of time every day rather than in huge flurries of activity followed by long breaks. If you set the book down for a prolonged period and come back to it, you will likely be hopelessly lost. Instead, try to budget 20-30 minutes per day and work
through the book in a fairly continuous manner. After you finish this book, you should invest in some other programming books, so that you can always be working on one aspect or another of your development as a programmer. If you take a long break from programming, you will find that you have forgotten a great many things upon your return.

While books are a great resource, be sure to take advantage of the internet as well. If any of the concepts or commands in this book are unclear to you, search for additional examples on the internet. With C++ programming in particular, many examples are available to solve almost any problem. Some of the solutions work better than others, so you will have to learn to separate the wheat from the chaff as you scour the internet for tips. Once you get used to recognizing the difference between good and bad internet programming advice, your search engine will become one of your most powerful problem-solving tools.

Install Visual Studio or Some Other C++ IDE

Probably the most widely used development environment for C++ is Microsoft’s Visual C++. Visual C++ is very complicated but very powerful. Fortunately, we will be able to ignore most of the complexities of Microsoft’s development environment. Visual C++ also has the great feature that an introductory version of Visual Studio, which includes Visual C++, is available for free, and this version is fully functional for our purposes. Our first step, if you are using Windows 10, is to download and install Visual Studio Community 2017. If you are using Windows 7, you should seriously consider upgrading to Windows 10. However, if you want to stick with Windows 7, you can still download and install Visual Studio Express 2013 for Windows Desktop. The installation is easily accomplished with a quick visit to Microsoft’s homepage. Follow the installation instructions and register your copy online. Registration is free, so with very little pain and no money at all you are now ready to get started.

If you are using a Linux or Mac machine, you will want an integrated development environment (IDE) or at least a text editor with some support of computer programming languages. I find that the easiest way to program for Linux is to write the programs in C++ on a Windows computer using Visual Studio, which has a lot of nice features in its editor, and then simply recompile the source files on a Linux machine. We will see how to set up a Linux virtual machine and how to compile C++ source code on it near the end of the book. For Mac, Xcode has a good reputation.

With Visual Studio installed on your computer, you are ready to create your first C++ program, so we will draw Chapter 1 to a close and move on to some real programming in Chapter 2.

Acknowledgements

The programming approaches described in this book emerged from a long-time collaboration with Dr. Steve Arnold, a professor of biology at Oregon State University, and Reinhard Bürger, a professor of mathematics at the University of Vienna. My collaboration with Steve and Reinhard has been one of the great highlights of my academic career. Much of this work was supported by grants from the National Institutes of Health and the National Science Foundation. I also would like to thank two of my former Ph.D. students, Dr. Nick Ratterman and Dr. Sarah Flanagan, who used earlier drafts of this document during the development of their dissertation research and provided feedback on these drafts. This book, as a tutorial, does not describe the science in much detail, so I would also like to thank all of the scientists upon whose work these computational models are based. In subsequent chapters, I will mention a few textbooks and papers where the interested reader can learn more about the development of these topics.

Chapter Summary

In this chapter, we got started by installing Visual Studio, or some other development environment for non-Windows machines. We oriented ourselves to some of the conventions used in the book and its supporting websites, and we committed ourselves to spend 20-30 minutes per day on learning C++.
Chapter 2. A Simulation Template

We are going to start by setting up a basic program template with some of the functions that will be common to almost every program we will want to write. Then it will be a simple matter to use this simple program as a starting place for each of the future programs. This sort of approach can save time, because we can put commonly used parts of the program into files that can easily be included in future projects.

If Visual Studio is not already open, open it now by finding it on your list of programs and activating it. You will probably find yourself staring at a start page, which gives you several options, such as opening an existing project or creating a new one. You might also see some news for developers displayed somewhere in the window. Since we are just starting out, we will obviously need to start with a new project. You might need to install the Visual C++ packages, depending on your version of Visual Studio, but just follow the prompts to do so.

Figure 2.1: Creating an empty C++ application. Arrows indicate the parts demanding attention.

On the start page, find the little blue link that reads, “Create a new project…”. If you do not see the link, just go to the menu item **File->New->Project**. A dialog box will pop up. Using the navigation
panel on the left of the pop-up window, click on “Visual C++”. A number of choices should appear in the main dialog window, and you will want to select “Empty Project”, as shown in Figure 2.1. Type a name for your program in the box labeled “Name” near the bottom of the window. Perhaps “Chapter2Example1” would be a good name. Notice that Visual Studio automatically uses the same name for the “Solution name”, which will be fine for our projects.

![Figure 2.2: The Project Property Page. Change the character set to “Not Set”.](image)

Click the OK button, and you will find yourself in the environment where you will be doing most of your work. We need to change one setting to allow us to write code, while avoiding the complications associated with Unicode text (look it up on the internet if you are interested). Use the menu bar to navigate to Project -> Properties. The Property Pages window appears, as shown in Figure 2.2. On the left-hand side, select Configuration Properties -> General. In the main window, you will see an entry labeled “Character Set”. Change the setting from “Use Unicode Character Set” to “Not Set”. By making this change, we are forsaking the Unicode Character Set. Consequently, we will use the ANSI character set, which will allow us to write code that is compatible with any modern C++ compiler on any operating system. If you decide at some point in the future to write a program with a nice graphical interface, then you will probably use the Unicode Character Set, which supports many additional characters beyond plain text. For our purposes, we do not need to worry about these sorts of nuances, but you may want to investigate this character set issue in some detail as you learn more about programming.
CHAPTER 2

We will need a source file in which to place our code, so find the Solution Explorer pane in Visual Studio. It should already be open by default. Use the menu to navigate to **Chapter2Example1** -> **Source Files**. Right click “Source Files”, and select **Add -> New Item…**. The “Add New Item” window will pop up. Select “C++ File (.cpp)”, give it a name (“Chapter2Example1” is fine for this file, too), and click the **Add** button in the lower right of the window. Now you will see the contents of this file in the main window of your Visual Studio working environment. We chose to create an empty project, so your new source file is completely blank.

Because we are exclusively developing console applications, we will not have to worry much about the user interface. When we run the program, a little console window will pop up, and all of our communication with the user will be via text. This interface has the advantage that if you wanted to compile your simulation to run on a supercomputer, you could do so with only minor changes to the code. The disadvantage is that the program will not be very user friendly. Creating a windows-based, graphical user interface is a more advanced topic, beyond the scope of this book. Everything we learn about C++ programming for a console application will still apply for windows-based programs, because the nuts and bolts of the programs will be the same, regardless of the form that the user interface takes.

Almost every programming book starts with the “Hello World!” program, which just prints that text on the screen. Type the following text into the blank space that is your “Chapter2Example1.cpp” source file:

```cpp
#include <iostream>
int main()
{
  int i;
  std::cout << "Hello World!";
  std::cin >> i;
  return 0;
}
```

A few points are worth making here. The most important point is that the syntax must be exactly right or you will confuse the compiler, which is the application that converts your code into a program that can run on your operating system. In particular, pay attention to the semicolons, which signify the end of a statement. Most, but not all, statements end with a semicolon. For instance, as you can see above, the `#include` statements do not. In addition, whenever there are braces ({}), you usually will not use a semicolon, because the braces also delimit a set of instructions. There are some exceptions to these rules, so keep that in mind, too.

The one thing the compiler does not care about is white space, which includes spaces, tabs and carriage returns. You could even put all of your statements on one single, seemingly endless line, and the program would still compile. However, such an approach would be horrible, as your code would be impossible to manage and understand. The fact that the compiler does not care about white space means that you can use white space to your advantage to organize your code. It is imperative that you get in the habit of making your code clear and legible. I suggest keeping braces on their own lines in most cases, so they are readily visible. I also suggest indenting everything within a particular set of braces, such that your lines get more and more indented as they are more and more nested within braces. Visual Studio will try to help you with this convention by automatically indenting code that occurs nested within other statements.

So what are the statements in our “Hello World” program doing? The `#include` statement tells the compiler that you will be using functions (or other useful things) that are in a different file than your source code. Some of these things are included with your C++ compiler. For instance, `iostream`
A SIMULATION TEMPLATE

includes a bunch of stuff related to input and output. Sometimes, you may also include files you have created yourself, and we will return to that topic soon.

The `int main()` command, followed by opening and closing braces, signifies the code that makes up the body of the program. All C++ programs have some version of `int main()`. The parts of the program that perform the desired actions are inside the braces, executed in order from the beginning of `int main()` to the end. Our first line within the brackets, `int i;`, declares a variable that is an integer with the name `i`. Notice the semicolon that ends this statement (and every other statement within the braces). The `std::cout` statement tells the program to output the text to the standard output stream, which is usually the screen. Note that this statement uses the `<<` symbol, which is called the **insertion operator** and points in the direction that information is moving. In this case, “Hello World!” is being sent to the standard output stream so the operator points toward `std::cout`.

![Figure 2.3: The “Hello World” Program. The arrow shows the button you need to push.](image)

On the next line, however, this operator points the other way (`>>`), and is now called the **extraction operator**. It directs data from the standard input, signified by `std::cin`, to the variable `i`. The standard input will almost always be the keyboard. Thus, the program will wait at that command until the user enters an integer and presses **Enter**. I have added this command just to make sure the console stays open until you signify that you want it to close by typing an integer and pressing **Enter**. Otherwise, Windows would run the program and immediately close the window, giving you no time to see what actually happened. The final statement, `return 0;`, is a standard part of a C++ program, so this command will be the last line of every console program that we write. Think of it as part of the `int main()` structure of the program and simply accept that every program will end with the `return 0;` command. After you type this program into your source code file, run it by pressing the button with the small green arrow (Local Windows Debugger) in the Visual Studio Tool Bar, as shown in Figure 2.3.
CHAPTER 2

That is all there is to writing a C++ program. You should see the console window pop up with the “Hello World!” text at the top. Input any integer (followed by Enter) and the program will end, closing the console window.

Random Number Generators

Almost every problem we will address in this tutorial will involve random numbers. We need to implement a portable subroutine capable of producing random numbers. This goal calls for the use of a function, a tool in C++ that we will discuss in more detail later. It also calls for some type of algorithm to produce pseudo-random numbers. So-called “random” numbers from computer programs are usually not actually random. Rather, they are produced by a mathematical algorithm that results in sequences of numbers with the same properties that would be expected of truly random numbers. However, given the same starting point (i.e., the seed for the random number generator), the algorithm will always produce exactly the same sequence of numbers. Random number generators often are seeded using the system clock, which virtually guarantees that the sequence will start in a different place each time the program runs.

Many possible random number generators are available. A search of the web will yield many options, and C++ even has some random number generators built in. For the types of simulations we will be developing, the choice of random number generator probably does not matter a whole lot, although some are computationally more efficient than others. A search of the internet shows that a random number generator known as the Mersenne Twister is highly regarded, so we will implement that one. This exercise will mainly involve cutting and pasting, as the details of what the algorithm is actually doing are beyond the scope of this book. To implement it correctly, we need to introduce some new concepts, so we are going to take a small detour into creating our own header files and understanding functions.

The Mersenne Twister Header File

In the solution explorer, just above the “Source Files” folder, you should see a folder marked “Header Files”. Right click this item, select Add->New Item…, and choose “Header File (.h)” from the window that pops up. Name this header file “MTwisterFunctions” and click Add. Copy and paste the text from Appendix 1 into this new empty file (or copy the text from the header file that is available on the book’s GitHub page). You will notice that the name of your file is “MTwisterFunctions.h”. The “.h” extension is used to indicate that it is a C++ header file. This header file includes a few new features, some of which you may not have seen before. For instance, the first line is #pragma once, which tells the compiler to include this header file only once (make sure that #pragma once only occurs once, at the very beginning of the header file). The second line is #include <cmath>, which tells the compiler to include the functions that occur in the standard C++ header file known as “cmath”. The next several lines are preceded by a double slash, //, which indicates a comment. The compiler will ignore anything after the double slash until the end of the line, so feel free to use comments liberally to clarify anything that might be confusing in your code. This particular comment is a disclaimer about the terms of use of the Mersenne Twister source code, and we are required to have it in our source code as a condition of use of this particular pseudorandom number generator, which the authors have generously decided to allow people to use for free.

For now, we are going to ignore most of the commands in the header file. There are some declarations of constants and variables and other items, most of which you will understand later. There are some parts that you will never have to understand, so do not fret over the details here too much. However, there is one aspect of this header file that you do need to understand. After the initial declarations of variables, constants and so forth, the rest of this header file consists of a number of functions, so we will take a short detour into C++ functions.
Functions in C++

A function is a small piece of code that you can invoke from elsewhere in your program to perform a specific task. Each function accepts arguments and possibly returns a value to the place from which the function was called. For the sake of illustration, we will consider a function that computes the area of a rectangle. Here is how such a function might look:

```cpp
double area(double length, double width)
{
    double RectangleArea;
    RectangleArea = length*width;
    return RectangleArea;
}
```

This function obviously has so little utility that it would be almost pointless in a real program, but it will serve to illustrate the parts of a function. The first word, `double`, indicates the type of variable the function will return. A `double` in C++ is a real number with a large number of significant digits. The word `area` is the name of the function, and it is followed by parentheses containing `double length` and `double width`. These phrases tell the compiler that the function expects two variables (the arguments), both of which are doubles (i.e., real numbers with potentially very large values), and it also gives these numbers the names “length” and “width”. These names will only be available within the function. Inside the brackets, we declare another double called `RectangleArea` and then set it equal to length times width with the statement `RectangleArea = length*width;`. Finally, the function returns the value of `RectangleArea` to the statement that called the function. Just to see how functions work, let us put this function into our “Hello World” program and try it out. Make your “.cpp” file look like the code below by adding the missing lines.

```cpp
#include <iostream>

double area(double length, double width)
{
    double RectangleArea;
    RectangleArea = length*width;
    return RectangleArea;
}

int main()
{
    int i;
    std::cout << "Hello World!\n";
    std::cout << "\n";
    std::cout << area(12, 3);
    std::cin >> i;
    return 0;
}
```

This bit of code shows a couple of important points. For instance, there are lots of places where you can put functions in your program, and one of them is right before the `int main()` statement. Second, when you call a function, it returns the value exactly where you called it. Thus, you have to be ready to do something with the value that comes back. Just like `x = 2 * 2;` would set the value of `x` to
CHAPTER 2

4. \( x = \text{area}(2, 2) \); would also set the value of \( x \) to 4. In our example, however, we are simply outputting the result to the screen. The other new statement here is `std::cout << "\n";`, which outputs a line break to the screen. In other words, the \( \backslash n \) signifies a special character, in this case a newline character.

Now look at the Marsenne Twister header file once more. It contains several different functions. We should try some of them out. Before we can try them out, however, we need to include the header file in our program, so in your “.cpp” file (i.e., “Chapter2Example1.cpp”), add (just below \#include <iostream>):

\[
\text{#include "MTwisterFunctions.h"}
\]

Now add the code shown below to your “Hello World” program just above the `std::cin >> i;` statement.

\[
\begin{align*}
\text{double} & \ \ drnd1, drnd2; \\
\text{std::cout} & \ \ << "\nUniformly Distributed:\t"; \\
drnd1 & \ \ = \text{genrand}(); \\
\text{std::cout} & \ \ << drnd1 << "\n"; \\
\text{std::cout} & \ \ << "\nBivariate Normally Distributed:\t"; \\
\text{randbivnorm}(5,2,0.8,drnd1,drnd2); \\
\text{std::cout} & \ \ << drnd1 << "\t" \ \ << drnd2 << "\n"; \\
\end{align*}
\]

If we ditch all the “Hello World” and “area” code, then the functional program that only retrieves random numbers would look like this:

\[
\begin{align*}
\text{#include <iostream>}
\text{#include "MTwisterFunctions.h"}

\text{int} & \ \ \text{main}() \\
\{ \\
\text{double} & \ \ \ drnd1, drnd2; \\
\text{std::cout} & \ \ << "\nUniformly Distributed:\t"; \\
\text{std::cout} & \ \ << drnd1 << "\n"; \\
\text{std::cout} & \ \ << "\nBivariate Normally Distributed:\t"; \\
\text{std::cout} & \ \ << drnd1 << "\t" \ \ << drnd2 << "\n"; \\
\text{std::cin} & \ \ >> i; \\
\text{return} & \ \ 0; \\
\}
\end{align*}
\]

What are the various parts of this program doing? Here is a statement by statement breakdown of what we are telling the compiler to do:

\[
\begin{align*}
\text{double} & \ \ \ drnd1, drnd2;
\end{align*}
\]

First we declare two `double` variables, \( drnd1 \) and \( drnd2 \). As mentioned above, doubles are numbers that can include decimal points (as opposed to integers, which cannot).
std::cout << "\nUniformly Distributed:";  

Here we just output the text “Uniformly Distributed:” to the screen. The \n is a new line character and the \t is a tab character.

drnd1 = genrand();

This statement both calls the genrand() function and sets drnd1 equal to the number returned by the function. Some functions return values and some do not. If the function returns a value, it must be called in a way similar to that shown here. In this case, genrand() returns a uniformly distributed number between 0 and 1 (not including 1).

std::cout << drnd1 << "\n";

Here we output the number stored in drnd1, which is the same number that was returned by genrand(), to the screen.

The concept behind the remainder of the code is similar, except now we are calling randbivnorm(). The randbivnorm function generates random numbers from a bivariate normal distribution. It requires five arguments. Yes, five! The first three numbers describe the bivariate normal distribution from which the numbers are drawn. These parameters are: the standard deviation for the first number, the standard deviation for the second number, and the correlation between the two numbers, in that order. Note that this function does not return a value. Its declaration starts with void, which means the function does not return anything. See “MTwisterFunctions.h” if you would like to see the code behind the function.

But if a function does not return a value, then how can we get random numbers from it? Here, we have encountered a substantial constraint imposed by the syntax of functions. The constraint is that a function can return only one value. However, a bivariate normal random number generator is only useful if it can return two numbers at the same time, because the correlation between the numbers is a key feature of a bivariate normal distribution. If you are getting confused by all of this distribution talk, look up “bivariate normal distribution” or “multivariate normal distribution” on Wikipedia.

Figure 2.4: The random number generator program output. Your numbers are probably different, but otherwise the output should look something like this screenshot.
CHAPTER 2

The way around this constraint is to pass variables to the function and then have the function alter the values of those variables. In C++, we do not actually have to make a copy of the variables. Rather, we just tell the function the location in memory where the variables are stored. Then the function fills that location with the values it generates. The ultimate effect is that the function changes the values of the variables whose names we pass to it. This way of giving a function access to a variable is known as “passing by reference”. A reference can be thought of as an alias for another variable. To see the syntax for passing by reference, examine the beginning of randbivnorm(), which is shown here.

```c++
inline void randbivnorm(double sigmaX, double sigmaY, double rho, double &bivN1, double &bivN2)
```

You can see that the last two arguments are preceded by &. This symbol signifies that we are creating a reference to the memory location of the variable being passed. Any changes that happen to the variables called bivN1 and bivN2 within the function will also happen to the variables in the main program passed to the function, in this case drnd1 and drnd2. Also note that bivN1 and bivN2 will initially be assigned (within the function) the values of the variables being passed. In other words, bivN1 and bivN2 in the function are using the same memory address as drnd1 and drnd2 in the main program, so for all intents and purposes, they are the same variables.

In short, if you want a function to modify more than one variable from your main program, you can pass the variables by reference. If you are confused by this aspect of functions, look up “passing by reference” in a C++ book or on the internet. Many, many examples are available to make this somewhat murky concept a little more clear.

Now whenever we run our program all of this code will execute, resulting in two lines of output (plus anything else you left in the program). Figure 2.4 shows the output from my version of the program. The first line has one uniformly distributed random number and the second line has two numbers drawn from a bivariate normal distribution. Do not be disturbed if you find that every time you run the program, you get exactly the same numbers. This somewhat unappealing feature is a consequence of the fact that we did not seed the random number generator, so it ends up using the same seed every time. We will fix this problem later. For now, be content that you have a header with a random number generator and you understand how functions work in C++.

**Chapter Summary**

In this chapter, we learned how to write and compile a simple “Hello World” program. We also learned some basic input and output operations for writing text to the screen and obtaining input from the user. This chapter also introduced functions in C++ and provided a brief overview of how they work, including passing variables to them by reference. Finally, we implemented a header containing the Mersenne Twister pseudo-random number generator. This header will be important for our future work, which will depend heavily on random numbers.
In this guide, the focus is primarily on individual-based simulations of evolutionary processes. In building our program, we will go in a slightly different order, perhaps, compared to how a person might proceed when building a model from scratch. There are two reasons to do it this way. First, we can start with simple principles and build on them as we go. Second, we can make the model building process seem more organized than it actually is. When you start building your own model, you will find that you frequently skip back and forth between the tasks that we are doing in each of the chapters of this book.

Our next task is to get used to object-oriented programming, which is one of the great selling points of C++. This type of programming takes advantage of a data structure known as a class. A class will be a place for our data to reside and also a place for us to keep the functions that act on the data. By using classes, we will be able to keep our code clean and organized. We will also be able to generate header files containing the classes, and these header files will be easy to move from one program to the next.

Introducing Classes

A class can be declared in your “.cpp” file, but it is usually more convenient, especially for very complex classes, to declare them in a header file. Let us start by creating a “simulation_engine” class that runs all of the little bits and pieces of our evolutionary simulation. The class will take care of important details like initializing parameters and variables, as well as generating random numbers and so forth. We can also use it to keep track of the data. All of these goals can be accomplished, while still keeping the program nice and organized, by using object-oriented programming.

The project from the previous chapter provides a good starting place, so open that project in Visual Studio. In the Solution Explorer, right click “Header Files” and select Add->New Item… Choose “Header File”, and name it “simulation_engine”. In general, resist the temptation to put spaces in the name of anything. Instead, use underscores. While spaces do not matter much to Windows operating systems, they do matter to Linux, so you can avoid a lot of headaches down the road by falling in love with underscores now.

When you open your new header file, you will see that it includes only one line of code, the #pragma once that we previously encountered. Add code to make the header look like this:

```cpp
#pragma once
#include <iostream>
#include "MTwisterFunctions.h"

class simulation_engine
{
    private:
        int NumberOfGenerations;
        int PopulationSize;

    public:
};
```

This code demonstrates the basics of declaring a class. We include the “MTwisterFunctions.h” header so the simulation_engine class will have access to the random number functions. Class declarations are
fairly simple, using the keyword `class`, followed by the name of the class, open braces, closing braces and a semicolon. All of the code corresponding to the class goes within the braces.

You will notice a couple of new keywords inside the braces delineating our class. The words `private` and `public` are very important. Variables and functions following the private keyword are only accessible from within the class, whereas variables and functions after the public keyword are accessible from anywhere in your program. Generally, good programming practices dictate that most of the data within your class should be private, and class functions should be implemented to manipulate the data. The functions will have to be public to be usable. We will usually follow these good practices, but sometimes, especially for very simple classes, we might not. The idea behind this programming practice is to limit accidental access to the members of a class. For very complex classes with many functions and data members, this practice is very important. The same holds true for very large projects on which many programmers are working simultaneously. However, for simple classes mainly designed to organize chunks of data, strictly limiting data to be private can be unnecessary and cumbersome.

**Keeping Track of Parameters and Variables**

Any simulation-based model will require quite a few parameters and variables. Parameters are aspects of the model that are set at the beginning of a given simulation run, and they remain constant throughout the run. Variables are the values that change during the run. If we are interested in how the population size affects the response of the mean body size to selection, then population size would be a parameter and the mean body size would be a variable. Confusingly, both parameters and variables in our model will be represented by variables in C++. The difference will be that some of these variables will be set at the beginning of the simulation run (the parameters) and others will be measured during the simulation run (the variables).

Probably the easiest way to keep track of variables and parameters for your simulation is to have them reside within your simulation_engine class. For now, simulation_engine already has two variables declared. Both of them are integers (int), and they are named `NumberOfGenerations` and `PopulationSize`. When naming variables, make the names descriptive rather than brief. Visual Studio will help you type long variable names, so there is no compelling reason to keep them short. Descriptive names will make your program less confusing.

For now, the variables declared within simulation_engine are private, which means we cannot access them from outside the class. If we want to set them to particular values, we will need to write a function to do so. Since the population size and number of generations are both parameters, a reasonable approach might be to set them to a certain value at the very beginning of a simulation run. We can accomplish this goal by initializing the simulation each time it runs. But first let us declare a couple of other variables to initialize. In the private section of your simulation header, type:

```cpp
double MutationalCorrelation, SelectionalCorrelation;
```

Recall that doubles are real numbers (with a decimal point) that can have very large values and very many significant digits. Also recall that variable names are case sensitive, so if you declare `MutationalCorrelation` and `mutationalcorrelation`, they are as different as `edwardowilson` and `stevenjaygould` in the eyes of the compiler.

Multiple variables of the same type can be declared on the same line by separating them with a comma, as shown above. We will add additional parameters later. For now, let us see how to set these parameters to a desired value when the program first starts up.
**Initializing the Simulation**

Before the simulation starts, we need to ensure that the values of all parameters are set to their desired starting values. An easy way to set parameter values is to design a function to do it. The function should reside within the class, so it can use all of the class variables and is automatically included in the class whenever the header is included.

Each class actually has a special function called a **constructor**. This constructor runs whenever an instance of the class is created in your program, and the constructor can take arguments just like a regular function. The flip side of a constructor is a **destructor**, which runs when the instance of the class is destroyed. One way to initialize our variables would be to do it in the constructor for our class.

The constructor for a class has the same name as the class but is a function that resides within the class. Add the code shown below after the **public**: but above the `};` that signifies the end of the class.

```cpp
simulation_engine() // Constructor: Initialize parameters and variables here
{
    NumberOfGenerations = 10;
    PopulationSize = 100;
    MutationalCorrelation = 0.5;
    SelectionalCorrelation = 0.2;
}
```

I have added a comment here to indicate that we are using the constructor to initialize the parameters. Because the parameter variables are private, we cannot touch them from our main program, not even to print them to the screen, so let us quickly write a function to display the parameter values. Just below the simulation_engine constructor, add the following function (you should still be within the public part of your class):

```cpp
void display_parameters()
{
    std::cout << "\nNumber of Generations: \t" << NumberOfGenerations;
    std::cout << "\nPopulation Size: \t" << PopulationSize;
    std::cout << "\nMutational Correlation: \t" << MutationalCorrelation;
    std::cout << "\nSelectional Correlation:\t" << SelectionalCorrelation;
}
```

Now we can put this class to use in our main program (even though it does not really do much). Navigate back to your “.cpp” file, which should still have code to produce random numbers, and add `#include "simulation_engine.h"` to the include statements. Also add the following code before the `std::cin >> i;` statement:

```cpp
simulation_engine mysim;
mysim.display_parameters();
```

The first statement declares an instance of the simulation_engine class. Classes are just like variables in that you have to declare an instance of them to use them. Each instance of a class is called an object, which is where the term object-oriented programming has its roots. Because you declare objects like variables, you could in principle have more than one simulation_engine declared at the same time within the same program. Once we have declared a simulation_engine object, we access its members, i.e., the functions and variables contained within the class, by using a period followed by the function or variable of interest. In this case, we are calling the function `display_parameters()`. We did not have to invoke the constructor, because it is invoked every time a new object is created.
CHAPTER 3

Seeding the Random Number Generator

We also need to seed the random number generator each time the program starts. You may have noticed that each run of your program has produced the same set of numbers, which suggests that maybe they are not so random after all. Random number generators use an algorithm to produce a sequence of numbers that looks and behaves like random numbers would, but the algorithm needs a starting point, which is the seed. Because the random number generator is a mathematical algorithm, each time it starts with the same seed it produces exactly the same sequence of numbers. Each time we run a simulation, we would like a different set of numbers, so we need to choose a different seed each time our program runs. Fortunately, C++ has built-in algorithms that we can use to seed the random number generator using the system clock (which will be set to a different value every time the program runs since time will continue to pass).

Back in our header file, “simulation_engine.h”, add one more include statement: #include <time.h>. Then in the simulation_engine constructor, near the top, add:

```
srand(time(NULL));
sgenrand(rand());
```

After adding this code, select Build Solution from the Build menu in Visual Studio. You will see a warning in the Output window: “conversion from ‘time_t’ to ‘unsigned int’, possible loss of data”. As long as you are sure what you are doing, you can ignore this type of warning. However, if you want the warning to go away, you can explicitly tell the compiler to convert the value returned by time() into an integer. This conversion can be accomplished using the static_cast() function. Change the code to read:

```
srand(static_cast<int>(time(NULL)));
sgenrand(rand());
```

Now the result from time() will be converted into an integer before being passed to srand(). If you build the program now, you will not see an error.

What are these statements doing? The first statement, srand(), seeds the built-in random number generator. It expects an integer, and time(NULL) provides said integer. The second statement uses the built-in random number generator to seed the random number generator in our “MTwisterFunctions” header. If you look at that header, you will see a function called sgenrand(), and you will also see that it expects a single unsigned long int as an argument. The built-in function rand() returns an integer between 0 and RAND_MAX (a constant defined by your particular compiler). If you want to know the value of RAND_MAX, just add `std::cout << RAND_MAX;` to your program somewhere. Of course, if we decide not to use the built-in random number generator at all, which would be a sound decision, we could eliminate the call to srand() and just seed the Marsenne Twister algorithm with the system clock with the following statement:

```
sgenrand(static_cast<int>(time(NULL)));
```

If you build your program now, you should get no errors or warnings. If you run it, you will be shocked to see that the random numbers it outputs are the same ones as before and are still the same every time. The problem now is that you seed the random number generator after you call the random number functions. If you rearrange the order, so that your simulation_engine is initialized before the code producing the random numbers, you should get different numbers each time you run the program. Here is what “simulation_engine.h” should look like:
### Getting Organized with Classes

```cpp
#pragma once
#include <iostream>
#include "MTwisterFunctions.h"
#include <time.h>

class simulation_engine
{
private:
    int NumberOfGenerations;
    int PopulationSize;
    double MutationalCorrelation, SelectionalCorrelation;

public:
    simulation_engine() // Constructor: Initialize parameters and variables here
    {
        srand(static_cast<int>(time(NULL)));
        sgenrand(rand());
        NumberOfGenerations = 10;
        PopulationSize = 100;
        MutationalCorrelation = 0.5;
        SelectionalCorrelation = 0.2;
    }

    void display_parameters()
    {
        std::cout << "\nNumber of Generations: \t" << NumberOfGenerations;
        std::cout << "\nPopulation Size: \t" << PopulationSize;
        std::cout << "\nMutational Correlation: \t" << MutationalCorrelation;
        std::cout << "\nSelectional Correlation: \t" << SelectionalCorrelation;
    }
};
```

And here is the main program:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int i;

    simulation_engine mysim;
    mysim.display_parameters();

    std::cout << "\nRAND_MAX = " << RAND_MAX << "\n";

    double drnd1, drnd2;
    std::cout << "\nUniformly Distributed: \t";
    drnd1 = genrand();
    std::cout << drnd1 << "\n";

    std::cout << "\nBivariate Normally Distributed: \t";
    randbivnorm(5, 2, 0.8, drnd1, drnd2);
    std::cout << drnd1 << "\t" << drnd2 << "\n";
```
std::cin >> i;
    return 0;
}

With the random number generator implemented and a class ready to hold our variables and functions, we are ready to try to make our program actually do something useful.

Chapter Summary

This chapter focused on some key programming principles for C++. We started with an introduction to classes, which form the backbone of this object-oriented programming language. Gone are the days of linear, monolithic programs. Rather, we need to think in terms of connecting data and the functions to manipulate the data in an organized, portable framework known as a class. We also created a “simulation_engine” class, which will be the place where we keep most of the important variables and functions related to our simulations. This chapter also introduced the class constructor, which is a function that runs automatically whenever an object (i.e., an instance of a class) is created. The chapter concluded by showing how to seed the random number generator using the system clock. The one difficult concept in this chapter might be the class. If you feel confused about classes, seek some outside resources, such as websites or another book, until you start to feel like you understand them. Over the next few chapters, you will develop a better feel for classes as we use our simulation_engine class to perform various interesting tasks.
Chapter 4. Iterating an Equation

One possible use of a computer in the analysis of evolutionary or ecological models is to iterate equations. This application is known as **numerical analysis**. We will experience only one example of numerical analysis to illustrate some basic principles, because this approach is much simpler than an individual-based model. If you can specify equations that govern the dynamics of your system, then numerical analysis can be a quite powerful tool. Even extremely complex systems of equations, which are not amenable to analytical solutions, can be solved with numerical analysis.

**Specifying the Model**

For the purposes of illustration, let us pick a very simple equation from population genetics. One of the most important evolutionary processes is natural selection, so we will use the equations for a model of viability selection. We will assume a single genetic locus with two alleles (A and B) in a diploid population. There will be three possible genotypes: AA, AB, and BB. We can assume different fitnesses for these genotypes and discover what happens to the allele frequencies each generation. If we were clever, we could just derive an analytical solution to this particular problem, but that is not the point. If we are willing to do numerical analysis, then we can find out what happens with a lot less math, even though this approach is admittedly not as pure and noble as analytical theory. If this model has you confused already, just check out any evolutionary biology textbook.

We can actually derive the equation for the change in the allele frequencies from one generation to the next surprisingly easily, so we might as well go over that here. Our population satisfies all of the assumptions of Hardy-Weinberg Equilibrium (see your evolution textbook if you do not know what that means), except “no selection”. Thus, we assume a very large population size, random mating, no migration and no mutation. As a consequence, the zygotes in any generation will start out in Hardy-Weinberg Equilibrium (in a diploid, one generation of random mating puts the population into Hardy-Weinberg Equilibrium). Let us assume the following fitnesses of the genotypes:

\[
\begin{array}{c|c|c|c}
    & AA & AB & BB \\
    \hline
    w_{AA} & w_{AB} & w_{BB} \\
\end{array}
\]

We can now easily specify the frequency of each genotype before selection and their relative proportions after selection. If we assume \( p \) is the frequency of A and \( q \) is the frequency of B, then we find:

\[
\begin{array}{c|c|c|c}
    & AA & AB & BB \\
    \hline
    \text{Frequency in zygotes} & p^2 & 2pq & q^2 \\
    \text{Relative proportion after selection} & p^2w_{AA} & 2pqw_{AB} & q^2w_{BB} \\
\end{array}
\]

The relative proportions after selection may not sum to one, and true frequencies must sum to one. We can convert the relative proportions into frequencies by dividing each relative proportion by the sum of all three. The sum of the relative proportions is actually the mean fitness in the population (it is a weighted
average), so we will just use a capital $W$ to represent it. Thus, $W = p^2 w_{AA} + 2pq w_{AB} + q^2 w_{BB}$. Now we can specify the frequency of each genotype after selection:

<table>
<thead>
<tr>
<th>Frequency after selection</th>
<th>AA</th>
<th>AB</th>
<th>BB</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$p^2 w_{AA}/W$</td>
<td>$2pq w_{AB}/W$</td>
<td>$q^2 w_{BB}/W$</td>
</tr>
</tbody>
</table>

We are interested in the changes in allele frequencies over time, so let us calculate the frequencies of $A$ and $B$. Recall that $p$ is the frequency of $A$ and $q$ is the frequency of $B$. Then $p$ is equal to the frequency of the AA genotype plus half the frequency of the AB genotype, because AA individuals have only A alleles, and AB individuals have half A alleles. If this does not make sense to you, look it up in an undergraduate evolutionary biology textbook. So, $p$ after one generation of selection is:

$$p' = p^2 w_{AA}/W + (1/2)2pq w_{AB}/W = p^2 w_{AA}/W + pq w_{AB}/W = (p^2 w_{AA} + pq w_{AB})/W = p(pw_{AA} + qw_{AB})/W$$

We will refer to the allele frequency after selection as $p'$ to indicate that it is one generation later than the original $p$. The same logic leads to the following expression for $q'$:

$$q' = q(pw_{AB} + qw_{BB})/W$$

Of course, $p$ and $q$ must sum to one, so $q' = 1 - p'$. Anyway, now we have the equations we need to find out what happens when selection acts at a single locus. We can iterate these equations from generation to generation to see what happens to $p$ and $q$ over time. This model obviously has several parameters that we will need to specify in our program. In particular, we will need variables for the starting allele frequencies, the number of generations, and the fitnesses of the different genotypes.

**Implementing the Model**

Create a new project in Visual Studio by following the steps outlined above. In short, launch Visual Studio, select **File->New->Project...**, choose “Empty Project” from the Visual C++ menu, name it “Chapter4Example1”, and finally go to **Project->Properties** and change the “Character Set” setting to “Not Set”. Next, add a new source file by right clicking on “Source Files” in the Solution Explorer window. Add a new item, a C++ file (.cpp), and name it “Chapter4example1”.

Numerical analysis, especially for a simple model like the one here, does not require a lot of heavy lifting, so we will not bother with a separate header file. Type the following code in your new source file to get started:

```cpp
#include <iostream>

int main()
{
    return 0;
}
```

Here we have the bare bones of a program, including **iostream** to facilitate communication with the user. You could run it, but the console window would just pop up and disappear immediately.

Next, declare some new variables to hold the parameters we will need for our simulation. Immediately after the opening bracket of **int main()**, declare five new doubles: Pstart, Qstart, Waa, Wab, and Wbb. Also declare a new integer to count the number of generations. You can declare them all by typing:
double Pstart, Qstart, Waa, Wab, Wbb;
int NumberOfGenerations;

We should initialize these parameters to have some starting values, so below their declarations, add the following lines of code:

Pstart = 0.5;
Qstart = 1 - 0.5;
Waa = 1.2;
Wab = 1.0;
Wbb = 0.8;
NumberOfGenerations = 20;

We finally have to face the crux of the problem, which is to instruct our program to use the provided parameters to determine some kind of result and communicate it to the user. Let us use a comment to indicate what the program is intended to do:

// Single-locus, constant viability selection
// in a diploid population with two alleles.

We want to take the initial allele frequencies given as parameters by the user and iterate the selection equation to see what happens to the allele frequencies over time. We will want to iterate it for the number of generations specified in the variable NumberOfGenerations. Ultimately, we will want to output the results in a way that is suitable for further analysis in a standard spreadsheet program, so we will need to be able to save the results somehow.

Knowing that we want to run the program for a specified number of generations is a clue that we will need some kind of loop, in which the program repeats the same actions for as many iterations as we specify. The most common type of loop is a for loop. To set up this type of loop, we will need an integer to keep track of the number of elapsed iterations. It is common practice to use single letter variables in loops (often starting with the letter i), so let us declare int i; near the beginning of our program. Let us also declare p, q, Pprime, and Qprime, which should be doubles, to keep track of the changing allele frequencies over time. Below are the lines of code that implement the suggested changes, and they should be placed near the beginning of the program, near where the other variables are declared.

int i;
double p, q;
double Pprime, Qprime;

Those seem to be all of the variables we will need, so we can proceed to the meat of the analysis. We should set p and q to their starting values by adding the following lines of code:

p = Pstart;
q = Qstart;

Loops are very critical in programming, so it is important to understand how they work. The for loop in C++ has a very specific syntax, which looks like this:

for (i = 0; i < NumberOfGenerations; i++)
{
}

21
The word **for** specifies that a loop is being declared. The variable $i$ can be any integer variable. That is, you can declare any integer and use it in the loop. This integer is the variable that counts the number of times you have been through the loop. The statement $i < 10$ will be evaluated at the beginning of each iteration through the loop. If the statement is true, the loop will run; if the statement is false the code contained within the loop will not run and execution will skip to the next line of code that comes after the loop’s closing brace. If the loop does run, all of the statements between the opening and closing brace will run each time through the loop. Finally, the $i++$ statement tells the loop to increment $i$ by one at the end of each run through the loop. Thus, in this case, $i$ will have the value of zero on the first run through the loop. On the second run, it will have the value of one, and so forth. This loop will run exactly 10 times, and $i$ will have the value of 10 when it ends, because on the last run through $i$ will reach this value, making $i < 10$ false and ending the loop. Note that within the loop, while it is running, $i$ will have the value of 0 the first time through and 9 the tenth time through. It will never have the value of 10 inside the loop – otherwise you would get 11 iterations of the loop instead of 10 – but $i$ does equal 10 at the end, thus preventing the loop from running. **Go back and read this section a few times until you understand it!**

To accomplish our current goal, we will want to create a loop that will have as many iterations as there are generations, so add the following statement to your program:

```cpp
for (i = 0; i < NumberOfGenerations; i++)
{
}
// end of i loop
```

The comment after the last brace, indicating the brace goes with the $i$ loop, is optional, but such a comment can help you keep track of your braces, especially when you have a lot of loops nested within other loops. Here is what the entire program looks like so far:

```cpp
#include <iostream>

int main()
{
    double Pstart, Qstart, Waa, Wab, Wbb;
    int NumberOfGenerations;
    int i;
    double p, q;
    double Pprime, Qprime;

    // Set parameters and starting conditions here

    Pstart = 0.5;
    Qstart = 1 - 0.5;
    Waa = 1.2;
    Wab = 1.0;
    Wbb = 0.8;
    NumberOfGenerations = 20;

    // Single-locus, constant viability selection
    // in a diploid population with two alleles.

    p = Pstart;
    q = Qstart;
```
ITERATING AN EQUATION

```cpp
for (i = 0; i < NumberOfGenerations; i++)
{
} // end of i loop

return 0;
}
```

If we set it up so the equations are applied during every iteration of the loop, then we will iterate the equations for the number of times specified by the number of generations. The implementation of this plan requires us to convert our equation for \( p' \) into C++ code that our computer can execute. As a reminder, here is the equation we wish to iterate:

\[
p' = \frac{p(pw_{AA} + qw_{AB})}{W}
\]

The average fitness term, \( W \), is pretty complicated, and it occurs in the equations for both \( p' \) and \( q' \). We probably should declare another variable for \( W \) so we can calculate it separately before we attempt to calculate \( p' \) and \( q' \). Near the top of the program, where we declared the other variables, declare another double by adding the statement:

```cpp
double Wbar;
```

Each time we calculate \( p' \) and \( q' \), we should first calculate \( Wbar \) and use it in our equation to produce clear, simple code. Inside the braces for your loop add:

```cpp
Wbar = p*p*Waa + 2 * p*q*Wab + q*q*Wbb;
pPrime = p*(p*Waa + q*Wab) / Wbar;
qPrime = q*(p*Wab + q*Wbb) / Wbar;
```

Amazingly enough, those three lines of code are pretty much all we need to implement this model of selection. It would be nice for the program to output the results, and here is an easy way to do it. Add the following code within your loop (immediately before the closing brace):

```cpp
std::cout << "\n" << i + 1 << "\t" << pPrime << "\t" << qPrime;
```

Recall that `std::cout` is a function (in `iostream`) that prints to the standard output – the screen in most cases. The `\n` and `\t` are new line and tab characters, respectively, and the `<<` tells the computer which direction the information is going. In this case, it is all going to `std::cout`. We first output the generation number. Since we are outputting the data within the loop, we need to add one to \( i \) to indicate the number of generations of selection that have elapsed. For example, the first time through, \( i \) is still zero when we get to the part that outputs the data, but the equations for \( p' \) and \( q' \) have already been applied at this point, so one generation of selection has elapsed.

Run the program and see what happens. It opens and closes so fast that you cannot see anything. At the end of the program, just before the `return 0;`, add the code shown below so that the program prompts us to enter a character before it closes.

```cpp
char end_it;
std::cout << "\nEnter any character to exit...";
std::cin >> end_it;
```
CHAPTER 4

Now when you run the program, you should see results that look something like the results shown in Figure 4.1.

![Image of program output](image)

**Figure 4.1: The results of the program as it now stands.**

The results look nice in the sense that they are lined up in columns, but they have the surprising feature that the allele frequencies do not change over time. We expect selection to result in a change in allele frequencies, so we should immediately suspect that our program contains a flaw. In addition to this apparent problem with our program’s logic, we can also see that we did not label the columns or give the starting values (i.e., generation zero) in our output.

Think for a moment about why the values failed to change from one generation to the next in our program, before you read the answer in the next sentence. The answer is that the values did not change because every time we calculate $P_{prime}$ and $Q_{prime}$, we use the starting values of $p$ and $q$. This problem is easily remedied by keeping a running tally of the values of $p$ and $q$. Just before the closing brace of your $i$ loop, add the following:

\[
p = P_{prime}; \\
q = Q_{prime};
\]

We can easily fix the problems with our output, too. Immediately before the statement declaring the $i$ loop, add this code:

```cpp
std::cout << "Gen\tqt\n";
std::cout << "0\t" << Pstart << "\t" << Qstart;
```

Run the program again, and you should see output like that shown in Figure 4.2.
ITERATING AN EQUATION

Figure 4.2: The program with the main problems fixed.

This output looks a lot better, especially since the equations seem to be working correctly. Allele A has higher fitness than allele B, and we are seeing the expected result that the frequency of A, given by \( p \), is increasing toward a value of one. However, our output looks sloppy, because the numbers do not line up well. We can make this output look better by reducing the number of significant digits reported to the user. An easy way to change the precision of the output for a floating point number (e.g., a double variable) is to use the `setprecision` function in the `iomanip` package. To implement this change, add the following code. Just below your `#include <iostream>` statement, add:

```cpp
#include <iomanip>
```

And change the `std::cout` statement in your loop to:

```cpp
std::cout << std::setprecision(4) << "\n" << i + 1 << "\t" << Pprime << "\t" << Qprime; 
```

This statement tells C++ to output a double with only four digits of precision, which will fit in the space between subsequent tab locations and consequently will line up. Now run the program and take a look at the numbers being output. The value of \( p \) is increasing and the value of \( q \) is decreasing. The two numbers always sum to one, except perhaps for a small amount of rounding error. Here is the complete program with all of the changes implemented:

```cpp
#include <iostream>
#include <iomanip>

int main()
{
    double Pstart, Qstart, Waa, Wab, Wbb;
    int NumberOfGenerations;
    int i;
```
double p, q;
double Pprime, Qprime;
double Wbar;

// Set parameters and starting conditions here
Pstart = 0.5;
Qstart = 1 - 0.5;
Waa = 1.2;
Wab = 1.0;
Wbb = 0.8;
NumberOfGenerations = 20;

// Single-locus, constant viability selection
// in a diploid population with two alleles.
p = Pstart;
q = Qstart;

std::cout << "Gen\tp\tq\n";
std::cout << "\t" << Pstart << "\t" << Qstart;

for (i = 0; i < NumberOfGenerations; i++)
{
    Wbar = p*p*Waa + 2 * p*q*Wab + q*q*Wbb;
Pprime = p*(p*Waa + q*Wab) / Wbar;
Qprime = q*(p*Wab + q*Wbb) / Wbar;

    std::cout << std::setprecision(4) << "\n" << i + 1 << "\t" << Pprime
        << "\t" << Qprime;

    p = Pprime;
    q = Qprime;
}

// end of i loop

char end_it;
std::cout << "\nEnter any character to exit...\";
std::cin >> end_it;

return 0;

Saving Output to a File
While output to a small console window is nice and easy to look at for a small project without much in
the way of results, bigger projects will almost always necessitate a save to a file. Fortunately, saving
results to a file is almost as easy as printing them to the screen in C++.

To save output to a file, you will need to use file streams, and the associated functions are included in
the header known as fstream. Add the following include statement to your program:

#include <fstream>

This header includes a variety of classes and functions related to file input and output. To save to a
file, we need to declare an object of the ofstream type. In addition, if we want to read from a file, we
use an ifstream object. You can remember the difference between these two classes, because “o” stands
for output and “i” stands for input in this case. To write to a file, the steps are simple. We declare the object, tell it to open a file, write to the file using the insertion operator (<<), and close the when we are done using it. Here are the commands we would use to write “Hello World” to a text file:

```cpp
std::ofstream out_file;
out_file.open("output.txt");
out_file << "Hello World\n";
out_file.close();
```

The file will be written in the active directory, which is usually the same one in which the compiled program resides. In this case, during the debugging process, the directory is probably deeply embedded in a folder somewhere on your hard drive. You should take some time now to find the folder associated with your Visual Studio project. In my case, on my Windows 10 machine, my “Chapter4Example1” project writes its output to a folder with the following path:

```
This PC > Local Disk (C:) > Users > agjon > source > repos > Chapter4Example1 >
Chapter4Example1
```

For easy access, I went ahead and made a shortcut to the “repos” folder and placed it on my desktop. This shortcut then allows me to get to the files associated with my programs, including source files, header files, and output files, without digging through my computer’s file structure every single time.

Returning to the issue of using the `ofstream` class, let us review what the code snippet above is actually doing. First, we declare an object of type `ofstream`. Note that the `std::` prefix indicates that `ofstream`, like `cout` and many of the other commands we have used, is in the standard (std) namespace. We will not discuss namespaces further here, other than to say that C++ uses them to keep commands organized. If you would like to know more about this topic, you should seek out additional information in a C++ book or on the internet. Second, we use the `open()` function to open a file called “output.txt”. This name could be any permissible filename, so feel free to change it. By putting a “.txt” extension at the end, we ensure that a double click on the file will results in its opening in a text editor like Notepad. Next, we write “Hello World” to the file by using the `<<` operator. Finally, we close the file, by calling the `close()` function for the `ofstream` class. As far as writing text to a file, that is all we need to know for the moment. Before you read onward, see if you can modify your program to write to a text file instead of the screen (or write to both if you are feeling exceptionally energetic). Here is the program modified to write to a text file and to the screen:

```cpp
#include <iostream>
#include <iomanip>
#include <fstream>

int main()
{
    double Pstart, Qstart, Waa, Wab, Wbb;
    int NumberOfGenerations;
    int i;
    double p, q;
    double Pprime, Qprime;
    double Wbar;

    std::ofstream out_file;
    out_file.open("output.csv");

    // Set parameters and starting conditions here

    return 0;
}
```
CHAPTER 4

Pstart = 0.5;
Qstart = 1 - Pstart;
Waa = 1.2;
Wab = 1.0;
Wbb = 0.8;
NumberOfGenerations = 20;

// Single-locus, constant viability selection
// in a diploid population with two alleles.

p = Pstart;
q = Qstart;

std::cout << "Gen\tp\tq\n";
std::cout << "\t" << Pstart << "\t" << Qstart;

out_file << "Gen,p,q\n";
out_file << "0," << Pstart << "," << Qstart;

for (i = 0; i < NumberOfGenerations; i++)
{
    Wbar = p*p*Waa + 2 * p*q*Wab + q*q*Wbb;
Pprime = p*(p*Waa + q*Wab) / Wbar;
Qprime = q*(p*Wab + q*Wbb) / Wbar;

    std::cout << std::setprecision(4) << "\n" << i + 1 << "\t" << Pprime
           << "\t" << Qprime;

    out_file << "\n" << i + 1 << "," << Pprime << "," << Qprime;

    p = Pprime;
    q = Qprime;
}

out_file.close();

char end_it;
std::cout << "\nEnter any character to exit...";
std::cin >> end_it;

return 0;
}

This version of the program only has a few changes compared to the last version. In the interest of complete transparency, here is a breakdown of what the new commands are doing. The statement

std::ofstream out_file; declares an object of type ofstream. Then we use the open() function that is a member of the ofstream class to open a file with a name of our choosing, in this case “output.csv”. Here, we have chosen the “.csv” extension to tell Windows that this file is of the “comma-delimited text” variety, which will allow us to open it in a spreadsheet program by double clicking on it. Next, we write to the file before the start of our i loop to add column headers and starting values of p and q. Note that the syntax is very similar to writing to the screen and that we have changed the tab characters to commas to be consistent with “.csv” format. Within the i loop, we write the values for each generation of our model. Finally, at the end of the program, we close the file with out_file.close();.
Now would be a good time to dig through the folders on your computer and take a look at “output.csv”. If you open this file in a spreadsheet program, you can easily graph the results. Under our starting parameter conditions, one allele goes toward fixation (i.e., $p$ increases toward a value of one) and the other allele decreases in frequency. Other values for the fitnesses ($w_{AA}$, $w_{AB}$ and $w_{BB}$) will result in different outcomes. For example, if we set the fitnesses so that $w_{AB}$ has the highest value, then both alleles will be maintained at an equilibrium frequency. The case where the heterozygote has the highest fitness is called overdominance, and this mechanism accounts for the maintenance of the allele that causes sickle-cell anemia in human populations.

![Graph of program's output](image)

**Figure 4.3**: A graph of the program’s output, produced in a spreadsheet program. This figure shows allele frequency changes when the starting frequency of the A allele, $p$, is 0.1, which means the B allele starts at frequency $q = 0.9$. The relative fitnesses are 0.8 for the AA genotype, 1.0 for the AB genotype and 0 for the BB genotype, so this example represents a case of overdominance where one homozygous genotype is lethal.

To see this effect, set $w_{AA}$ to 0.8, $w_{AB}$ to 1.0, and $w_{BB}$ to 0. Here, the BB genotype is lethal. Run the program three times with different values of $P_{start}$ (0.1, 0.5, and 0.9). Make sure that $Q_{start}$ is 1 minus $P_{start}$ (you can even change the code on that line to: $Q_{start} = 1-P_{start}$). Change the file name each time so the output is saved to different files for each run, and then graph the results to see the stable equilibrium.

If you look up “overdominance” in any evolution textbook, you can find the equation for the equilibrium allele frequency. The equation for the equilibrium frequency ($\hat{p}$) is:
CHAPTER 4

\[ \hat{p} = \frac{t}{s+t}. \]

This model assumes that the heterozygote has a relative fitness of one (that is, \( w_{AB} \) in our program has a value of one). The relative fitnesses of the AA and BB genotypes are 1-\( s \) and 1-\( t \), respectively, so for our fitnesses of \( w_{AA} = 0.8 \) and \( w_{BB} = 0 \), \( s \) and \( t \) are 0.2 and 1. If we plug these values into the equation, we get an expected equilibrium allele frequency of 1.0/1.2 = 0.833. Notice that regardless of the starting values, as long as \( P_{\text{start}} \) is between 0 and 1 (not including 0 or 1), the value of \( p \) goes to this equilibrium. The implication of this result is that overdominance can be a very powerful mechanism to maintain polymorphism in populations. Even a lethal allele, if it is beneficial in heterozygotes, will be retained in a population at pretty high frequencies.

Chapter Summary

In this chapter, we learned the basics of numerical analysis, which consists of iterating an equation to observe the dynamics of a system. Along the way, we covered some critical programming skills, the most important of which are for loops and file output, as implemented using the ofstream class. These meager skills are already pretty powerful, and we will build on them in the next chapter.
Chapter 5. Moving Toward Individual-Based Simulations

While the iteration of equations can be useful in many cases, such an approach reaches its limit quite quickly for many problems in evolution and ecology. For even slightly complex topics, it may be impossible to specify equations without making unrealistic assumptions. A solution to this limitation is to use individual-based simulations. Of course, the use of these simulations carries a tradeoff. The individual-based simulation will normally be a simpler model to design, but more difficult to program. In addition, the interpretation of individual-based simulations can be difficult, as they often cannot be analyzed as completely as mathematical models due to computational limitations. Regardless, in many cases, simulations are the only real option.

A simulation model is actually pretty simple in concept. Mainly it involves a lot of manipulation of numbers and record keeping. The record keeping is the hard part, but fortunately C++ has some features that will help us. To feel confident in the realm of individual-based simulations, we will have to delve into some more difficult issues in the C++ programming language.

Objects and Classes Revisited

The C++ programming language is an update from old-school, regular everyday C, and the biggest difference is that C++ is an object-oriented language. Thus, C++ uses objects and classes. Here, we will revisit some of the points from Chapter 3 and build on them until we have a functional class for our individual-based simulation. Remember, that a class is a data structure, complete with variables and functions. It is basically treated like any other variable (much like an int or double) by C++, except it can have lots of subvariables and its own functions that are defined by the programmer. An object is an instance of the class. In other words, a class is a template and an object is the actual realization of the template in your program. To continue the variable analogy, C++ defines what an int type variable is, which is analogous to defining a class. Then when you declare int my_integer, my_integer is an instance of the int variable. Say I call my class “address_book”. I could add the following line of code in my program: address_book my_addresses; Here, the class is “address_book” and “my_addresses” is an object of that class.

We learned how to declare a class in Chapter 3, while working with our simulation_engine class. Now we need to add to it and prepare it for individual-based simulations. One interesting aspect of a class is that it can contain other classes within it. In other words, you can declare a class and then use that class in a new, more inclusive class. Since our simulation Engine will be nearly all-encompassing, we will want some simple classes to keep track of variables within the simulation_engine.

Get started by opening Visual Studio and creating a new project, “Chapter5example1”. Make sure you create a new empty project, and be sure to set the “Character Set” to “Not Set” under Project->Properties.

Begin by adding a new source file, named “Chapter5example1”. You will recall that this feat can be accomplished by right-clicking on “Source Files” in the Solution Explorer. Select “C++ File (.cpp)” and type a filename into the dialog box. Now add some of the basics to this source file. In general, we will always want to start with:

#include <iostream>
CHAPTER 5

```cpp
int main()
{
    char end_it;
    std::cout << "\n\nEnter any character to exit...";
    std::cin >> end_it;
    return 0;
}
```

Back in Windows, navigate to the folder containing your new project. If you cannot recall how to find it, go back to Chapter 4, where we discovered the path to the folder in which our Visual Studio projects lurk on our hard drives. Within the project’s folder, you will see another folder with the same name, as well as some files created by Visual Studio. Open the folder named “Chapter5example1” and see what it contains. You should see that this folder is the one in which Visual Studio stores your “.cpp” files. Your header files will also go here.

Now go back to Visual Studio and click the arrow on the toolbar to compile and run your minimal program. A console window will pop up and it will say “Enter any character to exit…”, because that is the extent of its functionality. Enter a character, and the console window will disappear. Now take another look at the contents of the “Chapter5example1” folders. A “Debug” folder has now been added to each one, if it was not present already. These folders are used by Visual Studio to keep track of various important details during the debugging process. Note that one of the “Debug” folders contains “Chapter5example1.exe”, which is the compiled executable for your program. It is compiled in debug mode, which results in a less efficient program than if you had compiled it in release mode, but it is still a functional executable.

The main point of this exercise for now is to understand where Visual Studio is storing your source and header files, and the answer is that they are in the lowest level folder with the same name as the project. If you want to add a source or header file that you have already written to this project, the easiest way to do it is to copy the file into this folder. Then you simply add it to your project using the Solution Explorer window within Visual Studio.

In Chapters 2 and 3, we created two header files, one for random numbers and one for our simulation_engine, both of which are probably in your Chapter 2 example program (because Chapter 3 added to the existing Chapter 2 program). Rather than retype everything, we might want to use these header files in our present project and build on them. To add them to our current project, navigate to your Chapter 2 example folder, copy “MTwisterFunctions.h” and “simulation_engine.h”, navigate back to your Chapter 5 example folder, and paste these two files in the same folder that contains “Chapter5example1.cpp”. Now go back to Visual Studio and in the Solution Explorer right click on “Header Files”. Choose **Add -> Existing Item**, and add “MTwisterFunctions.h”, as well as “simulation_engine.h”. Both files should appear under “Header Files” in your Solution Explorer. You can double click on them to open them if you wish. Now they have been added to your project, but not to your program. To let the compiler know they are part of your program, you still need to use `#include` statements.

Before we make any dramatic changes to our program, we should get a little more practice with classes. When we declare a class, the name should be informative. Since this program will be an individual-based simulation, we will probably have to keep track of data for individuals, so it might make sense to make a class called “individual”. In “simulation_engine.h”, above the declaration for the simulation_engine class, add the following code:

```cpp
class individual
{
```
public:
    std::string name;
    int age;
    double weight;
    double height;

This class uses the “string” type included with C++, which is in the “string” header, so add the following include statement to the top of this header file:

#include <string>

Now we have created a new variable type (i.e., a class) called “individual”, and each time we declare a new individual, this instance of the variable (i.e., an object) will have all of the nested variable that we declared (i.e., name, age, weight, and height). Hopefully, it is starting to become obvious that this way of handling data could be convenient. Our class has very little functionality – it has no functions of its own and all of the member variables are public. Consequently, it is basically a “structure” (struct), which is a data type that was present in C and is essentially a stripped down class. Now that classes exist, there is no real need for structures, so we will not mention them again.

We can play around with the class a little to see how it works. Our simulation_engine class already has some variables, such as NumberOfGenerations and so forth. We will ignore those for now, but leave them where they are, and add some individuals. The individuals should be added under the “private:” heading of the simulation_engine class, as follows:

private:
    individual mom;
    individual dad;

Because these variables are private, they can only be accessed via functions contained within the simulation_engine, so if we want to set the values of their name, age, height and weight, we will need to write a function. An alternative, which would be considered a less than ideal programming practice, would be to declare mom and dad as public variables. Then we could access them directly. We will see examples of that approach later.

In the public part of the simulation_engine, let us write a few functions. We will need to set values for mom and dad, as well as output mom and dad. To set values, we can write one function for each parent and have each function set all values for the parent under consideration. Just below the simulation_engine() function, which you will recall is the constructor for the simulation_engine, type the following two functions:

```cpp
void set_dad(std::string dad_name, int dad_age, double dad_weight, double dad_height) {
    dad.name = dad_name;
    dad.age = dad_age;
    dad.weight = dad_weight;
    dad.height = dad_height;
}

void set_mom(std::string mom_name, int mom_age, double mom_weight, double mom_height) {
    mom.name = mom_name;
    mom.age = mom_age;
    mom.weight = mom_weight;
```
CHAPTER 5

    mom.height = mom_height;
}

You have enough experience now to understand easily what these functions do. The keyword void indicates that the function does not return a value. Each function accepts four arguments, which correspond to the variables in individual. Both mom and dad have these attributes by virtue of being objects of the individual class. The code in the bodies of the functions just sets the variables to the values of the corresponding arguments. You might recall that an object’s members can be accessed by using a period. There is not much point in setting these variables unless we use them for something, so for the purposes of demonstration, we will just output them to the screen. Since mom and dad are private, we need a function to handle the output, too. That is easy enough – we can write an output_mom() and an output_dad() function. Use the following code for these functions (also in the public section of the simulation_engine class):

    void output_mom()
    {
    std::cout << "Mom:\n";
    std::cout << "Name:\t" << mom.name;
    std::cout << "Age:\t" << mom.age;
    std::cout << "Height:\t" << mom.height;
    std::cout << "Weight:\t" << mom.weight;
    }

    void output_dad()
    {
    std::cout << "Dad:\n";
    std::cout << "Name:\t" << dad.name;
    std::cout << "Age:\t" << dad.age;
    std::cout << "Height:\t" << dad.height;
    std::cout << "Weight:\t" << dad.weight;
    }

So far, so good, but we need to implement something in our main program to make use of these classes. Navigate back to your source file (“Chapter5example1.cpp”) and add a new include statement:

    #include "simulation_engine.h"

Within the main body of the program, enclosed by int main() and its associated braces, add the missing code to make your program look like this:

    #include <iostream>
    #include "simulation_engine.h"

    int main()
    {
        simulation_engine my_sim;
        my_sim.set_dad("Fred", 44, 210, 69);
        my_sim.set_mom("Wilma", 44, 150, 64);

        my_sim.output_dad();
        std::cout << "\n";
        my_sim.output_mom();
    }
Perhaps this program makes perfect sense already, but in the interest of completeness, here is a rundown of what it is doing. We start, as usual, with include statements. First, we include `iostream`, which is built into C++. This header provides the tools we need for communication with the user via the standard output. Second, we include “simulation_engine.h”, which is the custom header that we just wrote. Note that the syntax is slightly different for built-in headers versus home-made ones. Within `int main()`, which specifies the main body of the program, we declare an instance of our `simulation_engine` class. Thus, `my_sim` is an object of type `simulation_engine`. You could thus have more than one `simulation_engine` declared at the same time, and they would function completely independently of one another.

![Figure 5.1: The output from our mom and dad functions.](image)

Next, we call the function `set_dad()` and provide it with four arguments. The order of these arguments has to be exactly the same as the order in which the function expects them, and you will recall that we arbitrarily set the order earlier when we wrote the class function. Conveniently, Visual Studio tells us what the function is expecting as we write our code, which should reinforce our desire to have informative variable names. We do the same thing for `mom`, and then we call our `output_dad()` and `output_mom()` functions to display the values on the screen. A call to a function always ends with a set of parentheses, even when the function takes no arguments. In this case, the parentheses are empty. We output a couple of `\n` (new line) characters to make the output more readable, and we end with `return`
Understanding Arrays

One of the most important concepts of any kind of computer programming is the array. Arrays are basically lists of variables that are organized in such a way that they can be accessed easily by the computer program. For starters, we will deal with an array with a single dimension. In our simulation_engine, let us declare an array of kids for this family. Just below individual dad; type:

```cpp
individual kid[3];
```

This statement declares an array of kids. This array has three elements, or three entries in the list, and they will be numbered 0, 1, and 2. This point is very important. In C++, elements of an array are numbered starting with 0 and ending with the number of elements in the array minus one. Not all programming languages follow this convention, so it can be a source of confusion. Thus, if we declared an array of integers by typing `int groupsize[10];` we would have 10 integers, known as `groupsize[0]`, `groupsize[1]`, and so forth, up to `groupsize[9]`. If we tried to access `groupsize[10]`, we would usually get an error for writing or reading memory that has not been allocated. Such a mistake can be a very serious problem for a program.

Now that we have declared three kids, let us design a function to set their values. Rather than writing a separate function for each individual kid, maybe we should develop a general function that can set the values for any arbitrary individual, including mom and dad. Such a general function will need to take an argument that specifies the individual under consideration. We have discussed one way to pass a variable to a function in a way that allows the variable to be altered by the function. There are other ways to accomplish this task, but the one we have mentioned, and the one we will use here, is known as passing by reference.

Passing by reference involves giving the memory location of the variable to the function; the function then treats it as a normal variable and modifies the information residing in that location. Thus, after the function is over, the variable of interest will be changed in whatever way the function manipulated it. To pass a variable by reference, you use an & in front of the variable name. Whereas `int my_number` would mean you are declaring a variable of type `int`, `&another_number` means you are declaring a reference to a variable of type `int`. We can declare references to classes in the same way we declare references to any other variables. With this knowledge in mind, we can write a `set_individual()` function, which will be very similar to `set_mom()` or `set_dad()`, except with one additional argument that is a reference to an `individual`. Add the following code to simulation_engine (note that the fact the first line is split onto two lines does not matter to C++, as the compiler ignores white space, including line breaks):

```cpp
void set_individual(individual &ind, std::string ind_name, int ind_age,
double ind_weight, double ind_height)
{
    ind.name = ind_name;
    ind.age = ind_age;
    ind.weight = ind_weight;
    ind.height = ind_height;
}
```
We also need a function to output an arbitrary individual, which will be similar to `output_dad()` or `output_mom()`, except that it needs to accept an argument for the identity of the individual. Here is one way to accomplish this goal:

```cpp
void output_individual(individual &ind) {
    std::cout << "\Name: \t" << ind.name;
    std::cout << "\Age: \t" << ind.age;
    std::cout << "\Height: \t" << ind.height;
    std::cout << "\Weight: \t" << ind.weight;
}
```

With these functions implemented, now we need to change the code in the main program to use the new functions. We will leave the original “mom” and “dad” functions, because they work, but now they are somewhat redundant. In the main program, add code to set values for the kids, which could go just below the calls to `set_dad()` and `set_mom()`.

Here we encounter a problem, however. From our main program, we do not have access to the variables `mom`, `dad` or `kid`. They are private. We could solve this problem by making the variables public. This workaround is probably the easiest but also represents a somewhat sloppy programming practice. If we are going to make everything public, then why do classes even have private variables? They must be there for a reason. The reason is to hide them a little so you definitely know it when you change them. It keeps the code cleaner, safer and more manageable. It does require a little more programming effort, however.

The other alternative, which allows us to keep the variables private, is to write a separate function to use the `set_individual()` function specifically on kids. This function should also go in the `simulation_engine`.

```cpp
void set_kid(int whichkid, std::string kid_name, int kid_age, double kid_weight, double kid_height) {
    set_individual(kid[whichkid], kid_name, kid_age, kid_weight, kid_height);
}
```

We need to do the same thing to output the values for kids. Here’s the function to do that:

```cpp
void output_kid(int whichkid) {
    std::cout << "\Kid " << whichkid + 1;
    output_individual(kid[whichkid]);
}
```

All of this may seem very convoluted, but when we use the function in the main program, it will be very simple, and we will know exactly what it is doing. Modify your source file to look like this:

```cpp
#include <iostream>
#include "simulation_engine.h"

int main() {
    simulation_engine my_sim;
    my_sim.set_dad("Fred", 44, 210, 69);
```
CHAPTER 5

```cpp
my_sim.set_mom("Wilma", 44, 150, 64);
my_sim.set_kid(0, "Joe", 4, 35, 40);
my_sim.set_kid(1, "Marge", 6, 44, 46);
my_sim.set_kid(2, "Kelly", 8, 65, 50);
```

```cpp
my_sim.output_dad();
std::cout << "\n";
my_sim.output_mom();
std::cout << "\n";
my_sim.output_kid(0);
std::cout << "\n";
my_sim.output_kid(1);
std::cout << "\n";
my_sim.output_kid(2);
std::cout << "\n";

return 0;
```

Note that we access the elements of the kid array with `kid[whichkid]`. Thus, the first kid in the array is `kid[0]`, the second is `kid[1]`, and the third is `kid[2]`. We can use a variable to access the kids, too, so if we say `i = 2`, then access `kid[i]`, we will get `kid[2]`. Run the program, and you will see the information for the kids and parents appear in the console window.

Another Array Example

Suppose we would like to make a list of names. At the very least, we might want to output the names to the screen. Close your current project and create a new, empty project in Visual Studio ("Chapter5example2"). Be sure to change the “Character Set” property to “Not Set”. Add a new source file, and type the following code:

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name[6];
    size_t i;
    name[0] = "Mike";
    name[1] = "Sarah";
    name[2] = "Fred";
    name[3] = "Carrie";
    name[4] = "Ted";
    name[5] = "Julie";

    for (i = 0; i < 6; i++)
    {
        std::cout << name[i] << "\n";
    }

    char end_it;
```
First, we declare an array of strings called name, which has six elements. In other words, it is a list of six strings. We also declare a variable, i, of type size_t. The size_t variable type is an unsigned integer type commonly used as an iterator in C++ programs. We could also have declared int i; and the program would still have worked. Next, we set each name to a value, which in this case must be a string indicated by quotation marks. Finally, we use a for loop to iterate through the array and output each name followed by a newline character. This loop should make perfect sense. The first time through the loop, i has a value of 0, so name[0] (“Mike”) is output. The next time through the loop, the value of i is 1, so we see name[1], and so forth. Note that the final time through the loop, i has a value of 5, so name[6], which does not exist, is never accessed. That is good, because an attempt to access name[6] would cause the program to crash. However, after the loop is over, i has a value of 6. This value occurs because the i++ statement increases the value of i by 1 at the end of each iteration of the loop, and i < 6 is evaluated at the beginning of each time through the loop. When i = 6, at the end of the sixth run through the loop, then i < 6 evaluates to false and the loop stops running. To keep it simple, just remember that for any loop with the format for (i = 0; i < n; i++), the loop will run a total of n times and the value of i will range from 0 to n-1 while the loop is running.

To be sure an understanding of arrays and loops is firmly entrenched in your mind, let us add some additional features to this simple program. For each individual, we will add a random phone number, including an area code, prefix and number. To keep the example simple, we will use one of the built-in random number generators included with C++. This random number generator should not be used for most scientific applications. The Visual Studio implementation of C++ actually does include a number of random number generators that would be suitable for scientific applications, if you include the proper header files, but we are not going to discuss them here. Add the following include statement:

```cpp
#include <time.h>
```

Also declare the following arrays (under size_t i;):

```cpp
int area_code[6];
int prefix[6];
int number[6];
```

Next, add the statements to seed the random number generator and to randomly set the phone numbers:

```cpp
std::srand(time(NULL));
for (i = 0; i < 6; i++)
{
    area_code[i] = rand() % 900 + 100;
    prefix[i] = rand() % 900 + 100;
    number[i] = rand() % 9000 + 1000;
}
```

The first statement here just seeds the random number generator (and generates a warning from the compiler, which you can safely ignore in this case). The loop calls rand() three times per iteration to generate three numbers, one for the area code, one for the prefix, and one for the rest of the number. Here we are using the % or modulus operator, which returns the remainder from the division of two integers.
CHAPTER 5

The **rand()** function returns a pseudorandom number between 0 and **RAND_MAX**, as we have seen before. The number returned by **rand() % 900** can be anything from 0 to 899. The remainder cannot be larger than the number by which you are dividing, for obvious reasons. Then we also add 100 to the number to get a number between 100 and 999. This choice of range is just a matter of convenience – the numbers print out better if they are three digits in the case of the area code and prefix, and for digits in the case of the rest of the number. The logic behind the **number[i]** statement is the same, except now we are going for a four-digit number.

The final step in this project is to output the phone numbers, a task we can accomplish by modifying the **std::cout** statement in the loop we used to output the names. Change this statement to the following:

```cpp
std::cout << name[i] << "\t(" << area_code[i] << ") " << prefix[i] << "-" << number[i] << "\n";
```

Above the loop, add some column labels:

```cpp
std::cout << "Name\tNumber\n";
```

All told, the new program should look like this:

```cpp
#include <iostream>
#include <string>
#include <time.h>

int main()
{
    std::string name[6];
    size_t i;
    int area_code[6];
    int prefix[6];
    int number[6];

    name[0] = "Mike";
    name[1] = "Sarah";
    name[2] = "Fred";
    name[3] = "Carrie";
    name[4] = "Ted";
    name[5] = "Julie";

    std::srand(time(NULL));
    for (i = 0; i < 6; i++)
    {
        area_code[i] = rand() % 900 + 100;
        prefix[i] = rand() % 900 + 100;
        number[i] = rand() % 9000 + 1000;
    }

    std::cout << "Name\tNumber\n";
    for (i = 0; i < 6; i++)
    {
        std::cout << name[i] << "\t(\" << area_code[i] << ") " << prefix[i] << "-" << number[i] << "\n";
    }
}
```

40
Run the program, and you should see the output shown in Figure 5.2.

![Image of program output]

Figure 5.2: The output from our practice program that uses arrays and loops. Your numbers will be different, because they are randomly generated.

It should be pretty clear that this way of accessing elements in an array can be extremely useful, especially for very long lists. In fact, without arrays, computer programs would be almost pointless. The great thing about a computer is that it can do repetitive tasks very well, even for lists containing thousands or hundreds of thousands of entries. Arrays and loops give us the ability to write fairly concise code that can perform potentially complex operations on every element of a potentially huge array. Consequently, the mastery of loops and arrays is a key skill for any programmer, regardless of the choice of programming language.

Returning to the Kid Array

After the previous example, you may conclude that our “Chapter5Example1” program could have benefitted from a loop. This conclusion is correct, of course, and the obvious place for a loop is during the procedure that outputs the kids to the screen. Close “Chapter5Example2” and open “Chapter5Example1”. In the source code, we see that each snippet of code that outputs a kid is exactly the same, except that a different integer is provided as the argument to the function. Rather than writing a separate bit of code for each kid, we can write one loop that takes care of all the kids. This change is not hugely necessary for a program dealing with three records, but imagine what a pain it would be to deal with 100 records without a loop. In the main source file, replace the calls to my_sim.output_kid() with the following code:
CHAPTER 5

for (size_t i = 0; i < 3; i++)
{
    my_sim.output_kid(i);
    std::cout << "\n";
}

This loop will now do the same thing, but with less code. It also can be generalized to any number of kids with only small modifications. Run the program, and you will see that the program has exactly the same functionality as it did before (Figure 5.3).

Figure 5.3: Output of project “Chapter5example1”, which uses classes, arrays and loops.

We have also done something a little different than we are used to in our loop here. In this code snippet, we declare the variable \texttt{i} within the loop and initialize it to 0 in a single step. This practice is common in C++, and it brings up the concept of \textit{scope}. The scope of a variable refers to where it exists and is accessible in a C++ program. If you declare a variable within a loop, then the variable will only exist within that loop. The program will discard it at the end and free up the associated memory. To see the effect of this variable’s limited scope, try setting \texttt{i} to a new value outside of the loop by adding the statement \texttt{i = 10;} somewhere outside the braces enclosing your loop. Even if you add this statement after the loop where \texttt{i} is declared, your compiler will act like it has no idea what you are talking about. This error arises because the variable simply does not exist outside the loop. In general, variables will only exist within the most immediate bit of code where they are declared. The variables for your classes only exist within your objects and die with the objects; variables declared within loops only exist within that particular loop; and so forth. If you would like to declare a \textit{global} variable, which is generally not recommended but would be visible to your entire program, declare it before the initial \texttt{int main()}

42
statement that signifies the main body of your program. One of the advantages of this scope concept, other than the savings on system resources provided by discarding variables not currently in use, is that we do not need to worry about the fact that we may have used the variable \( i \) for a loop in our source file as well as a header file. To C++, these variables, with their different scopes, are treated as completely different entities by the compiler.

**Chapter Summary**

This chapter covered a lot of ground. We gained a better understanding of *classes* and *object-oriented programming*. We also saw how important *arrays* can be and how they naturally interface with *loops*. A few miscellaneous topics arose along the way, but the most important take-away from this chapter is that arrays and loops are indispensible features of nearly any meaningful computer program.
Chapter 6. Setting Up the Simulation

At this point, we have most of the key skills we need to create an evolutionary simulation. The next steps are to make our individual class a little more useful and to learn a bit about memory management. We will start by working on our class. For an evolutionary simulation, we will need to keep track of genetic loci. If you do not know the first thing about evolutionary biology or genetics, you may find yourself lost and confused in a few minutes. However, you might be able to save yourself by reading a little about population genetics and quantitative genetics in an introductory evolution text. You do not need a deep understanding of these topics, but you do need to know a little bit about genetic variation and inheritance. Here, we are going to go ahead put together a foundation that will ultimately lead to a fairly sophisticated model describing the evolution of quantitative traits.

An Additive Genetic Model
The simplest quantitative genetic model involves traits that are determined only by additive effects. Under this scenario, each locus affecting the trait may have several alleles, and each allele has a particular effect on the quantitative trait. The trait value is determined by summing allelic effects across alleles and loci. The term “additive” refers to the fact that you just add allelic effects to get the trait value. There may also be environmental variation in the trait, which includes almost all non-genetic effects on trait values. Consider, for example, body size. While much of the variation among individuals in body size may be due to genetic effects, some of it is probably due to an individual’s diet, or nutritional environment. The variation among individuals in body size caused by diet would be considered environmental variation in the realm of quantitative genetics. Fortunately, simple environmental variance is easy to handle in the model.

We would like to design a model with two quantitative traits. The reason for this decision is that some very important features of quantitative genetics, such as genetic correlations among traits, cannot be demonstrated using a model with only one trait. Thus, to have an interesting and representative model, we need more than one trait. We will have to make decisions about how genetic loci affect the two traits. In our case, some loci will affect one of the traits, some will affect the other trait, and some will affect both traits. We will set it up so we can alter the numbers of these loci as parameters of the model. The first step, then, is to change our individual class to accommodate the genetics of the model. We will need arrays of loci, a genotypic value (i.e., the individual’s expected trait value as determined by genes), and a phenotypic value (i.e., the individual’s actual trait value after the environment alters it).

To start down this path, we will need a new project. We will build on this project throughout the rest of the book, so we can call it something like “QGmodel”. You know the routine by now: Open Visual Studio, select **File->New->Project...**, create an empty Visual C++ project with the appropriate name, and change its properties so that “Character Set” is “Not Set”. You will need to add a source file, so right click on “Source Files” in the Solution Explorer, and select **Add->New Item...** Add a C++ file with whatever name you want to give it (“QGmodel” would be fine).

We also need some header files, and rather than type everything in from scratch, we can use some of the ones we have already developed. Back in Windows, navigate to the folder for “Chapter5example1” and copy both “MTwisterFunctions.h” as well as “simulation_engine.h”. Make your way to your new “QGmodel” folder and paste them into the folder containing the source file you created a moment ago. It should be the “QGmodel” folder that is nested within another “QGmodel” folder. Back in Visual Studio,
right click on “Header Files” in the Solution Explorer, select Add->Existing Item..., and add the two header files you copied a few moments ago. You should see “MTwisterFunctions.h” and “simulation_engine.h” in the Solution Explorer under Header Files.

We will begin by working on the individual class. Double click on “simulation_engine.h” in the Solution Explorer to open it. We will not need the name, age, weight or height for our individuals, so we need to replace those variables with actual useful ones. For an evolutionary simulation, what will we need? Well, we certainly need a way to keep track of genotypes and phenotypes. Since we are dealing with diploid, sexually reproducing organisms, we will need a way to keep track of the two copies of each locus (one on each chromosome) and sex. Fortunately, these issues are easily handled by using classes and arrays. Replace everything after public: with the following:

```cpp
double Allele1trait0[50];
double Allele2trait0[50];
double Allele1trait1[50];
double Allele2trait1[50];
double Allele1both[50][2];
double Allele2both[50][2];
double Genotype[2];
double Phenotype[2];
bool Female;
```

Now the Individual class should look like this:

```cpp
class individual
{
public:
    double Allele1trait0[50];
double Allele2trait0[50];
double Allele1trait1[50];
double Allele2trait1[50];
double Allele1both[50][2];
double Allele2both[50][2];
double Genotype[2];
double Phenotype[2];
bool Female;
};
```

Some of these variables require a little explanation. The first array, Allele1trait0[50], represents an array of alleles that affect only trait zero (with two traits, it is convenient to call them trait zero and trait one). Because we are dealing with a diploid, we need two alleles per locus per individual (note that more than two different alleles could be present in the population, but each individual can have at most two of these in a diploid). We could use a number of different approaches to keep track of the two alleles in an individual, but I find it convenient to use two arrays, one array for the alleles inherited from the mother and one array for the alleles inherited from the father. Thus, the Allele1trait0[50] array represents the maternal allele at each of 50 loci affecting trait zero, and the Allele2trait0[50] array represents the paternal allele at each of 50 loci affecting trait zero. Consequently, an individual’s genotype at the first locus affecting trait zero would be given by Allele1trait0[0] and Allele2trait0[0]. The next two arrays, Allele1trait1[50] and Allele2trait1[50], are basically the same, except for trait one instead of trait zero.
The next two arrays require more explanation. These arrays represent the pleiotropic loci. In this case, each allele has effects on both traits, so we need to keep track of both allelic effects and make sure they are inherited together. The first part of the array name, Allele1 or Allele2, indicates whether the allele was inherited from the mother or father. The next part, both, indicates that these loci affect both traits. Then there are two numbers in brackets, 50 and 2. This type of array is known as a multidimensional array (in this case two-dimensional), and it can be interpreted as a table instead of a simple list. In our case, the first number indexes the locus, and the second number indexes the effects of the allele by trait. For example, Allele1both[9][1] would give the effect of the maternally inherited allele at locus 9 (i.e., the 10th locus because they are numbered starting with 0) on trait one. Because these loci are pleiotropic, this allele would also have an effect on trait zero, which would be given by Allele1both[9][0]. The logic behind the multidimensional array may not be entirely clear at this point, but it should become more obvious as we actually use it.

The last variable we have added introduces a new variable type. The bool operator declares a variable of type Boolean, which can have the values “true” or “false”. In our case, we can use this type of variable to keep track of sex by declaring the variable bool Female; Thus, if Female is true, the individual is a female and otherwise the individual is a male.

Some Points about Pointers

To make this program work the way we want it to, we need to introduce another important C++ concept known as a pointer. A pointer merely points to a place in memory where variables are stored. It is up to you to specify the variables that are stored there, so for our purposes pointers will work like regular variables, except that they will be declared a little differently than the ones we have been using so far. You might wonder why we should bother with pointers if they turn out to be functionally identical to ordinary variables. In addition to lots of advantages of pointers beyond the scope of this book, we need pointers to allocate memory in a way that will work with our potentially very large arrays of individuals and genotypes.

From your program’s standpoint, not all memory is created equally. When you type int i;, for instance, the program allocates memory from the stack, a limited supply of memory that is at the immediate disposal of the program. However, when you allocate memory to a pointer, the memory comes from the free store, which includes all of the other memory that exists in your computer, including hard drive space if physical memory gets used up (depending on how your operating system manages memory). Hence, the stack provides a limited supply of memory, whereas the free store is practically a bottomless pit of memory. A program can still run out of memory, but by using the free store we can have access to a much, much larger chunk of memory than is available on the stack.

We will want arrays of individuals to represent adults and their progeny in our population. There could be many such individuals, and each individual has several reasonably large arrays of alleles, so the individuals potentially take up a lot of memory. This type of array, then, is a candidate for a pointer. How do we declare a pointer that will be available for the entire life of the simulation? This process is slightly involved, but the following steps will walk you through the process. The first step is to declare the pointer in our simulation_engine class. Where your program now declares mom, dad and kid, change it to declare the following:

```cpp
individual *adult;
individual *progeny;
int NadultMax, NprogMax;
```

This step declares two pointers to a variable of type individual. The asterisk before the name of the variable tells the compiler that adult and progeny are pointers. However, we have not yet specified the
number of elements in the array. Currently, these pointers do not point to anything, so we will eventually have to allocate memory for them. One advantage of pointers is that they permit dynamic memory allocation, so in principle you could get information from the user and use that information to inform how many adults and progeny you need in your program. We have also declared two integers, which we will use to keep track of the maximum allowable number of adults and progeny in our simulation.

Where should we initialize our pointers? Since we will want the arrays to be available to the entire program, it would make sense to do it in the constructor for the simulation_engine (remember the constructor?). Because the constructor function runs every time you declare a new simulation_engine object, this function is a good place to initialize our simulation. You might recall that we already have some code in our constructor. Currently, we seed our random number generator and set the values of a handful of parameters for the model in the constructor. Add the following code to the constructor for the simulation_engine class:

```cpp
// Set the maximum number of adults and progeny in the population
// Allocate memory for the pointers for the adults and progeny
NadultMax = 2000;
NprogMax = 10000;
adult = new individual[NadultMax];
progeny = new individual[NprogMax];
```

The `new` operator allocates a block of memory large enough to house the number of elements indicated in the brackets for the variable type indicated. In this case, we use two variables to set the maximum number of adults and progeny that we will allow. We will have to be careful not to write past the ends of these arrays, because doing so would be disastrous to the program. By using variables like `NadultMax` to keep track of the sizes of our arrays, we can easily increase the number of adults or progeny in the future by just changing the values of these variables. Here is what your constructor should look like now:

```cpp
simulation_engine() // Constructor: Initialize parameters and variables here
{
    srand(static_cast<int>(time(NULL)));
    sgenrand(rand());

    // Set the maximum number of adults and progeny in the population
    // Allocate memory for the pointers for the adults and progeny
    NadultMax = 2000;
    NprogMax = 10000;
adult = new individual[NadultMax];
    progeny = new individual[NprogMax];

    NumberOfGenerations = 10;
    PopulationSize = 100;
    MutationalCorrelation = 0.5;
    SelectionalCorrelation = 0.2;
}
```

One more step is necessary for the creation of pointers to arrays. Every time you use `new` to allocate memory, you need to use the `delete` command to free up the memory when you are done with it. Otherwise you will have a `memory leak`, a situation in which a program keeps allocating memory without freeing any up. The ultimate outcome of a memory leak is to use up all the system resources and crash the computer, so memory leaks can be very bad. One of the causes of instability of earlier versions of
CHAPTER 6

Windows, for example, was probably the existence of various memory leaks. Before you put in a delete statement, you have to understand the constructor’s shadowy nemesis.

As we have seen before, the evil counterpart of the constructor is the destructor. It is not actually evil, of course, and it also is not as destructive as it sounds. Every class has a destructor, whether or not it is explicitly declared. The destructor is the function that runs when an object is done being used. In other words, when the object goes out of scope and needs to be discarded, the destructor is called. For our class, then, the destructor is an ideal place to free up the memory associated with our pointers.

A destructor has the same name as the class, like the constructor, except the destructor is preceded by a ~. To write a custom destructor, then, we can use the following syntax:

```cpp
~simulation_engine()
{
    delete[] adult;
    delete[] progeny;
}
```

Add this destructor code to your simulation_engine class, immediately after the end of the code for the constructor. The brackets [] affiliated with the delete command are necessary when deleting an array, as they tell the compiler to free up the memory for all of the elements of the array. Pointers have a lot more utility in C++ than we will cover here, so it might be worth reading up on them in a C++ manual. However, for our purposes, we will use them just like any old variable, but with a more complicated declaration.

Now we have added to and altered several aspects of the program. In fact, we changed it enough that it will no longer compile and run. Go ahead and delete all of the functions other than the constructor, destructor, and display_parameters(). Your simulation_engine header file should look like this:

```cpp
#pragma once
#include "MTwisterFunctions.h"
#include <time.h>
#include <string>
#include <iostream>

class individual
{
public:
    double Allele1trait0[50];
    double Allele2trait0[50];
    double Allele1trait1[50];
    double Allele2trait1[50];
    double Allele1both[50][2];
    double Allele2both[50][2];
    double Genotype[2];
    double Phenotype[2];
    bool Female;
};

class simulation_engine
{
private:
    individual *adult;
    individual *progeny;
    int NadultMax, NprogMax;
    int NumberOfGenerations;
```
SETTING UP THE SIMULATION

```
int PopulationSize;
double MutationalCorrelation, SelectionalCorrelation;

public:
simulation_engine() // Constructor: Initialize parameters and variables here
{
    srand(static_cast<int>(time(NULL)));
    sgenrand(rand());

    // Set the maximum number of adults and progeny in the population
    // Allocate memory for the pointers for the adults and progeny
    NadultMax = 2000;
    NprogMax = 10000;
    adult = new individual[NadultMax];
    progeny = new individual[NprogMax];

    NumberOfGenerations = 10;
    PopulationSize = 100;
    MutationalCorrelation = 0.5;
    SelectionalCorrelation = 0.2;
}

~simulation_engine()
{
    delete[] adult;
    delete[] progeny;
}

void display_parameters()
{
    std::cout << "\nNumber of Generations: \t" << NumberOfGenerations;
    std::cout << "\nPopulation Size: \t" << PopulationSize;
    std::cout << "\nMutational Correlation: \t" << MutationalCorrelation;
    std::cout << "\nSelectional Correlation: \t" << SelectionalCorrelation;
}
```

We should make sure our code actually compiles, so type the code below into your source file ("QGmodel.cpp").

```
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    simulation_engine my_sim;

    char end_it;
    std::cout << "\n\nEnter any character to exit...";
    std::cin >> end_it;
    return 0;
}
```
Compile and run the program. It does not do anything, other than open a console window and wait for you to enter a character, but this exercise allows you to check whether or not the program has any major problems that will prevent it from compiling.

**Individual-Based Quantitative Genetics**

Now would probably be a good time to think about what we would like to accomplish with our model. If you had been developing a model to address a scientific question, rather than as a learning exercise as you are doing with this book, you would have addressed this problem long ago, before you began writing any code. So far we have been putting elements in place in our model that are pretty general, and might be useful for a wide array of models addressing almost any topic in evolutionary biology. We are at a point now, however, where our program will start to specialize, so we need to have a roadmap to follow. In this particular case, we are building a model of quantitative genetics, so at the minimum, our model needs to at least simulate the life cycle of our focal organism. We conveniently already have variables for population size and the number of generations, two parameters that we will certainly need as we move forward. We should consider what other variables and parameters we should add to our model.

For convenience, we will assume that our diploid organism has non-overlapping generations, so the life cycle will start with progeny, the progeny will grow up to be adults, the adults will mate and produce progeny, and then the adults will die. The whole cycle will start over again with this new set of progeny. At some point in the life cycle, we will also impose natural selection, so an individual’s probability of surviving to adulthood might depend on its trait values. In addition, we will want to include mutation, so some gametes will carry alleles that differ to some degree from the parental alleles. All of this sounds a little daunting, but we can handle it step by step by implementing functions for each feature of the life cycle in our simulation_engine class.

---

**Box 6.1: Summary of Steps for Adding a Parameter**

It will often be necessary to add parameters to the model, so it is worth having an easily accessible list of the admittedly few steps involved. Here are the steps:

1. **Declare the variable in the** `private:` **section of the** `simulation_engine` **class.** This variable will only be accessible through `simulation_engine` functions.

2. **Initialize the variable in the constructor for the class, which is a function with the same name as the class:** `simulation_engine()` **in this case.** Every time you declare a new `simulation_engine` object, this constructor will run and initialize the variables.

3. **If any of your variables are pointers with memory allocated using the** `new` **operator, use** `delete` **to free up the memory in the destructor function:** `~simulation_engine()`. Any time you use `new` in the constructor, you need a `delete` in the destructor to go with it. In general, you will be using `new` to create arrays (as in `int *my_int = new int[10];`), so you will usually want to use `delete[]` (i.e., `delete[] my_int;`).
Just thinking about the life cycle makes it clear that we will need to add some parameters to the model. Let us contemplate the parameters that we will need. We will need parameters specifying the mutation rate, the strength of natural selection, the distribution of mutational effects, and possibly many other parameters that are not immediately obvious. Each time we encounter a new parameter or variable that we failed to anticipate at the outset, we can add it to the model as we make it more complete and realistic.

We can add some of the obvious additional variables and initialize them now. The place to declare them is at the beginning of the simulation_engine, under the keyword `private`, and they should be initialized in the simulation_engine’s constructor function. Add the following variable declarations to your simulation_engine class:

```cpp
int NumLociTrait0, NumLociTrait1; // The number of non-pleiotropic loci for each trait
int NumLociBoth; // The number of pleiotropic loci (affecting both traits)
int Fecundity; // Number of offspring each female can produce
double MutationRatePerLocus; // The per-locus mutation rate
int CarryingCapacity; // The maximum adult population size
```

We should initialize the parameters. The carrying capacity is a special case, because we want the population size to start at the carrying capacity. Rather than having to change two parameters and make sure they match, we can just set `CarryingCapacity` to the value of `PopulationSize`. Go to the simulation_engine constructor and type the following (somewhere below `PopulationSize = 100;`):

```cpp
NumLociTrait0 = 10;
NumLociTrait1 = 10;
NumLociBoth = 10;
Fecundity = 4;
MutationRatePerLocus = 0.0002;
CarryingCapacity = PopulationSize;
```

Adding a Few More Parameters that Will become Necessary

We need to add a few more parameters to specify the mutational distribution and the shape of the selection surface. Do not worry about the meaning of the variables right now, as we will go over them in some detail when we actually use them. Your program should already have declarations for `MutationalCorrelation` and `SelectionalCorrelation`, both of which are doubles. We need an additional four parameters to describe the shape of the selection surface and the distribution of the effects of new mutations. Add the following declarations to your class in the usual place:

```cpp
double MutationalVariance[2];
double SelectionStrength[2];
```

Note that these are arrays, but very short ones with only two elements each. We need a mutational variance for each trait and a selection strength for each trait, so the two elements of each array will correspond to trait 0 and trait 1.

We should also initialize these variables, so add the following code to the simulation_engine constructor:

```cpp
MutationalVariance[0] = 0.05;
MutationalVariance[1] = 0.05;
SelectionStrength[0] = 49;
SelectionStrength[1] = 49;
```
Compile and run your program to make sure it still works. It still does nothing, but we are checking here to make sure we have not introduced any bugs that prevent the program from compiling.

Now we have the parameters we need to simulate at least simple evolutionary processes. We also have a place to declare new parameters, to initialize them, and to clean up any memory we happen to allocate. You may also want to put your parameters in a more meaningful order and add some comments, especially in the area where they are initialized. This step will make it easier to find specific parameters when you want to change them, and it will also make the code more readable to other programmers and your future self. After rearranging the code a bit, here is what my simulation_engine class looks like now:

```cpp
#pragma once
#include "MTwisterFunctions.h"
#include <time.h>
#include <string>
#include <iostream>

class individual
{
public:
    double Allele1trait0[50];
    double Allele2trait0[50];
    double Allele1trait1[50];
    double Allele2trait1[50];
    double Allele1both[50][2];
    double Allele2both[50][2];
    double Genotype[2];
    double Phenotype[2];
    bool Female;
};

class simulation_engine
{
private:
    individual *adult;
    individual *progeny;
    int NadultMax, NprogMax;
    int NumberOfGenerations;
    int PopulationSize;
    double MutationalCorrelation, SelectionalCorrelation;
    int NumLociTrait0, NumLociTrait1; // The number of non-pleiotropic loci for each trait
    int NumLociBoth; // The number of pleiotropic loci (affecting both traits)
    int Fecundity; // Number of offspring each female can produce
    double MutationRatePerLocus; // The per-locus mutation rate
    int CarryingCapacity; // The maximum adult population size
    double MutationalVariance[2];
    double SelectionStrength[2];

public:
    simulation_engine() // Constructor: Initialize parameters and variables here
    {
        srand(static_cast<int>(time(NULL)));
        sgenrand(rand());

        // Set the maximum number of adults and progeny in the population
        // Allocate memory for the pointers for the adults and progeny
        NadultMax = 2000;
```

52
NprogMax = 10000;
adult = new individual[NadultMax];
progeny = new individual[NprogMax];

// Initialize the Parameters

// Demographic Parameters
NumberOfGenerations = 10;
PopulationSize = 100;
CarryingCapacity = PopulationSize;
Fecundity = 4;

// Genetic Parameters
NumLociTrait0 = 10;
NumLociTrait1 = 10;
NumLociBoth = 10;

// Mutational Parameters
MutationalVariance[0] = 0.05;
MutationalVariance[1] = 0.05;
MutationalCorrelation = 0.5;
MutationRatePerLocus = 0.0002;

// Selection Parameters
SelectionStrength[0] = 49;
SelectionStrength[1] = 49;
SelectionalCorrelation = 0.2;

~simulation_engine()
{
    delete[] adult;
    delete[] progeny;
}

void display_parameters()
{
    std::cout << "\nNumber of Generations: \t" << NumberOfGenerations;
    std::cout << "\nPopulation Size: \t" << PopulationSize;
    std::cout << "\nMutational Correlation: \t" << MutationalCorrelation;
    std::cout << "\nSelectional Correlation: \t" << SelectionalCorrelation;
}

Chapter Summary
Even though the program does not yet do much, we have covered a lot of concepts in this chapter. The important concepts from this chapter include the use of pointers, including how to allocate memory for them using new and free up the memory using delete. We also learned more about how to use a constructor to our advantage, and we implemented a destructor, a function that runs when your program is finished with an object and provides a convenient place to free up any memory we allocated for our object. We also added a number of key variables to our simulation_engine class, and these variables will hold the important parameters related to our simulation.
Chapter 7. Setting Up the Life Cycle

One of the most important components of our model is the life cycle. We have discussed it some in the previous chapter, but have yet to implement anything that simulates the lives of our hypothetical organisms. We could choose a number of different ways to start our life cycle, but the choice is largely arbitrary. Here, we will start with a population of adults, have them reproduce, and then have the resulting progeny mature to replace the original adults. Then the process is simply repeated generation after generation. It is the circle of life. We already have most elements in place, so let us continue to build upon the “QGmodel” program. Start up Visual Studio and open your project.

Output the Parameter Values

Whenever you build a simulation model, a good design feature is to have the output include the parameter values. Otherwise, the results can get separated from the parameter values, and results produced under unknown conditions are completely worthless.

In the interest of writing clean code, we should create a separate function to output the parameter values. In fact, we have already created such a function, and it is called display_parameters(). We can add the output for the remaining parameter values to the existing function. For now, we will output them to the screen, but this solution will not be a permanent one, as information displayed in the console window is not generally useful for downstream analyses. Amend display_parameters() as follows:

```cpp
void display_parameters()
{
    std::cout << "Parameter Values:\n";
    // Demographic Parameters
    std::cout << "Demographic_Parameters:\n";
    std::cout << "No_Generations: \t" << NumberOfGenerations << "\n";
    std::cout << "Initial_Pop_Size: \t" << PopulationSize << "\n";
    std::cout << "Carrying_Capacity: \t" << CarryingCapacity << "\n";
    std::cout << "Female_Fecundity: \t" << Fecundity << "\n";
    // Genetic Parameters
    std::cout << "Genetic_Parameters:\n";
    std::cout << "No_Loci_Trait0: \t" << NumLociTrait0 << "\n";
    std::cout << "No_Loci_Trait1: \t" << NumLociTrait1 << "\n";
    std::cout << "No_Loci_Pleiotrop: \t" << NumLociBoth << "\n";
    // Mutational Parameters
    std::cout << "Mutational_Parameters:\n";
    std::cout << "Mut_Var_Trait0: \t" << MutationalVariance[0] << "\n";
    std::cout << "Mut_Var_Trait1: \t" << MutationalVariance[1] << "\n";
    std::cout << "Mut_Correlation: \t" << MutationalCorrelation << "\n";
    std::cout << "Mutation_Rate: \t" << MutationRatePerLocus << "\n";
    // Selection Parameters
    std::cout << "Selection_Parameters:\n";
    std::cout << "Omega_Trait0: \t" << SelectionStrength[0] << "\n";
    std::cout << "Omega_Trait1: \t" << SelectionStrength[1] << "\n";
    std::cout << "Selection_Corr: \t" << SelectionalCorrelation << "\n";
}
```
As a reminder, the keyword `void` indicates that this function does not return a value, and the fact that the parentheses after the function name are empty means that this function also accepts no arguments. The rest of the function should be self-explanatory (`\t` is a tab character and `\n` is a newline character).

![Image of the output of the display_parameters() function]

Figure 7.1: The output of the `display_parameters()` function, which shows the default parameter values set by the simulation_engine constructor. A good practice is to get used to avoiding the use of the space character. Use the underscore instead to save yourself many future headaches.

Now we can modify our source code to call this function, and we should see our initial parameter values, as set in the class constructor, appear in the console window. First, we have to declare an object of the simulation_engine class and then we merely call the `display_parameters()` function. Add or delete code as necessary to obtain the following source file:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    simulation_engine my_sim;
    my_sim.display_parameters();

    char end_it;
    std::cout << "\n\nEnter any character to exit...";
    std::cin >> end_it;
    return 0;
}
```

Compile and run your program. You should see output that looks something like that shown in Figure 7.1.

**Initialize a Population**

We need a starting point for the simulation, so the first order of business is to set up a population of adults. We could initialize our population in the constructor for the simulation_engine, but we would run
the risk of making the constructor extremely large and awkward. Instead, we can choose to write a different function to set up the starting population. This approach has the advantage that if we want our simulation to start over or run under a different set of parameter values, we could have our program change the parameter values and then initialize a new population. If we had the population initialization in the constructor, then the population would always be initialized under the parameter values that are hard-coded into the model. This distinction matters if we wish to have our program automatically run many replicate simulations under different parameter values and save the results.

Add a new function, named `initialize_population()` to your `simulation_engine` class. You can place this function right after `display_parameters()`. One question is whether or not we want this new function to return a value when it is called. For instance, we could have it return “true” if successful and “false” if unsuccessful. In that case, the function would return a Boolean variable. Use the following code to declare this function:

```cpp
bool initialize_population()
{
}
```

By now the `simulation_engine` class is getting pretty large, requiring a fair amount of scrolling through command after command to find a particular place in the code. Fortunately, you can ask Visual Studio to hide the content of functions by clicking the small box with the minus sign next to the function name (Figure 7.2). Once a function is complete and working as intended, there is no reason to pore over the code again and again. These finished functions can just be hidden from view to reduce the eyestrain and confusion caused by huge, monolithic collections of C++ commands.
At this point in the program we have to start being very careful about certain details. One really important issue is the size of the arrays we have allocated. We have to be sure that we do not make more adults than we have space to store. Remember that we used a class-level variable, \texttt{NadultMax}, to specify the size of the array of adults. Of course, you should set \texttt{NadultMax} to be a value larger than the maximum population size in which you are interested. There are limits on how large you can make it, though, because memory is not an infinite resource. To be on the safe side, we should have the program check to make sure that the population size will never exceed the size of the array. Type the following code in your new \texttt{initialize_population()} function:

```cpp
bool good_initialization = true;
int i, j;
// Make sure the population size is not larger than the size of the adult array
if (PopulationSize > NadultMax)
{
    PopulationSize = NadultMax;
    good_initialization = false;
}
```

The first two lines declare variables we will need for this function, and the comment indicates the point of the statement that follows. The next line plus the contents of the braces represent an \texttt{if statement}. The way the \texttt{if} statement works is that the program will evaluate the statement in the parentheses. If that statement is true, then the code contained within the braces associated with the \texttt{if} statement will execute. Otherwise, nothing will happen, and the program will skip the \texttt{if} statement and move on to the next line of code. Note that the line of the \texttt{if} statement with the parentheses has no semicolon after it.

In this case, the \texttt{if} statement checks to see if the population size is larger than the array that we allocated to hold the adults (given by \texttt{NadultMax}). If the population size is too large, the program lowers it to the size of the array to prevent an error associated with accessing memory beyond the extent of the array. If this situation comes to pass, the \texttt{if} statement also sets our Boolean variable, \texttt{good_initialization}, to false, so the function can report that something went wrong. If the population size is smaller than the array, as it should be, then nothing happens and \texttt{good_initialization} remains true.

By now, you should already suspect what programming construct we will need to prepare our population of adults. We will need a loop, of course. We will loop through all of the starting adults in the population and set their values to something. How many times should the loop run? Well, it should run through once per adult, so add the following code to your function:

```cpp
// Set the starting allelic values for the adults in the population
for (i = 0; i < PopulationSize; i++)
{
}
```

While it is not necessary, I often put a comment on the closing brace (\texttt{)}} so I know the loop to which it belongs (as I have mentioned before). The Visual Studio editor also helps you keep track: if you click on a brace with your mouse, Visual Studio will highlight the other brace that goes with it. However, even simple programs often have loops nested within other loops, if statements, and other brace-using statements, so having comments after closing braces can help reduce potential confusion.

This loop will repeat a number of times equal to the value of \texttt{PopulationSize}, so we can use it to set values for each adult in the population. The next several statements will go inside this loop. We need to set genotypes for all of the individuals at all of the loci, we need to determine phenotypes, and we need
to set a sex for each individual. The genotypes will require additional loops within the i loop. These are referred to as nested loops. Why do we need nested loops? The reason is that every individual in our array of individuals has an array of alleles of its own, so for each individual we will need to have another loop that runs through its alleles.

For now, let us start with a population with no genetic variation. We will allow the genetic variation to build up through mutation. We will arbitrarily assume that the optimum for each trait is zero, so in a sense a trait value can almost be interpreted as a deviation from the optimal trait value. The absolute values do not matter as much as the relative values in this type of model. For instance, if we decided to start with a trait value of 15, we would have the same evolutionary dynamics, but traits would be scaled differently. A genetically uniform population with a mean of zero is easy to create by setting all allelic effects to zero. The following code will accomplish this goal. Type the following inside the brackets of the i loop:

```c
// Set allelic values to zero for loci affecting trait 0
for (j = 0; j < NumLociTrait0; j++)
{
    adult[i].Allele1trait0[j] = 0;
    adult[i].Allele2trait0[j] = 0;
} // end of j
```

This loop takes care of the starting allelic values for trait 0. We need to do the same thing for trait 1 and the pleiotropic loci. The code below will finish off the loci. Notice that we can reuse j, provided that the previous loop using it is finished with it. However, we cannot use i for a loop within the i loop, because the nested loop would change the value of i, causing the higher level loop to either run forever or end prematurely. We could, however, reuse i later in the program after the i loop ends. We can also use i in other functions, because this particular instance of i only exists within the initialize_population() function, and it will be destroyed when the function ends.

The next j loop will be just like the first one, except for trait 1 instead of trait 0. However, the loop for the pleiotropic loci is a little more complicated, because each allele affects both traits (the very definition of pleiotropy). Type the following code within the braces for the i loop:

```c
// Set allelic values to zero for loci affecting trait 1
for (j = 0; j < NumLociTrait1; j++)
{
    adult[i].Allele1trait1[j] = 0;
    adult[i].Allele2trait1[j] = 0;
} // end of j

// Set allelic values to zero for pleiotropic loci
for (j = 0; j < NumLociBoth; j++)
{
    adult[i].Allele1both[j][0] = 0;
    adult[i].Allele1both[j][1] = 0;
    adult[i].Allele2both[j][0] = 0;
    adult[i].Allele2both[j][1] = 0;
} // end of j
```

Do not allow yourself to be confused by the pleiotropic loci. Any given adult has two alleles, Allele1 and Allele2, because we are dealing with a diploid organism. Each allele has an effect on trait 0 (indicated by the [0] after the [j]) and an effect on trait 1 (indicated by the [1]). For these loci to behave like actual pleiotropic loci, the effects of an allele (i.e., Allele1both[j][0] and Allele1both[j][1]) must be inherited together. We will revisit this issue later.
To complete our individuals, we need to calculate their genotypes and phenotypes. Our model is additive, so the genotype is simply calculated by summing allelic effects across loci. Obviously, these values will come out to 0, but we should write the code to sum across loci in case we decide to change the starting values of alleles later. Of course, summing across loci requires more loops.

This procedure is also the kind of repetitive activity that lends itself to being placed in a function. Because the genotype is strictly a property of the individual, the logical place for this function to reside is within the individual class. In the `public:` region of `individual`, below the variable declarations, add:

```c++
void calculate_genotypic_values(int n_loci_0, int n_loci_1, int n_loci_both)
{
    int i;
    Genotype[0] = 0;
    for (i = 0; i < n_loci_0; i++)
        Genotype[0] = Genotype[0] + Allele1trait0[i] + Allele2trait0[i];
    for (i = 0; i < n_loci_both; i++)
        Genotype[0] = Genotype[0] + Allele1both[i][0] + Allele2both[i][0];

    Genotype[1] = 0;
    for (i = 0; i < n_loci_1; i++)
    for (i = 0; i < n_loci_both; i++)
}
```

This function starts with the keyword `void` because it does not return a value. Rather, it sets the values of variables that already reside within the individual class. The function also takes three arguments, because the individual class does not have access to the numbers of the various types of loci. Consequently, we have to pass those values as arguments to the function. This constraint causes some overhead, as the program has to create the variables and copy the values to the correct memory location every time the function is called. We might want to eliminate this step eventually if we decide to optimize our code to make it run faster. For now, however, we will not worry too much about it. You also might notice that this function uses the variable `i`, which is also used in the loop containing the calls to the function. This approach is fine, because the variables in the function are specific to that function and consequently have no effect on variables declared elsewhere in the program. The function, in fact, does not have the ability to access variables from other functions or from the main program unless you explicitly give it that ability.

The code within this function uses a fairly standard way to sum a bunch of numbers. First, we set the variable that will eventually hold the sum to zero, a goal accomplished by the `Genotype[0] = 0;` statement. Then, we loop through the list of values, adding a value to the sum during each iteration of the loop.

A couple of points about these loops are worth making here. First, if a loop has only one statement, it does not require braces. If the braces are left off, the loop will only complete the first statement (up to the first semi-colon) during each run through the loop. Second, these statements use a programming convenience that actually makes no sense from a mathematical point of view. In C++ (or essentially any other programming language), the expression `x = x + y;` is interpreted to mean that the new value of `x` is set to the original value of `x` with `y` added to it. Thus, if `x` had the value of 4, and you used the command `x = x + 3;`, the new value of `x` would be 7. Clearly, the mathematical expression `x = x + 3` is stupid, but this way of manipulating variables in a programming language is extremely convenient. We use this approach to add up the genotypic value for an individual by sequentially adding all of the allelic effects to the genotype. Also note that for `Genotype[0]`, we have two sets of loci: the loci that only
affect trait 0 and the pleiotropic loci. For the pleiotropic loci, we are just using the allelic effect on trait 0, which is indicated by the value in the last bracket.

With this new function in hand, we should use it in our loop that initializes the population. After setting all of the allelic values to zero, still within the i loop in our initialize_population() function, add the following call to our new function:

```cpp
adult[i].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
```

We are finally getting close to wrapping up our population initialization function. The last step in initializing the adults is to add environmental variance to the genotypic value to obtain a phenotype. For now, we will use an environmental variance of 1, but we may want to change this value at some later date. Let us declare a variable for environmental variance. It turns out that our random number generator uses the standard deviation, which is the square root of the variance, so it would be convenient to place that value in a variable as well. In the public: section of simulation_engine, add the following declarations:

```cpp
double EnvironmentalVariance[2];
double EnvironmentalStDev[2];
```

Also add statements to output the environmental variance in display_parameters(), and to initialize these variables in the constructor. If we change “Genetic Parameters” to “Quantitative Genetic Parameters” in our display_parameters() function, then we can put our new variables under that heading. The environmental variance is not a genetic parameter per se, but it is a major determinant of quantitative trait values. Change that section of code as follows:

```cpp
// Quantitative Genetic Parameters
std::cout << "Quantitative_Genetic_Parameters:\n";
std::cout << "No_Loci_Trait0: \t" << NumLociTrait0 << "\n";
std::cout << "No_Loci_Trait1: \t" << NumLociTrait1 << "\n";
std::cout << "No_Loci_Pleiotrop:\t" << NumLociBoth << "\n";
std::cout << "Env_Variance_Trt0:\t" << EnvironmentalVariance[0] << "\n";
std::cout << "Env_Variance_Trt1:\t" << EnvironmentalVariance[1] << "\n";
```

There is no need to output both the environmental variances and the environmental standard deviations, because the latter is the square root of the former. In the class constructor, add the following statements to initialize the values:

```cpp
EnvironmentalVariance[0] = 1;
EnvironmentalVariance[1] = 1;
EnvironmentalStDev[0] = sqrt(EnvironmentalVariance[0]);
EnvironmentalStDev[1] = sqrt(EnvironmentalVariance[1]);
```

This code obviously sets the environmental variances to 1 and the environmental standard deviations to the square root of the variances (in this case also 1). Compile and run the program. It should compile without errors or warnings and should now display all of the parameter values, including the new environmental variances in the console window.

The next step to prepare our population for prime time is to calculate each individual’s phenotype, which simply involves adding the environmental variance to the genotypic values for each trait. In the interest of complete transparency and practice using functions, we will add another function to calculate the phenotype within our individual class. This approach might not yield the fastest possible code, as we
will be calling a couple of functions per individual every single generation. We can revisit this issue later if we find our program to be prohibitively slow. Add the following function to the individual class:

```cpp
void calculate_phenotype(double env_st_dev_0, double env_st_dev_1)
{
    Phenotype[0] = Genotype[0] + randnorm(0, env_st_dev_0);
    Phenotype[1] = Genotype[1] + randnorm(0, env_st_dev_1);
}
```

This function requires two arguments, the environmental standard deviations for the two traits, because these variables reside in the simulation_engine class rather than in `individual`. The function then simply calls `randnorm()`, which is part of our “MTwisterFunctions” header, twice and adds the values returned by `randnorm()` to the genotypic values that are calculated in `calculate_genotypic_values()`. The `randnorm()` function takes two arguments, a mean and a standard deviation. It then returns a random value drawn from a normal distribution (also known as a Gaussian distribution) with the specified mean and standard deviation. In other words, if we drew many such numbers and made a histogram, their frequencies would be a bell-shaped normal distribution, with the specified mean and standard deviation. The number returned by `randnorm()` is added to the genotypic value to simulate environmental effects on the trait value.

We need to call our new function in our `initialize_population()` function, and it is important to make this call after we have already calculated the genotypes. Below the statement that calls `calculate_genotypic_values()`, add:

```cpp
adult[i].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
```

The final touch is to determine the sex of each individual. We can add yet another function in `individual` to perform this procedure, and we can use a random number to determine sex. This function will do the trick:

```cpp
void set_sex()
{
    if (genrand() < 0.5)
        Female = true;
    else
        Female = false;
}
```

The `genrand()` function is also defined in “MTwisterFunctions.h”. This function returns a (uniformly distributed) number between 0 and 1, not including 1, and it takes no arguments. If the number returned is less than 0.5, we will call the individual a female by setting `Female` to true. The `else` command can be used with an `if` statement to tell the program what to do if the statement in the parentheses is false. Also note that an `if` or `else` with only one line of code does not require braces. In summary, `genrand()` returns uniformly distributed random number between 0 and 1, and if the number is less than 0.5, we assign the individual’s sex as female; otherwise, the individual is male.

We can add a call to this function after the phenotype is determined in our `initialize_population()` function. The function takes no arguments and returns no values, so here is the appropriate line of code:

```cpp
adult[i].set_sex();
```
Finally, you might recall from the beginning of this section that we decided to have the function return a Boolean value, which is intended to indicate whether or not the initialization was successful. The variable `good_initialization` stores this information, so at the end of the function, we have to return its value. Before the closing brace of the function (and after the closing brace of the `i` loop), add:

```c
return good_initialization;
```

Here is the entire `initialize_population()` function:

```c
bool initialize_population()
{
    bool good_initialization = true;
    int i, j;
    // Make sure the population size is not larger than the size of the adult array
    if (PopulationSize > NadultMax)
    {
        PopulationSize = NadultMax;
        good_initialization = false;
    }

    // Set the starting allelic values for the adults in the population
    for (i = 0; i < PopulationSize; i++)
    {
        // Set allelic values to zero for loci affecting trait 0
        for (j = 0; j < NumLociTrait0; j++)
        {
            adult[i].Allele1trait0[j] = 0;
            adult[i].Allele2trait0[j] = 0;
        }
        // end of j

        // Set allelic values to zero for loci affecting trait 1
        for (j = 0; j < NumLociTrait1; j++)
        {
            adult[i].Allele1trait1[j] = 0;
            adult[i].Allele2trait1[j] = 0;
        }
        // end of j

        // Set allelic values to zero for pleiotropic loci
        for (j = 0; j < NumLociBoth; j++)
        {
            adult[i].Allele1both[j][0] = 0;
            adult[i].Allele1both[j][1] = 0;
            adult[i].Allele2both[j][0] = 0;
            adult[i].Allele2both[j][1] = 0;
        }
        // end of j

        adult[i].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
        adult[i].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
        adult[i].set_sex();
    }
    // end of i

    return good_initialization;
}
```
Now our `initialize_population()` function is ready, so we just have to invoke it from our main program after we create an object of `simulation_engine`. The initialization function returns a Boolean value, so we also need to declare a variable to accept the value. Right now, initialization only fails when the population size is larger than the maximum number of allowable adults in the associated array. However, other details may crop up, so we may need to allow other circumstances under which an initialization might fail. Anyway, we need to declare the variable, invoke the initialization function, and deal with a failed initialization. Modify your source code file to look like this:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    simulation_engine my_sim;
    my_sim.display_parameters();

    bool initialization_success;
    initialization_success = my_sim.initialize_population();
    if (!initialization_success)
    {
        std::cout << "\nSimulation Initialization Failure!\n";
        return 0;
    }

    char end_it;
    std::cout << "\n\nEnter any character to exit...";
    std::cin >> end_it;
    return 0;
}
```

The only new feature here is the exclamation point in the `if` statement. The “!” means “not”. Thus, if the statement is not true, the contents of the `if` will run. In other words, if `initialization_success` is true, then `!initialization_success` evaluates to false (preventing the `if` statement from running). We could have accomplished the same effect by using the phrase `if (initialization_success == false)`. Notice that in C++ a single equal sign (`=`) sets the value of one item to the value of another, whereas a double equal sign (`==`) evaluates whether or not two items are equal. The other small point about this section of code is that the `return 0;` in the `if` statement essentially aborts the program. Program execution ends within the main program when the first `return` statement is encountered. If the `if` evaluates to false (that is, `initialization_success` is true), this particular `return` statement will not be executed, because it is within the braces associated with the `if` statement. In this case, program execution will proceed.

Compile and run the program to make sure everything is still working. The program’s functionality will not change from the last time you ran it. It still just outputs the parameter values, but now it is getting the population ready behind the scenes. If you want to see what happens when the initialization fails, set the population size to a value larger than `NadultMax` (it will exit and close its window before you can see what is happening).

### Outputting the Adult Data

Even though our simulation will deal mainly with summary statistics, it is important as we go along to have the program output some key variables so we can be sure it is doing what we want it to. In
particular, it will be important to check to make sure that the adult genotypes look okay and that the progeny look correct when we get around to producing them down the road.

Again, this purpose is well served by the implementation of a new function. It would be nice to output the adults in a table-like format, so we might want to put some headings at the beginning of the output. We may want to number them by their order in the array. We can call this number an individual’s ID. Then we have the genotypes at all the loci, the genotypic value, the phenotype and sex. We can easily make headings for all of these things. Let us add a function to our simulation_engine class, including the statement that will output this first line of column labels. While this function will output the adults to the screen of the console window, slight modifications of it would allow us to output the same information to a file. Add the code shown below to your simulation_engine class.

```cpp
void output_adults()
{
    std::cout << "\n|ID|tSex|tTrait|tGeno|tPheno|t"
}
```

In addition to these columns, we also need columns for the individual’s locus-by-locus genotypes. Remember that we have three types of loci, so we will need three loops. We also have to think of some terminology. How about “Locus_{x}Traint_{y}” for locus “x” affecting trait “y”, and “Locus_{x}Pleio_{y}” for pleiotropic locus “x”? These loops will be simple, as they will just add some text to the same line we have already started. This output will only be useful for small numbers of loci, because when the output exceeds the width of the console window, it will spill onto a second line and become very difficult to interpret. However, if we later use this code to save the adults to a file, we will no longer face this constraint. Use the following code:

```cpp
int i, j;
for (j = 0; j < NumLociTrait0; j++)
    std::cout << "Locus_" << j << "Trait0\t";
for (j = 0; j < NumLociTrait1; j++)
    std::cout << "Locus_" << j << "Trait1\t";
for (j = 0; j < NumLociBoth; j++)
    std::cout << "Locus_" << j << "Pleio\t";
```

These loops just add additional text to the first line of the output. Each time a new bit is added, it ends with the \t character, which tells the compiler to add a tab character, as you already know. Head over to your source file and, below the area where the population is initialized, add:

```cpp
my_sim.output_adults();
```

Compile and run the program. We can see immediately that this output will be unintelligible, because it spills onto five lines! Thus, to examine the output for the adults, and eventually the progeny, we should reduce the number of loci to one per type. Go to the simulation_engine class constructor and change the number of loci from 10 to 1 for each type of locus. Now compile and run the program – the output should look quite a bit cleaner.

The next step is to add two lines per adult in the population. This task actually takes a fair amount of code compared to most of the things we have done so far. We will go through it in some detail here, however, because it provides good practice using loops. Make your way back to your `output_adults()` function. Add the following comment (immediately above the closing brace):

```cpp
// Output two rows for each adult
// The first row will be for trait 0
```
The next step is to add a loop. Obviously, we need to loop through all of the adults in the population, but within adults we will have to loop through the two traits (one trait for each row) and through the loci affecting the traits. Thus, our task calls for three levels of loops. One loop will be nested within another loop that is nested within a third. Keeping track of these types of routines, and knowing when they should be used, may be the most difficult part of programming, at least for the relatively simple programs with which we will be dealing. For the outer loop, we will use i, so type this:

```cpp
for (i = 0; i < PopulationSize; i++)
{
}
```

Since we need three loops, we will need one more integer. Scroll up to the beginning of the function and change `int i, j;` to `int i, j, k;`. We have been using j to loop through loci, so let us maintain this convention. We will use k to loop through traits. Inside the i loop braces, type:

```cpp
for (k = 0; k < 2; k++)
{

}
```

It will be easier to keep track of where you are in the hierarchy of loops if you indent the k loop a little bit more than the i loop (Visual Studio should indent it for you automatically). The k loop will execute twice for each adult, once for each trait. We will go ahead and put the ID, sex, and other information about the adult on both lines, even though doing so is a bit redundant. Within the k loop, add the following (indented yet a little further):

```cpp
std::cout << "\n" << i << "\t";
if (adult[i].Female)
    std::cout << "female\t";
else
    std::cout << "male\t";
std::cout << k << "\t";
std::cout << adult[i].Genotype[k] << "\t";
std::cout << adult[i].Phenotype[k] << "\t";
```

Run the program, and inspect the output. It should show the ID number, whether the individual is male or female, a genotypic value and a phenotypic value. All of the genotypic values are zero, but the phenotypic values are not. This result is expected, because we set all allelic effects to zero, and the genotypic value is a sum of allelic effects across loci. The phenotypic values, however, include environmental variance, which we drew from a normal distribution with a mean of zero and a standard deviation of one. You will see that most of the values are between about -2 and 2, which is within 2 standard deviations of zero (the mean). For a normally distributed random variable, we expect about 95 percent of the values to fall within 1.96 standard deviations of the mean in either direction, so the environmental variation is consistent with our expectations.

However, the environmental variance has so many significant digits that it is going to prevent some of the values from lining up into neat columns. Presumably, the genotypes and allelic effects would have the same problem if they were not set to exactly zero. Do you recall how to fix this problem? In Chapter 4,
we used a function called `setprecision()` to format our output. This function is included in the header known as “iomanip”, so include it in your header file:

```cpp
#include <iomanip>
```

If we use `setprecision()`, it will set the number of significant digits in the output. For example, if we invoked `setprecision(3)`, then 1.03342237 would be output as 1.03. However, 0.00041134 would be 0.000411, so our problem of columns lining up would not be completely solved. If we pair `setprecision()` with another function, called `fixed`, however, we can set the number of digits after the decimal place. If we used `setprecision(4)` and `fixed`, then the numbers above would come out as 1.0334 and 0.0004, which would line up nicely. Change your lines of code that output the genotype and phenotype to the following:

```cpp
std::cout << std::setprecision(3) << std::fixed << adult[i].Genotype[k] << "\t";
std::cout << std::setprecision(3) << std::fixed << adult[i].Phenotype[k] << "\t";
```

Compile and run the program. The columns of numbers should be nice and orderly.

For the next little bit of code, we will need to know if we are on the first line for the current adult (and dealing with trait 0) or the second line (trait 1). How can we solve this problem? An `if` statement would be useful in this context. After the code to output the genotype and phenotype, still within the `k` loop, type the following:

```cpp
if (k == 0)
{
}
else
{
}
```

The code between the braces associated with `if (k == 0)` will execute the first time through the `k` loop and the code associated with the `else` will execute the second time through. This `if` statement actually brings up one peculiarity with regard to the C++ programming language, which we covered a few pages ago but should nevertheless reiterate here. It is important to distinguish between two types of equals signs in C++. A single `=` sets one variable equal to another, whereas `==` compares two values. Thus, if you use a single `=` in an `if` statement, you will be rewarded with undesirable results. The program will set the variable equal to whatever follows the `=` and evaluate the statement as true; the result will be that the code associated with the `if` will always be executed. In general, always use `==`, rather than `=`, with an `if` statement. If you use `=` by mistake, the compiler should warn you that you have a possible incorrect usage of the `=` when you compile the program.

On the first row for each adult, we want the allelic effects for trait 0. The trait 0 loci and pleiotropic loci affect trait 0, but the trait 1 loci do not. Hence, we will need to output the allelic effects at the trait 0 and pleiotropic loci, while leaving empty spaces for the trait 1 loci. On the second row for each adult, we want to do the same thing, but with the trait 1 effects output and blank spaces for the trait 0 loci. We will start by dealing with the non-pleiotropic. Add the following code to our `if` statement:

```cpp
if (k == 0)
{
    for (j = 0; j < NumLociTrait0; j++)
```
SETTING UP THE LIFE CYCLE

```cpp
std::cout << std::setprecision(3) << std::fixed << adult[i].Allele1trait0[j] << "/" << adult[i].Allele2trait0[j] << "\t";
for (j = 0; j < NumLociTrait1; j++)
    std::cout << " \t";
} // end of if (k == 0)
else
{
    for (j = 0; j < NumLociTrait0; j++)
        std::cout << " \t";
    for (j = 0; j < NumLociTrait1; j++)
        std::cout << std::setprecision(3) << std::fixed << adult[i].Allele1trait1[j] << "/" << adult[i].Allele2trait1[j] << "\t";
} // end of else

See if you can understand why this code does what it does. In each case, the first j loop deals with the trait 0 loci and the second j loop deals with the trait 1 loci. In the j loops associated with if (k == 0), we output the effect of each allele for a given locus, with the allelic effects separated by a /. In this case the second j loop just outputs some spaces and tabs so that everything lines up nicely. The code in the else part of the statement performs the same task, except that the spaces are output for the trait 0 loci and the allelic effects are output for the trait 1 loci.

It is actually a little easier to deal with the pleiotropic loci, so just below the closing brace for the else, enter the following:

```cpp
for (j = 0; j < NumLociBoth; j++)
{
    std::cout << std::setprecision(3) << std::fixed << adult[i].Allele1both[j][k] << "/" << adult[i].Allele2both[j][k] << "\t";
}

This loop will output values for the pleiotropic loci on both rows. However, when k is 0 (i.e., on the first row), the allelic effects on trait 0 will be output, because we use the [k] in Allele1both[j][k] to index the trait. The second time through the k loop (i.e., on the second row for the adult), k will equal 1, so we will be outputting the trait 1 effects. Here is what the complete output_adults() function looks like:

```cpp
void output_adults()
{
    std::cout << "\n\nID\nSex\nTrait\nGeno\nPheno\n";
    int i, j, k;
    for (j = 0; j < NumLociTrait0; j++)
        std::cout << "Locus_" << j << "Trait0\t";
    for (j = 0; j < NumLociTrait1; j++)
        std::cout << "Locus_" << j << "Trait1\t";
    for (j = 0; j < NumLociBoth; j++)
        std::cout << "Locus_" << j << "Pleio\t";

    // Output two rows for each adult
    // The first row will be for trait 0
    // The second row will be for trait 1
    for (i = 0; i < PopulationSize; i++)
    {
        for (k = 0; k < 2; k++)
```
```cpp
std::cout << "\n" << i << "\t";
if (adult[i].Female)
    std::cout << "female\t";
else
    std::cout << "male\t";
std::cout << k << "\t";
std::cout << std::setprecision(3) << std::fixed
    << adult[i].Genotype[k] << "\t";
std::cout << std::setprecision(3) << std::fixed
    << adult[i].Phenotype[k] << "\t";
if (k == 0)
{
    for (j = 0; j < NumLociTrait0; j++)
        std::cout << std::setprecision(3) << std::fixed
            << adult[i].Allele1trait0[j] << "/" <<
            adult[i].Allele2trait0[j] << "\t";
    for (j = 0; j < NumLociTrait1; j++)
        std::cout << "\t";
} // end of if (k == 0)
else
{
    for (j = 0; j < NumLociTrait0; j++)
        std::cout << "\t";
    for (j = 0; j < NumLociTrait1; j++)
        std::cout << std::setprecision(3) << std::fixed
            << adult[i].Allele1trait1[j] << "/" <<
            adult[i].Allele2trait1[j] << "\t";
} // end of else
for (j = 0; j < NumLociBoth; j++)
{
    std::cout << std::setprecision(3) << std::fixed
        << adult[i].Allele1both[j][k] << "/" <<
        adult[i].Allele2both[j][k] << "\t";
}
} // end of k loop
} // end of i loop
```

If you reduce the population size to 6 or 8, then all of the output will fit in the console window. The output should look pretty decent. If things are not lined up quite right, you can alter the number of spaces in your output or the number of significant digits for each number to make it fit. The output should look something like that shown in Figure 7.3.

Thus far, the program does not yet simulate anything. It sets up a population in anticipation of a simulation, outputs parameter values, and outputs data on the adults in the population. Someday, we might want to know about their progeny, too, but they do not yet have any progeny! This chapter illustrates that one of the most time-consuming challenges in writing a program is to build features that communicate with the user. Both inputting and outputting data in meaningful ways almost always present major challenges for any program with substantial functionality. The challenge is not that the coding for these aspects is technically difficult. Rather, it tends to be relatively straightforward but very tedious and repetitive (as we have seen here).
Chapter Summary

This chapter dealt with some of the details involved in setting up a functional simulation. Here, we learned the importance of initializing the population and communicating key details to the user. In the course of doing so, we gained additional practice with functions, if statements, and loops. We also learned the importance of not writing past the end of arrays for which we have allocated memory. Doing so will cause the program to crash. Now that we have added some key features with respect to initialization and communication, we are ready to tackle the next chapter, where our actual simulation will start to take form.
Chapter 8. More on the Life Cycle

In this chapter, we are picking up where we left off and adding some key features to the simulation model. We currently have the ability to set parameters, initialize our population of adults, and output the adults to the screen, but the program still does not do very much. In this chapter, we will continue to move toward a functional simulation model.

Initialize the Population in a More Interesting Way

If we initialize the population with all zeros for allelic effects, as we currently do, we cannot easily see what the model is doing because everyone has a genotype of zero. If we added reproduction to the model, the progeny would simply inherit zeros from their parents and nothing would change. If we start with some genetic variation, however, the population will be more interesting, and we will have an easier time seeing what is happening from one generation to the next.

Let us alter the initialization so that the population starts with a variety of alleles rather than all zeros. We can accomplish this goal by using one of our random number routines. We might imagine that allelic effects have a normal distribution, so we can use our randnorm() function to draw random numbers. But what standard deviation should we use? If the initial population is too variable at the outset, we will experience problems when we try to impose stabilizing selection. One approach is to divide the mutational variance (which we will discuss later) by the number of loci. This smaller value can then be used to set up the initial allelic effects. We will introduce some variation but not an unreasonable amount. This decision is somewhat arbitrary, so it is based on prior experience with these types of models. It has no real biological justification, however, as a population never starts de novo; rather, each generation of an organism is a product of previous generations, spanning billions of years of evolution. In our simulation, though, we have to break into the evolutionary process somewhere, without simulating the evolution of our diploid organism from a single-celled ancestor. We have to hope that these small assumptions, such as the starting levels of variation in the population, do not affect the general conclusions that can be drawn from the model. For parameter values that we find concerning, we can test this assumption by running the simulation under different starting conditions. In this particular case, previous work shows the starting variation in the population does not have a major effect, because the population quickly reaches a mutation-selection-drift quasi-equilibrium, even if the population starts with no variation.

Find the initialize_population() function in your simulation_engine header. We need to declare a few additional variables. Below int i, j; declare:

    double dIP1, allelic_std_dev_0, allelic_std_dev_1;

One important point, to which we will return later, is that any calculations involving multiplication or division should use only doubles instead of integers. If you multiply or divide integers, you may get strange results because the program will automatically round numbers to integers at unexpected times. Thus, if you want to multiply or divide an integer, you should set a temporary double to the value of the integer and use the double for the calculations. We will use dIP1 for this purpose. The number of loci affecting trait 0 includes the trait 0 loci and the pleiotropic loci, so let us set dIP1 to the sum of these
values. Then we need to calculate the mutational standard deviation and divide it by $dIP_1$. Immediately below the comment that starts with // Set the starting allelic values..., type:

```cpp
    dIP1 = NumLociTrait0 + NumLociBoth;
    if (dIP1 > 0)
        allelic_std_dev_0 = sqrt(MutationalVariance[0]) / dIP1;
    else
        allelic_std_dev_0 = 0;
```

Recall that the standard deviation is the square root of the variance. The function $\text{sqrt()}$ is a standard C++ function that returns the square root of a double. The if statement ensures that we will only perform the division if $dIP_1$ is greater than zero. Whenever you divide, you must be sure that the denominator is non-zero. If your program tries to divide by zero, it will crash. In general, you should always put this type of if statement before any division, unless you are absolutely sure that the number in the denominator has no chance whatsoever of having the value of zero. If $dIP_1$ does happen to be zero, we still need to set the value of $\text{allelic\_std\_dev\_0}$ to something, so we have an else statement to do so. All variables should be set to some value before you use them; it is not safe to assume that they will start with a value of zero. We can perform the same procedure for trait 1 with the following:

```cpp
    dIP1 = NumLociTrait1 + NumLociBoth;
    if (dIP1 > 0)
        allelic_std_dev_1 = sqrt(MutationalVariance[1]) / dIP1;
    else
        allelic_std_dev_1 = 0;
```

The final step is to use our $\text{randnorm()}$ function to set the values of allelic effects. Anywhere you see a statement like `adult[i].Allele1trait0[j] = 0;`, change the zero to the right of the $=$ to `randnorm(0, allelic_std_dev_0)` or `randnorm(0, allelic_std_dev_1)`, depending on whether you are setting the allelic effect for trait 0 or trait 1. Here is what the `initialize_population()` function should look like now:

```cpp
bool initialize_population()
{
    bool good_initialization = true;
    int i, j;
    double dIP1, allelic_std_dev_0, allelic_std_dev_1;

    // Make sure the population size is not larger than the size of the adult array
    if (PopulationSize > NadultMax)
    {
        PopulationSize = NadultMax;
        good_initialization = false;
    }

    // Set the starting allelic values for the adults in the population
    dIP1 = NumLociTrait0 + NumLociBoth;
    if (dIP1 > 0)
        allelic_std_dev_0 = sqrt(MutationalVariance[0]) / dIP1;
    else
        allelic_std_dev_0 = 0;

    dIP1 = NumLociTrait1 + NumLociBoth;
```
if (dIP1 > 0)
    allelic_std_dev_1 = sqrt(MutationalVariance[1]) / dIP1;
else
    allelic_std_dev_1 = 0;

for (i = 0; i < PopulationSize; i++)
{
// Set allelic values to zero for loci affecting trait 0
    for (j = 0; j < NumLociTrait0; j++)
    {
        adult[i].Allele1trait0[j] = randnorm(0, allelic_std_dev_0);
        adult[i].Allele2trait0[j] = randnorm(0, allelic_std_dev_0);
    } // end of j

// Set allelic values to zero for loci affecting trait 1
    for (j = 0; j < NumLociTrait1; j++)
    {
        adult[i].Allele1trait1[j] = randnorm(0, allelic_std_dev_1);
        adult[i].Allele2trait1[j] = randnorm(0, allelic_std_dev_1);
    } // end of j

// Set allelic values to zero for pleiotropic loci
    for (j = 0; j < NumLociBoth; j++)
    {
        adult[i].Allele1both[j][0] = randnorm(0, allelic_std_dev_0);
        adult[i].Allele1both[j][1] = randnorm(0, allelic_std_dev_1);
        adult[i].Allele2both[j][0] = randnorm(0, allelic_std_dev_0);
        adult[i].Allele2both[j][1] = randnorm(0, allelic_std_dev_1);
    } // end of j

    adult[i].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
    adult[i].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
    adult[i].set_sex();
} // end of i

return good_initialization;

Run the program. If you set the number of loci to one for each type of locus and the number of adults to a small value, the output will look something like that shown in Figure 8.1. Notice that the genotypic value for an individual is the sum of the allelic values affecting that trait.

Implementing a Mating System

We are starting with a population of adults, which is an arbitrary decision, but given this choice we need to proceed to the next stage of the life cycle: the production of offspring. We are simulating a sexually reproducing diploid, so individuals will have to find mates and exchange gametes to produce zygotes. While we have set up the simulation to have separate sexes, slight modifications could allow it to handle hermaphroditic organisms or almost anything else we could imagine.

A convenient way to implement mating is to add another function to our simulation_engine class. For starters, we will implement a polygynous mating system in which each female can mate with only one male but each male can mate with many females. The plan will be to cycle through the adults. If the adult is female, she will choose a mate from among the males in the population and produce offspring. If there
are no males, then the population will go extinct. We should add a comment describing the mating system. Add the following function to simulation_engine:

```cpp
void polygynous_mating()
{
    // This function implements strict polygyny.
    // Under this mating system, each female mates once, but
    // each male can mate an unlimited number of times.
    // Females choose males at random.
}
```

![Simulation output](image)

Figure 8.1: The output of the simulation program with starting allelic effects initialized to random values drawn from a normal distribution.

Eventually, we will implement mating preferences, such that females no longer mate at random, and at that point we might find that we need to limit the number of encounters per female. In other words, females might be time-limited in some cases and unable to mate if they cannot find a suitable male in time. We can add a new parameter to the model to accommodate this possibility, and we will want it to be a class-level variable. Declare an integer, in the `private:` section of the simulation_engine class:

```cpp
int MaxMatingEncounters;
```

Initialize it in the class constructor, under `// Demographic Parameters`, to a value of 500:
MaxMatingEncounters = 500;

Also add a line to your `display_parameters()` function to output the value of this variable, as shown below.

```cpp
std::cout << "Max_Mating_Enc: \t" << MaxMatingEncounters << "\n";
```

Run your program to make sure all of this new code works as intended. If all went well, you should see the value of `MaxMatingEncounters` displayed in the console window.

Now return your attention to the `polygynous_mating()` function. Attempting to mate is pointless if no males exist in the population, so we need to ensure that at least one male does exist. If no males exist, then the population will go extinct. We can have a loop quickly check to see if there are males present in the population. The following code should do the trick:

```cpp
int i, j, m;

// Check to make sure at least one male is present in the population
bool males_present = false;
for (i = 0; i < PopulationSize; i++)
{
    if (!adult[i].Female)
        males_present = true;
}
```

Recall that the ! means “not”. Thus, `if (adult[i].Female)` means the same thing as `if (adult[i].Female == true)`, whereas `if (!adult[i].Female)` means `if (adult[i].Female == false)` or `if (adult[i].Female != true)`. That last symbol might be new to you. For comparisons in C++, == means “equals”, as we have seen, whereas != means “does not equal”. In short, there are many ways to skin this cat, but the way we have done it uses the fewest number of characters, is still completely clear, and appears commonly in C++ programming. If there are no males in the population, then `males_present` will stay false, but if at least one male is present, then `males_present` will be changed to true by the if statement.

The next step is to perform the mating and production of offspring. We will need to keep track of the number of offspring produced, so declare an integer called `iPC` near where you declared `i`, `j`, and `m`, with the following:

```cpp
int iPC;
```

We need a loop to cycle through the adults. Within that loop, we only need to take action when the current adult is female and there are males present in the population (this approach actually is not the most efficient, but it is more clear and the loss of computational speed is nominal), so type the following:

```cpp
iPC = 0;
for (i = 0; i < PopulationSize; i++)
{
    if (adult[i].Female && males_present)
    {
    }
    // end of if (adult[i].Female && males_present)
} // end of i loop
```
The rest of our code that performs the actual mating and production of offspring will be within the braces associated with the `if` statement. Thus, if no males are present in the population, no mating will take place. The same thing will happen if the population has no females. This approach follows the constraints imposed by the sexually reproducing system we have decided to model. If you wanted to model a species capable of parthenogenesis or changing sex, then you would want to take a different approach. You may have noticed a new symbol here, `&&`, which stands for “and”. An `if` statement with two or more conditions connected by `&&` will only run if all conditions evaluate to true.

Notice that the mother under consideration will have the index `i`. We will use the loop to ensure that every female mates, or at least looks for a mate. We know the mother’s ID on each cycle of the loop, but we do not know the father yet. The next step is to choose the father. For now, we will use random mating. If we wished to impose sexual selection or assortative mating, any non-random mating algorithm would come into play at this point in the program. We will have to keep track of whether or not a female finds a mate, the mate’s ID (i.e., its index number in the array of adults), and the number of adults the female has encountered (since the number of encounters is potentially limited). Declare the following new variables (up near the beginning of the function):

```c
bool mate_found;
int mateID, counter, rnum;
```

We will use `rnum` to store a random number that determines which adult the female encounters. We will allow females to encounter both sexes, so it will be harder to find a mate when the sex ratio is severely skewed. Such an approach seems biologically realistic, but it may not hold for all systems. A slightly more complex algorithm could ensure that females only encounter males (and we will implement such an algorithm in a later chapter). The code below will choose a mate for the current female, and it should go within the braces of our `if (adult[i].Female && males_present)` statement.

```c
// Find a mate for this female
// Mating is random, so any male will do

mate_found = false;
counter = 0;
while (counter < MaxMatingEncounters && !mate_found)
{
    rnum = randnum(PopulationSize);
    if (!adult[rnum].Female)
    {
        mateID = rnum;
        mate_found = true;
    }
    counter++;
} // end of while
```

This bit of code requires some explanation. It is our first exposure to the `while` loop, which continues to loop until the expression in parentheses evaluates to false. Be careful with `while` loops, because their misuse can result in an endless loop in your program, which will cause the program to hang. Our `while` loop will run as long as `counter` is less than `MaxMatingEncounters` and `mate_found` is false. By setting `mate_found` to false and `counter` to zero right before the `while` statement, we ensure that this loop will run at least once. Then, as the `while` loop runs, we increase the value of `counter` by one each time through (i.e., `counter++;`), so that eventually, if no male is found, the loop will end. On the other
CHAPTER 8

hand, if a male is found, we change mate_found to true, preventing the loop from running again. Once the loop is over, execution proceeds to the next statement after the closing brace.

Each pass through the loop, we choose a new random number with a value between 0 and PopulationSize-1, which encompasses the entire array of adults. In other words, the individual encountered could be any of the adults in the population. We will interpret the female encountering herself, which will occasionally happen, as a failure to find any other individual. Since mating is random, the female will mate with any male. The if statement checks to see if the encountered individual is male. If it is a male, then we set mate_found to true and set mateID to the index of that individual in the array (i.e., rnum). When the while loop ends, two outcomes are possible. One possibility is that no suitable mate was found (i.e., the female did not encounter a male). In this case, mate_found will still be false, mateID will not have been set to a value, and counter will be equal to MaxMatingEncounters. The other possibility is that the female will have encountered a male. In this case, mate_found will be true, mateID will be set to the index in the array of the encountered male, and counter could have any value between one and MaxMatingEncounters. We need to handle both of these possible outcomes.

If the female did not find a mate (i.e., mate_found is false), we do not want to create any progeny, but if mate_found is true, we do want to create progeny for the female. In other words, the function should do nothing if mate_found is false and it should do something if mate_found is true, so we need another if statement conditioned on the value of mate_found.

Immediately after the closing brace of the while statement, we need to add an if statement containing a loop to create a certain number of progeny per female. Each female will produce Fecundity progeny. Type the following:

```c
// If a mate is found, produce progeny
if (mate_found) {
    for (m = 0; m < Fecundity; m++) {
    }
} // end of if (mate_found)
```

The production of progeny will occur within this m loop. Notice that we have loops nested within if statements and other loops. You should try to get used to this type of situation now, because it is typical. This common occurrence illustrates the importance of setting up loops in a clear, logical manner, as we are doing now. If you plan your code carefully, it should remain clear and easy to follow.

Anyway, within the m loop, we need to start producing progeny. We have already identified the mother, whose ID is given by i, and the male, whose index is mateID. Now we just have to pass alleles from those two parents to the offspring. We have three types of loci, so we should handle the inheritance one type of locus at a time. Let us start with the loci that only affect trait zero. Before we tackle this problem, however, we have to be sure that we are not creating more offspring than the array of progeny can handle. Recall that iPC is counting the number of progeny we have created. You should have set it to zero just before the start of the i loop – make sure you have that statement in the correct place. Inside our m loop, type:

```c
if (iPC >= NprogMax) 
    iPC = NprogMax - 1;
```

Since we will use iPC to index our new progeny, this if statement will prevent any progeny from having an index that is outside the array allocated for the progeny. In general, we do not want this statement to come into play, so we should run the program only under parameter values that will produce
fewer than \texttt{NprogMax} progeny. In other words, \texttt{NprogMax} should be set to a value that is larger than the maximum number of adults (given by the carrying capacity) times the fecundity. Regardless, writing beyond the end of an array is such a serious problem that we should take precautions within the program to ensure it cannot occur.

Still within the \texttt{m} loop, we need to cycle through the loci and choose one allele at random from each parent for each new offspring. Which allele the offspring gets is basically a coin flip, so we need a new variable for a random number. Up where the variables for this function are declared, add one more declaration:

\begin{verbatim}
double dRnum;
\end{verbatim}

We need yet another loop to cycle through the loci, so type the following code within the \texttt{m} loop, after the \texttt{if} statement we just added:

\begin{verbatim}
// First, let's take care of Mendelian assortment of the trait0 loci
for (j = 0; j < NumLociTrait0; j++)
{
    // The progeny needs one maternal allele and one paternal allele.
    // The maternal allele will be from adult[i] (the mother), and
    // we determine which allele with basically a coin flip. The
    // function genrand() produces a value in the range [0,1), so
    // if this number is less than 0.5 we choose one allele. Otherwise,
    // we choose the other allele.
    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele1trait0[j] = adult[i].Allele1trait0[j];
    else
        progeny[iPC].Allele1trait0[j] = adult[i].Allele2trait0[j];

    // The procedure is the same for the father, adult[mateID].
    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele2trait0[j] = adult[mateID].Allele1trait0[j];
    else
        progeny[iPC].Allele2trait0[j] = adult[mateID].Allele2trait0[j];
}
\end{verbatim}

What this code is doing should be pretty obvious, and I have included some comments to explain it further. First, we choose the maternal allele and store it as \texttt{Allele1trait0[j]} for the current progeny. We randomly choose one of the alleles at that locus from the mother by using \texttt{genrand()}, which returns a number between 0 and 1. Then we do the same thing for the father and store it in \texttt{Allele2trait0[j]}. The variable \texttt{j} indexes the locus, so the loop ultimately goes through all loci affecting trait one. By implementing inheritance in this way, every individual’s \texttt{Allele1} is always from the mother and \texttt{Allele2} is always from the father. This feature of the model does not really matter, but we need to keep it in mind if we try to do something down the road and find that it does actually matter. For the loci affecting trait one, the approach is the same:

\begin{verbatim}
// Second, take care of Mendelian assortment for the trait1 loci.
// The procedure is essentially identical to that for trait0.
for (j = 0; j < NumLociTrait1; j++)
\end{verbatim}
CHAPTER 8

{ 
    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele1trait1[j] = adult[i].Allele1trait1[j];
    else
        progeny[iPC].Allele1trait1[j] = adult[i].Allele2trait1[j];

dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele2trait1[j] = adult[mateID].Allele1trait1[j];
    else
        progeny[iPC].Allele2trait1[j] = adult[mateID].Allele2trait1[j];
} // end of j loop

Now that the loci affecting only trait 0 or only trait 1 are done, we can tackle the slightly more complicated problem of the pleiotropic loci. The logic is almost exactly the same as for the non-pleiotropic loci, except that we must take some care to make sure the allelic effects on the two traits stay together when passed to offspring. The following piece of code handles this situation:

// The pleiotropic loci are more complicated. We have to be sure to // keep the allelic effects (on the two traits) together for each allele. // Otherwise, the loci would not behave realistically like actual // pleiotropic loci.
for (j = 0; j < NumLociBoth; j++)
{  
    dRnum = genrand();
    if (dRnum < 0.5)
        {  
            progeny[iPC].Allele1both[j][0] = adult[i].Allele1both[j][0];
            progeny[iPC].Allele1both[j][1] = adult[i].Allele1both[j][1];
        }
    else
        {  
            progeny[iPC].Allele1both[j][0] = adult[i].Allele2both[j][0];
            progeny[iPC].Allele1both[j][1] = adult[i].Allele2both[j][1];
        }

    dRnum = genrand();
    if (dRnum < 0.5)
        {  
            progeny[iPC].Allele2both[j][0] = adult[mateID].Allele1both[j][0];
            progeny[iPC].Allele2both[j][1] = adult[mateID].Allele1both[j][1];
        }
    else
        {  
            progeny[iPC].Allele2both[j][0] = adult[mateID].Allele2both[j][0];
            progeny[iPC].Allele2both[j][1] = adult[mateID].Allele2both[j][1];
        }
} // end of j loop

The final steps in creating an offspring are to set the sex, calculate the genotype and determine the phenotype. We already have functions in our individual class to take care of these steps, so we merely need to invoke them after the close of the last j loop. This code will take care of these steps:

progeny[iPC].set_sex();
progeny[iPC].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
I have tagged one additional step on there to update the number of progeny we have made. At the end of the creation of each offspring, we need to be sure that we are keeping a running tally of the number of offspring we have made so we can keep track of them later (and continue to make new offspring without overwriting the existing ones).

An additional important step is to set a variable, with a class-level scope (which means it is available to the entire class), that keeps track of the current number of progeny in the population. Within this function, we used $iPC$, but we declared $iPC$ within the function, which means its scope was also within the function. In other words, when the function ends, $iPC$ will be destroyed and its associated memory will be released along with any information stored there. At the very top of the simulation_engine class, in the private: section where the other variables are declared, add a new declaration for an integer called Nprogeny, as follows:

```cpp
int Nprogeny;
```

Then just before the closing brace of the `polygynous_mating()` function, add the statement:

```cpp
Nprogeny = iPC;
```

After all of that, we should have the following `polygynous_mating()` function:

```cpp
void polygynous_mating()
{
    // This function implements strict polygyny.
    // Under this mating system, each female mates once, but
    // each male can mate an unlimited number of times.
    // Females choose males at random.

    int i, j, m;
    int iPC;
    bool mate_found;
    int mateID, counter, rnum;
    double dRnum;

    // Check to make sure at least one male is present in the population
    bool males_present = false;
    for (i = 0; i < PopulationSize; i++)
    {
        if (!adult[i].Female)
            males_present = true;
    }

    iPC = 0;
    for (i = 0; i < PopulationSize; i++)
    {
        if (adult[i].Female && males_present)
        {
            // Find a mate for this female
            // Mating is random, so any male will do

            mate_found = false;
            counter = 0;
```
while (counter < MaxMatingEncounters && !mate_found) {
    rnum = randnum(PopulationSize);
    if (!adult[rnum].Female)
    {
        mateID = rnum;
        mate_found = true;
    }
    counter++;
} // end of while

// If a mate is found, produce progeny
if (mate_found) {
    for (m = 0; m < Fecundity; m++) {
        if (iPC >= NprogMax)
            iPC = NprogMax - 1;

        // First, let's take care of Mendelian assortment of the trait0 loci
        for (j = 0; j < NumLociTrait0; j++) {
            // The progeny needs one maternal allele and one paternal allele.
            // The maternal allele will be from adult[i] (the mother), and
            // we determine which allele with basically a coin flip. The
            // function genrand() produces a value in the range [0,1), so
            // if this number is less than 0.5 we choose one allele. Otherwise,
            // we choose the other allele.
            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele1trait0[j] = adult[i].Allele1trait0[j];
            else
                progeny[iPC].Allele1trait0[j] = adult[i].Allele2trait0[j];

        // The procedure is the same for the father, adult[mateID].
        dRnum = genrand();
        if (dRnum < 0.5)
            progeny[iPC].Allele2trait0[j] = adult[mateID].Allele1trait0[j];
        else
            progeny[iPC].Allele2trait0[j] = adult[mateID].Allele2trait0[j];

    } // end of j loop

    // Second, take care of Mendelian assortment for the trait1 loci.
    // The procedure is essentially identical to that for trait0.
    for (j = 0; j < NumLociTrait1; j++) {
        dRnum = genrand();
        if (dRnum < 0.5)
            progeny[iPC].Allele1trait1[j] = adult[i].Allele1trait1[j];
        else
            progeny[iPC].Allele1trait1[j] = adult[i].Allele2trait1[j];

        dRnum = genrand();
        if (dRnum < 0.5)
            progeny[iPC].Allele2trait1[j] = adult[mateID].Allele1trait1[j];
        else
            progeny[iPC].Allele2trait1[j] = adult[mateID].Allele2trait1[j];
    } // end of j loop
}

else
    progeny[iPC].Allele2trait1[j] = adult[mateID].Allele2trait1[j];
} // end of j loop

// The pleiotropic loci are more complicated. We have to be sure to
// keep the allelic effects (on the two traits) together for each allele.
// Otherwise, the loci would not behave realistically like actual
// pleiotropic loci.
for (j = 0; j < NumLociBoth; j++)
{
    dRnum = genrand();
    if (dRnum < 0.5)
    {
        progeny[iPC]. Allele1both[j][0] = adult[i].Allele1both[j][0];
        progeny[iPC]. Allele1both[j][1] = adult[i].Allele1both[j][1];
    }
    else
    {
        progeny[iPC]. Allele1both[j][0] = adult[i].Allele2both[j][0];
        progeny[iPC]. Allele1both[j][1] = adult[i].Allele2both[j][1];
    }
    dRnum = genrand();
    if (dRnum < 0.5)
    {
        progeny[iPC]. Allele2both[j][0] = adult[mateID].Allele1both[j][0];
        progeny[iPC]. Allele2both[j][1] = adult[mateID].Allele1both[j][1];
    }
    else
    {
        progeny[iPC]. Allele2both[j][0] = adult[mateID].Allele2both[j][0];
        progeny[iPC]. Allele2both[j][1] = adult[mateID].Allele2both[j][1];
    }
} // end of j loop

progeny[iPC].set_sex();
progeny[iPC].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
progeny[iPC].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
iPC++;

} // end of m loop
} // end of if (mate_found)
} // end of if (adult[i].Female & males_present)
} // end of i loop
Nprogeny = iPC;

After implementing any major change, we should check to make sure the program is working the way we think it should. Save your changes and run the program to make sure it still works. It does not do anything new yet, but if it does not compile, then there must a mistake of some kind in the code. If an error or warning appears, go back and try to fix it. Otherwise, we can continue to the next step.

We should output some of the progeny to make sure they are getting alleles from their mothers and fathers as they should. Create a new function within the simulation_engine class called output_progeny(). Insert the code below, which is very similar to the output_adults() function, except with some small changes to make it output information from the array of progeny instead of the adults.
`void output_progeny()
{
    std::cout << "\n\nID\nSex\nTrait\nGeno\nPheno\n";
    int i, j, k;
    for (j = 0; j < NumLociTrait0; j++)
        std::cout << "Locus_" << j << "Trait0";
    for (j = 0; j < NumLociTrait1; j++)
        std::cout << "Locus_" << j << "Trait1";
    for (j = 0; j < NumLociBoth; j++)
        std::cout << "Locus_" << j << "Pleio";

    // Output two rows for each offspring
    // The first row will be for trait 0
    // The second row will be for trait 1
    for (i = 0; i < Nprogeny; i++)
    {
        for (k = 0; k < 2; k++)
        {
            std::cout << "\n" << i << "\t";
            if (progeny[i].Female)
                std::cout << "female\t";
            else
                std::cout << "male\t";
            std::cout << k << "\t";
            std::cout << std::setprecision(3) << std::fixed << progeny[i].Genotype[k] << "\t";
            std::cout << std::setprecision(3) << std::fixed << progeny[i].Phenotype[k] << "\t";

            if (k == 0)
            {
                for (j = 0; j < NumLociTrait0; j++)
                    std::cout << std::setprecision(3) << std::fixed
                        << progeny[i].Allele1trait0[j] << "/";
                for (j = 0; j < NumLociTrait1; j++)
                    std::cout << "\t";
            } // end of if (k == 0)
            else
            {
                for (j = 0; j < NumLociTrait0; j++)
                    std::cout << "\t";
                for (j = 0; j < NumLociTrait1; j++)
                    std::cout << std::setprecision(3) << std::fixed
                        << progeny[i].Allele1trait1[j] << "/";
                for (j = 0; j < NumLociBoth; j++)
                    std::cout << "\t";
            } // end of else

            for (j = 0; j < NumLociBoth; j++)
            {
                std::cout << std::setprecision(3) << std::fixed
                    << progeny[i].Allele1both[j][k] << "/";
            }
        }
    }
}
Figure 8.2: Output showing adults and their progeny, as produced by the polygynous_mating() function. The parameter values have scrolled off the top of the window. Notice that three of the adults are female, so they produce 12 total offspring (Fecundity = 4). The first four offspring are from the first female. For each locus, the allele to the left of the “/” is the maternal allele, and you can see that each offspring did indeed inherit a random allelic value from its mother. The output does not indicate who sired each offspring, but the father’s identity can be deduced easily from the genotypes. The female with ID 0 clearly mated with the male with ID 5, as her offspring inherited his alleles, too. Also notice that the allelic effects are inherited together for the pleiotropic loci, such that the mother’s effects left of the “/” for traits 0 and 1 stay together, as do those to the right of the “/”.

Now we just have to invoke these functions by adding them to our main program in the “.cpp” file. Alter your source file to look like this:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
```
#include "simulation_engine.h"

int main()
{
    simulation_engine my_sim;
    my_sim.display_parameters();

    bool initialization_success;
    initialization_success = my_sim.initialize_population();
    if (!initialization_success)
    {
        std::cout << "\nSimulation Initialization Failure!\n";
        return 0;
    }

    std::cout << "\nAdults:"
    my_sim.output_adults();
    my_sim.polygynous_mating();
    std::cout << "\nProgeny:"
    my_sim.output_progeny();

    char end_it;
    std::cout << "\nEnter any character to exit..."
    std::cin >> end_it;
    return 0;
}

Run the program. If your carrying capacity is set to a small value, like 6, and the fecundity is not too high (a value of 3 or 4 is good), then you should see output something like that shown in Figure 8.2.

Chapter Summary

In this chapter, we learned how to initialize the population to have random starting allelic values, to design a simple mating system and to produce progeny using Mendel’s laws. A major hurdle in all programs is to communicate effectively with the user, and we experienced how tedious it can be to output the large amounts of information hiding behind the scenes in any moderately complex simulation. In terms of programming skills, we learned the meaning of the && symbol (it means “and”) and the != symbol (“not equal”). We also reiterated the importance of not accessing elements past the end of an array, as such an error will cause a program to crash or otherwise behave in unexpected ways. Finally, we introduced the concept of a while loop, and gained a fair amount of experience with nested loops. In the next chapter, we will continue to flesh out our life cycle, and we will soon see actual data from the simulation.
Chapter 9. Mutation and Selection

In an evolutionary model, mutation and selection are often important parts of the life cycle. Particularly in real life, mutation is the ultimate source of genetic variation, so evolution eventually becomes impossible without this input of new variation into the system. Selection is equally important, because most of the ecologically important traits of organisms have been shaped to some degree by natural or sexual selection. Your ability to read these words and deft facility to raise a cup of coffee to your mouth are both products of natural selection. A complete model of evolution should include these phenomena. However, it is also important to realize that not all models need to include everything. For instance, if we were interested in short-term simulations of evolutionary change over a few generations, new mutations might not have any meaningful impact over such a short time frame, and we could leave them out. Similarly, if we were interested in simulating genetic drift (i.e., random changes in allele frequencies over time), we could leave selection out.

Our model will be a general model of complex trait evolution, however, so we will add subroutines to handle mutations and non-random survival. We can also add the ability to set the associated parameters to values that remove these processes, making “no mutation” and “no selection” scenarios special cases of our more general model. You will probably guess that the easiest way to add these phenomena is to add functions to our simulation_engine class.

Mutation

Start by adding a function called **mutation()** to the simulation_engine class with the following code:

```c
void mutation()
{
}
```

Now that we have the function, we need to think about the best way to implement mutations in this type of model. One way would be to cycle through every locus in every offspring generated in the **polygynous_mating()** subroutine and draw a random number between 0 and 1 for each allele. If the random number comes out to less than the mutation rate, then we assume a mutation occurred and change the allelic effects. However, this type of approach can be remarkably slow, because it requires that hundreds or thousands of random numbers be drawn every single generation. Drawing random numbers is actually one of the more time-consuming functions of the program.

A quicker option would be to determine mutations on a per-individual basis. Thus, we could draw one random number per offspring, and if the number comes out to less than $2 \times \text{(the number of loci)} \times \text{(mutation rate)}$, we assume a mutation occurred in that individual. Then, we can choose a locus and allele at random within that individual to which we apply the mutational effect. This approach is a bit of an approximation, because it does not allow offspring with two mutations, an event that could occasionally occur in a natural system. However, with small mutation rates, a vanishingly small number of offspring would have two mutations in any case, so this approximation should have virtually no effect on the dynamics of the simulation. Of course, this approach only works when the mutation rate is low, such that each individual has a probability much lower than one of carrying a new mutation at the simulated loci. We will use this latter option to ensure that our program runs at a decent clip.
Between the braces of your `mutation()` function, declare the following variables, which we will need for the mutation routine:

```c
int i;
double dRnd1, dRnd2;
int iRnd1;
double mut_rate_per_ind, total_number_loci;
int i_total_number_loci;
int mutated_locus;
double mutational_effect[2];
double mutational_std_dev[2];
```

We also need to calculate a few things to figure out which loci to mutate and how big the mutational effects will be. Type the following code below the variable declarations:

```c
mutational_std_dev[0] = sqrt(MutationalVariance[0]);
mutational_std_dev[1] = sqrt(MutationalVariance[1]);
i_total_number_loci = NumLociTrait0 + NumLociTrait1 + NumLociBoth;
total_number_loci = i_total_number_loci;
mut_rate_per_ind = 2.0 * total_number_loci * MutationRatePerLocus;
```

The first two lines calculate the mutational standard deviation by taking the square root of the mutational variance. The standard approach in the evolutionary literature is to talk about quantitative traits in terms of variances, so we enter the parameters as such. However, our random number generator requires the standard deviation. We also need to sum up the total number of loci, including those that affect trait 0, those that affect trait 1, and those that affect both traits. We actually need both a double variable and an integer variable for this value for reasons that will become apparent later, so we set `total_number_loci` to the value of `i_total_number_loci`. Finally, we need to calculate the mutation rate per individual by multiplying the mutation rate per locus (which is actually per allele per locus) times the number of loci and times two because each individual has two alleles per locus. Here, we use 2.0 instead of 2 to emphasize that we are dealing with doubles rather than integers. Also recall that if you multiply a double by an integer, the result will be rounded to an integer by dropping everything after the decimal point. In other words, if we multiplied `2 * i_total_number_loci * MutationRatePerLocus`, the result would be zero. If you want a double value at the end of a mathematical operation, always ensure that all variables involved in the operation are also doubles. This sort of bug can be very hard to track down when you finally notice that your program has been giving funny results.

To put these values to use, we need a loop that cycles through all of the offspring in the population. A simple `for` loop will do nicely:

```c
for (i = 0; i < Nprogeny; i++)
{
}
```

The rest of the code for the mutation function will be nested within this `for` loop. The next order of business is to draw a random number to see if the current offspring will be affected by a mutation, easily accomplished by adding (within the `i` loop):

```c
dRnd1 = genrand();
```
If this random number is less than the mutation rate per individual, then we should mutate one of the alleles in the current individual. Otherwise, nothing should happen. Thus, we need an **if** statement without an **else**. Below `dRnd1 = genrand();`, still inside the `i` loop, add an **if** statement of the following form:

```c
if (dRnd1 < mut_rate_per_ind)
{
}
```

We actually could accomplish the same goal in a single step with the code below, which would be a little more efficient because it would not require the use of an additional double variable. The additional overhead involved in setting a double variable equal to the random number before using it is quite small, however, so use whichever form you find more comforting. The slightly more streamlined version looks like this:

```c
if (genrand() < mut_rate_per_ind)
{
}
```

The effects of a new mutation will occur inside this statement. Once we are inside the **if** statement, we know a mutation has occurred, but we have yet to determine which locus and which allele at that locus will be affected. The total number of loci is the sum of loci affecting trait 0, loci affecting trait 1, and pleiotropic loci, so if we pick a random number between 0 and the total number of loci minus 1, then we can use this number to choose the locus targeted by the mutation. If there are 10 of each type of locus, for example, then we have a total of 30 loci. They will be numbered 0 to 9, 0 to 9 and 0 to 9. If we sum them all up and start numbering with 0, then the 30 loci will be numbered 0 to 29, which explains why we want our random number to have a maximum value of one less than the total number of loci. We have a random number subroutine that produces an integer within a specified range and it is called `randnum()`. For example, `randnum(10)` would return a pseudo-random value between 0 and 9 (including 0 and 9). Let us add a statement to this effect (the **if** statement is repeated here for clarity):

```c
if (genrand() < mut_rate_per_ind)
{
    iRnd1 = randnum(i_total_number_loci);
}
```

If this random number is less than the number of loci affecting trait 0, then the mutation should hit one of those loci. If the random number is greater than or equal to the number of loci affecting trait 0, however, then the mutation should hit one of the other classes of loci, as mentioned above. To give another example, if there are 5 loci of each type, then there will be 15 loci total. If the random number is from 0 to 4, then the mutation should affect the trait 0 loci, values of 5-9 should affect the trait 1 loci, and values of 10-14 should affect the pleiotropic loci. Remember, though, that within each class the loci are numbered 0-4, so we will need to keep this feature in mind when we index our arrays. Suppose the random number comes out to 8 in this example. Then we would know that the mutation should affect one of the trait 1 loci, but which one? Well, if the trait 1 loci are 5-9 in the list of all loci, then we need to subtract 5 (which is the number of trait 0 loci) from our random number to have it correctly index the trait 1 loci. Subtracting 5 from 8 yields 3, so trait 1 locus with the index of 3 (i.e., the fourth locus) is the one that should be mutated. Using this sort of logic, we can use three **if** statements to sort out which class of
locus is being mutated and what the index number of the affected locus should be. Add the following code right under `iRnd1 = randnum(iTotalNumberLoci);`:

```c
if (iRnd1 < NumLociTrait0)
{
    mutated_locus = iRnd1;
} // end of if -- mutation affecting trait 0 loci
if (iRnd1 >= NumLociTrait0 && iRnd1 < (NumLociTrait0 + NumLociTrait1))
{
    mutated_locus = iRnd1 - NumLociTrait0;
} // end of if -- mutation affecting trait 1 loci
if (iRnd1 >= (NumLociTrait0 + NumLociTrait1))
{
    mutated_locus = iRnd1 - (NumLociTrait0 + NumLociTrait1);
} // end of if -- mutation affecting pleiotropic loci
```

The variable `mutated_locus` is the index of the locus being mutated within the class of loci affected by the mutation. If the rationale behind this variable is still confusing, reread the previous paragraph a few times to try to make sense of it.

The final step in the mutation function is to add some code within each of these `if` statements to actually change the allelic effect of the mutated locus. In this model, we change the allelic effects by drawing a mutational effect from a Gaussian mutational distribution with a mean of zero and a mutational variance that is specified as a parameter of the model. Thus, the model has some memory in that the current allelic effect has some bearing on what the allelic effect will be after the mutation. In addition, there is no mutational bias, since positive and negative mutations are equally likely. We have enough information to decide which locus will be affected from our random number stored in `iRnd1`, which we have used to set the value of `mutated_locus`. Now we need to decide which allele in the offspring (i.e., paternal or maternal) will be affected. We also need to decide the size of the mutational effect (or effects in the case of pleiotropic loci). For the non-pleiotropic loci, the approach is basically identical for both trait 0 loci and trait 1 loci. Add the new code shown here to your `if` statements corresponding to the non-pleiotropic loci to make them look as follows:

```c
if (iRnd1 < NumLociTrait0)
{
    mutated_locus = iRnd1;
    dRnd2 = genrand();
    mutational_effect[0] = randnorm(0, mutational_std_dev[0]);
    if (dRnd2 < 0.5)
        progeny[i].Allele1trait0[mutated_locus] =
        progeny[i].Allele1trait0[mutated_locus] + mutational_effect[0];
    else
        progeny[i].Allele2trait0[mutated_locus] =
        progeny[i].Allele2trait0[mutated_locus] + mutational_effect[0];
} // end of if -- mutation affecting trait 0 loci
if (iRnd1 >= NumLociTrait0 && iRnd1 < (NumLociTrait0 + NumLociTrait1))
{
    mutated_locus = iRnd1 - NumLociTrait0;
    dRnd2 = genrand();
    mutational_effect[1] = randnorm(0, mutational_std_dev[1]);
    if (dRnd2 < 0.5)
        progeny[i].Allele1trait1[mutated_locus] =
```

88
We are using `dRnd2` to decide whether the maternal or paternal allele gets mutated. The function `genrand()` produces a random number between 0 and 1, and we use a cutoff of 0.5 to decide which allele will be mutated, which is like flipping a coin. Then we use the variable `mutational_effect[0]` to store the mutational effect on trait 0, which is drawn from a Gaussian distribution by the use of the `randnorm()` function. Recall that `mutational_std_dev[0]` is the mutational standard deviation for trait 0. The `if-else` statement adds the mutational effect to either allele 1 or allele 2, depending on the value of `dRnd2`, at the locus with the index of `mutated_locus`. The other `if` statement, for mutations affecting trait 1, is very similar, except that a different set of loci is under consideration and the mutational effects are for trait 1.

The pleiotropic loci are slightly trickier because we have to change allelic effects at both traits whenever a mutation occurs. Here is the code for that `if` statement:

```plaintext
if (iRnd1 >= (NumLociTrait0 + NumLociTrait1))
{
    mutated_locus = iRnd1 - (NumLociTrait0 + NumLociTrait1);
    dRnd2 = genrand();
    randbivnorm(mutational_std_dev[0], mutational_std_dev[1], MutationalCorrelation, mutational_effect[0], mutational_effect[1]);
    if (dRnd2 < 0.5)
    {
        progeny[i].Allele1both[mutated_locus][0] =
        progeny[i].Allele1both[mutated_locus][0] + mutational_effect[0];
        progeny[i].Allele1both[mutated_locus][1] =
        progeny[i].Allele1both[mutated_locus][1] + mutational_effect[1];
    }
    else
    {
        progeny[i].Allele2both[mutated_locus][0] =
        progeny[i].Allele2both[mutated_locus][0] + mutational_effect[0];
        progeny[i].Allele2both[mutated_locus][1] =
        progeny[i].Allele2both[mutated_locus][1] + mutational_effect[1];
    }
} // end of if -- mutation affecting pleiotropic loci
```

The main difference here is that each mutation changes two allelic effects, because each locus affects two different traits. Thus, the mutations are drawn from a bivariate Gaussian distribution with means of zero. The shape of the bivariate distribution is specified by two standard deviations and a mutational correlation. The mutational correlation can be between -1 and 1, and it specified the degree to which effects on trait 0 are non-independent of effects on trait 1. If the mutational correlation is 0.9, for example, then mutations with a positive effect on trait 0 will almost always have a positive effect on trait 1 as well. We draw effects from a bivariate normal distribution with our function called `randbivnorm()`, which requires five arguments: the mutational standard deviation for trait 0, the mutational standard deviation for trait 1, the mutational correlation, and two variables where the random values will be stored (in our case `mutational_effect[0]` and `mutational_effect[1]`). Then, we use `dRnd2` to flip a coin as
above. If the maternal allele is mutated, then the mutational effects are added to both of the corresponding allelic effects on the two traits. The same goes if the paternal allele is mutated. Do not forget that for pleiotropic loci the allelic effects on traits 0 and 1 must remain together as a unit. This constraint is necessary for the pleiotropic loci to behave properly in the model. Hence, they are inherited as a unit (whenever the effect on trait 0 is passed to an offspring, so is the effect on trait 1) and they are mutated as a unit (whenever the effect on trait 0 is mutated, so is the effect on trait 1, with the constraint that the correlation between the mutational effects is determined by the mutational correlation).

**Calculate Progeny Genotypes and Phenotypes**

When we made our progeny during the `polygynous_mating()` action, we ended it by tallying up genotypic effects across loci to determine genetic values and phenotypic values. We also determined the sex of each embryo. While the sex will still be valid after a mutation, both the genetic and phenotypic values will be incorrect. For the offspring that received a mutation, then, we need to tally genetic and phenotypic values again. We could probably think of a way to perform these steps more efficiently, but for now we are mainly concerned with getting the program running in a way that results in a functional simulation. Fortunately, we already have functions to determine genotypic and phenotypic values, so we can invoke them from within the `if` loop that checked to see if the progeny was a mutant. Here are the requisite statements:

```
progeny[i].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
progeny[i].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
```

Now we have finished crafting our mutation subroutine, and the final product follows this paragraph. Notice that I have also removed the declaration for `dRnd1` because the way I chose to use `genrand()` no longer called for it. If you declare a variable and never use it, the compiler should give you a warning. You should generally heed the warnings and try to discover why you have triggered them, because sometimes they will result from coding mistakes that will affect the behavior of your program in a serious way. If you have code that generates warnings, but you are sure it is working the way you want it to, the warnings can be ignored without affecting the functionality of the program.

```c
void mutation()
{
    int i;
    double dRnd2;
    int iRnd1;
    double mut_rate_per_ind, total_number_loci;
    int i_total_number_loci;
    int mutated_locus;
    double mutational_effect[2];
    double mutational_std_dev[2];

    mutational_std_dev[0] = sqrt(MutationalVariance[0]);
    mutational_std_dev[1] = sqrt(MutationalVariance[1]);
    i_total_number_loci = NumLociTrait0 + NumLociTrait1 + NumLociBoth;
    total_number_loci = i_total_number_loci;
    mut_rate_per_ind = 2.0 * total_number_loci * MutationRatePerLocus;

    for (i = 0; i < Nprogeny; i++)
    {
        if (genrand() < mut_rate_per_ind)
        {
```
iRnd1 = randnum(i_total_number_loci);
if (iRnd1 < NumLociTrait0)
{
  mutated_locus = iRnd1;
  dRnd2 = genrand();
  mutational_effect[0] = randnorm(0, mutational_std_dev[0]);
  if (dRnd2 < 0.5)
    progeny[i].Allele1trait0[mutated_locus] =
    progeny[i].Allele1trait0[mutated_locus] + mutational_effect[0];
  else
    progeny[i].Allele2trait0[mutated_locus] =
    progeny[i].Allele2trait0[mutated_locus] + mutational_effect[0];
} // end of if -- mutation affecting trait 0 loci
if (iRnd1 >= NumLociTrait0 && iRnd1 < (NumLociTrait0 + NumLociTrait1))
{
  mutated_locus = iRnd1 - NumLociTrait0;
  dRnd2 = genrand();
  mutational_effect[1] = randnorm(0, mutational_std_dev[1]);
  if (dRnd2 < 0.5)
    progeny[i].Allele1trait1[mutated_locus] =
    progeny[i].Allele1trait1[mutated_locus] + mutational_effect[1];
  else
    progeny[i].Allele2trait1[mutated_locus] =
    progeny[i].Allele2trait1[mutated_locus] + mutational_effect[1];
} // end of if -- mutation affecting trait 1 loci
if (iRnd1 >= (NumLociTrait0 + NumLociTrait1))
{
  mutated_locus = iRnd1 - (NumLociTrait0 + NumLociTrait1);
  dRnd2 = genrand();
  randbivnorm(mutational_std_dev[0], mutational_std_dev[1], MutationalCorrelation, mutational_effect[0], mutational_effect[1]);
  if (dRnd2 < 0.5)
    { 
      progeny[i].Allele1both[mutated_locus][0] =
      progeny[i].Allele1both[mutated_locus][0] + mutational_effect[0];
      progeny[i].Allele1both[mutated_locus][1] =
      progeny[i].Allele1both[mutated_locus][1] + mutational_effect[1];
    }
  else
    { 
      progeny[i].Allele2both[mutated_locus][0] =
      progeny[i].Allele2both[mutated_locus][0] + mutational_effect[0];
      progeny[i].Allele2both[mutated_locus][1] =
      progeny[i].Allele2both[mutated_locus][1] + mutational_effect[1];
    }
} // end of if -- mutation affecting pleiotropic loci
progeny[i].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
progeny[i].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
} // end of if
} // end of i loop
Natural Selection

Natural selection obviously is an important part of evolutionary biology, so we need to incorporate it into our model. We will assume a Gaussian fitness surface, which will have a single optimum. Fitness then drops off as individuals depart from the optimum with respect to phenotype. This implementation of selection allows for both stabilizing and directional selection. Stabilizing selection will occur when the population mean is at the optimum. Directional selection will occur when the population mean is displaced from the optimum, and selection will act in such a way as to move the population mean back toward the optimum.

You can probably already guess the steps involved in implementing selection. The first step is to add a function to our simulation_engine class. Selection is actually one of the simplest features to add to this type of model. It merely involves calculating a probability of survival for each individual, drawing a random number, and marking the individual as alive or dead.

In the case of selection on a single trait, we need some parameters to describe the individual fitness surface, which relates phenotypic values to fitness. For instance, we will need to know the position of the optimum, and we will need a parameter specifying the steepness of the selection surface. We already have parameters for the steepness of the selection surface called SelectionStrength[2], and we will not bother with a parameter for the optimum yet. Instead, we will assume the optimum is always zero, so this version of the model will have a stationary optimum at an arbitrary trait value. Imagine that we have standardized the trait so that it is measured on a scale where the mean is near zero. In this case, the population will generally experience stabilizing selection that will keep the mean trait value from departing much from this arbitrary value of zero.

To calculate an individual’s probability of survival, we will use the probability density function for the Gaussian distribution. It can be found in any statistics book or online. Here it is:

\[
P(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}
\]

The mean is given by \(\mu\), \(\sigma^2\) is the variance, and \(e\) is the base of the natural logarithm. The first term (to the left of \(e\)) is a normalization term, which guarantees that the area under the curve sums to one, a necessary condition for a probability density function. However, in our case we want the maximum value to be one, so we will drop the normalization term. Thus, we end up with the following equation to specify our selection surface:

\[
W(z) = e^{-\frac{(z-\theta)^2}{2\omega}}
\]

The fitness of an individual with phenotype \(z\) is given by \(W(z)\), \(\theta\) is the position of the optimum, and \(\omega\) is the parameter that specifies the steepness of the selection surface (sometimes it is also called \(\omega^2\) in the literature, especially in the single-trait case, so be careful).

These considerations are all well and good, but how do we implement them? We will need a few variables, of course, so declare a new function (in your simulation_engine) with the following variables:

```c
void natural_selection()
{
    int i;
    double survival_prob, dRnum1, optimum[2];
```
We also need a way to keep track of which progeny are alive and which are dead. The easiest way to keep track of this important detail is probably to add another variable to our individual class. Scroll up in "simulation_engine.h" until you find the declaration for class individual. Immediately below bool Female;, add bool Alive;. The individual class will now look like this (with its functions minimized):

```cpp
class individual
{
 public:
    double Allele1trait0[50];
    double Allele2trait0[50];
    double Allele1trait1[50];
    double Allele2trait1[50];
    double Allele1both[50][2];
    double Allele2both[50][2];
    double Genotype[2];
    double Phenotype[2];
    bool Female;
    bool Alive;

    void calculate_genotypic_values(int n_loci_0, int n_loci_1, int n_loci_both) { ... }
    void calculate_phenotype(double env_st_dev_0, double env_st_dev_1) { ... }
    void set_sex(){ ... }
};
```

We will need to set our optima to zero. We will also need to implement a loop that calculates a survival probability for each progeny and uses a random number to determine whether it survives or not. Implement these steps by adding code to your new natural_selection() function until it looks like this:

```cpp
void natural_selection()
{
    int i;
    double survival_prob, dRnum1, optimum[2];

    optimum[0] = 0;
    optimum[1] = 0;

    for (i = 0; i < Nprogeny; i++)
    {
        survival_prob = exp(-1.0 * (progeny[i].Phenotype[0]-
                                                 optimum[0])*(progeny[i].Phenotype[0]-optimum[0]) /
                                                (2*SelectionStrength[0]));
        dRnum1 = genrand();
        if (dRnum1 < survival_prob)
            progeny[i].Alive = true;
        else
            progeny[i].Alive = false;
    } // end of i loop
}
```
That is all it takes to implement selection on trait zero. Be very careful with the parentheses and mathematical symbols used in the statement that calculates the survival probability, because even a very minor mistake can unhinge the entire operation, possibly without generating a compiler error. Under this selection subroutine, trait zero will experience directional or stabilizing selection, depending on the population mean, but trait one will be under no selection whatsoever. Instead, evolution at trait one would be governed by genetic drift, which affects all finite populations.

If we want both traits to be under selection simultaneously, then we have to make some changes. In the simplest case, we could assume that the two traits independently affect survivorship. In this case, we could add a second round of selection exactly like the first but based on trait one instead of trait zero. However, a more useful way to implement selection on both traits is to consider their joint effects on fitness. If we take this approach, then we can specify that certain combinations of traits have higher fitness than other combinations of traits. For example, we may want to study a case in which individuals with large values of trait zero have high fitness if they also have large values of trait one but have low fitness if they have small values of trait one. This situation could arise in real populations if proportional individuals have a fitness advantage. Thus, larger individuals might have high fitness if they also have long legs but low fitness if they have short legs. The selectional correlation specifies the extent to which traits interact to produce total organismal fitness. If the selectional correlation is zero, then the traits affect fitness independently, but if the selection correlation is non-zero then the two traits interact to determine the fitness of the individual. We can implement this type of selection by using a bivariate Gaussian distribution. Here is the probability density function for the bivariate Gaussian distribution:

$$P(x_1, x_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} e^{-\frac{a}{2(1-\rho^2)}},$$

where

$$a = \frac{(x_1 - \mu_1)^2}{\sigma_1^2} + \frac{(x_2 - \mu_2)^2}{\sigma_2^2} - 2\rho(x_1 - \mu_1)(x_2 - \mu_2).$$

The values $\mu_1$ and $\mu_2$ are the two means, $\sigma_1$ and $\sigma_2$ are the standard deviations, and $\rho$ is the correlation. Once again, for our purposes, we can drop the normalization term to the left of $e$, which leaves us with the following equation for fitness in terms of an individual’s trait zero and trait one phenotypes, $z_0$ and $z_1$:

$$W(z_0, z_1) = \exp\left[\left(\frac{-1}{2(1-r_s^2)}\right)\left(\frac{(z_0 - \theta_0)^2}{\omega_0} + \frac{2r_s(z_0 - \theta_0)(z_1 - \theta_1)}{\sqrt{\omega_0\omega_1}} + \frac{(z_1 - \theta_1)^2}{\omega_1}\right)\right].$$

The position of the bivariate optimum is given by $\theta_0$ and $\theta_1$, whereas $\omega_0$, $\omega_1$ and $r_s$ specify the shape of the selection surface. It is worth noting that in the selection literature, the shape of the selection surface is often represented by the matrix $\omega$, where
Here, the variable $\omega_{01}$ is analogous to the covariance, so it is related to $r_s$ in the following way:

$$r_s = \frac{\omega_{01}}{\sqrt{\omega_0 \omega_1}}.$$ 

This relationship is based on the definition of the correlation coefficient in terms of the covariance, and you can find this definition online or in any statistics book.

Implementation of this bivariate function for the calculation of survival probabilities is relatively straightforward. For slightly complex equations like this one, it is often desirable to break the calculation into several parts in the computer program, so we will declare a few additional variables to help out. Near the beginning of your function, type the following:

```c
double dSu, dSv, dSc;
double SSsqrt[2];
dSc = 2 * (1 - SelectionalCorrelation*SelectionalCorrelation);
SSsqrt[0] = sqrt(SelectionStrength[0]);
SSsqrt[1] = sqrt(SelectionStrength[1]);
```

Here we declare a few variables, set one of them to $2(1-r_s^2)$, and produce two variables equal to the square roots of $\omega_0$ and $\omega_1$. We calculate the values of these latter two variables outside of the loop, because we want to avoid making many unnecessary calls to the `sqrt()` function. Invoking functions usually carries a performance cost, so we adopt a general policy to minimize the usage of functions whenever we can. When we add mating preferences to the model, we will go back to selection on a single trait, so for now we should just “comment out” the code that calculates a survival probability on the basis of trait 0. Type `//` at the beginning of each line of the `survival_prob = ...` statement. It also might be useful to put a comment above that statement to indicate what it does. The last step is to calculate a new survival probability based on the bivariate normal probability density function. Underneath the recently commented-out univariate selection probability calculation and above `dRnum1 = genrand();`, type the following:

```c
dSu = (progeny[i].Phenotype[0] - optimum[0]) / SSqrt[0];
dSv = (progeny[i].Phenotype[1] - optimum[1]) / SSqrt[1];
survival_prob = exp((2*SelectionalCorrelation*dSu*dSv - dSu*dSu - dSv*dSv) / dSc);
```

Convince yourself that this code calculates the desired survival probability by comparing it to the equation above for $W(z_0, z_1)$. With that calculation, we are already done with our `natural_selection()` function. Here is the finished product:

```c
void natural_selection()
{
    int i;
    double survival_prob, dRnum1, optimum[2];
    double dSu, dSv, dSc;
    double SSqrt[2];
    dSc = 2 * (1 - SelectionalCorrelation*SelectionalCorrelation);
```
SSsqrt[0] = sqrt(SelectionStrength[0]);
SSsqrt[1] = sqrt(SelectionStrength[1]);

optimum[0] = 0;
optimum[1] = 0;

for (i = 0; i < Nprogeny; i++)
{
    // Univariate Gaussian selection on trait 0:
    // survival_prob = exp(-1.0 * (progeny[i].Phenotype[0] -
    // optimum[0])*(progeny[i].Phenotype[0] - optimum[0])
    // / (2*SelectionStrength[0]));
    
    // Bivariate selection on traits 0 and 1:
    dSu = (progeny[i].Phenotype[0] - optimum[0]) / SSsqrt[0];
    dSv = (progeny[i].Phenotype[1] - optimum[1]) / SSsqrt[1];
    survival_prob = exp((2*SelectionalCorrelation*dSu*dSv - dSu*dSu - dSv*dSv)
    / dSc);
    dRnum1 = genrand();
    if (dRnum1 < survival_prob)
        progeny[i].Alive = true;
    else
        progeny[i].Alive = false;
} // end of i loop

Using the Mutation and Selection Functions

The final step in this chapter is to add mutation and natural selection to our organism’s life cycle. Head over to your “.cpp” source file and add calls to mutation() and natural_selection() after my_sim.polygynous_mating() but before std::cout << "\n\nProgeny:"

We should also update our output_progeny() function to tell us whether each offspring is alive or not after natural selection. Add the following statement to output_progeny() immediately after the j loops that output the locus headings (i.e., the first set of three j loops in the function):

std::cout << "Alive";
Near the end of the function, immediately before the closing brace of the \( k \) loop, add:

```cpp
if (progeny[i].Alive)
    std::cout << "Alive";
else
    std::cout << "Dead";
```

Compile and run the program. It should compile and run without any errors or warnings. The output now contains an additional column indicating whether the individual survived natural selection. If you look carefully at the results, you will probably see that every offspring survived and also detect no evidence of a mutation. Our default selection strength is actually very permissive, and a mutation rate on the order of 1 in 5000 is unlikely to produce a mutation in any of the 10 or so offspring you have created.
Figure 9.1: Some of the output from the program after the addition of mutation and selection. Adult number 2 is the mother of progeny 0 through 3, so offspring number 1 obviously has a mutation at the Locus_0Trai10 gene (yellow arrow). The allele “-0.095” does not appear in the mother, nor does it appear in any of the adults. In addition, offspring 1 did not survive the selection phase of the life cycle (orange box), which is understandable given its large value for trait 0 (i.e., a phenotypic value of 2.115). You might need to run your program a few times before you see an obvious mutation.

To make sure our program is behaving as expected, change the population size to 10, the number of each type of loci to 1, the mutation rate per locus to 0.05, and both selection strengths to 9 (smaller values result in stronger selection). Run the program and look at the output under these new conditions. You might see something like the output shown in Figure 9.1. Mutations should be present in some offspring, and you should be able to pick out some of the mutants by comparing the mother’s allelic values to those in the offspring. If a new allele is present, then it was the product of a novel mutation. You should also see that some of the offspring failed to survive the episode of selection, although there is still a possibility they will all survive. At this point, we have almost all of the phases of the life cycle incorporated into the model. So far, the program runs for only one generation, so we still have some work to do before we start to get interesting results. The next step will be to run the program for many generations and keep track of the population-level variables we are interested in studying.

Chapter Summary
In this chapter, we added some key elements to the life cycle. Mutation is necessary in evolutionary models to maintain genetic variation over long periods of time. Selection is one of the most important evolutionary mechanisms resulting in phenotypic change in natural populations of organisms. This chapter did not call for new programming concepts, but it put our current programming skills to good use. We also learned about the normal and bivariate normal probability density functions and how these can be used to simulate an individual selection surface that relates phenotypic values to fitness.
Chapter 10. Completing the Life Cycle

In a few more steps, we will have a complete life cycle. So far, we have implemented mating, mutation, and selection. However, our population has no density regulation, so it will either shrink to the point of extinction or grow without bounds unless exactly enough offspring survive to replace the adults. The solution to this problem is to add some sort of population regulation. The easiest approach, and the one we will implement here, is to have a hard carrying capacity. If the number of offspring exceeds the carrying capacity, then we randomly kill off the excess. If the number is smaller than the carrying capacity, then we let everyone survive.

Implementing a Carrying Capacity

We will add a new function to our simulation_engine to regulate the population size. We can call it \texttt{population\_regulation()}. Go ahead and declare it as follows:

\begin{verbatim}
void population_regulation()
{
    //
}
\end{verbatim}

This function is an important part of the circle of life, because we will discard the old adults and replace them with progeny, which will be promoted to adult status. We need to think about what might be the best way to accomplish this goal. For instance, we might be tempted to pick the first $K$ progeny, where $K$ is the carrying capacity. For some models, this approach might work, but for our model it would not be a good idea. Think for a moment about this last statement. What feature of our model makes this approach a poor choice? The answer should be apparent from consideration of how we produce the offspring. We cycle through the parents in order, so the first four offspring are from the first female, the next four are from the second female, and so on. Consequently, if we choose offspring in order from the list, the early females will have more offspring represented than later females. This bias would actually have evolutionary implications, because some females would be more likely to produce offspring than others by virtue of an artificial list, rather than a biological reason. We would get a reduction in the effective population size, and an increase in the importance of genetic drift relative to expectations. We would like our model to behave predictably from an evolutionary theory standpoint, so we need to choose offspring in a random order.

Algorithms have been worked out for choosing a certain number of individuals at random from a list, without choosing the same individual twice; we can use one of these algorithms here. Add the following code to your new function (we will discuss what it is doing below):

\begin{verbatim}
void population_regulation()
{
    int i;
    double carrying_capacity_unfilled, progeny_left, keep_prob;
    int number_adults_chosen;
    double drnd1;

    progeny_left = 0;
\end{verbatim}
for (i = 0; i < Nprogeny; i++)
    if (progeny[i].Alive)
        progeny_left++;

carrying_capacity_unfilled = CarryingCapacity;
number_adults_chosen = 0;
for (i = 0; i < Nprogeny; i++)
{
    if (progeny[i].Alive)
    {
        keep_prob = carrying_capacity_unfilled / progeny_left;
        drnd1 = genrand();
        if (drnd1 < keep_prob)
        {
            adult[number_adults_chosen] = progeny[i];
            carrying_capacity_unfilled = carrying_capacity_unfilled - 1;
            number_adults_chosen++;
        }
        progeny_left = progeny_left - 1;
    } // end of if (progeny[i].Alive)
} // end of i
PopulationSize = number_adults_chosen;

This subroutine works by first counting up the number of progeny that are still alive after selection. The variable\_progeny\_left\_stores that number, and the first i loop does the counting. We also need a variable for the number of individuals still to be chosen before we reach the carrying capacity, and \_carrying\_capacity\_unfilled\_serves this purpose. We use \_number\_adults\_chosen\_to index and keep track of the new adults chosen.

The second i loop does the choosing. The technique we are using first calculates the proportion of progeny we need to choose to reach the carrying capacity. Thus, if the number of surviving progeny is 100 and the carrying capacity is 20, we need to promote 20 percent of the progeny to adults to end up with 20 adults. However, we cannot merely run through the progeny promoting each one with a probability of 0.2, because such an approach might not result in exactly 20 adults. Instead, we update the probability as we go along to ensure that we end up with exactly 20 adults. Thus, if we have 100 surviving progeny, for the first one we draw a uniformly distributed number between 0 and 1. If the number is less than 0.2, we promote that individual to adulthood. For the next progeny, however, we update the probability. In our example if the first progeny was chosen, then the probability for the second progeny will be 19/99, because we need 19 more adults and there are 99 surviving progeny left. If the first progeny had not been chosen, then the probability for the second progeny would be 20/99 because we would still need 20 adults and there would be 99 live progeny left to check. If the carrying capacity is larger than the number of progeny, then the probability will always be greater than one and all progeny will be promoted to adulthood.

Remember that the first i loop counted up the number of surviving progeny. Only the surviving progeny matter for this subroutine, so the program proceeds as if the dead progeny do not even exist. Note that in the second i loop, nothing happens if the progeny under consideration is not alive. If the progeny is alive, however, we use keep\_prob to calculate the probability that we should promote that individual. As noted in the previous paragraph, keep\_prob is equal to carrying\_capacity\_unfilled, which is the number of adults we still need to promote to reach the carrying capacity, divided by progeny\_left, which is keeping track of the number of live progeny we still need to provide with an opportunity to be promoted to adulthood (i.e., the number of living progeny for which we have not yet drawn a random
number). We use `genrand()` to draw a uniformly distributed random number between zero and one, and we store it in `drnd1`. If this number is less than `keep_prob`, then we promote the offspring to adulthood. Note that the value of `keep_prob` is calculated using only double variables, because including integers anywhere in the calculation will result in an integer value, which would be either zero or one in this case and would not produce the behavior we want.

The statement `adult[number_adults_chosen] = progeny[i];` sets all of the values for the adult indexed by `number_adults_chosen` equal to the values for `progeny[i]`. In other words, all of the allelic values, the sex, whether the individual is alive or dead (although they all should be alive), and so on are passed to the `adult` array. Remember that `number_adults_chosen` is being used to keep track of the number of adults we have accumulated in the population. Thus, the first time `drnd1` is less than `keep_prob`, `number_adults_chosen` will be equal to zero, so we will set `adult[0]` equal to `progeny[i]`. Then we also need to increment `number_adults_chosen` upward by one to indicate that we have a single adult in our `adult` array now. Thus, when the loop is finished, `number_adults_chosen` will be equal to the number of adults in the `adult` array. Each time we choose an adult, we also need to decrease by one our variable that keeps track of how many more adults we need to reach the carrying capacity. This task is accomplished by `carrying_capacity_unfilled = carrying_capacity_unfilled - 1;`. The final step in this second `i` loop is to update the number of live progeny we have yet to give a chance to be promoted to adulthood. Whether the offspring was chosen or not, we need to decrease `progeny_left` by one for each living progeny we run through. Otherwise, we will not properly update our probabilities, as described in the previous paragraph.

Finally, we need to set our class-level variable, `PopulationSize`, which keeps track of the number of adults in the population, equal to the number of progeny we have promoted to adulthood. The statement `PopulationSize = number_adult_chosen;` serves this purpose.

We need to be clear on the difference between `PopulationSize` and `CarryingCapacity`. With respect to C++, they are both variables. In terms of the model, however, one is a variable and one is a parameter. The carrying capacity is a parameter, because it specifies the maximum number of adults that can exist at any given time in the simulation, and it does not change during a simulation run. The population size, however, specifies the current number of adults and is a variable because it can, in principle, drop below the carrying capacity. For most parameter combinations, the adult population size will stay at the carrying capacity, but some parameter combinations, especially when selection is strong, could cause a population to be unable to replace itself. You may recall that we set the population size and carrying capacity to be equal when we initialized the population, so we are assuming that we are dealing with a population that is starting at the carrying capacity.

Now we are prepared to test our new `population_regulation()` subroutine. For now, we should still output the adults and progeny. After population regulation, we can also output the new generation of adults, which will be the grown up versions of the progeny. Alter the source code where your main program resides to look like this (the new parts are immediately above the `return 0;` statement):

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    simulation_engine my_sim;
    my_sim.display_parameters();

    bool initialization_success;
    initialization_success = my_sim.initialize_population();
```
if (!initialization_success)
{
    std::cout << "Simulation Initialization Failure!\n";
    return 0;
}

std::cout << "Adults:"
my_sim.output_adults();
my_sim.polygynous_mating();
my_sim.mutation();
my_sim.natural_selection();
std::cout << "Progeny:"
my_sim.output_progeny();
my_sim.population_regulation();
std::cout << "New Adults:"
my_sim.output_adults();
std::cout << "\n"

char end_it;
std::cout << "Enter any character to exit..."
std::cin >> end_it;
return 0;

Try running the program with the population size set to 10. You should see a list of ten adults, followed by a bunch of progeny, followed by a second list of 10 adults. The second list of 10 adults should be different than the first list. If you look carefully, you will see that the adults in the second list are actually identical to some of the individuals in the list of progeny, because these adults are progeny that survived to be the adults of the next generation. None of the offspring listed as dead should have made it to the next generation. The world is cruel, though, because even some of the survivors of selection never made it to adulthood, possibly as a result of random misadventure. Regardless, the survivors of selection that failed to make it to adulthood are a random subset of the survivors with respect to the phenotypic traits we are modeling.

Calculating Some Variables of Interest

We have a complete life cycle now, but to observe any meaningful evolution we will need to iterate the simulation over many generations and calculate population-level summary statistics each generation. Thus, we must put additional code into the program to calculate those statistics that interest us. For now, we might be interested in quantities such as the phenotypic mean, genetic mean, phenotypic variances and covariances, and genetic variances and covariances. We will not need to output all of the adults and progeny every generation, so we can eliminate the calls to the functions that output them. We can leave those functions in our class in case we need them later.

To calculate the population-level statistics of interest, we will need a new function in our simulation_engine class. Usually, these values are calculated in the progeny before selection, so we will use this approach. Our simulation starts with adults, so we will consider generation 0 to be the progeny produced by these adults after one round of random mating but no selection. We will come back to this issue later, but we definitely want to take some care to be sure we know when in the life cycle we are measuring the variables in our model.

Add a new function named calculate_values_progeny() to the simulation_engine. If we use local variables to calculate the values of interest, they will go out of scope after the function runs every
COMPLETING THE LIFE CYCLE

generation. This situation might not be ideal, because we might want other functions to have access to the population-level statistics. For instance, we might want to either output the values to the screen or save them to a file. We might also want to compare values across generations. Rather than put all the code to save and output the variables in one huge function, let us have one function that calculates the values and other functions that output them to the screen or a file. To implement this approach, we will need to declare some variables in the private: section of the class (i.e., up where we declared variables like PopulationSize, CarryingCapacity, and so on). Declare the following variables (and include a comment delimiting them from the other variables):

```c++
// Variables corresponding to population-level summary statistics:
double phenotypic_mean[2], genotypic_mean[2];
double phenotypic_variance[2], genotypic_variance[2];
double phenotypic_covariance, genotypic_covariance;
double phenotypic_correlation, genotypic_correlation;
```

The need for these variables will become apparent in a moment, but most of them will be used to store the means, variances and covariances that will be calculating. Because the means, variances, and covariances can be calculated by accumulating terms across the population, the first step in our new function is to set some of the variables equal to zero before we start adding up the terms. Setting them equal to zero is easy. Add the following lines of code to your new `calculate_values_progeny()` function:

```c++
void calculate_values_progeny()
{
    int i;
    for (i = 0; i < 2; i++)
    {
        phenotypic_mean[i] = 0;
        genotypic_mean[i] = 0;
        phenotypic_variance[i] = 0;
        genotypic_variance[i] = 0;
    }
    phenotypic_covariance = 0;
    genotypic_covariance = 0;
}
```

We have variables for phenotypic means, variances and covariances as well as for genetic means, variances and covariances. We also have variables for phenotypic and genetic correlations, but these are calculated from the covariances and variances, so we do not have to set them to zero in preparation for summing terms. We will need to divide by the number of progeny, so we also need a double variable equal to the number of progeny in the population. Recall that if you use an `int` variable in mathematical operations that involve non-integers, you may observe unexpected results. We are using `dNP` for this purpose, so add this line of code:

```c++
double dNP = Nprogeny;
```

Next we need to loop through the progeny to add up values for the mean. Remember that the mean is just the sum of the observations divided by the total number of observations. We have two traits, which is why we declared the mean and variance variables as very small arrays with two elements each. If we had more than two traits, then we would need larger arrays to accommodate all of the trait means. The strategy here will be to loop through the progeny in the population, adding the value of the current
progeny to a running tally of the total sum of trait values. This approach can easily be accomplished by using the following loop (added after the loop where we set everything to zero):

```c
// Calculate the Means
for (i = 0; i < Nprogeny; i++)
{
    phenotypic_mean[0] = phenotypic_mean[0] + progeny[i].Phenotype[0];
    phenotypic_mean[1] = phenotypic_mean[1] + progeny[i].Phenotype[1];
    genotypic_mean[0] = genotypic_mean[0] + progeny[i].Genotype[0];
}
phenotypic_mean[0] = phenotypic_mean[0] / dNP;
genotypic_mean[0] = genotypic_mean[0] / dNP;
```

The logic behind the mean calculation is extremely simple. If we take `phenotypic_mean[0]`, for example, it has a value of zero at the start of the loop. Each cycle of the loop adds the trait 0 phenotype of a different offspring to `phenotypic_mean[0]`, and by using a `for` loop we are ensuring that every offspring gets added. Thus, at the end of the loop, `phenotypic_mean[0]` holds the sum of all phenotypic values for trait 0 across all progeny. To convert that value to a mean, we merely divide by the number of progeny immediately after the conclusion of the loop. Similar logic applies to the phenotypic mean for trait 1 and the genetic means for both traits.

We need another loop to calculate variances and covariances. We were forced to calculate means first because the formulae for variances and covariances include the mean. The variance is the average squared deviation from the mean, which is given by

$$
\sigma_x^2 = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2,
$$

where $n$ is the population size, $x_i$ is the value for individual $i$, and $\bar{x}$ is the population mean. Here we are using the population variance rather than the sample variance (which would be divided by $n-1$ instead of $n$), because we are calculating the variance based on all individuals in the population. Similarly, the equation for the covariance is

$$
\sigma_{xy} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}).
$$

The symbols have the same meaning as in the variance equation, except that the $y$’s correspond to the second variable of interest (i.e., in our case the $x$’s would be trait 0 values and the $y$’s would be trait 1 values). The implementation of these equations in our program is entirely straightforward. We accumulate the summation term as we did for the means and then divide by the population size, as follows:

```c
// Calculate Variances and Covariances
for (i = 0; i < Nprogeny; i++)
{
    phenotypic_variance[0] = phenotypic_variance[0]
    + (progeny[i].Phenotype[0] - phenotypic_mean[0])
        * (progeny[i].Phenotype[0] - phenotypic_mean[0]);
```
The variances and covariances are useful, but the correlation has intuitive appeal so we will want to calculate it as well. A correlation is a covariance standardized to fall in the range -1 to 1, and the equation for the correlation is:

$$ \rho_{xy} = \frac{\sigma_{xy}}{\sqrt{\sigma_x^2 \sigma_y^2}}. $$

You might have noticed that the correlation is undefined if either of the variances is zero. We must take great care in our program to avoid dividing by zero, because any attempt to do so will result in a crash. Thus, we should first check to make sure the variances are both greater than zero before calculating the correlation coefficient. Here is one way to implement this calculation:

```cpp
// Calculate the phenotypic and genotypic correlations
double dtemp;
dtemp = sqrt(phenotypic_variance[0] * phenotypic_variance[1]);
if (dtemp > 0)
    phenotypic_correlation = phenotypic_covariance / dtemp;
else
    phenotypic_correlation = 0;
dtemp = sqrt(genotypic_variance[0] * genotypic_variance[1]);
if (dtemp > 0)
    genotypic_correlation = genotypic_covariance / dtemp;
else
    genotypic_correlation = 0;
```

We are using `dtemp` as a temporary double to calculate the denominator (i.e., bottom) of the equation for the correlation. If `dtemp` comes out greater than zero, then we use it. Otherwise, we assign a value of zero.
to the correlation (even though technically it is undefined). The code above performs the procedure first for the phenotypic correlation and then for the genetic correlation.

We are making a lot of progress here, and we have a barebones set of population-level statistics calculated. However, calculating the values is pointless unless we plan to use them for something. In this case, we should output the values for inspection. While we could handle the output from within the `calculate_values_progeny()` function, our code will probably be more versatile if we create separate functions to either output the data to the screen or save the results to a file.

**Outputting Variables to the Screen**

Add a new function called `output_population_variables()` to your `simulation_engine` class. This function will have no way of knowing how many generations of the simulation have elapsed, so we need to take that value as an argument. Before you skip to the code below, try to remember how to declare a function that accepts an argument. This function should also be declared as `void` because it will not return a value. The rest of the function is straightforward. Here it is:

```c++
void output_population_variables(int generation)
{
    if (generation == 0) // Output the header in generation zero
    {
        std::cout << "\nGen\tzbar0\tzbar1\tP00\tP11\tP12\ttr(P)\tgbar0\tgbar1\tG00\tG11\tG01\ttr(G)";
    }

    std::cout << "\n" << generation;
    std::cout << std::setprecision(3) << std::fixed;
    std::cout << "\t" << phenotypic_mean[0];
    std::cout << "\t" << phenotypic_mean[1];
    std::cout << "\t" << phenotypic_variance[0];
    std::cout << "\t" << phenotypic_variance[1];
    std::cout << "\t" << phenotypic_covariance;
    std::cout << "\t" << phenotypic_correlation;
    std::cout << "\t" << genotypic_mean[0];
    std::cout << "\t" << genotypic_mean[1];
    std::cout << "\t" << genotypic_variance[0];
    std::cout << "\t" << genotypic_variance[1];
    std::cout << "\t" << genotypic_covariance;
    std::cout << "\t" << genotypic_correlation;
}
```

Once we implement this function in the main program, the calculated variables will be output to the console screen. We will find that outputting variables to the screen is a less-than-ideal solution, so we will soon wish to save the variables to a text file. Saving to a file is a bit more involved (but still not hard), and we will get there in short order. For now, though, we will take the easy path of dumping the values onto the screen. All that remains is to call the function from within our simulation’s “.cpp” file.

We will not want to output every individual’s genotype during a simulation run, because the output would be unmanageable and not very useful. Imagine how long it would take for all of this information to scroll past on the screen for a population of hundreds of individuals for thousands of generations. And how would we make sense of all of these numbers? This task would be hopeless. Instead, a better approach is to calculate summary statistics each generation and output those. Under some circumstances, we might be interested in individual genotypes or other details that would require information on specific
organisms, and in these special cases, we would have to make special arrangements, involving additional code, to get the data of interest.

Before we delete the code associated with outputting the adults and progeny to the screen, however, we should test our new function that outputs summary statistics. In your “.cpp” file, after the calls to all the other functions, add the following code:

```cpp
my_sim.calculate_values_progeny();
my_sim.output_population_variables(0);
```

Go ahead and run the program. The simulation will still output all of the genotypic information for every individual in the simulation, but we will have a summary of the genetic values of interest at the very bottom of the screen. If that step worked, then we can feel comfortable that everything is functioning as intended, and we can safely delete the code that outputs the adults and progeny. If you are a hoarder and are uncomfortable with the complete removal of code, feel free to comment out the lines of interest rather than summarily deleting them. After the removal of these now unnecessary commands, the “.cpp” file should look like this:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main() {
    simulation_engine my_sim;
    my_sim.display_parameters();
    bool initialization_success;
    initialization_success = my_sim.initialize_population();
    if (!initialization_success) {
        std::cout << \n"Simulation Initialization Failure!\n";
        return 0;
    }
    my_sim.polygynous_mating();
    my_sim.mutation();
    my_sim.natural_selection();
    my_sim.population_regulation();
    my_sim.calculate_values_progeny();
    my_sim.output_population_variables(0);
    char end_it;
    std::cout << \nEnter any character to exit...\n ;
    std::cin >> end_it;
    return 0;
}
```

The output should look something like that shown in Figure 10.1.

Let us reflect on what the program is doing so far. First, we initialize the simulation with a population of adults, and these adults start with arbitrary genotypes at three loci that determine the values of our two quantitative traits. For the purposes of bookkeeping, let us refer to this initial population of adults as generation -1. Then, these adults reproduce, a feat that results in the production of a new population of progeny, the progeny of generation 0. Some of these progeny may even carry new mutations. Next, we
implement natural selection and population regulation, which results in the population of adults for the next generation – these generation 0 adults will produce the generation 1 progeny. Finally, we calculate the summary statistics for the progeny, but note that we have not yet allowed the generation 0 adults to reproduce. Oddly, the progeny of generation 0 and the adults of generation 0, which the progeny will eventually grow up to become, coexist simultaneously in our simulation (in two different arrays). In other words, at the end of the commands implemented so far, we are left with two arrays of individuals – the progeny of generation 0 and the adults of generation 0. We decided to calculate our summary statistics for the progeny, so our row of statistics corresponds to the progeny from generation 0. The adults are ignored for now. If we continued to generation 1, however, the adults would be very important, because they would be called upon to create the progeny for generation 1.

![Image](image.png)

**Figure 10.1:** Output from the current program. Gone are the genotypes of every last individual. Instead, we now have a row of summary statistics, including the phenotypic means, phenotypic variances and covariances, and genetic variances and covariances.

In short, the program runs a single generation and outputs summary statistics for the generation 0 offspring. That level functionality is actually not very useful, because meaningful evolutionary change usually requires dozens, hundreds or thousands of generations (although there are examples of major evolutionary effects in just a few generations, given a strong enough selective pressure). Regardless, we want our simulation to run for more than one generation. If we want the same basic set of instructions to be repeated over and over for a specified number of iterations, then we should be talking about a for loop. We will need to place our commands inside a loop.

Look at your “.cpp” file and consider which commands should be inside the loop and which should be outside, if the intent of our loop is to simulate many consecutive generations of evolution. Obviously (I hope), the #include statements should not be inside the loop, as each #include only needs to be called once during the execution of the program. Similarly, the int main() statement and its associated braces delineate the executable portion of our program, so there is no logical way to shove it inside a for loop. What about the declaration of our simulation_engine object? Or the population initialization steps? Well,
it seems like those tasks also need to be performed only once during the entire simulation run. If we initialized the population every generation, we would effectively be erasing the population’s history at every turn. Once we have an initialized population, however, we are off to the races, and we need to simulate mating, mutation, natural selection, and population regulation every generation. Each new generation will work with the adults from the previous generation. We can ensure this happens by putting the code from `my_sim.polygynous_mating();` to `my_sim.output_population_variables(0);` inside a loop that counts generations. Hopefully, it is also obvious to you that the zero in that last statement will also have to change. If it is not obvious, pause and take another look at your source code to figure out why a zero will be inadequate as the simulation runs.

We will need to declare a new variable to iterate our loop. A convenient place to declare variables is just after the opening brace for `int main()`. We should call the variable something informative, so let us call it `generations`. Type the following line of code in the appropriate place:

```cpp
int generations;
```

Now add a `for` loop and surround your statements pertaining to the life cycle with it. We have a name for the variable we are using as the iterator (i.e., `generations`), but do we know how many iterations it should run? If you peruse the “simulation_engine.h” header file, you might notice that we actually already declared a variable that specifies the number of generations we want the simulation to run. This variable is called, appropriately enough, `NumberOfGenerations`. You might also notice that we declared this variable as private, which means that we cannot access it from outside the class. We have a couple of options here. We could simply change the variable to public by moving it into the section of the class following the `public:` label. However, this approach would probably be frowned upon. The other option is to write a member function that retrieves the value of the variable. We will take the latter path. Add the following function to your `simulation_engine` class:

```cpp
int getNumberOfGenerations()
{
    return NumberOfGenerations;
}
```

This function is the last piece of the puzzle we needed to implement our loop. The loop should look like this:

```cpp
for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
{
}
```

The functions that implement the life cycle of the model should be within the braces. Once you have made these changes, here is what the “.cpp” file should look like:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations;

    simulation_engine my_sim;
```
my_sim.display_parameters();

bool initialization_success;
initialization_success = my_sim.initialize_population();
if (!initialization_success)
{
    std::cout << "Simulation Initialization Failure!\n";
    return 0;
}

for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
{
    my_sim.polygynous_mating();
    my_sim.mutation();
    my_sim.natural_selection();
    my_sim.population_regulation();
    my_sim.calculate_values_progeny();
    my_sim.output_population_variables(generations);
}

char end_it;
std::cout << "Enter any character to exit...";
std::cin >> end_it;
return 0;

Ensure that your NumberOfGenerations variable is initialized to a relatively small value (like 10) in your simulation_engine class constructor and run the program. You should see output something like that shown in Figure 10.2.

Now we have a respectable program that actually outputs some useful data. In principle, we could iterate it for any number of generations and see some interesting evolutionary patterns emerging. For example, we could use it to investigate the evolutionary dynamics of the genetic variances and covariance under different selection regimes and different mutational parameters.

Saving to a File

If it has not yet occurred to you, dealing with hundreds or thousands of generations of output as it scrolls by in a console window is simply impossible for anyone other than Lieutenant Commander Data from Star Trek. If you were Data, you could simply watch the text as it scrolled past and commit it all to memory or extract any analysis you wish on the fly as the numbers whiz by. But you are not Data, are you? Consequently, those of us who are mere humans will need to store the information so that we can deal with it later when our busy schedules allow. Of course, this problem has a solution, and the solution is to save the data to a file.

Saving information to a file is very much like outputting it to the screen, except instead of sending the information to the standard output stream (in this case the screen), we need to send it to a file. As we have seen before, reading and writing files is implemented as a class in C++, and this class is included as part of the compiler environment. The name of the header containing the class is “fstream”, so add the statement #include <fstream> to “simulation_engine.h” near the top of the file, where you will see the other include statements you have already added.

Before we start typing commands, we will review the implementation of the “fstream” header. As noted above, the functions we will use to read or write files are contained in a class, so the class will have member variables and functions. Essentially, we will declare an instance of this class (an object) and use
its member functions to access the file system. One such function might set the name of the file. The “fstream” header actually includes more than one class, so we might want to declare different objects depending on our goals. In this case, we will want to write to a file, so we will use the class ofstream. The class for exclusively reading from a file is called ifstream. If you search for “fstream C++” in a search engine, you will be treated to more information than you ever wanted to know about the fstream header and its classes.

![Figure 10.2: Output of summary statistics spanning multiple generations.](image)

Now that we have decided to use ofstream, we should review how it works. The member functions of greatest interest to us are open() and close(). We can also use the << operator to put data into an open file. We will see the usage of these functions in a minute, but a few important details are worth mentioning here. The close() function just closes the file and should be called when you are finished manipulating its contents. This function does not require any arguments, that is, nothing goes in the parentheses. The open() function is a little more complicated, because it requires at least one argument: a string or a null terminated character array with the name of the file. If open() is used with no other arguments, then a new file is created, destroying any other file with the same name that happens to exist in the target directory. This destruction will happen with absolutely no fanfare, so be careful. Alternatively, we have the option to add a second argument to tell the function to append the new information to the existing file. This argument is std::fstream::app.

The operations of opening and closing files, as well as writing or reading data to or from them, carry quite bit of overhead, so any disk access will slow your program considerably. For now, we are not going to worry too much about how fast our program runs, so we will just open and close the file every
CHAPTER 10

generation. However, down the road, we will want to optimize this part of the program so that all or most of the file creation is done in one step (or as few steps as possible). This sort of optimization will require a little bit of work and some thinking about how to store the information while it is waiting to be written to the hard drive, so we will return to it later.

Our function to save to a file turns out to be a modification of our function that outputs data to the screen. Use the code below to add the function to your simulation_engine class. An explanation of this code is provided below.

```cpp
void save_population_variables(int generation)
{
    std::ofstream outfile;
    if (generation == 0) // Output the header in generation zero
    {
        outfile.open("output.csv");
        outfile << "Gen,zbar0,zbar1,P00,P11,P12,r(P),gbar0,gbar1,G00,G11,G01,r(G)"
                << std::endl;
        outfile.close();
    }
    outfile.open("output.csv", std::fstream::app);
    outfile << std::endl << generation;
    outfile << "," << phenotypic_mean[0];
    outfile << "," << phenotypic_mean[1];
    outfile << "," << phenotypic_variance[0];
    outfile << "," << phenotypic_variance[1];
    outfile << "," << phenotypic_correlation;
    outfile << "," << genotypic_mean[0];
    outfile << "," << genotypic_mean[1];
    outfile << "," << genotypic_variance[0];
    outfile << "," << genotypic_variance[1];
    outfile << "," << genotypic_covariance;
    outfile << "," << genotypic_correlation;
    outfile.close();
}
```

As you can see, the code is very similar to that found in `output_population_variables()`, but there are some key differences. For example, we declare an object of the `ofstream` class with the statement `std::ofstream outfile;`. Then we retain the `if` statement that determines whether or not the `generations` loop is in generation zero. In generation zero, we need to open a new file and put the header on the first line, so the body of this if statement uses the `open()` function with only the filename as an argument, which creates a new file called “output.csv”. If another file by that name already exists, it will be destroyed in favor of the new one. The filename has a “.csv” extension, which signifies that it is a comma-delimited text file. This file format provides a standard way of saving spreadsheet data as text, so a double-click on it will open it in the default spreadsheet program on your computer. The column breaks are signified by commas in this sort of file, so the spreadsheet program will sort the text into columns based on the positions of the commas. As part of our switch to the comma-delimited format, we have also replaced the tab characters, `	`, that we used to output data to the screen with commas throughout the function. At the end of the `if` statement, we close the file.

The code after the `if` statement is executed every generation, including generation zero, and this code reopens the file for appending, so that any new characters will be added to the end without destroying the original file. Then we simply use the `<<` operator to send our data to our object, which knows to direct the
data to a file with the filename that we proposed when we called the `open()` function. Each generation, we start with a new line by adding a \n character to the file. Then we output each piece of data with commas separating the values. At the end of the function, we once more close our file. You can see that we are opening and closing the file like mad, and for long simulation runs, we might end up opening and closing the file thousands of times. Luckily, the file has no door or hinges that can wear out, so this approach works fine, especially for short simulations. If we become interested in simulating very long spans of evolutionary time, however, this part of the program is one that we could target for modification with more efficient code.

The final step in implementing this new function is to call it from our “.cpp” file. We will leave everything else alone, including the output to the screen, just to make sure everything works fine. Below the call to `my_sim.output_population_variables(generations);`, but still within the `for` loop, add the following line:

```
my_sim.save_population_variables(generations);
```

Run the program. Everything should look exactly the same as it did before, except that, unbeknownst to the user, the program created a new file. Dig through your file system and look for the file called “output.csv”. You should find it in the folder containing the other files associated with your Visual Studio project. If you have a spreadsheet program installed, you should be able to double-click “output.csv” and have it automatically open in your spreadsheet. The output should be identical to the output shown in the console window where your program is running, with the exception that we did not save the parameter values in “output.csv”. It might be a good idea to have the program automatically save the parameter values, but we probably want them in a separate file from our generation-by-generation output, because our spreadsheet will be more manageable without a bunch of lines of parameter values at the very top.

**Chapter Summary**

In this chapter, we took significant steps toward making our model useful. We completed the life cycle by adding population regulation. In this case, we used a simple carrying capacity, by randomly culling adults each generation until the adult population size is equal to the carrying capacity. Every model needs some sort of population regulation algorithm, and the one we have used is one of the simplest. Other, more complicated models could also be implemented and might add more ecological realism. In this chapter, we also learned how to calculate some important population-level summary statistics, such as the elements of the $P$-matrix (the phenotypic variance-covariance matrix) and the $G$-matrix (the additive genetic variance-covariance matrix). Then we wrote functions to output the variables to the screen and to write them to a file on the computer’s hard drive. In terms of programming, no new concepts were introduced. Rather, we redundantly reviewed the “fstream” class for file input and output, in case you missed that section in Chapter 4.
Chapter 11. Non-random Mating

For the purposes of this tutorial, we wish to create a general simulation model that can incorporate both natural selection and sexual selection. Sexual selection results from differences in fitness that arise from differences among individuals with respect to how many times they mate successfully. The previous statement is an oversimplification, of course, but it is a good enough definition to get us started. Our model has two traits already, so one approach might be to think of trait 0 as the male trait and trait 1 as the value of the female preference. For sexual selection to act, a male’s success at obtaining mates has to be associated with his phenotype in some way, and our model does not yet have mate choice. Instead, females mate at random, and random mating will not produce sexual selection. In addition, the model, as it stood at the end of Chapter 10, imposed natural selection on both traits. To model sexual selection, we probably want viability selection to act on the male trait only when it is expressed in males, and we might not want any natural selection on the female preference. Selection on the female preference thus would arise entirely from the outcome of her mating choices.

The biggest addition here is non-random mating, so let us start with that. The non-random mating function will be similar in many ways to the polygynous_mating() function, except that females will have the ability to reject some males on the basis of their phenotypic values for trait 0. The first step is to add yet another function to our simulation_engine class. We will call it gaussian_mating(), so add the following function to your simulation_engine class:

```cpp
void gaussian_mating()
{
}
```

Note that we are declaring it as void, because it will not return a value, and it has no arguments, just like polygynous_mating(). Start by adding a comment that addresses what the function is intended to do:

```cpp
// This function implements gaussian mate choice in a
// polygynous mating system. It should be used instead
// of other mating functions (like polygynous_mating).
// In this function, each female mates at most once, and
// each male can mate an unlimited number of times, so
// the mating system is polygynous.

// Females choose males based on their trait values.
// Mating preferences are Gaussian in the sense that each female
// has an ideal preferred male phenotype and her preferences
// fall off as males depart from her preferred phenotype.
// The drop in mating probability as the male departs from
// the preferred phenotype is modeled as a Gaussian-shaped
// function.

// Absolute or relative preferences? This function uses relative
// preferences, and they are relative to the phenotypic mean of
// the males. Consequently, if a female's preference trait has
```
Before we start to delve too deeply into this function, I am going to go ahead and unleash the spoiler that we will need additional parameter values to handle mating preferences. For a Gaussian preference, however, the only additional parameter we need will specify the rate at which a female’s mating probability drops off as the male trait deviates from her preference. In the case of a Gaussian preference function, this parameter is basically the variance of a normal distribution, so a high value means the preference drops off slowly and a low value specifies a steep drop-off.

As sexual selection becomes stronger, we are creating a greater risk of population extinction for a variety of reasons. We also need to declare another variable to track whether or not our population has gone extinct. Scroll up to the private: section of your class, where you will find the declarations for a number of important class-level variables. Add the following lines of code:

```cpp
double GaussianPreferenceVariance;
bool PopulationExtinct;
```

Each time we add additional model parameters, we create a number of additional tasks. First, we have to update the `display_parameters()` function to output them, assuming their initial values are of interest. In this case, `GaussianPreferenceVariance` is of interest to the user but `PopulationExtinct` is actually just a bookkeeping variable. Its initial value will always be false, and we will obviously have to report something to the user when it becomes true, but we do not need to report its initial value. Second, we have to initialize the values of any new parameters or variables, whenever their starting value plays a role in the model. Find the `display_parameters()` function, and change it to look like this:

```cpp
void display_parameters()
{
    std::cout << "Parameter_Values:\n";
    // Demographic Parameters
    std::cout << "Demographic_Parameters:\n";
    std::cout << "No_Generations: \t" << NumberOfGenerations << "\n";
    std::cout << "Initial_Pop_Size: \t" << PopulationSize << "\n";
    std::cout << "Carrying_Capacity: \t" << CarryingCapacity << "\n";
    std::cout << "Female_Fecundity: \t" << Fecundity << "\n";
    // Mating Parameters
    std::cout << "Mating_Parameters:\n";
    std::cout << "Max_Mating_Enc.: \t" << MaxMatingEncounters << "\n";
    std::cout << "Gaussian_Pref_Var.:\t" << GaussianPreferenceVariance << "\n";
    // Quantitative Genetic Parameters
    std::cout << "Quantitative_Genetic_Parameters:\n";
    std::cout << "No_Loci_Trait0: \t" << NumLociTrait0 << "\n";
    std::cout << "No_Loci_Trait1: \t" << NumLociTrait1 << "\n";
    std::cout << "No_Loci_Pleiotrop: \t" << NumLociBoth << "\n";
    std::cout << "Env_Variance_Trt0: \t" << EnvironmentalVariance[0] << "\n";
    std::cout << "Env_Variance_Trt1: \t" << EnvironmentalVariance[1] << "\n";
}
```
CHAPTER 11

// Mutational Parameters
std::cout << "Mutational_Parameters:\n";
std::cout << "Mut_Var_Trait0: \t" << MutationalVariance[0] << "\n";
std::cout << "Mut_Var_Trait1: \t" << MutationalVariance[1] << "\n";
std::cout << "Mut_Correlation: \t" << MutationalCorrelation << "\n";
std::cout << "Mutation_Rate: \t" << MutationRatePerLocus << "\n";

// Selection Parameters
std::cout << "Selection_Parameters:\n";
std::cout << "Omega_Trait0: \t" << SelectionStrength[0] << "\n";
std::cout << "Omega_Trait1: \t" << SelectionStrength[1] << "\n";
std::cout << "Selection_Corr: \t" << SelectionalCorrelation << "\n";
}

The changes here involve creating a new category of “Mating Parameters”, moving “Max._Mating_Enc.” into that category, and adding a line of code that outputs the new Gaussian parameter. To initialize the variables, go to the class constructor function (simulation_engine(): you may recall that constructors have the same name as the class), and make the following changes between the // Initialize the Parameters comment and the // Genetic Parameters comment:

// Initialize the Parameters

// Demographic Parameters
NumberOfGenerations = 10;
PopulationSize = 10;
CarryingCapacity = PopulationSize;
Fecundity = 4;
PopulationExtinct = false;

// Mating Parameters
MaxMatingEncounters = 500;
GaussianPreferenceVariance = 100;

// Genetic Parameters

Leave the rest of the constructor unchanged. A preference variance of 100 will impose extremely weak mate choice. We will see exactly how weak later. For the purposes of building and testing the function, we want the effects to be fairly small in case large effects have unintended consequences for the rest of our simulation. Now return to your new gaussian_mating() function, which so far contains only a series of comment lines. The next step is to declare some of the important local variables for the function. This variable list starts with the ones from polygynous_mating() and then adds a few new ones that will come in handy. Type the code below into your gaussian_mating() function.

int i, j, m;
int iPC;
bool mate_found;
int mateID, counter, rnum;
double dRnum;
double mate_prob;
double dNmales;
double mean_male_trait0;
double ideal;
The added variables will be involved mainly in the calculation of the mean male trait value and in keeping track of each female’s preferred male phenotype. As in `polygynous_mating()`, we only want the function to continue if there are males present in the population. If no males are present, then the population will be extinct in a single generation, because our sexually reproducing population will be unable to produce any progeny to replace the adults, all of whom will die. We can kill two birds with one stone here by also calculating the mean of trait 0 in males as we check to see if any males are present. We need the mean of the trait, because female preferences are relative to the mean. We need to restrict the calculation of the mean to males only, because females do not express the trait. They do, however, have the genes for trait 0, so they have a latent, hidden trait 0 value. We could accidentally include it in our mean calculation and our simulation would be unrealistic. Interestingly, this feature occurs in living systems as well. Even though females usually lack the ornaments present in males, they do have the genes for the ornament, which they will pass to their sons. Unfortunately, we have no way of knowing whether a female would have had an awesome ornament or a crappy one had she been male.

Anyway, the following code will check to see if any males are present in the population and will also calculate the mean of trait 0 in the males that do exist. If no males are present, the mean does not matter, because no mating will take place, so we have to deal with that possibility as well.

```cpp
bool males_present = false;
dNmales = 0;
mean_male_trait0 = 0;
for (i = 0; i < PopulationSize; i++)
{   
    if (!adult[i].Female)
    {
        males_present = true;
        dNmales++;
        mean_male_trait0 = mean_male_trait0 + adult[i].Phenotype[0];
    }
}
```

This code counts up the number of males by incrementing `dNmales` each time a male is encountered and sums up the male phenotypic values. We use the phenotypic value rather than the genotypic value for the trait, because females from a real population have access to male phenotypes but not their genotypes. By summing the phenotypes, we have set the stage to calculate the mean, but have not yet calculated it. Use the code below to calculate the mean or to flag the simulation that the population has gone extinct.

```cpp
if (dNmales > 0)
{   
    mean_male_trait0 = mean_male_trait0 / dNmales;
}
else
{   
    mean_male_trait0 = 0;
    PopulationExtinct = true;
}
```

The rest of `gaussian_mating()` is quite similar to `polygynous_mating()`, so here is the `gaussian_mating()` function in its entirety, including the parts we just discussed:

```cpp
void gaussian_mating()
{
    // This function implements gaussian mate choice in a
CHAPTER 11

// polygynous mating system. It should be used instead
// of other mating functions (like polygynous_mating).
// In this function, each female mates at most once, and
// each male can mate an unlimited number of times, so
// the mating system is polygynous.

// Females choose males based on their trait values.
// Mating preferences are Gaussian in the sense that each female
// has an ideal preferred male phenotype and her preferences
// fall off as males depart from her preferred phenotype.
// The drop in mating probability as the male departs from
// the preferred phenotype is modeled as a Gaussian-shaped
// function.

// Absolute or relative preferences? This function uses relative
// preferences, and they are relative to the phenotypic mean of
// the males. Consequently, if a female's preference trait has
// a value of 0.72, then that particular female's ideal male
// has a trait value 0.72 units greater than the male mean trait
// value. If she encountered such a male, she would mate with
// probability 1. Her probability of mating would drop off for
// males with trait values larger or smaller than the preferred
// value.

int i, j, m;
int iPC;
bool mate_found;
int mateID, counter, rnum;
double dRnum;
double mate_prob;
double dNmales;
double mean_male_trait0;
double ideal;

// Check to make sure at least one male is present in the population.
// Also calculate the mean of the male ornament trait (trait 0).

bool males_present = false;
dNmales = 0;
mean_male_trait0 = 0;
for (i = 0; i < PopulationSize; i++)
{
    if (!adult[i].Female)
    {
        males_present = true;
        dNmales++;
        mean_male_trait0 = mean_male_trait0 + adult[i].Phenotype[0];
    }
}

if (dNmales > 0)
{
    mean_male_trait0 = mean_male_trait0 / dNmales;
}
else
{
    mean_male_trait0 = 0;
}
PopulationExtinct = true;
}
iPC = 0;
for (i = 0; i < PopulationSize; i++)
{
    if (adult[i].Female && males_present)
    {
        // Find a mate for this female
        // She only gets MaxMatingEncounters tries to find
        // someone. We also include encounters with females
        // in the count, so it will be harder to find a mate
        // when the sex ratio is extremely female-biased.
        // If she doesn't find a mate in the allotted number
        // of tries, then she produces no progeny.

        // For each female, we first have to determine her
        // ideal mate phenotype. Trait 1 is the female preference
        // trait and it describes the deviation from the male
        // mean of her preferred mate at Trait 0, the ornament.

        ideal = adult[i].Phenotype[1] + mean_male_trait0;

        mate_found = false;
        counter = 0;
        while (counter < MaxMatingEncounters && !mate_found)
        {
            rnum = randnum(PopulationSize);
            if (!adult[rnum].Female)
            {
                // Calculate the focal female's (adult[i]) probability
                // of mating with this random male (adult[rnum]). The
                // probability is calculated using the Gaussian
                // probability density function without the normalization
                // term.
                mate_prob = exp(-1 * (adult[rnum].Phenotype[0] - ideal)*
                                (adult[rnum].Phenotype[0] - ideal)
                                / (2 * GaussianPreferenceVariance));

                // Generate a random number (roll the dice!)
                dRnum = genrand();

                if (dRnum < mate_prob)
                {
                    mateID = rnum;
                    mate_found = true;
                }
            }
            counter++;
        } // end of while

        // If a mate is found, produce progeny
        if (mate_found)
        {
            for (m = 0; m < Fecundity; m++)
            {
                if (iPC >= NprogMax)
                
                119
\[ iPC = NprogMax - 1; \]

// First, let's take care of Mendelian assortment of the trait0 loci
for (j = 0; j < NumLociTrait0; j++)
{
    // The progeny needs one maternal allele and one paternal allele.
    // The maternal allele will be from adult[i] (the mother), and
    // we determine which allele with basically a coin flip. The
    // function genrand() produces a value in the range [0,1), so
    // if this number is less than 0.5 we choose one allele.
    // Otherwise, we choose the other allele.
    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele1trait0[j] = adult[i].Allele1trait0[j];
    else
        progeny[iPC].Allele1trait0[j] = adult[i].Allele2trait0[j];

    // The procedure is the same for the father, adult[mateID].
    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele2trait0[j]=adult[mateID].Allele1trait0[j];
    else
        progeny[iPC].Allele2trait0[j]=adult[mateID].Allele2trait0[j];
}

// Second, take care of Mendelian assortment for the trait1 loci.
// The procedure is essentially identical to that for trait0.
for (j = 0; j < NumLociTrait1; j++)
{
    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele1trait1[j] = adult[i].Allele1trait1[j];
    else
        progeny[iPC].Allele1trait1[j] = adult[i].Allele2trait1[j];

    dRnum = genrand();
    if (dRnum < 0.5)
        progeny[iPC].Allele2trait1[j]=adult[mateID].Allele1trait1[j];
    else
        progeny[iPC].Allele2trait1[j]=adult[mateID].Allele2trait1[j];
}

// The pleiotropic loci are more complicated. We have to be sure to
// keep the allelic effects (on the two traits) together for each allele.
// Otherwise, the loci would not behave realistically like actual
// pleiotropic loci. [Note that this part isn't properly indented in
// this text document - it should be indented a bit more in your actual
// .cpp file. Keep in mind, however, that indentation is just cosmetic
// The C++ compiler ignores all spaces and tabs.]
for (j = 0; j < NumLociBoth; j++)
{
    dRnum = genrand();
    if (dRnum < 0.5)
    {
        progeny[iPC].Allele1both[j][0] = adult[i].Allele1both[j][0];
    }
That is quite a bit of code, and there are a few differences between this action and `polygynous_mating()` that we have not yet discussed. For example, we use the variable `ideal` to keep track of the phenotype of each female’s ideal mate. The value of `ideal` is the sum of the mean male phenotype at trait 0 plus the female’s value for trait 1. Hence, a positive value for trait 1, our preference trait, indicates that the female prefers a male above the mean with respect to trait 0, our male ornament. A negative value of trait 1 would indicate a preference for males with trait values smaller than the mean. We evaluate the male’s trait for the purposes of mating by using the following code (here the comments are removed):

```plaintext
mate_prob = exp(-1 * (adult[rnum].Phenotype[0] - ideal) *
               (adult[rnum].Phenotype[0] - ideal) / (2 * GaussianPreferenceVariance));
dRnum = genrand();
if (dRnum < mate_prob)
{
    mateID = rnum;
    mate_found = true;
}
```

The mating probability is calculated according to a Gaussian-shaped function. Note the similarity between the calculation of `mate_prob` here and the equation we used for selection on a single trait in the `natural_selection()` action. Thus, `mate_prob` gives the probability that a female will mate with a male that differs from her ideal as much as the male under consideration does. We draw a uniformly
distributed random number and store it in \texttt{dRnum}. If this number is less than the mating probability, then the female mates with the male under consideration. The rest of the code in this action produces the offspring once a male is selected, in the same way we did in \texttt{polygynous_mating}().

We have a few other bookkeeping items to address before this program is ready to run. We need to deal with our population extinction issue, for instance. Head up to the \texttt{save_population_variables()} function. One thing we will want to do is to output the present population size, so we need to add a column in our output for it. Adding an “N” column should do the trick. Change the line of code in \texttt{save_population_variables()} that outputs the header for our comma-delimited file to look like this:

\begin{verbatim}
outfile << "Gen,N,zbar0,zbar1,P00,P11,P12,r(P),gbar0,gbar1,G00,G11,G01,r(G)";
\end{verbatim}

Also add a line to this function to output the population size each generation. The line should be sandwiched between the lines that output the generation number and the phenotypic mean of trait 0. Here are those two lines plus the new one:

\begin{verbatim}
outfile << "\n" << generation;
outfile << "," << PopulationSize;
outfile << "," << phenotypic_mean[0];
\end{verbatim}

Back over in the “.cpp” file, we will want to include an \texttt{if} statement within the \texttt{generations} loop so that we stop trying to produce offspring after the population goes extinct. Recall that our extinction flag, \texttt{PopulationExtinct}, is private, so we cannot access it directly from the “.cpp” file. How do we access it, then? We need to write a function. In this case, we want the function to return “true” if the population is extinct and “false” if it is not. By now, you should be able to write such a function with minimal effort. You would likely come up with something like:

\begin{verbatim}
bool is_extinct()
{
    return PopulationExtinct;
}
\end{verbatim}

Great job! That function will work. Add it to your \texttt{simulation_engine} class. Now we just have to put our various life history events inside an \texttt{if} statement, so they will not execute when the population is flagged as extinct. We also want to change our statement that executes \texttt{polygynous_mating()} so that it now executes \texttt{gaussian_mating()}. Once these changes are implemented, the “.cpp” file should look something like this:

\begin{verbatim}
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations;

    simulation_engine my_sim;
    my_sim.display_parameters();

    bool initialization_success;
    initialization_success = my_sim.initialize_population();
    if (!initialization_success)
    {
        return -1;
    }

    // Other code...

    return 0;
}
\end{verbatim}
\{ 
    std::cout << "\nSimulation Initialization Failure!\n";
    return 0;
\}

for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
{ 
    if (!my_sim.is_extinct())
    { 
        my_sim.gaussian_mating();
        my_sim.mutation();
        my_sim.natural_selection();
        my_sim.population_regulation();
        my_sim.calculate_values_progeny();
        my_sim.save_population_variables(generations);
    }
}

char end_it;
std::cout << "\nEnter any character to exit...";
std::cin >> end_it;
return 0;

We have also eliminated the call to output_population_variables(), as we will be dealing mainly with output saved to the hard drive from here on out.

Now that we are facing extinction head on, we need to investigate some other unexpected consequences and take steps to ameliorate them. In particular, the way we have implemented our extinction checkpoint essentially checks once per generation, even though extinction could happen midway through a generation if, for example, adults were present but they were all female. Consequently, we need to check our code very carefully to make sure that the program will not crash in the event that adults or progeny are missing from the population. The most common problem, which will cause an embarrassing hard crash of the simulation, is a division by zero. We have to scour our code, then, and make sure that whenever we divide, we literally never have even a slight chance that the denominator would be zero. So, scour through the code (all of it!) and look for anything that could cause a problem.

Not unexpectedly, almost all of the offending code appears in the calculate_values_progeny() function. In particular, we divide by dNP like crazy, and dNP is just a double that corresponds to the number of progeny in the population. If the population has no progeny and we invoke this function, then the program will crash. We actually took some precautions in other parts of the function. For instance, near the end, where we calculate correlations, we check to make sure the variances are not zero. However, throughout the function, we need to make sure dNP is not zero. Because the number of progeny cannot be negative, as that would make no biological sense, we need to ensure that dNP is greater than zero before we divide by it. Basically, we just have to put all of the calculations involving dNP inside a huge if statement. Here is what the function should look like now:

\begin{verbatim}
void calculate_values_progeny()
{
    int i;
    double dNP = Nprogeny;
    for (i = 0; i < 2; i++)
    {
        phenotypic_mean[i] = 0;
    }
    // Rest of the function...
}
\end{verbatim}
CHAPTER 11

genotypic_mean[i] = 0;
phenotypic_variance[i] = 0;
genotypic_variance[i] = 0;
}
phenotypic_covariance = 0;
genotypic_covariance = 0;

if (dNP > 0)
{
    // Calculate the Means
    for (i = 0; i < Nprogeny; i++)
    {
        phenotypic_mean[0] = phenotypic_mean[0] + progeny[i].Phenotype[0];
        phenotypic_mean[1] = phenotypic_mean[1] + progeny[i].Phenotype[1];
        genotypic_mean[0] = genotypic_mean[0] + progeny[i].Genotype[0];
    }
    phenotypic_mean[0] = phenotypic_mean[0] / dNP;
    genotypic_mean[0] = genotypic_mean[0] / dNP;

    // Calculate Variances and Covariances
    for (i = 0; i < Nprogeny; i++)
    {
        phenotypic_variance[0] = phenotypic_variance[0] + (progeny[i].Phenotype[0] - phenotypic_mean[0]) * (progeny[i].Phenotype[0] - phenotypic_mean[0]);
        phenotypic_covariance = phenotypic_covariance + (progeny[i].Phenotype[0] - phenotypic_mean[0]) * (progeny[i].Phenotype[1] - phenotypic_mean[1]);
        genotypic_variance[0] = genotypic_variance[0] + (progeny[i].Genotype[0] - genotypic_mean[0]) * (progeny[i].Genotype[0] - genotypic_mean[0]);
        genotypic_covariance = genotypic_covariance + (progeny[i].Genotype[0] - genotypic_mean[0]) * (progeny[i].Genotype[1] - genotypic_mean[1]);
    }
    phenotypic_variance[0] = phenotypic_variance[0] / dNP;
    genotypic_variance[0] = genotypic_variance[0] / dNP;
    phenotypic_covariance = phenotypic_covariance / dNP;
    genotypic_covariance = genotypic_covariance / dNP;
}

// Calculate the phenotypic and genotypic correlations
double dtemp;
dtemp = sqrt(phenotypic_variance[0] * phenotypic_variance[1]);
if (dtemp > 0)
\[
\text{phenotypic\_correlation} = \frac{\text{phenotypic\_covariance}}{\text{dtemp}};
\]

\[
\text{else}
\]
\[
\text{phenotypic\_correlation} = 0;
\]

\[
\text{dtemp} = \sqrt{\text{genotypic\_variance}[0] * \text{genotypic\_variance}[1]};
\]

\[
\text{if} (\text{dtemp} > 0)
\]
\[
\text{genotypic\_correlation} = \frac{\text{genotypic\_covariance}}{\text{dtemp}};
\]

\[
\text{else}
\]
\[
\text{genotypic\_correlation} = 0;
\]

Because we check the value of \text{dNP} after we set the variables of interest to zero, we will get zeros for everything if there are no progeny in the population. This outcome is better than getting the previous generation’s values, which is what would happen if we checked at the very beginning or checked before invoking the function. Technically, these values are incorrect, because with no progeny present everything is undefined. The values have no meaning. There is no mean, there is no variance, and so forth, because all of these values presuppose that something exists to measure. We will need to remember this feature of the simulation. If we get values of zero across the board, they indicate population extinction and have no meaning. In reality there are no values for any of our summary statistics in a generation during which the population fails to produce offspring.

\textit{Chapter Summary}

This chapter was pretty straightforward, as we had only one goal. The goal was to include mate choice in the model. We used a Gaussian mate-choice algorithm, in which females choose males based on their phenotype at trait 0. The female preference is encoded by trait 1. We added a new parameter to specify the strength of the preference, and we created a new function, \texttt{gaussian\_mating()}, to include mate choice. We also took some steps to make our program extinction-proof. The population might still go extinct under some circumstances, but at least extinction will not crash the program now. This chapter did not require any new programming skills, as by now you possess a fairly solid set of basic programming skills.
Chapter 12. Small but Important Modifications

We have a quite functional model of quantitative trait evolution, including natural selection and mate choice. However, the model still has some unrealistic qualities. For example, if we decide to focus on sexual selection, treating trait 0 as a male ornament and trait 1 as a female preference, then only males should express trait 0 and only females should express trait 1. In our current implementation of natural selection, both traits are under selection in both sexes. To model sexual selection, we will need to change the way natural selection acts on the population. Another possible problem is that the way we have implemented mating preferences could be very inefficient when the number of potential mates is exceptionally small. In our current implementation, a female seeking a male will encounter adults at random, including females, and decide each time whether to mate. Females are not considered potential mates, as they do not produce the sperm needed to fertilize eggs, so these encounters are wasted. In a real organism, we might expect females to evolve mechanisms to seek out males in particular, so perhaps this algorithm could also be improved in our model.

Sex-Limited Selection

As just mentioned, our implementation of natural selection is not quite right for a sexual selection model. Careful examination of the natural_selection() function indicates that natural selection is acting on both sexes for both the trait (trait 0) and the preference (trait 1). Rather than change natural_selection(), which we might need later, let us create a new function called sex_limited_selection(). For this function, we will assume that the ornament (trait 0) is expressed only in males and the preference (trait 1) is expressed only in females. In addition to making selection sex specific, we also want it to be possible to have no selection whatsoever. We can make it so that a value of 0 for the selection strength parameter indicates no selection. We will need many of the same variables that we used in natural_selection(). Create a new function in your simulation_engine class, and add some of the lines of code from natural_selection() as shown below.

```c
void sex_limited_selection()
{
    int i;
    double survival_prob, dRnum1, optimum[2];
    optimum[0] = 0;
    optimum[1] = 0;
}
```

The rest of the action also uses the ideas already implemented in natural_selection(), except that we use different criteria for determining survival probabilities for males and females. This approach is easily implemented with the following:

```c
for (i = 0; i < Nprogeny; i++)
{
    // Calculate survival probabilities for males and females separately
    if (!progeny[i].Female)
    {
        if (SelectionStrength[0] > 0)
```
SMALL BUT IMPORTANT MODIFICATIONS

```
{  
survival_prob = exp(-1.0 * 
    (progeny[i].Phenotype[0] - optimum[0])*
    (progeny[i].Phenotype[0] - optimum[0])
    / (2 * SelectionStrength[0]));
} 
else  
{  
survival_prob = 1; // SelectionStrength = 0 means no selection.
}
} 
else  
if (SelectionStrength[1] > 0)  
{  
survival_prob = exp(-1.0 * 
    (progeny[i].Phenotype[1] - optimum[1])*
    (progeny[i].Phenotype[1] - optimum[1])
    / (2 * SelectionStrength[1]));
}  
else  
{  
survival_prob = 1; // SelectionStrength = 0 means no selection.
}
}
// Generate a random number to see if the individual survives 
dRnum1 = genrand(); 
if (dRnum1 < survival_prob)  
progeny[i].Alive = true;
else  
progeny[i].Alive = false; 
} // end of i loop
```

That should do it. An inspection of this code will show that males and females will have their survival probabilities calculated separately, with male fitness determined by trait 0 and female fitness determined by trait 1. Moreover, if the selection strength parameter is zero then everyone being evaluated on the basis of the corresponding trait will survive. Now just flip over to your “.cpp” file and change it to invoke the `sex_limited_selection()` function instead of the `natural_selection()` function.

**Making Gaussian Mating More Efficient**

As the program is now written, we can run into problems when males become rare. This situation could happen when mate choice is extremely strong, leading to males with extremely exaggerated traits. These males will have low survivorship, but will make up for it by fathering many offspring. The problem is that if we have females encounter adults at random and then determine whether or not the adult is a male, we may waste a lot of time during which females are encountering other females. The solution to this problem is to make a list of males and make sure each female only encounters males when looking for a mate. Before we do anything, we need to decide whether to modify our existing `gaussian_mating()` function or create a new function. The cost of keeping too many functions is that our class will be cluttered with code that we may never use, and too much clutter might become confusing eventually. The cost of changing it is that we will not have the old function should we want to use it sometime down the road. In this case, I think the new function will be better than the old function in every possible context, so I am going to change the old function, resulting in a new version of
**CHAPTER 12**

`gaussian_mating()` with the changes shown below. To illustrate the changes, I present the new version of `gaussian_mating()` in its entirety, with changes indicated by “< NEW” to the right of the line of code containing the change. We will discuss the changes below.

```c
void gaussian_mating()
{
    // This function implements gaussian mate choice in a
    // polygynous mating system. It should be used instead
    // of other mating functions (like polygynous_mating).
    // In this function, each female mates at most once, and
    // each male can mate an unlimited number of times, so
    // the mating system is polygynous.

    // Females choose males based on their trait values.
    // Mating preferences are Gaussian in the sense that each female
    // has an ideal preferred male phenotype and her preferences
    // fall off as males depart from her preferred phenotype.
    // The drop in mating probability as the male departs from
    // the preferred phenotype is modeled as a Gaussian-shaped
    // function.

    // Absolute or relative preferences? This function uses relative
    // preferences, and they are relative to the phenotypic mean of
    // the males. Consequently, if a female's preference trait has
    // a value of 0.72, then that particular female's ideal male
    // has a trait value 0.72 units greater than the male mean trait
    // value. If she encountered such a male, she would mate with
    // probability 1. Her probability of mating would drop off for
    // males with trait values larger or smaller than the preferred
    // value.

    int i, j, m;
    int iPc;
    bool mate_found;
    int mateID, counter, rnum;
    double dRnum;
    double mate_prob;
    double dNmales;
    double mean_male_trait0;
    double ideal;
    int iNmales; //new < NEW
    int iMaleIndex; //new < NEW
    int *iMaleList = new int[PopulationSize]; // new < NEW

    // Check to make sure at least one male is present in the population.
    // Also calculate the mean of the male ornament trait (trait 0).

    bool males_present = false;
    iNmales = 0; //new < NEW
    mean_male_trait0 = 0;
    for (i = 0; i < PopulationSize; i++)
    {
        if (!adult[i].Female)
        {
            males_present = true;
            iMaleList[iNmales] = i; // new < NEW
```
iNmales++; // new → NEW
mean_male_trait0 = mean_male_trait0 + adult[i].Phenotype[0];
}
}     // new → NEW

dNmales = iNmales; // new → NEW

if (dNmales > 0)
{
    mean_male_trait0 = mean_male_trait0 / dNmales;
} else
{
    mean_male_trait0 = 0;
    PopulationExtinct = true;
}

iPC = 0;
for (i = 0; i < PopulationSize; i++)
{
    if (adult[i].Female && males_present)
    {
        // Find a mate for this female
        // She only gets MaxMatingEncounters tries to find
        // someone.

        // She only encounters males.

        // If she doesn't find a mate in the allotted number
        // of tries, then she produces no progeny.

        // For each female, we first have to determine her
        // ideal mate phenotype. Trait 1 is the female preference
        // trait and it describes the deviation from the male
        // mean of her preferred mate at Trait 0, the ornament.

        ideal = adult[i].Phenotype[1] + mean_male_trait0;

        mate_found = false;
        counter = 0;
        while (counter < MaxMatingEncounters && !mate_found)
        {
            rnum = randnum(iNmales); // new → NEW
            iMaleIndex = iMaleList[rnum]; // new → NEW

            // Calculate the focal female's (adult[i]) probability
            // of mating with this random male (adult[iMaleIndex]). The
            // probability is calculated using the Gaussian
            // probability density function without the normalization
            // term.

            if (GaussianPreferenceVariance > 0) //new (whole if-else) → NEW
            {
                mate_prob = exp(-1 * (adult[iMaleIndex].Phenotype[0] - ideal)*
                                (adult[iMaleIndex].Phenotype[0] - ideal) /
                                (2 * GaussianPreferenceVariance)); → NEW
            } else //new (whole if-else) → NEW
            {
            
            }  //new (whole if-else) → NEW

            counter += 1;
        }
    
    } //new (whole if-else) → NEW
}
mate_prob = 1;
}

// Generate a random number (roll the dice!)
DRnum = genrand();
if (DRnum < mate_prob)
{
    mateID = iMaleIndex; // new ← NEW
    mate_found = true;
}
counter++;
} // end of while

// If a mate is found, produce progeny
if (mate_found) {...} // This section is collapsed -- no changes.
} // end of if (adult[i].Female && males_present)
} // end of i loop

Nprogeny = iPC;
delete[] iMaleList; // new ← NEW

The changes are small but important. We declare iMaleList as a pointer to an array of integers with PopulationSize elements, because there can be no more than PopulationSize males in the population. We then check each adult to see if it is male or female. If it is male, then we add its index number in our adult array to our list of males. We are using iNmales to count the number of males in iMaleList. This procedure might be better understood from a concrete example. Suppose we have eight adults in the population and half are male. Then our adult array, with respect to the variable Female, could look like:

adult[0].Female = true;
adult[1].Female = false;
adult[2].Female = true;
adult[3].Female = true;
adult[4].Female = false;
adult[5].Female = false;
adult[6].Female = true;
adult[7].Female = false;

If we make a list of males, as the code for gaussian_mating() now will do, then get the following list:

iMaleList[0] = 1;
iMaleList[1] = 4;
iMaleList[2] = 5;
iMaleList[3] = 7;

After this procedure, iNmales will have a value of four. Thus, by using rnum = randnum(iNmales);, we obtain a number from 0 to 3, which is a male from our list of male identification numbers stored in the iMaleList array. Remember that iMaleList contains every male in the population and iNmales is the total number of males in the population. Thus, if we pick a male at random from the list, we obtain a potential mate without having to check whether the chosen individual is male or female. He has to be male to be on the list. However, the random number we have just chosen is not the index of the individual in the adult array. Rather, we have to look up that index number by using
our **iMaleList** array. Imagine that our random number is 2 in the above example. Then the value stored in **iMaleList[2]** gives us the position of the male we want in the **adult** array. In this case **iMaleList[2] = 5**, so the male we have chosen is **adult[5]**. To simplify the code a little, we are using the variable **iMaleIndex** to keep track of the chosen male’s position in the **adult** array. In this example, we would set **iMaleIndex** to equal **iMaleList[2]**, so **iMaleIndex** would have a value of 5 and **adult[iMaleIndex]** (that is, **adult[5]**) would be the male under consideration.

The other small change is that we have also included an **if-else** statement that calculates **mate_prob** if **GaussianPreferenceVariance** is greater than zero but sets **mate_prob** to one otherwise. This change will allow us to impose random mating by setting **GaussianPreferenceVariance** to zero. We can use this feature to perform control runs that lack mating preferences. At the end of the action, we free the memory allocated to our array with the statement `delete[] iMaleList;`.

### Controlling the Starting Conditions

It might be nice to have more control over the starting conditions in our model. One approach is to have some initial generations during which mating is random, where we set strengths of selection and the optimal trait values in such a way as to give us desired distributions of male traits and female preferences before the sexual selection generations start. As usual, we will need some additional parameters, in this case to set the conditions for the initial generations. The main parameters should be the position of the optimum and the initial selection strength on each trait. In the **private**: section of the `simulation_engine` class, declare the following variables:

```cpp
int NumberOfInitialGenerations;
double InitialSelectionStrength[2], InitialSelectionalCorrelation;
double InitialOptimum[2];
```

We will want to set these parameters to some reasonable initial values. In the constructor for the `simulation_engine` class (`simulation_engine()`), pick a spot for the parameters for the initial generations and add the following code:

```cpp
// Initial Generations Parameters
NumberOfInitialGenerations = 200;
InitialSelectionStrength[0] = 49;
InitialSelectionStrength[1] = 49;
InitialSelectionalCorrelation = 0;
InitialOptimum[0] = 0;
InitialOptimum[1] = 0;
```

Now we just need to add a loop in our “.cpp” file that runs through these initial generations. It might be a good idea to use a different variable for the initial generations loop to keep it distinct from the experimental generations loop, so we will declare a new `int` for this purpose. Our initial generations loop should go after the population is initialized but before the **generations** loop. However, before we type the code for the loop, we need to have access to the **NumberOfInitialGenerations** parameter. Because this variable is a private member of the `simulation_engine`, we need to write a simple function to retrieve its value. Add the code below to your `simulation_engine` class.

```cpp
int getNumberOfInitialGenerations()
{
    return NumberOfInitialGenerations;
}
```
Back over in the “.cpp” file, add the following code just above the `generations` loop:

```c++
int ig;
for (ig = 0; ig < my_sim.getNumberOfInitialGenerations(); ig++)
{
    my_sim.polygynous_mating();
    my_sim.mutation();
    my_sim.natural_selection();
    my_sim.population_regulation();
}
```

If you compile and run the program with this new subroutine, be sure to increase the population size to something like 200. If you run it with a population size of 10, the population will almost certainly go extinct. We are not quite ready for prime time, however, because our `natural_selection()` function does not distinguish between the initial and experimental generations. Rather, it always uses the parameter values for the experimental generations, so the initial generations are not doing what we want them to. A solution to this problem is to make some small modifications. We can add an argument to `natural_selection()` that allows us to indicate whether or not we are in the initial generations. Then, within `natural_selection()`, when the simulation is flagged as initial generations, we can make sure we are using the appropriate parameter values. Change `natural_selection()` to look like this:

```c++
void natural_selection(bool init_gens)
{
    int i;
    double survival_prob, dRnum1;
    double dSu, dSv, dSc;
    double SSqrt[2];
    double local_sel_str[2], local_sel_corr, local_opt[2];

    if (init_gens)
    {
        for (i = 0; i < 2; i++)
        {
            local_sel_str[i] = InitialSelectionStrength[i];
            local_opt[i] = InitialOptimum[i];
        }
        local_sel_corr = InitialSelectionCorrelation;
    }
    else
    {
        for (i = 0; i < 2; i++)
        {
            local_sel_str[i] = SelectionStrength[i];
            local_opt[i] = Optimum[i];
        }
        local_sel_corr = SelectionalCorrelation;
    }

    dSc = 2 * (1 - local_sel_corr*local_sel_corr);
    SSSqrt[0] = sqrt(local_sel_str[0]);
    SSSqrt[1] = sqrt(local_sel_str[1]);
    for (i = 0; i < Nprogeny; i++)
```

132
SMALL BUT IMPORTANT MODIFICATIONS

{
    //Bivariate selection on traits 0 and 1:
    dSu = (progeny[i].Phenotype[0] - local_opt[0]) / SSsqrt[0];
    dSv = (progeny[i].Phenotype[1] - local_opt[1]) / SSsqrt[1];
    survival_prob = exp((2* local_sel_corr*dSu*dSv - dSu*dSu - dSv*dSv) / dSc);

    dRnum1 = genrand();
    if (dRnum1 < survival_prob)
        progeny[i].Alive = true;
    else
        progeny[i].Alive = false;
} // end of i loop

The function now takes a Boolean variable as an argument, which indicates whether the selection parameters in use should be the ones from the initial generations or the experimental generations. If init_gens is true, then we set the local versions of the relevant parameters (i.e., local_sel_str, local_sel_corr, local_opt) to the parameter values for the initial generations. Otherwise, we set them to the values for the experimental generations. We do not actually have variables for the position of the optima in the experimental generations yet, so you will have to declare them. To the private section of the simulation_engine class, add:

    double Optimum[2];

And initialize these variables in the class constructor, using:

        Optimum[0] = 0;
        Optimum[1] = 0;

Back in the natural_selection() function, make sure that you replace all occurrences of SelectionStrength, SelectionCorrelation, and Optimum (after the if-else loop) with the new local versions of the variables. Also, notice that we no longer need the local variable optimum (with a lowercase “o”), so you can delete its declaration (i.e., double optimum[2];).

Navigate back over to the “.cpp” file and change the my_sim.natural_selection(); call, which should now be showing an error due to a missing argument, to:

        my_sim.natural_selection(true);

We do not need to make changes to mutation() or population_regulation() because these phenomena will work the same in the initial generations as in the experimental generations. Compile the simulation and run it to make sure everything is working fine. Make sure the population size is set to a relatively large value (like 200) or the population might go extinct. You can set it to a small value, like 10, just to see what happens. In this case, you will probably end up with one line of output that is all zeros. Recall that our simulation sets all variables to zero when the population is extinct. This result partially explains why we want to maintain populations of endangered or threatened species at much larger population sizes than ten. Also notice that we do not keep track of anything that happens during the initial generations. In principle, we could, but for now we are using the initial generations just to set up the population for the experimental generations, which we do care about.

It would also be nice to have our program output these new parameter values, so add the code pasted below to the display_parameters() function. The new code all goes at the end, but for clarity I have
included some of the existing code. For now, we will continue to output our parameter values to the screen, but we will want to save them to the hard drive eventually.

```cpp
// Selection Parameters
std::cout << "Selection_Parameters:\n";
std::cout << "Omega_Trait0: \t" << SelectionStrength[0] << "\n";
std::cout << "Omega_Trait1: \t" << SelectionStrength[1] << "\n";
std::cout << "Selection_Corr: \t" << SelectionalCorrelation << "\n";
std::cout << "Optimum_Trait0 \t" << Optimum[0] << "\n";
std::cout << "Optimum_Trait1 \t" << Optimum[1] << "\n";

// Initial Generations
std::cout << "Initial_Generations_Parameters:\n";
std::cout << "No_Initial_Gens: \t" << NumberOfInitialGenerations << "\n";
std::cout << "Init_Omega_Trait0: \t" << InitialSelectionStrength[0] << "\n";
std::cout << "Init_Omega_Trait1: \t" << InitialSelectionStrength[1] << "\n";
std::cout << "Init_Sel_Corr: \t" << InitialSelectionalCorrelation << "\n";
std::cout << "Init_Opt_Trt0 \t" << InitialOptimum[0] << "\n";
std::cout << "Init_Opt_Trt1 \t" << InitialOptimum[1] << "\n";
```

With these changes, we have some control over the starting values of the male trait and the female preference. We also can use the selection parameters to set the starting values of the genetic variances and genetic covariance. Later, we will use these tools to investigate the effects of starting conditions on the sexual selection process. Before we do that, however, we need our program to calculate a wider variety of summary statistics each generation. And before we do that, we want to ensure that our parameter values are set to reasonable starting values.

**More Interesting Starting Parameter Values**

The parameter values we have been using so far are good for testing the program, but not good enough to produce much in the way of interesting results. Let us change the starting parameter values to something a little more interesting. Change the // Initialize the Parameters section of the class constructor to look like the following code.

```cpp
// Initialize the Parameters

// Initial Generations Parameters
NumberOfInitialGenerations = 1000;
InitialSelectionStrength[0] = 49;
InitialSelectionStrength[1] = 49;
InitialSelectionalCorrelation = 0;
InitialOptimum[0] = 0;
InitialOptimum[1] = 0;

// Demographic Parameters
NumberOfGenerations = 2000;
PopulationSize = 500;
CarryingCapacity = PopulationSize;
Fecundity = 4;
PopulationExtinct = false;

// Mating Parameters
MaxMatingEncounters = 50;
GaussianPreferenceVariance = 9;
```
These changes are completely straightforward, so there is nothing to explain here.

**Calculating Additional Variables**

Now that we have better starting parameter values, we also need to be sure we are calculating everything we need to know. There are lots of things that would be interesting to know about what is happening on a generation-by-generation basis as our population evolves. Some of these things are obvious and others are not so obvious. We will start with some of the more obvious ones. These calculations might get a little tedious, but they are necessary, so let us take a deep breath and jump in.

To access the variables in which we are interested, we will have to make a few changes here and there throughout the program to make things a little more convenient. Otherwise, we risk becoming hopelessly confused. We can start easy by just adding some columns to our output. When you see how many columns, your heart may sink, however.

Navigate over to the `save_population_variables()` function in your `simulation_engine`. After the first line that outputs “Gen,N,…,r(G)”, add the following:

```cpp
outfile << ",Lambda1,Lambda2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1"
outfile << ",~Lmbd1,~Lmbd2,~Ang,~Size,~Ecc,~G00,~G11,~G01,~r(g)"
outfile << ",ASR,Im,If,MdifM,MdifF"
```

We will leave the initial generations loop the way it is, but we need to change the experimental generations loop slightly. We are going to turn the life cycle a half turn before the experimental generations start. This change will make interpretation of our variables a little bit easier. The reason is that we will need to calculate some summary statistics in progeny and others in adults. It is desirable to have the calculations that appear on the same line of our output be from the same generation. In other words, we will calculate some values from progeny and others from adults, but we want to make sure that the progeny we use to calculate variables are the same individuals that ultimately comprise the adult population of the same generation (minus those that die). Before the `generations` loop in your “.cpp” file, add the following two lines of code:

```cpp
// Turn the life cycle halfway, so the
```
Of course, this change also necessitates a change within the experimental generations loop. We are going into the loop with newly created offspring (zygotes, really), so we need to start the loop with viability selection. Everything just needs to be rearranged slightly. Change the experimental generations loop to match the code below (note that the lines of code we just added above are also repeated here).

```cpp
// Turn the life cycle halfway, so the
// experimental generations start with progeny
my_sim.polygynous_mating();
my_sim.mutation();

for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
{
    if (!my_sim.is_extinct())
    {
        my_sim.sex_limited_selection();
        my_sim.calculate_values_progeny();
        my_sim.population_regulation();
        my_sim.gaussian_mating();
        my_sim.mutation();
        my_sim.save_population_variables(generations);
    }
}
```

Note that `calculate_values_progeny()` must occur before the adults reproduce and replace the current progeny with the next generation of progeny. After `sex_limited_selection()`, we will know which ones are going to survive selection, but we need not act on this information unless we want to. Even the progeny destined to die are still in the progeny array. We will need another action to calculate values in adults, so create a new function in your simulation_engine called `calculate_values_adults()`. That is, add:

```cpp
void calculate_values_adults()
{
}
```

We will use this function to calculate various summary statistics in the adults. We will call this function immediately after mating and mutation each generation, so the progeny of the next generation will already exist when we invoke it. However, the creation of these progeny does not erase the adults or change them in any way, so we can still reliably measure variables describing them. After we add a call to this new function (which currently does nothing), here is what our entire “.cpp” file should look like:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations;
```
simulation_engine my_sim;
my_sim.display_parameters();

bool initialization_success;
initialization_success = my_sim.initialize_population();
if (!initialization_success)
{
    std::cout << "Simulation Initialization Failure!\n";
    return 0;
}

int ig;
for (ig = 0; ig < my_sim.getNumberOfInitialGenerations(); ig++)
{
    my_sim.polygynous_mating();
    my_sim.mutation();
    my_sim.natural_selection(true);
    my_sim.population_regulation();
}

// Turn the life cycle halfway, so the experimental generations start with progeny
my_sim.polygynous_mating();
my_sim.mutation();

for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
{
    if (!my_sim.is_extinct())
    {
        my_sim.sex_limited_selection();
        my_sim.calculate_values_progeny();
        my_sim.population_regulation();
        my_sim.gaussian_mating();
        my_sim.mutation();
        my_sim.calculate_values_adults();
        my_sim.save_population_variables(generations);
    }
}

char end_it;
std::cout << "\nEnter any character to exit...";
std::cin >> end_it;
return 0;

To get the information we need from our program, we will have to add some calculations to both calculate_values_progeny() and calculate_values_adults(). We will deal with calculate_values_progeny() first. In addition to the variables our program already calculates, we will need additional variables that more completely characterize the G-matrix. The G-matrix is the additive genetic variance-covariance matrix, so in our case it has three elements: the additive genetic variance for trait 0, the additive genetic variance for trait 1, and the additive genetic covariance between traits 0 and 1. In our model, all genetic variation is additive, so the total genetic variance equals the additive genetic variance. You may be thinking that we have already set up the program to calculate the elements of the G-matrix, and you are correct. However, there are other ways to describe G-matrices that
can be more useful under some circumstances. For the sake of clarity, here is what our \( G \)-matrix looks like:

\[
G = \begin{bmatrix}
G_{00} & G_{01} \\
G_{01} & G_{11}
\end{bmatrix},
\]

where \( G_{00} \) and \( G_{11} \) are the additive genetic variances for traits 0 and 1, and \( G_{01} \) is the additive genetic covariance. Because there is only one additive genetic covariance for the two-trait case, \( G_{01} \) and \( G_{10} \) are the same value, so I am using \( G_{01} \) both above and below the diagonal. More generally, every \( G \)-matrix, regardless of the number of traits, has the same values above and below the diagonal (in other words, \( G \)-matrices are symmetrical).

In addition to variances and covariances, another way to characterize a matrix is to calculate the \textit{eigenvalues} and \textit{eigenvectors}. This approach is particularly useful in quantitative genetics, because the eigenvectors of the matrix will be vectors that point some direction in phenotypic space. The reason they are enlightening in this case is that the “leading eigenvector” of the \( G \)-matrix points in the direction of phenotypic space with the greatest amount of additive genetic variation. In other words, it points in the direction of phenotypic space for which a response to selection can occur most readily, given the standing genetic variation in the population. A two-trait \( G \)-matrix has only two eigenvectors, and the second eigenvector is constrained to be orthogonal to the leading eigenvector (i.e., at a 90 degree angle). The eigenvalues tell us how much genetic variation is associated with each eigenvector. Thus, we can describe a \( G \)-matrix using this eigenvector and eigenvalue approach, and this description of a \( G \)-matrix is more easily interpreted in light of evolutionary change than raw variances and covariances. Now the big question is how to implement this approach in the program.

Navigate over to your \texttt{calculate_values_progeny()} function and scroll down to the end of the calculations involving the genetic variances, covariance and correlation. You should be just above the closing brace for the function. Add a new comment here that says: \textit{// Calculate the eigenvectors and eigenvalues of the \( G \)-matrix}. We will work just below this comment. We will need additional variables to perform these calculations, so declare the following:

```cpp
double a, b, c, d;
double ev[2];
a = genotypic_variance[0];
b = genotypic_covariance;
c = genotypic_covariance;
d = genotypic_variance[1];
const double m_pi = 3.14159265358979323846;
```

This code also set the values of \( a, b, c, \) and \( d \) to be equal to the elements of the \( G \)-matrix, and declares a constant variable, \texttt{m\_pi}, which stores the value of pi.

We will want some values that we calculate to be available at the level of the class, as opposed to locally within the \texttt{calculate_values_progeny()} function, so declare the following in the \texttt{private:} area of your class definition (which by now has quite a lot of variables declared):

```cpp
double EigenValue[2];
double EigenVector1[2];
double EigenVector2[2];
double LeadAngle, Sigma, Epsilon;
```
The calculation of eigenvalues for a 2x2 matrix is completely straightforward. For larger matrices, the calculations become non-trivial, but we do not need to concern ourselves with such issues at this point. Type the following code to calculate the eigenvalues:

```c
// Use the formula for a 2x2 matrix to calculate eigenvalues
if ((a + d)*(a + d) > 4 * (a*d - b*c))
{
    ev[0] = ((a + d) + sqrt((a + d)*(a + d) - 4 * (a*d - b*c))) / 2;
    ev[1] = ((a + d) - sqrt((a + d)*(a + d) - 4 * (a*d - b*c))) / 2;
}
else
{
    ev[0] = 0;
    ev[1] = 0;
}

// Make sure EigenValue[0] is the larger of the two
if (ev[0] > ev[1])
{
    EigenValue[0] = ev[0];
    EigenValue[1] = ev[1];
}
else
{
    EigenValue[0] = ev[1];
    EigenValue[1] = ev[0];
}
```

The calculation of the eigenvectors is a similarly simple matter. Type the following code:

```c
// Calculate eigenvectors using 2x2 matrix formulae
if (c != 0)
{
    EigenVector1[0] = (EigenValue[0] - d) / sqrt((EigenValue[0] - d)*(EigenValue[0] - d) + c*c);
    EigenVector1[1] = c / sqrt((EigenValue[0] - d)*(EigenValue[0] - d) + c*c);
    EigenVector2[0] = (EigenValue[1] - d) / sqrt((EigenValue[1] - d)*(EigenValue[1] - d) + c*c);
}
else
{
    if (d > a)
    {
        EigenVector1[0] = 0;
        EigenVector1[1] = 1;
        EigenVector2[0] = 1;
        EigenVector2[1] = 0;
    }
    else
    {
        EigenVector1[0] = 1;
        EigenVector1[1] = 0;
        EigenVector2[0] = 0;
        EigenVector2[1] = 1;
    }
}
```
CHAPTER 12

140

// Calculate the angle of the leading eigenvector in degrees
if (EigenVector1[0] != 0)
{
    LeadAngle = atan(EigenVector1[1] / EigenVector1[0])*(180 / m_pi);
}
else
{
    LeadAngle = 90;
}

The first if statement, if (c != 0)…, calculates the eigenvectors when the covariance is not equal to zero. If the covariance is 0, then the eigenvector either points along the x-axis or along the y-axis in a two-dimensional plot of trait 0 (on the x-axis) versus trait 1 (on the y-axis). The eigenvector points along the x-axis if trait 0 has the greater genetic variance and along the y-axis if trait 1 has the greater genetic variance. These situations are handled by the second if-else statement.

Finally, we need to convert our vectors from coordinates to an angle, and the last if-else statement accomplishes this task. We have calculated two vectors, and each vector has two elements. We are most concerned with the leading angle, which is the angle associated with the largest eigenvalue. In this case, the vector we care about is in the array EigenVector1, which has two elements. The two elements correspond to coordinates, so EigenVector1[0] is a position along the x-axis (corresponding to trait 0) and EigenVector1[1] is a position along the y-axis (corresponding to trait 1). We can convert these coordinates into an angle by using the arctangent function. Remember, the tangent of an angle is equal to the opposite side of a triangle divided by the adjacent side of the triangle. Thus, if we take the position on the y-axis and divide it by the position on the x-axis, then the arctangent of that value will give us the angle of the point in question relative to the origin. We use this fact to calculate the variable known as LeadAngle in the code above. However, the C++ implementation of the arctangent function, which is called atan(), returns the value in radians. To convert from radians to degrees, we multiply by 180/pi. One other complication occurs when EigenVector1[0] is zero. Under this scenario, the value we are attempting to feed to atan() will be undefined, resulting in a situation that would crash our program, so we need to address this possibility. A value of 0 for EigenVector1[0] means that the vector points directly along the y-axis, and we are measuring the angle relative to the x-axis. Thus, in this case the angle will be 90 degrees. With this final bit of code, we have our eigenvalues and eigenvectors (we only need to keep track of the leading eigenvector because we know the second eigenvector meets it at an angle of 90 degrees).

Is that enough? Of course not! Two other summary statistics related to the G-matrix are the “sum” (represented by sigma, Σ) and the “eccentricity” (represented by epsilon, ε). The sum is simply the sum of the additive genetic variances, which is exactly equal to the sum of the eigenvalues, and it is a measure of the total amount of genetic variance across all traits. The eccentricity is the smaller eigenvalue divided by the larger eigenvalue, and it measures how much the G-matrix deviates from a circle. An eccentricity near one indicates a circular G-matrix, while smaller values indicate more cigar-shaped G-matrices. Calculate these values with the two lines of code below.

Sigma = EigenValue[0]+EigenValue[1];
Epsilon = EigenValue[1]/EigenValue[0];

The final values of interest that we can calculate from the progeny are the selection differentials, arising from viability selection, on trait 0 (the male ornament) and trait 1 (the female preference). These selection differentials will only deal with the viability component of the life cycle, so the selection
differentials we are about to calculate are purely natural selection (not sexual selection), and they are related to the distance that trait values are displaced from their naturally selected optima. In this case, we will calculate the total selection differential, including both sexes, even though selection may only act on one sex, depending on whether we decide to have sex-limited selection in our model. When selection acts on only one sex, and the sex ratio is 1:1, the selection differential is half what it would be if selection acted on both sexes with equal strength. First, declare two new variables in the private: section of your class, by adding the following statement:

```c
  double sel_diff_trt_0, sel_diff_trt_1;
```

Now add the following code to the end of your calculate_values_progeny() function:

```c
// Calculate selection differentials on trait 0 and trait 1
// These selection differentials represent viability
// selection, because they include only progeny survival
// during the juvenile phase of the life cycle.

// The selection differential is the covariance between
// trait values and fitness. Here, fitness is based on
// whether or not the individual survived viability selection.

// We use some of the variables calculated above:
// phenotypic_mean[2], dNP (number of progeny).

double mean_fitness;

// Tally the mean fitness of the offspring
mean_fitness = 0;
for (i = 0; i < Nprogeny; i++)
{
  if (progeny[i].Alive)
  {
    mean_fitness++;
  }
}
mean_fitness = mean_fitness / dNP;

sel_diff_trt_0 = 0;
sel_diff_trt_1 = 0;
for (i = 0; i < Nprogeny; i++)
{
  if (progeny[i].Alive) // Alive and dead have different fitnesses (1 vs. 0)
  {
    sel_diff_trt_0 = sel_diff_trt_0 + (progeny[i].Phenotype[0] - phenotypic_mean[0]) * (1.0 / mean_fitness - 1.0);
    sel_diff_trt_1 = sel_diff_trt_1 + (progeny[i].Phenotype[1] - phenotypic_mean[1]) * (1.0 / mean_fitness - 1.0);
  }
  else
  {
    sel_diff_trt_0 = sel_diff_trt_0 + (progeny[i].Phenotype[0] - phenotypic_mean[0]) * (-1.0);
    sel_diff_trt_1 = sel_diff_trt_1 + (progeny[i].Phenotype[1] - phenotypic_mean[1]) * (-1.0);
  }
}
Recall that the selection differential is the covariance between trait values and relative fitness. In this case, we are working with each individual’s realized fitness, in the sense that we only keep track of whether each individual lived or died. Living individuals have a fitness of 1, and dead individuals have a fitness of 0. This type of unstandardized fitness is known as absolute fitness. A selection differential is, by definition, the covariance between trait values and relative fitness, where relative fitness is standardized to have a mean of 1. Thus, an individual’s relative fitness is just absolute fitness divided by the population mean fitness. Here, we calculate mean fitness by adding 1 to a tally for each living individual. We ignore dead individuals, because they would add zero to the tally, leaving it unchanged. Then we divide by the total number of individuals, including all living and dead individuals. Hence, if some individuals die, the mean fitness will be some value between 0 and 1, and it will reflect the proportion of individuals that lived.

The above code next calculates the covariance between relative fitness and the trait value for each trait. We have seen the formula for the covariance before, but here it is again in case you have forgotten it:

$$
\sigma_{xy} = \frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y}).
$$

The covariance is the average product of the deviations from the mean for the two variables under consideration. Thus, the code above loops through the individuals, calculating this product for each individual and summing across individuals. We are calculating two covariances simultaneously, so the sums are accumulating in `sel_diff_trt_0` and `sel_diff_trt_1`. The terms being added to the sum each iteration might look a little strange. For instance, `progeny[i].Phenotype[0] - phenotypic_mean[0]` looks just like $x_i - \bar{x}$, so it makes sense. However, $1 / mean_fitness - 1$ does not look much like $y_i - \bar{y}$. Remember, though, that we need to use relative fitness, so each living individual has an absolute fitness of 1, which must be divided by mean fitness to give relative fitness, so $1 / mean_fitness$ is just the current individual’s relative fitness. In addition, relative fitness is standardized to have a mean of 1, so the “$- 1$” is just subtracting mean fitness. In short, this term really is equivalent to $y_i - \bar{y}$. Dead individuals have a fitness of 0, so their relative fitness is $0 / mean_fitness$ and $0 / mean_fitness - 1 = -1$, which explains why the second term in the product is just $-1.0$ for the dead progeny.

In principle, we could calculate selection differentials based on an individual’s survival probability, which is known in a simulation but not in real life, where we just see the outcome of the survival probability. Such an approach might give a more precise estimate of the selection differential, especially in smaller populations, but we would have to keep track of each individual’s survival probability. This task could easily be accomplished by adding a variable to our individual class and setting it equal to the survival probability that we calculate during our function that implements selection. We are not going to calculate selection differentials in this way, but it is worth knowing that it would be easy to do so.

Before we continue, we need to take one last look at `calculate_values_progeny()` to make sure it is crash proof. In particular, are there instances where we might divide by zero? Also, what happens if we call the function and there are no progeny in the population? We have already crash-proofed the function up to the new code that we just added. Examine the code from the G-matrix calculations down. Do you see any problems?
SMALL BUT IMPORTANT MODIFICATIONS

It looks to me like we will be okay as long as both traits have non-zero variances. If either trait has no variance, then the $G$-matrix has no real meaning. We should put our calculations inside an if statement that checks to see if the traits both have non-zero genetic variances. If one or both traits does not, then we also need statements to handle those possibilities. Finally, the last section that calculates the selection differentials should also be inside an if statement that checks to make sure at least one progeny is present in the population. In addition, these calculations will run into problems if mean fitness is either zero (all offspring died) or one (all offspring lived), so we need to handle these possibilities as well. I have added these cautionary changes, and the entire function is pasted below.

```c
void calculate_values_progeny()
{
    int i;
    double dNP = Nprogeny;
    for (i = 0; i < 2; i++)
    {
        phenotypic_mean[i] = 0;
        genotypic_mean[i] = 0;
        phenotypic_variance[i] = 0;
        genotypic_variance[i] = 0;
    }

    phenotypic_covariance = 0;
    genotypic_covariance = 0;

    if (dNP > 0)
    {
        // Calculate the Means
        for (i = 0; i < Nprogeny; i++)
        {
            phenotypic_mean[0] = phenotypic_mean[0] + progeny[i].Phenotype[0];
            phenotypic_mean[1] = phenotypic_mean[1] + progeny[i].Phenotype[1];
            genotypic_mean[0] = genotypic_mean[0] + progeny[i].Genotype[0];
        }

        phenotypic_mean[0] = phenotypic_mean[0] / dNP;
        genotypic_mean[0] = genotypic_mean[0] / dNP;

        // Calculate Variances and Covariances
        for (i = 0; i < Nprogeny; i++)
        {
            phenotypic_variance[0] = phenotypic_variance[0]
                + (progeny[i].Phenotype[0] - phenotypic_mean[0])
                * (progeny[i].Phenotype[0] - phenotypic_mean[0]);
                + (progeny[i].Phenotype[1] - phenotypic_mean[1])
                * (progeny[i].Phenotype[1] - phenotypic_mean[1]);
            phenotypic_covariance = phenotypic_covariance
                + (progeny[i].Phenotype[0] - phenotypic_mean[0])
                * (progeny[i].Phenotype[1] - phenotypic_mean[1]);

            genotypic_variance[0] = genotypic_variance[0]
                + (progeny[i].Genotype[0] - genotypic_mean[0])
                * (progeny[i].Genotype[0] - genotypic_mean[0]);
                + (progeny[i].Genotype[1] - genotypic_mean[1])
                * (progeny[i].Genotype[1] - genotypic_mean[1]);
        }
    }
}
```


CHAPTER 12

       + (progeny[i].Genotype[1] - genotypic_mean[1])
       * (progeny[i].Genotype[1] - genotypic_mean[1]);
  genotypic_covariance = genotypic_covariance
       + (progeny[i].Genotype[0] - genotypic_mean[0])
       * (progeny[i].Genotype[1] - genotypic_mean[1]);

} // end of i
phenotypic_variance[0] = phenotypic_variance[0] / dNP;
genotypic_variance[0] = genotypic_variance[0] / dNP;
phenotypic_covariance = phenotypic_covariance / dNP;
genotypic_covariance = genotypic_covariance / dNP;

} // Calculate the phenotypic and genotypic correlations
double dtemp;
dtemp = sqrt(phenotypic_variance[0] * phenotypic_variance[1]);
if (dtemp > 0)
   phenotypic_correlation = phenotypic_covariance / dtemp;
else
   phenotypic_correlation = 0;

dtemp = sqrt(genotypic_variance[0] * genotypic_variance[1]);
if (dtemp > 0)
   genotypic_correlation = genotypic_covariance / dtemp;
else
   genotypic_correlation = 0;

// Check to make sure the variances are both non-zero before
// attempting to calculate the G-matrix.
if (genotypic_variance[0] > 0 && genotypic_variance[1] > 0)
{
   // Calculate the eigenvectors and eigenvalues of the G-matrix

double a, b, c, d;
double ev[2];
a = genotypic_variance[0];
b = genotypic_covariance;
c = genotypic_covariance;
d = genotypic_variance[1];
const double m_pi = 3.14159265358979323846;

// Use the formula for a 2x2 matrix to calculate eigenvalues
if ((a + d)*(a + d) > 4 * (a*d - b*c))
{
   ev[0] = ((a + d) + sqrt((a + d)*(a + d) - 4 * (a*d - b*c))) / 2;
   ev[1] = ((a + d) - sqrt((a + d)*(a + d) - 4 * (a*d - b*c))) / 2;
}
else
{
   ev[0] = 0;
   ev[1] = 0;
}

// Make sure EigenValue[0] is the larger of the two
if (ev[0] > ev[1])

144
SMALL BUT IMPORTANT MODIFICATIONS

```
{  
    EigenValue[0] = ev[0];  
    EigenValue[1] = ev[1];  
}
else  
{
    EigenValue[0] = ev[1];  
    EigenValue[1] = ev[0];  
}

// Calculate eigenvectors using 2x2 matrix formulae  
if (c != 0)  
{
    EigenVector1[0] = (EigenValue[0] - d) / sqrt((EigenValue[0] - d)*(EigenValue[0] - d) + c*c);  
    EigenVector1[1] = c / sqrt((EigenValue[0] - d)*(EigenValue[0] - d) + c*c);  
    EigenVector2[0] = (EigenValue[1] - d) / sqrt((EigenValue[1] - d)*(EigenValue[1] - d) + c*c);  
}
if (c == 0)  
{
    if (d > a)  
    {
        EigenVector1[0] = 0;  
        EigenVector1[1] = 1;  
        EigenVector2[0] = 1;  
        EigenVector2[1] = 0;  
    }
    else  
    {
        EigenVector1[0] = 1;  
        EigenVector1[1] = 0;  
        EigenVector2[0] = 0;  
        EigenVector2[1] = 1;  
    }
}

// Calculate the angle of the leading eigenvector in degrees  
if (EigenVector1[0] != 0)  
{
    LeadAngle = atan(EigenVector1[1] / EigenVector1[0])*(180 / m_pi);  
}
else  
{
    LeadAngle = 90;  
}

Sigma = EigenValue[0] + EigenValue[1];  
Epsilon = EigenValue[1] / EigenValue[0];  
}
else // deal with the case where one or both genetic variances are zero  
{
    // If either variance is zero, then the covariance is also zero.  
    // If the covariance is zero, then the eigenvalues are the same as the variances.
```
EigenValue[0] = genotypic_variance[0];
EigenValue[1] = genotypic_variance[1];
Sigma = EigenValue[0] + EigenValue[1];
Epsilon = 0; // If either variance is zero, so is epsilon

// if only one variance is zero, we will define the angle to
// point along the axis of the trait with the non-zero variance.

EigenVector1[0] = 0;
EigenVector1[1] = 0;
EigenVector2[0] = 0;
EigenVector2[1] = 0;
LeadAngle = 0;

if (EigenValue[0] > 0)
{
    EigenVector1[0] = 1;
    EigenVector1[1] = 0;
    EigenVector2[0] = 0;
    EigenVector2[1] = 1;
}

if (EigenValue[1] > 0)
{
    EigenVector1[0] = 0;
    EigenVector1[1] = 1;
    EigenVector2[0] = 1;
    EigenVector2[1] = 0;
    LeadAngle = 90;
}

// Calculate selection differentials on trait 0 and trait 1
// These selection differentials represent viability
// selection, because they include only progeny survival
// during the juvenile phase of the life cycle.

// The selection differential is the covariance between
// trait values and fitness. Here, fitness is based on
// whether or not the individual survived viability selection.

// We use some of the variables calculated above:
// phenotypic_mean[2], dNP (number of progeny).

double mean_fitness;
if (dNP > 0)
{
    // Tally the mean fitness of the offspring
    mean_fitness = 0;
    for (i = 0; i < Nprogeny; i++)
    {
        if (progeny[i].Alive)
        {
            mean_fitness++;
        }
    }
    mean_fitness = mean_fitness / dNP;
}
if (dNP > 0 && mean_fitness > 0 && mean_fitness < 1) {
    sel_diff_trt_0 = 0;
    sel_diff_trt_1 = 0;
    for (i = 0; i < Nprogeny; i++) {
        if (progeny[i].Alive)//Alive and dead have different fitnesses (1 vs. 0)
        {
            sel_diff_trt_0 = sel_diff_trt_0 + (progeny[i].Phenotype[0] - phenotypic_mean[0]) * (1.0 / mean_fitness - 1.0);
            sel_diff_trt_1 = sel_diff_trt_1 + (progeny[i].Phenotype[1] - phenotypic_mean[1]) * (1.0 / mean_fitness - 1.0);
        }
        else
        {
            sel_diff_trt_0 = sel_diff_trt_0 + (progeny[i].Phenotype[0] - phenotypic_mean[0]) * (-1.0);
            sel_diff_trt_1 = sel_diff_trt_1 + (progeny[i].Phenotype[1] - phenotypic_mean[1]) * (-1.0);
        }
    }
    sel_diff_trt_0 = sel_diff_trt_0 / dNP;
    sel_diff_trt_1 = sel_diff_trt_1 / dNP;
} else {
    sel_diff_trt_0 = 0;
    sel_diff_trt_1 = 0;
}
}

The only changes to the above code, beyond what we covered in detail, consist of a few `if` statements that will prevent the program from trying to calculate values that are impossible to calculate. The final step in our current programming effort is to make sure our new values get saved to our output file. Luckily, this task is easily accomplished by adding a few lines of code to `save_population_variables()`. Make your way over to this function and, immediately above `outfile.close()`, place the following lines of code:

```
outfile << "," << EigenValue[0];
outfile << "," << EigenValue[1];
outfile << "," << EigenVector1[0];
outfile << "," << EigenVector1[1];
outfile << "," << LeadAngle;
outfile << "," << Sigma;
outfile << "," << Epsilon;
outfile << "," << sel_diff_trt_0;
outfile << "," << sel_diff_trt_1;
```

If you compile and run the program (which might take a little longer than usual due to the larger number of generations and larger population size), you should get output with numbers in these additional columns of data. For now, we will not worry about interpreting these numbers, but we should take some solace in the fact that our program can produce an expansive, mind-numbing wall of high-precision numerals.
CHAPTER 12

Calculate Changes from One Generation to the Next

Sometimes we may wish to keep track of changes in a subset of the variables from one generation to the next. To do so, we will need to store the previous generation’s values of the variables of interest. For our present purposes, we are interested in how much the $G$-matrix changes, so we will want to store the eigenvalues, angle, sigma, epsilon, and the elements of $G$. The implementation of this plan requires some thought. For instance, when and where do we want to store the previous values? Where do we calculate the change from one generation to the next? Good answers to these questions will result in a better program than bad answers. One approach, and the one we will take here, is to set the previous generation values at the very end of the `calculate_values_progeny()` function. The variables will have to be class-level, and we might as well make them private, so they can go up where all the other class-level variables are declared, in the `private:` section of the simulation_engine class. We also need some variables to keep track of the single-generation changes for each variable, so add the following code:

```cpp
double PrevEval[2], PrevAngle, PrevSigma;
double PrevEpsilon, PrevG00, PrevG11, PrevG01, PrevRg;
double cPrevEval[2], cPrevAngle, cPrevSigma;
double cPrevEpsilon, cPrevG00, cPrevG11, cPrevG01, cPrevRg;
```

Back in the `calculate_values_progeny()` function, we need to set these variables equal to the current values just before the very end of the function. This way, the next time we call the function, the previous generation’s values will be available right up until the end of the function, when they are overwritten by the current generation’s values (for use in the next generation). Use the following code to set the values:

```cpp
PrevEval[0] = EigenValue[0];
PrevEval[1] = EigenValue[1];
PrevAngle = LeadAngle;
PrevSigma = Sigma;
PrevEpsilon = Epsilon;
PrevG00 = genotypic_variance[0];
PrevG11 = genotypic_variance[1];
PrevG01 = genotypic_covariance;
PrevRg = genotypic_correlation;
```

Now that we have access to the previous generation’s values, we can calculate the single-generation change for each of these values within `calculate_values_progeny()` for every generation, except the very first time that we execute the action. Oddly enough, we need to calculate the single-generation changes in `calculate_values_progeny()` just before we set the previous generation’s values. If we do it after we set the previous generation’s values, then we will get nothing but zeros. So, just above the code you just added, still within `calculate_values_progeny()`, add:

```cpp
cPrevEval[0] = fabs(PrevEval[0] - EigenValue[0]);
cPrevAngle = fabs(PrevAngle - LeadAngle);
cPrevSigma = fabs(PrevSigma - Sigma);
cPrevEpsilon = fabs(PrevEpsilon - Epsilon);
cPrevG00 = fabs(PrevG00 - genotypic_variance[0]);
cPrevG11 = fabs(PrevG11 - genotypic_variance[1]);
cPrevG01 = fabs(PrevG01 - genotypic_covariance);
```
SMALL BUT IMPORTANT MODIFICATIONS

\[ c_{\text{PrevRg}} = \text{fabs}(\text{PrevRg} - \text{genotypic}\_\text{correlation}); \]

We are interested in the absolute value of the changes, so the order of the subtraction does not matter. The function **fabs()** returns the absolute value of a float or double variable. Plain old **abs()** is the absolute value function that applies to integers only.

One of these variables, **cPrevAngle** needs special consideration. This variable is the change in the angle of the leading eigenvector of the \( G \)-matrix from one generation to the next, and we have constrained the angle to fall between -90 and 90 degrees, where angles from 0 to 90 point up and to the right, and angles from 0 to -90 point down and to the right. The eigenvector, however, has no real directionality when we talk about the \( G \)-matrix, so a vector with an angle of 45 degrees also points at an angle of -135 under our numbering scheme. This point becomes important when we consider a change in angle from, say, -85 degrees to 85 degrees. Our program would calculate this angle to be a change of 85 – (-85), which equals 170. However, the -85 degree eigenvector also points up and to the left at 95 degrees in our numbering scheme. Thus the actual change in angle is only 85 – 95, or -10 degrees, with an absolute value of 10 degrees. In general, whenever we calculate an angle change with an absolute value greater than 90 degrees, we should subtract it from 180 to get the smaller change in eigenvector associated with its other end. Thus, after the code that calculates **cPrevAngle**, add:

```cpp
// Ensure that we are calculating the smallest // possible change in eigenvector angle.
if (cPrevAngle > 90)
    cPrevAngle = 180 - cPrevAngle;
```

With that settled, we have a couple of additional bookkeeping issues to address. First, never use a variable without first setting it to a value. If you think carefully about our code, you will realize that the first time we call the **calculate_values_progeny()** function, we will be using the previous values of the target variables before we have actually set the previous values. This unappealing feature is a consequence of the fact that we set the variables to values at the very end of the function. A solution to this problem is to set these variables to a value of zero when we initialize our population. In the class constructor, **simulation_engine()**, add the following code near the end:

```cpp
// Initialize previous generation variables to zero
PrevEval[0] = 0;
PrevEval[1] = 0;
PrevAngle = 0;
PrevSigma = 0;
PrevEpsilon = 0;
PrevG00 = 0;
PrevG11 = 0;
PrevG01 = 0;
PrevRg = 0;
```

Our second bookkeeping task is to save the single-generation changes for each value. For these changes to be valid during the first experimental generation, we actually have to call **calculate_values_progeny()** before the first generation of progeny for our experimental generations is created. If you look at the code, you will realize that this generation of progeny is actually created during the space between our \( \text{ig} \) loop and our \( \text{generations} \) loop, when we call **my_sim.polygynous_mating()**. Consequently, we have to call **my_sim.calculate_values_progeny()** just before that. If we take this step, then the first call to this function in the \( \text{generations} \) loop will have access to the previous generation’s values (i.e., the last
CHAPTER 12

generation of the initial generations). In your “.cpp” file, add the following code immediately after the end of the ig loop and before the between-loop call to my_sim.polygynous_mating():

```cpp
my_sim.calculate_values_progeny();
```

We are now in a position to add these single-generation change variables to our output file. Add these statements to your save_population_variables() function:

```cpp
outfile << "," << cPrevEval[0];
outfile << "," << cPrevEval[1];
outfile << "," << cPrevAngle;
outfile << "," << cPrevSigma;
outfile << "," << cPrevEpsilon;
outfile << "," << cPrevG00;
outfile << "," << cPrevG11;
outfile << "," << cPrevG01;
outfile << "," << cPrevRg;
```

If you compile and run the program now, you will be rewarded with an even larger wall of numbers than before. We have now filled the columns preceded by a “~”, the symbol we are using to signify that the value in the column reports the absolute value of the single-generation change in the variable of interest.

**Calculate Values in Adults**

If you have not done so already, create a new function in your simulation_engine called calculate_values_adults(). The variables we would like to calculate in the adults are the adult sex ratio, the opportunities for sexual selection for both sexes, and the mating differentials for both sexes. To keep track of these values, we will need to declare some variables at the level of the class, so start by adding the code below to the private: section of the simulation_engine class.

```cpp
double SexRatio, Im, If, MdiffMales, MdiffFemales;
```

Within our calculate_values_adults() function, we will need to tally a number of quantities, such as the number of individuals of each sex and variables related to mating success. Hence, we should start by declaring the variables we will need and setting them equal to zero. Add lines of code to make your function look like this:

```cpp
void calculate_values_adults()
{
    int i;
    double MeanMSmales, MeanMSfemales, MeanTrait0males, MeanTrait1females;
    double VarMSmales, VarMSfemales, NMma, Nfe;

    NMma = 0;
    Nfe = 0;
    MeanMSmales = 0;
    MeanMSfemales = 0;
    MeanTrait0males = 0;
    MeanTrait1females = 0;
}
```
The code to count up the number of males and females as well as to calculate mean mating success and trait values is completely straightforward, except that we do not keep track of mating success for the adults anywhere in our program. We will need to add a variable to keep track of adult mating success each generation. We will also have to add code to set it to a value of zero for each newborn individual and then to increment it each time an individual mates. This task will send us on a merry tour of our code, with the first stop at the individual class. The easiest way to add a variable for mating success, and the preferred way for an object-oriented program, is to add it to a class. Scroll up to the individual class and add this variable to the `public:` section:

```cpp
double MatingSuccess;
```

Simple enough, but how do we count up the mating success of males and females? The answer is that we will have to set the variable to zero when the individual is created and then count up the number of times each individual mates during our mating function. Every time we create a group of progeny, they all go through the routine in the `mutation()` function, where we count up their breeding values and add environmental effects. We can just set their mating success to zero there as well, and be sure that every individual will enter whatever mating action we happen to be using with a mating success of zero. Find the `mutation()` function and scroll down to the very end. Just before the closing bracket of the `i` loop (make sure you are outside of the `if` statement), add the following line of code:

```cpp
progeny[i].MatingSuccess = 0;
```

Now we just need to head over to our mating routine and count up the number of time each individual succeeds in mating. Because we use both `polygynous_mating()` and `gaussian_mating()` at different times in the program, we should update both.

The logic behind this change is pretty simple. If a female successfully chooses a mate, then she produces offspring with that mate, so her mating success should go up by one. Given our mating system, females can only have a mating success of 0 or 1. If a male is chosen by a female as a mate, then he mates with her and his mating success should also go up by one. Males differ from females in that a male can potentially mate many times. If we just find the `if` statement that starts with `if (bMateFound)`, we can put the statements we need just inside the closing brace. There are two ways to find the closing brace. One way is to select the opening brace. The code editor should highlight both the opening and closing brace, so scroll down until you find the highlighted closing brace. The other way is to look for our comment (if we added one), which should say `// end of if (mate_found)` or something to that effect. Just above this brace (and below the closing brace for the `m` loop), add the following two lines of code:

```cpp
adult[i].MatingSuccess++;
adult[mateID].MatingSuccess++;
```

Both `gaussian_mating()` and `polygynous_mating()` have the same structure in this part of the function, so make the same change to both functions.

Return to the `calculate_values_adults()` function, now that all the pieces are in place to calculate the variables of interest. Add the following loop below the code you have already added:

```cpp
// Calculate mean mating success and mean trait values
    for (i = 0; i < PopulationSize; i++)
    {
        if (adult[i].Female)
The loop runs through all of the adults in the population, counting the number of males and females. It also accumulates terms to calculate the mean mating success of males and females as well as the mean preference in females and the mean trait value in males.

The calculation of the sex ratio is super easy, so add the code below to the `calculate_values_adults()` function.

```cpp
// Calculate the Sex Ratio.
if (Nma > 0)
    SexRatio = Nma / (Nma + Nfe);
else
    SexRatio = 0;
```

The next step is to calculate the opportunities for sexual selection and the mating differentials. The opportunity for sexual selection is the variance in mating success divided by the square of mean mating success. We have already calculated the mean, so we still need to calculate the variances for the two sexes. The mating differential is the covariance between trait values and relative mating success, so its calculation is very similar to that of the selection differential, which we previously calculated for the progeny. Both of these calculations can be accomplished at the same time with the following code:

```cpp
// Calculate opportunities for sexual selection
// and mating differentials
VarMSmales = 0;
VarMSfemales = 0;
MdiffMales = 0;
MdiffFemales = 0;

for (i = 0; i < PopulationSize; i++)
{
    if (adult[i].Female)
    {
        VarMSfemales = VarMSfemales + (adult[i].MatingSuccess - MeanMSfemales) *(adult[i].MatingSuccess - MeanMSfemales);
        if (MeanMSfemales > 0)
            MdiffFemales = MdiffFemales +
                (adult[i].Phenotype[1] - MeanTrait1females) *(adult[i].MatingSuccess / MeanMSfemales - 1);
    }
    else
    {
        VarMSmales = VarMSmales + (adult[i].MatingSuccess - MeanMSmales) *(adult[i].MatingSuccess - MeanMSmales);
        if (MeanMSmales > 0)
            MdiffMales = MdiffMales + (adult[i].Phenotype[0] - MeanTrait0males) *(adult[i].MatingSuccess / MeanMSmales - 1);
    }
} // end of i

if (Nfe > 0)
{
    VarMSfemales = VarMSfemales / Nfe;
    MdiffFemales = MdiffFemales / Nfe;
}
else
{
    VarMSfemales = 0;
    MdiffFemales = 0;
}

if (Nma > 0)
{
    VarMSmales = VarMSmales / Nma;
    MdiffMales = MdiffMales / Nma;
}
else
{
    VarMSmales = 0;
    MdiffMales = 0;
}

if (MeanMSmales > 0)
    Im = VarMSmales / (MeanMSmales*MeanMSmales);
else
    Im = 0;

if (MeanMSfemales > 0)
    If = VarMSfemales / (MeanMSfemales*MeanMSfemales);
CHAPTER 12

else
    If = 0;

In this code, the loop accumulates terms for the variances and the covariances. Then we divide these sums by either the number of females or the number of males, taking care not to allow the program to divide by zero. Finally, we calculate the opportunities for sexual selection by dividing the variances in mating success by the squares of the appropriate means. The only thing left to do is to output these values each generation, and we can tackle this challenge by adding the code below to save_population_variables(). just above the statement outfile.close().

    outfile << "", << SexRatio;
    outfile << "", << Im;
    outfile << "", << If;
    outfile << "", << MdiffMales;
    outfile << "", << MdiffFemales;

For the sake of completeness, here is the entire calculate_values_adults() function:

    void calculate_values_adults()
    {
        int i;
        double MeanMSmales, MeanMSfemales, MeanTrait0males, MeanTrait1females;
        double VarMSmales, VarMSfemales, Nma, Nfe;

        Nma = 0;
        Nfe = 0;
        MeanMSmales = 0;
        MeanMSfemales = 0;
        MeanTrait0males = 0;
        MeanTrait1females = 0;

        // Calculate mean mating success and mean trait values
        for (i = 0; i < PopulationSize; i++)
        {
            if (adult[i].Female)
            {
                Nfe++;
                MeanMSfemales = MeanMSfemales + adult[i].MatingSuccess;
                MeanTrait1females = MeanTrait1females + adult[i].Phenotype[1];
            }
            else
            {
                Nma++;
                MeanMSmales = MeanMSmales + adult[i].MatingSuccess;
                MeanTrait0males = MeanTrait0males + adult[i].Phenotype[0];
            }
        } // end of i

        if (Nma > 0)
        {
            MeanMSmales = MeanMSmales / Nma;
            MeanTrait0males = MeanTrait0males / Nma;
        }
        else
        {

    154
SMALL BUT IMPORTANT MODIFICATIONS

MeanMSmales = 0;
MeanTrait0males = 0;
}
if (Nfe > 0)
{
MeanMSfemales = MeanMSfemales / Nfe;
MeanTrait1females = MeanTrait1females / Nfe;
}
else
{
MeanMSfemales = 0;
MeanTrait1females = 0;
}

// Calculate the Sex Ratio.
if (Nma + Nfe > 0)
   SexRatio = Nma / (Nma + Nfe);
else
   SexRatio = 0;

// Calculate opportunities for sexual selection
// and mating differentials
VarMSmales = 0;
VarMSfemales = 0;
MdiffMales = 0;
MdiffFemales = 0;
for (i = 0; i < PopulationSize; i++)
{
   if (adult[i].Female)
   {
      VarMSfemales = VarMSfemales + (adult[i].MatingSuccess - MeanMSfemales) *(adult[i].MatingSuccess - MeanMSfemales);
      if (MeanMSfemales > 0)
         MdiffFemales = MdiffFemales + (adult[i].Phenotype[1] - MeanTrait1females) *(adult[i].MatingSuccess / MeanMSfemales - 1);
   }
   else
   {
      VarMSmales = VarMSmales + (adult[i].MatingSuccess - MeanMSmales) *(adult[i].MatingSuccess - MeanMSmales);
      if (MeanMSmales > 0)
         MdiffMales = MdiffMales + (adult[i].Phenotype[0] - MeanTrait0males) *(adult[i].MatingSuccess / MeanMSmales - 1);
   }
} // end of i
if (Nfe > 0)
{
   VarMSfemales = VarMSfemales / Nfe;
   MdiffFemales = MdiffFemales / Nfe;
}
else

CHAPTER 12

```c
{  
    VarMSfemales = 0;  
    MdiffFemales = 0;  
}
if (Nma > 0)
{
    VarMSmales = VarMSmales / Nma;  
    MdiffMales = MdiffMales / Nma;  
} else
{
    VarMSmales = 0;  
    MdiffMales = 0;  
}
if (MeanMSmales > 0)
    Im = VarMSmales / (MeanMSmales*MeanMSmales);
else
    Im = 0;
if (MeanMSfemales > 0)
    If = VarMSfemales / (MeanMSfemales*MeanMSfemales);
else
    If = 0;
}

Now we have a functional program that can be used to investigate the evolutionary dynamics of a system subject to sexual selection or a number of other topics in evolutionary biology. The program should compile with no errors or warnings. If you run it, you should see that you now have values for every column in the output file.

Adding Some Additional Flexibility

As it stands now, our program uses trait 0 as the ornament in males and trait 1 as the preference in females, but we might want our program to be a little more general, with sexual selection as a special case. For instance, we might want to be able to run a simulation without any sexual selection and without sex-limited selection. In this case, the two traits could be any arbitrary naturally selected traits. In fact, we should make our program general, so that we can compare our results to those of similar simulations that have been run in the past. If we can replicate the results of at least a few published studies, then we can feel confident that our simulation is working as intended.

We already have the option of having no mate choice, by setting GaussianPreferenceVariance to zero, and we also have the option of having no selection in our sex_limited_selection() function by setting the selection strength parameters to zero. However, our natural_selection() function does not yet allow us to have no selection, and we use this function during the initial generations of the simulation. In this function, if we set the selection strength equal to zero, we will end up dividing by zero and crashing the program.

We should allow our program to have no selection in both the initial generations and the experimental generations, in case we decide to study the effects of genetic drift. Navigate over to your natural_selection() function and find the loop that cycles through the progeny. It should look something like:
for (i = 0; i < Nprogeny; i++)
{
    //Bivariate selection on traits 0 and 1:
    dSu = (progeny[i].Phenotype[0] - local_opt[0]) / SSqrt[0];
    dSv = (progeny[i].Phenotype[1] - local_opt[1]) / SSqrt[1];
    survival_prob = exp((2 * local_sel_corr*dSu*dSv - dSu*dSu - dSv*dSv) / dSc);

    dRnum1 = genrand();
    if (dRnum1 < survival_prob)
        progeny[i].Alive = true;
    else
        progeny[i].Alive = false;
} // end of i loop

We will have to modify our code to deal with four possible scenarios: selection on both traits (it already does that), selection on trait 0 only, selection on trait 1 only, and no selection. Obviously, we will need some if statements. We can use if and else or just a series of if statements, depending on how much we want to think. For now, I will just go with if statements to keep it simple. Change this loop to look like the code below.

for (i = 0; i < Nprogeny; i++)
{
    //Bivariate selection on traits 0 and 1:
    if (SSqrt[0] > 0 && SSqrt[1] > 0)
    {
        dSu = (progeny[i].Phenotype[0] - local_opt[0]) / SSqrt[0];
        dSv = (progeny[i].Phenotype[1] - local_opt[1]) / SSqrt[1];
        survival_prob = exp((2 * local_sel_corr*dSu*dSv - dSu*dSu - dSv*dSv) / dSc);
    }

    // Selection on trait 0 only
    if (SSqrt[0] > 0 && SSqrt[1] == 0)
    {
        survival_prob = exp(-1.0 *
            (progeny[i].Phenotype[0] - local_opt[0])*
            (progeny[i].Phenotype[0] - local_opt[0])
            / (2 * local_sel_str[0]));
    }

    // Selection on trait 1 only
    if (SSqrt[0] == 0 && SSqrt[1] > 0)
    {
        survival_prob = exp(-1.0 *
            (progeny[i].Phenotype[1] - local_opt[1])*
            (progeny[i].Phenotype[1] - local_opt[1])
            / (2 * local_sel_str[1]));
    }

    // No selection on either trait
    if (SSqrt[0] == 0 && SSqrt[1] == 0)
    {
        survival_prob = 1;
    }

    dRnum1 = genrand();
    if (dRnum1 < survival_prob)
progeny[i].Alive = true;
else
progeny[i].Alive = false;
} // end of i loop

The rest of `natural_selection()` should stay the way it is. The equations for the survival probability when selection is acting on only one trait should look familiar, because they are the same as the equations we used in `sex_limited_selection()`. The other option we might want is to toggle between sex-limited selection and natural selection on both sexes. Consequently, we need to make it possible for sex-limited selection to occur in the initial generations. We also need to make it possible for non-sex-limited selection, as implemented in our `natural_selection()` function to occur in the experimental generations. We will need a class-level variable to keep track of the type of selection we want in the initial and experimental generations. We will also have to modify `sex_limited_selection()` to have the correct selection strengths in the initial generations and experimental generations. In the `private:` section of `simulation_engine`, declare:

```cpp
bool InitialSelectionSexLimited, ExperimentalSelectionSexLimited;
```

And in the class constructor, set these variables to values. For now, we will set them to “true”, so that selection is sex-limited throughout each simulation run. Somewhere in the `// Initialize the Parameters` section of `simulation_engine()`, add:

```cpp
InitialSelectionSexLimited = true;
ExperimentalSelectionSexLimited = true;
```

Next, we need to head over to our `sex_limited_selection()` function, which we will modify to be cognizant of whether the simulation is in the initial or experimental phase of the simulation. First, change the declaration of the function to accept an argument, as follows:

```cpp
void sex_limited_selection(bool init_gens)
```

As in `natural_selection()`, we need to use either the selection strengths from the initial generations or the values for the experimental generations, so we need to declare new variables. Add the following declaration at the beginning of `sex_limited_selection()`:

```cpp
double local_sel_str[2];
```

Then, depending on whether we are in the initial or experimental generations, we need to set the local variables equal to the appropriate selection strengths and the proper optima, as we have previously done in `natural_selection()`. The code pasted below provides the appropriate `if-else` statement and should be included immediately below the variable declarations.

```cpp
if (init_gens)
{
    local_sel_str[0] = InitialSelectionStrength[0];
    local_sel_str[1] = InitialSelectionStrength[1];
    optimum[0] = InitialOptimum[0];
    optimum[1] = InitialOptimum[1];
}
else
{
```
SMALL BUT IMPORTANT MODIFICATIONS

local_sel_str[0] = SelectionStrength[0];
local_sel_str[1] = SelectionStrength[1];
optimum[0] = Optimum[0];
optimum[1] = Optimum[1];
}

Finally, for the rest of the function, anywhere you see SelectionStrength, you should change it to local_sel_str. There are only two places, so make the two changes. At the end of all of this, the new sex_limited_selection() function should look like this:

```cpp
void sex_limited_selection(bool init_gens)
{
    int i;
    double survival_prob, dRnum1, optimum[2];
    double local_sel_str[2];

    if (init_gens)
    {
        local_sel_str[0] = InitialSelectionStrength[0];
        local_sel_str[1] = InitialSelectionStrength[1];
        optimum[0] = InitialOptimum[0];
        optimum[1] = InitialOptimum[1];
    } else
    {
        local_sel_str[0] = SelectionStrength[0];
        local_sel_str[1] = SelectionStrength[1];
        optimum[0] = Optimum[0];
        optimum[1] = Optimum[1];
    }

    for (i = 0; i < Nprogeny; i++)
    {
        // Calculate survival probabilities for males and females separately
        if (!progeny[i].Female)
        {
            if (local_sel_str[0] > 0)
            {
                survival_prob = exp(-1.0 *
                    (progeny[i].Phenotype[0] - optimum[0])*
                    (progeny[i].Phenotype[0] - optimum[0])
                    / (2 * local_sel_str[0]));
            } else
            {
                survival_prob = 1; // SelectionStrength = 0 means no selection.
            }
        } else
        {
            if (local_sel_str[1] > 0)
            {
                survival_prob = exp(-1.0 *
                    (progeny[i].Phenotype[1] - optimum[1])*
                    (progeny[i].Phenotype[1] - optimum[1])
                    / (2 * local_sel_str[1]));
            } else
            {
                survival_prob = 1; // SelectionStrength = 0 means no selection.
            }
        }
    }
}
CHAPTER 12

} else
{
  survival_prob = 1; // SelectionStrength = 0 means no selection.
}

// Generate a random number to see if the individual survives
dRnum1 = genrand();
if (dRnum1 < survival_prob)
  progeny[i].Alive = true;
else
  progeny[i].Alive = false;
} // end of i loop

We have the functions we need, but we do not yet call them appropriately in the main part of the program. If you look at the “.cpp” file, you will see that we call natural_selection() in the initial generations and sex_limited_selection() in the experimental generations. Obviously, we need these calls to depend on the values of our new Boolean variables, which indicate whether we want sex-limited or non-sex-limited selection. One way to implement this change would be to add if statements within the ig loop and the generations loop. A more elegant solution might be to add a new function to our simulation_engine that automatically chooses the correct type of selection based on the parameter values. Back in your simulation_engine class, add another function called selection() by using the following code:

```cpp
void selection(bool initial_generations)
{
  if (initial_generations)
  {
    if (InitialSelectionSexLimited)
      sex_limited_selection(true);
    else
      natural_selection(true);
  }
  else
  {
    if (ExperimentalSelectionSexLimited)
      sex_limited_selection(false);
    else
      natural_selection(false);
  }
}
```

This function takes an argument indicating whether we are in the initial generations of the function or not. If we are in the initial generations, then the function uses the InitialSelectionSexLimited variable to decide whether to call sex_limited_selection() or natural_selection(). It also passes along the fact that we are in the initial generations to the respective function. If we are not in the initial generations, then this function uses the ExperimentalSelectionSexLimited variable to choose which function to call. Now we can use selection() as our function wherever we want any type of selection in our model. In principle, if we wished to add other types of selection in the future, we could just add a new variable to indicate what type of selection we want, and then modify selection() to sort out which function should be called. So the remaining step is to replace any call to
SMALL BUT IMPORTANT MODIFICATIONS

sex_limited_selection() or natural_selection() in our “.cpp” file with a call to selection(). After these changes, your “.cpp” file should look like this:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations;

    simulation_engine my_sim;
    my_sim.display_parameters();

    bool initialization_success;
    initialization_success = my_sim.initialize_population();
    if (!initialization_success)
    {
        std::cout << "Simulation Initialization Failure!\n";
        return 0;
    }

    int ig;
    for (ig = 0; ig < my_sim.getNumberOfInitialGenerations(); ig++)
    {
        my_sim.polygynous_mating();
        my_sim.mutation();
        my_sim.selection(true);
        my_sim.population_regulation();
    }

    // Turn the life cycle halfway, so the
    // experimental generations start with progeny
    my_sim.calculate_values_progeny();
    my_sim.polygynous_mating();
    my_sim.mutation();

    for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
    {
        if (!my_sim.is_extinct())
        {
            my_sim.selection(false);
            my_sim.calculate_values_progeny();
            my_sim.population_regulation();
            my_sim.gaussian_mating();
            my_sim.mutation();
            my_sim.calculate_values_adults();
            my_sim.save_population_variables(generations);
        }
    }

    char end_it;
    std::cout << "Enter any character to exit...";
    std::cin >> end_it;
    return 0;
}
```
CHAPTER 12

We might want to output whether we are using sex-limited selection or not with our other parameter values. This task is easy to accomplish by modifying our display_parameters() function. Change the code for this function to look like this:

```cpp
void display_parameters()
{
    std::cout << "Parameter Values:\n";
    // Demographic Parameters
    std::cout << "Demographic Parameters:\n";
    std::cout << "No_Generations: \t" << NoOfGenerations << "\n";
    std::cout << "Initial_Pop_Size: \t" << PopulationSize << "\n";
    std::cout << "Carrying_Capacity: \t" << CarryingCapacity << "\n";
    std::cout << "Female_Fecundity: \t" << Fecundity << "\n";

    // Mating Parameters
    std::cout << "Mating_Parameters:\n";
    std::cout << "Max_Mating_Enc.: \t" << MaxMatingEncounters << "\n";
    std::cout << "Gaussian_Pref_Var:\t" << GaussianPreferenceVariance << "\n";

    // Quantitative Genetic Parameters
    std::cout << "Quantitative_Genetic_Parameters:\n";
    std::cout << "No_Loci_Trait0: \t" << NumLociTrait0 << "\n";
    std::cout << "No_Loci_Trait1: \t" << NumLociTrait1 << "\n";
    std::cout << "Env_Variance_Trt0:\t" << EnvironmentalVariance[0] << "\n";
    std::cout << "Env_Variance_Trt1:\t" << EnvironmentalVariance[1] << "\n";

    // Mutational Parameters
    std::cout << "Mutational_Parameters:\n";
    std::cout << "Mut_Var_Trait0: \t" << MutationalVariance[0] << "\n";
    std::cout << "Mut_Var_Trait1: \t" << MutationalVariance[1] << "\n";
    std::cout << "Mut_Correlation: \t" << MutationalCorrelation << "\n";
    std::cout << "Mutation_Rate: \t" << MutationRatePerLocus << "\n";

    // Selection Parameters
    std::cout << "Selection_Parameters:\n";
    std::cout << "Omega_Trait0: \t" << SelectionStrength[0] << "\n";
    std::cout << "Omega_Trait1: \t" << SelectionStrength[1] << "\n";
    std::cout << "Selection_Corr: \t" << SelectionalCorrelation << "\n";
    std::cout << "Optimum_Trait0 \t" << Optimum[0] << "\n";
    std::cout << "Optimum_Trait1 \t" << Optimum[1] << "\n";
    if (ExperimentalSelectionSexLimited)
        std::cout << "Sex_Limited_Sel: \tttrue\n";
    else
        std::cout << "Sex_Limited_Sel: \tfalse\n";

    // Initial Generations
    std::cout << "Initial_Generations_Parameters:\n";
    std::cout << "No_Initial_Gens: \t" << NoOfInitialGenerations << "\n";
    std::cout << "Init_Omega_Trait0: \t" << InitialSelectionStrength[0] << "\n";
    std::cout << "Init_Omega_Trait1: \t" << InitialSelectionStrength[1] << "\n";
    std::cout << "Init_Sel_Corr: \t" << InitialSelectionalCorrelation << "\n";
    std::cout << "Init_Opt_Trt0 \t" << InitialOptimum[0] << "\n";
    std::cout << "Init_Opt_Trt1 \t" << InitialOptimum[1] << "\n";
    if (InitialSelectionSexLimited)
```

162
The only changes here are the additions of two `if-else` statements, which output either true or false, depending on the values of `ExperimentalSelectionSexLimited` and `InitialSelectionSexLimited`.

Chapter Summary

In this chapter, we made some important changes to the program, improving it to be more complete, more effective, and more general. For instance, we added sex-limited selection for studies involving sexually selected traits and preferences, and we improved the Gaussian mate choice function, so that receptive females encounter only males during their search for Mr. Right. We added some initial generations, during which we do not keep track of variables of interest, so that we can let the simulation arrive at a quasi-equilibrium before we start measuring population-level variables. We also added calculations for a number of important variables in progeny and adults, and we made sure that we were calculating them at the correct point of the life cycle so that all of the variables on one row of output would be from the same generation’s individuals. Finally, we modified the program in a way that allows specification of whether or not we wish to have sex-limited selection in the initial and experimental generations. This change gives us the flexibility to model either traits and preferences or an arbitrary pair of traits subject to natural selection in both sexes. As far as programming skills, this chapter used the skills you already have to modify the program, without introducing any new programming concepts.
Chapter 13. Validating the Simulation

Whether you are writing simulations for fun or to conduct serious evolutionary analysis, you will want to check to make sure your simulation is really doing what you think it is. Small typographical errors and failures of logic can introduce bugs into a program. Some of these bugs will cause the program to crash. While these types of bugs are annoying and can be difficult to diagnose, their presence is known thanks to the complete failure of the program. The more nefarious kind of bug is one that does not result in a crash but instead produces a program that runs just fine except that it reports bogus results. These types of bugs can go unnoticed for years in some cases, and they can have devastating effects. The best way to make sure your program contains no hidden bugs is to compare it against another program that has been used to produce published results. In science, each development builds upon past developments, so each time we develop a new approach, we should endeavor to replicate the older approach upon which the new approach is based. The model in this tutorial is based upon work from the early 2000s designed to examine the evolution of the genetic architecture (as described by the G-matrix) in a two-trait system. Thus, we should be able to use our new program to generate data that agrees quite closely with the data from previous simulation-based models.

We should start with the paper entitled “Stability of the G-matrix in a population experiencing pleiotropic mutation, stabilizing selection, and genetic drift” by A. G. Jones, S. J. Arnold, and R. Bürger (Evolution, 2003, 57(8): 1747-1760). If we read through the methods section of this paper, we will see many similarities to our simulation, but there are some differences as well. Unfortunately, showing that our simulation is similar to the previous work despite some important differences does not represent sufficiently reliable validation. We need to make our model identical, insofar as possible, to the model described by Jones et al. (2003). Consequently, we need to scour through their methods and determine the aspects that we can achieve by setting parameter values in our model versus those that require additional programming. Fortunately, most of our model is nearly identical to their model, but our model does have some noticeable differences.

The first, and most troubling, difference is that our model always uses some form of polygynous mating, where each female mates exactly once and each male can mate with any number of females. The Jones et al. (2003) model, however, uses monogamous mating, where each individual has exactly one mate and the sex ratio is always exactly half male and half female. This difference may seem somewhat small, but it actually matters quite a bit. The mating system influences the effective population size, and a monogamous mating system results in a larger effective population size compared to a polygynous mating system. The effective population size has a direct impact on the rate of genetic drift, and the dynamics of the genetic architecture are strongly influenced by drift. We can conclude from these considerations that our model with polygynous mating will not be directly comparable to the Jones et al. (2003) model. Thus, we will need to implement our own monogamous mating function.

Monogamous Mating

The implementation of monogamous mating is more difficult than it might appear at first. For instance, it might be tempting to have adult[0] mate with adult[1], and adult[2] mate with adult[3], and so forth. However, given how we produce progeny and implement population regulation, siblings are located adjacent to one another in the array of adults. If we implemented monogamy in the way described here, we would have a disproportionate amount of inbreeding, which would certainly
affect the variables we are interested in tracking. This example illustrates how a nefarious bug might make its way into a program. We are free to implement the type of monogamy I just described, and the program would run just fine. In fact, without knowing more about how our various functions work, this form of monogamy might seem completely justified. But we do know how our functions work, and we have to always be watching out for unintended consequences of the decisions we make. The bottom line is that we will need a function that takes a more randomized approach.

Even though this approach may not be the most efficient, we will just randomly choose half of the individuals, checking each time that the individual in question has not already been chosen, and then we will assume that these individuals are males. The other half will be females. Then we will iterate, in order, through the females, choosing a male for each female from the randomized list of males. If the sex ratio is unequal, one of the individuals will not mate. In implementing this approach, we are ignoring the sex assigned at birth and assigning a new sex for the purposes of monogamous mating. For this model, this way of doing things is fine, because the assignment of sex was arbitrary anyway.

Start by creating a new function in your simulation_engine called `monogamy()`. Add lines of code and comments until it looks something like the code pasted below.

```c++
void monogamy()
{
    // This function implements strict monogamy.
    // Under this mating system, each female mates once,
    // and each male also mates once. Sex is assigned
    // on the fly, and the sex ratio is equalized.
    // If there's an uneven number of individuals,
    // one individual does not mate.

    int i, j, m;
    int iPC;
    int maleID, femaleID, rnum;
    double dRnum;
    int number_females;

    bool *already_chosen = new bool[PopulationSize];
    int *male_list = new int[PopulationSize];
    int *female_list = new int[PopulationSize];

    for (i = 0; i < PopulationSize; i++)
    {
        adult[i].Female = true;
        already_chosen[i] = false;
    }

    delete[] already_chosen;
    delete[] male_list;
    delete[] female_list;
}
```

This code describes what the function is intended to do and declares some variables. The code also declares some pointers and allocates memory for the pointers using the `new` operator. At the end of the function, we use `delete[]` to release this memory. If we forget the `delete[]` statements, we have a memory leak and our program will just keep eating more and more memory until it crashes. We also use a `for` loop to set every adult’s sex to female and to set the values of our `already_chosen` array to false.
We will use the latter array to keep track of which adults are converted to males and added to our list of males.

The remainder of the code will be added between the closing brace of the $i$ loop and the first `delete[]` statement. Add the following:

```c
// We need to make half the population male
// If the population contains an odd number of
// individuals, then we need one more male than
// female, so below when all the females mate
// they do not run out of males.

int number_males = PopulationSize / 2;
if (PopulationSize % 2 != 0)
    number_males++;

// Now make a list of unique males
for (i = 0; i < number_males; i++)
{
    rnum = randnum(PopulationSize);
    while (already_chosen[rnum])
        rnum = randnum(PopulationSize);
    adult[rnum].Female = false;
    already_chosen[rnum] = true;
    male_list[i] = rnum;
}

// Make a list of females
number_females = 0;
for (i = 0; i < PopulationSize; i++)
{
    if (adult[i].Female)
    {
        female_list[number_females] = i;
        number_females++;
    }
}
```

The above code first determines the number of males we need for our monogamous mating system. Basically, we need half of the population to be male. Remember that division involving an integer always rounds down to the nearest integer, so if the population size is 51, $\text{PopulationSize} / 2 = 25$. The plan is to cycle through the females in the population, as we have done before, so we need to ensure that we have at least as many males as females. For an even population size, it is easy enough to have exactly half males and half females, but for an odd population size, we need to have one more male than the number of females. The first `if` statement takes care of this issue. Recall that the modulus operator ($\%$) returns the remainder from an integer division, so $\text{PopulationSize} \% 2$ will be zero if $\text{PopulationSize}$ is an even number and one if $\text{PopulationSize}$ is an odd number. Effectively, the division operation coupled with the `if` statement just divides by two, with rounding up instead of down.

The next little bit of code populates our list of males. We randomly choose an adult, and check to make sure that adult has not been chosen. The `while` loop just keeps picking random numbers until we get an individual who has not been chosen. Then we set the newly chosen individual’s sex to male, update our list of those chosen, and add the new male to our list of males. In the third bit of code above, we cycle
through all the individuals in the population and make a list of adults that remained females. With these lists of males and females in hand, we can easily produce progeny. Because we populated our list of males by randomly choosing individuals, we can just pair our lists in order for the production of progeny, and mating will be random.

The last section of code we need to add handles the production of progeny. Add the following (again, above the first delete[] statement):

```c
iPC = 0;
for (i = 0; i < number_females; i++)
{
    maleID = male_list[i];
    femaleID = female_list[i];

    // Produce progeny
    for (m = 0; m < Fecundity; m++)
    {
        if (iPC >= NprogMax)
            iPC = NprogMax - 1;

        // First, let's take care of Mendelian assortment of the trait0 loci
        for (j = 0; j < NumLociTrait0; j++)
        {
            // The progeny needs one maternal allele and one paternal allele.
            // The maternal allele will be from adult[femaleID] (the mother), and
            // we determine which allele with basically a coinflip. The
            // function genrand() produces a value in the range [0,1), so
            // if this number is less than 0.5 we choose one allele. Otherwise,
            // we choose the other allele.
            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele1trait0[j] = adult[femaleID].Allele1trait0[j];
            else
                progeny[iPC].Allele1trait0[j] = adult[femaleID].Allele2trait0[j];

            // The procedure is the same for the father, adult[maleID].
            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele2trait0[j] = adult[maleID].Allele1trait0[j];
            else
                progeny[iPC].Allele2trait0[j] = adult[maleID].Allele2trait0[j];
        } // end of j loop

        // Second, take care of Mendelian assortment for the trait1 loci.
        // The procedure is essentially identical to that for trait0.
        for (j = 0; j < NumLociTrait1; j++)
        {
            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele1trait1[j] = adult[femaleID].Allele1trait1[j];
            else
                progeny[iPC].Allele1trait1[j] = adult[femaleID].Allele2trait1[j];
```
dRnum = genrand();
if (dRnum < 0.5)
    progeny[iPC].Allele2trait1[j] = adult[maleID].Allele1trait1[j];
else
    progeny[iPC].Allele2trait1[j] = adult[maleID].Allele2trait1[j];
} // end of j loop

// The pleiotropic loci are more complicated. We have to be sure to
// keep the allelic effects (on the two traits) together for each allele.
// Otherwise, the loci would not behave realistically like actual
// pleiotropic loci.
for (j = 0; j < NumLociBoth; j++)
{
    dRnum = genrand();
    if (dRnum < 0.5)
    {
        progeny[iPC].Allele1both[j][0] = adult[femaleID].Allele1both[j][0];
        progeny[iPC].Allele1both[j][1] = adult[femaleID].Allele1both[j][1];
    }
    else
    {
        progeny[iPC].Allele1both[j][0] = adult[femaleID].Allele2both[j][0];
        progeny[iPC].Allele1both[j][1] = adult[femaleID].Allele2both[j][1];
    }
    dRnum = genrand();
    if (dRnum < 0.5)
    {
        progeny[iPC].Allele2both[j][0] = adult[maleID].Allele1both[j][0];
        progeny[iPC].Allele2both[j][1] = adult[maleID].Allele1both[j][1];
    }
    else
    {
        progeny[iPC].Allele2both[j][0] = adult[maleID].Allele2both[j][0];
        progeny[iPC].Allele2both[j][1] = adult[maleID].Allele2both[j][1];
    }
} // end of j loop

progeny[iPC].set_sex();
progeny[iPC].calculate_genotypic_values(NumLociTrait0, NumLociTrait1, NumLociBoth);
progeny[iPC].calculate_phenotype(EnvironmentalStDev[0], EnvironmentalStDev[1]);
iPC++;
} // end of m loop
adult[femaleID].MatingSuccess++;
adult[maleID].MatingSuccess++;
// This function implements strict monogamy.
// Under this mating system, each female mates once,
// and each male also mates once. Sex is assigned
// on the fly, and the sex ratio is equalized.
// If there's an uneven number of individuals,
// one individual does not mate.

int i, j, m;
int iPC;
int maleID, femaleID, rnum;
double dRnum;
int number_females;

bool *already_chosen = new bool[PopulationSize];
int *male_list = new int[PopulationSize];
int *female_list = new int[PopulationSize];

for (i = 0; i < PopulationSize; i++)
{
    adult[i].Female = true;
    already_chosen[i] = false;
}

// We need to make half the population male
// If the population contains an odd number of
// individuals, then we need one more male than
// female, so below when all the females mate
// they do not run out of males.

int number_males = PopulationSize / 2;
if (PopulationSize % 2 != 0)
    number_males++;

// Now make a list of unique males
for (i = 0; i < number_males; i++)
{
    rnum = randnum(PopulationSize);
    while (already_chosen[rnum])
        rnum = randnum(PopulationSize);
    adult[rnum].Female = false;
    already_chosen[rnum] = true;
    male_list[i] = rnum;
}

// Make a list of females

number_females = 0;
for (i = 0; i < PopulationSize; i++)
{
    if (adult[i].Female)
    {
        female_list[number_females] = i;
        number_females++;
    }
}
iPC = 0;
for (i = 0; i < number_females; i++)
{
    maleID = male_list[i];
    femaleID = female_list[i];

    // Produce progeny

    for (m = 0; m < Fecundity; m++)
    {
        if (iPC >= NprogMax)
            iPC = NprogMax - 1;

        // First, let's take care of Mendelian assortment of the trait0 loci
        for (j = 0; j < NumLociTrait0; j++)
        {
            // The progeny needs one maternal allele and one paternal allele.
            // The maternal allele will be from adult[femaleID] (the mother), and
            // we determine which allele with basically a coinflip. The
            // function genrand() produces a value in the range [0,1), so
            // if this number is less than 0.5 we choose one allele. Otherwise,
            // we choose the other allele.

            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele1trait0[j] = adult[femaleID].Allele1trait0[j];
            else
                progeny[iPC].Allele1trait0[j] = adult[femaleID].Allele2trait0[j];

            // The procedure is the same for the father, adult[maleID].
            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele2trait0[j] = adult[maleID].Allele1trait0[j];
            else
                progeny[iPC].Allele2trait0[j] = adult[maleID].Allele2trait0[j];
        } // end of j loop

        // Second, take care of Mendelian assortment for the trait1 loci.
        // The procedure is essentially identical to that for trait0.
        for (j = 0; j < NumLociTrait1; j++)
        {
            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele1trait1[j] = adult[femaleID].Allele1trait1[j];
            else
                progeny[iPC].Allele1trait1[j] = adult[femaleID].Allele2trait1[j];

            dRnum = genrand();
            if (dRnum < 0.5)
                progeny[iPC].Allele2trait1[j] = adult[maleID].Allele1trait1[j];
            else
                progeny[iPC].Allele2trait1[j] = adult[maleID].Allele2trait1[j];
        } // end of j loop

        // The pleiotropic loci are more complicated. We have to be sure to keep
To convert our simulation to monogamy, all that remains is to flip over to the “.cpp” file and change the calls to “polygynous_mating()” and “gaussian_mating()” to “monogamy()”. Go ahead and make these changes. We can change them back later if we wish, after we verify that our simulation is capable of producing results that match those reported by Jones et al. (2003). The program should compile without any errors. If you run it, you will see that the output is similar to the output obtained under the other mating systems, with the exception of the variables related to the mating system (i.e., the last five columns). Under monogamy, the sex ratio is always 0.5, and the other mating system variables are all zero, because monogamy results in no variation in mating success (everyone has a single mate).
CHAPTER 13

Means and “Means of Means”

A quick perusal of the Jones et al. (2003) paper also shows that most of the values they reported were means from 20 independent runs of the simulation. First they calculated the mean from each of 20 runs and then reported the mean across those twenty runs in their tables. Our program outputs the data from a single run into a file, which we can open in a spreadsheet program. In principle, we could run the program 20 times for each parameter combination, open each file in a spreadsheet, calculate the means from each run, copy these means into a new spreadsheet, with one row for each of 20 runs, and then calculate the means in that spreadsheet. That sounds like it would take a long time. In addition, that time spent would be an embarrassing waste for someone who can program. Why not just automate all of this, as Jones et al. (2003) surely must have done?

The structure of our program is continuing to get more and more complex, with layers of complexity piled upon one another. As long as we keep it all organized and compartmentalized, however, we should be fine. We are coming face-to-face with the need to add yet another layer of complexity. We need to somehow set up a program that will perform many simulation runs consecutively (or at the same time), store the data, and calculate summary statistics across generations and runs. This new program architecture is different enough from our current program, that I think we should consider starting a new project. One of the great aspects of object-oriented programming is that we can use the classes we have already developed as the basis for the new project.

We have worked with the same project for so long now that you have probably forgotten how to create a new one. Open Visual Studio 2017 and select File -> New -> Project... from the File menu. In the dialog window, go to the Visual C++ menu and click on “Empty Project”. Name your project by typing a name in the text window near the bottom of the screen. Something like “Chapter13project” would be a fine name, but feel free to use any name you wish. Make sure the checkbox next to “Create directory for solution” is checked (it should be by default), and click the OK button. Click on the Project -> Properties menu item and, under the Configuration Properties -> General menu, change “Character Set” to “Not Set”. Click OK to apply this change and close the dialog.

You will notice that your new, empty project, contains nothing, which should be expected. We will want to import our header files from our previous project. We might also want to import the “.cpp” file just to give us a starting place. To import the header files, you need to navigate to the folder on your computer where Visual Studio saves your projects. On my computer, the folder is in the following path: This PC > Local Disk (C:) > Users > agjon > source > repos. I actually made a shortcut to repos and placed this shortcut on my desktop for easy access. Within your repos folder, you should find a folder named for the project we have been building over the last several chapters, and within that folder, you should find another folder with the same name. Your header files (“.h”) and C++ file (“.cpp”) should be there. Copy these files (two “.h” files and one “.cpp” file) and navigate to your new project folder. Paste these files into the same relative location in the folders associated with your new project. In my case, I would paste the files into This PC > Local Disk (C:) > Users > agjon > source > repos > Chapter13project > Chapter13project. That folder should also contain files with names like “Chapter13project.vcxproj” and “Chapter13project.vcxproj.filters”. While you are here, go ahead and rename the “.cpp” file to something more accurate, like “Chapter13example.cpp”.

Return to Visual Studio, where your project should still be open. In the Solution Explorer window, right click on “Header Files” and select Add -> Existing Item. A dialog should pop up to allow you to pick a file to add to your project. The dialog will default to the ... repos > Chapter13project > Chapter13project directory, where you will see the files you just copied over. Go ahead and add the two header files, “MTwisterFunctions.h” and “simulation_engine.h”. They should appear under “Header Files” in your Solution Explorer window. Now right click on “Source Files” and select Add - ->
**Existing Item.** Choose your “.cpp” file, which you recently renamed something like “Chapter13example.cpp”. That file should now appear in your Solution Explorer window under “Source Files”. If you compile and run the program, it should work fine, and in fact will be the exact same program that we finished developing earlier in this chapter, except with a different name.

Probably the easiest way to run multiple simulations is to create multiple objects of the simulation_engine, each of which runs one replicate of the simulation under the proposed parameter combinations. Before we make this addition, however, we need to ensure that our simulation_engine is doing everything we need for the scaling up process. One obvious shortcoming is that the simulation_engine does not keep track of the means of the variables across the simulation run. We will need to change the code to calculate the means of variables of interest. We can tackle this problem in a number of different ways. For example, we could sum up each variable as the simulation runs and then divide by the number of generations to get a mean. Alternatively, we could keep track of all of the data and then calculate the means at the end. The latter approach would take more time and memory than the former. We might be able to save time in the end, however, by not saving the data every single generation. Instead, we could save the data at the very end of each simulation run. This latter approach also demonstrates some additional programming techniques, so we will take this path.

**A Side Trip into the Standard Template Library**

To modify the simulation_engine so that it keeps track of its data, we obviously need a variable (or set of variables) in which to store the information. Some useful data containers are already included in C++ in what is known as the Standard Template Library (STL). To illustrate the STL and its usage, we will focus on one container in particular, which is known as a **vector**. The STL contains several other types of containers, such as the array, deque, forward_list, and list, all of which can be useful for various purposes. If you wish to know more about these other containers, just search the internet and read up on the STL. One of the most useful containers is the vector, which is basically a one-dimensional list, to which we can add elements as we go along. The advantage to a vector is that its memory management is handled automatically, so we do not need to use the **new** and **delete** operators to allocate and free memory, and it has a number of built-in functions that allow easy manipulation of the data it contains. We will see some of these functions as we illustrate their use by example.

To store the data for our simulation runs, while keeping everything simple and organized, I suggest we introduce a new class in which to store the values from each generation and eventually the means across generations. Each variable will have a list of values, one for each generation, so we can use vectors to store these lists. We will need quite a few vectors, however, because each variable needs its own vector and we already have many variables. To use vectors in your program, you have to invoke the proper **#include** statement, so add the following to the very top of your “simulation_engine.h” header file, along with the other **#include** statements:

```cpp
#include <vector>
```

Now add the class declaration pasted below. You will want to add it below the individual class declaration and above the simulation_engine class declaration.

```cpp
class simulation_data
{
public:
  std::vector<double> vN, vZbar0, vZbar1;
  std::vector<double> vP00, vP11, vP01, vRp, vGbar0, vGbar1;
  std::vector<double> vG00, vG11, vG01, vRg, vLambda1, vLambda2;
  std::vector<double> vEvecX, vEvecY, vAngle, vSize, vEccen;
```
CHAPTER 13

```cpp
std::vector<double> vStrt0, vStrt1, vClmbd1, vClmbd2, vcAng;
std::vector<double> vcSize, vcEcc, vcG00, vcG11, vcG01, vcRg;
std::vector<double> vASR, vIm, vIf, vMdifM, vMdifF;

double meanN, meanZbar0, meanZbar1;
double meanP00, meanP11, meanP01, meanRp, meanGbar0, meanGbar1;
double meanG00, meanG11, meanG01, meanRg, meanLambda1, meanLambda2;
double meanEvecX, meanEvecY, meanAngle, meanSize, meanEccen;
double meanStrt0, meanStrt1, meancLmbd1, meancLmbd2, meancAng;
double meancSize, meancEcc, meancG00, meancG11, meancG01, meancRg;
double meanASR, meanIm, meanIf, meanMdifM, meanMdifF;
```

The simulation outputs a lot of variables, so we need a lot of vectors to hold the data. In the declaration of a vector, we must specify what type of variable it will hold, and this specification is placed within the <> symbols. Vectors can contain virtually anything, including classes, so the vector container is quite flexible. We also declared variables to hold the mean for each vector, which will allow us to calculate the means when we get around to doing so. In fact, we might as well add a function to this new class to calculate the means. Add the following function to the new simulation_data class:

```cpp
void calculate_means()
{
    size_t i;
    double dNgens = 0;

    // Set all means to zero
    meanN = 0;
    meanZbar0 = 0;
    meanZbar1 = 0;
    meanP00 = 0;
    meanP11 = 0;
    meanP01 = 0;
    meanRp = 0;
    meanGbar0 = 0;
    meanGbar1 = 0;
    meanG00 = 0;
    meanG11 = 0;
    meanG01 = 0;
    meanRg = 0;
    meanLambda1 = 0;
    meanLambda2 = 0;
    meanEvecX = 0;
    meanEvecY = 0;
    meanAngle = 0;
    meanSize = 0;
    meanEccen = 0;
    meanStrt0 = 0;
    meanStrt1 = 0;
    meancLmbd1 = 0;
    meancLmbd2 = 0;
    meancAng = 0;
    meancSize = 0;
    meancEcc = 0;
    meancG00 = 0;
    meancG11 = 0;
    ```
VALIDATING THE SIMULATION

```cpp
meanCG01 = 0;
meanRg = 0;
meanASR = 0;
meanIm = 0;
meanIf = 0;
meanMdifM = 0;
meanMdifF = 0;

for (i = 0; i < vN.size(); i++)
{
    meanN = meanN + vN[i];
    meanZbar0 = meanZbar0 + vZbar0[i];
    meanZbar1 = meanZbar1 + vZbar1[i];
    meanP00 = meanP00 + vP00[i];
    meanP11 = meanP11 + vP11[i];
    meanP01 = meanP01 + vP01[i];
    meanRp = meanRp + vRp[i];
    meanGbar0 = meanGbar0 + vGbar0[i];
    meanGbar1 = meanGbar1 + vGbar1[i];
    meanG00 = meanG00 + vG00[i];
    meanG11 = meanG11 + vG11[i];
    meanG01 = meanG01 + vG01[i];
    meanRg = meanRg + vRg[i];
    meanLambda1 = meanLambda1 + vLambda1[i];
    meanLambda2 = meanLambda2 + vLambda2[i];
    meanEvecX = meanEvecX + vEvecX[i];
    meanEvecY = meanEvecY + vEvecY[i];
    meanAngle = meanAngle + vAngle[i];
    meanSize = meanSize + vSize[i];
    meanEccen = meanEccen + vEccen[i];
    meanStrt0 = meanStrt0 + vStrt0[i];
    meanStrt1 = meanStrt1 + vStrt1[i];
    meanclmbd1 = meanclmbd1 + vclmbd1[i];
    meanclmbd2 = meanclmbd2 + vclmbd2[i];
    meanAng = meanAng + vAng[i];
    meanSize = meanSize + vSize[i];
    meanEcc = meanEcc + vEcc[i];
    meanCG00 = meanCG00 + vCG00[i];
    meanCG11 = meanCG11 + vCG11[i];
    meanCG01 = meanCG01 + vCG01[i];
    meanRg = meanRg + vRg[i];
    meanASR = meanASR + vASR[i];
    meanIm = meanIm + vIm[i];
    meanIf = meanIf + vIf[i];
    meanMdifM = meanMdifM + vMdifM[i];
    meanMdifF = meanMdifF + vMdifF[i];
}

dNgens++;
```

meanN = meanN / dNgens;
meanZbar0 = meanZbar0 / dNgens;
meanZbar1 = meanZbar1 / dNgens;
meanP00 = meanP00 / dNgens;
meanP11 = meanP11 / dNgens;
meanP01 = meanP01 / dNgens;
meanRp = meanRp / dNgens;
```
CHAPTER 13

\[
\begin{align*}
\text{meanGbar}_0 &= \text{meanGbar}_0 / \text{dNgens}; \\
\text{meanGbar}_1 &= \text{meanGbar}_1 / \text{dNgens}; \\
\text{meanG}_0 &= \text{meanG}_0 / \text{dNgens}; \\
\text{meanG}_1 &= \text{meanG}_1 / \text{dNgens}; \\
\text{meanRg} &= \text{meanRg} / \text{dNgens}; \\
\text{meanLambda}_1 &= \text{meanLambda}_1 / \text{dNgens}; \\
\text{meanLambda}_2 &= \text{meanLambda}_2 / \text{dNgens}; \\
\text{meanEvecX} &= \text{meanEvecX} / \text{dNgens}; \\
\text{meanEvecY} &= \text{meanEvecY} / \text{dNgens}; \\
\text{meanAngle} &= \text{meanAngle} / \text{dNgens}; \\
\text{meanSize} &= \text{meanSize} / \text{dNgens}; \\
\text{meanEccen} &= \text{meanEccen} / \text{dNgens}; \\
\text{meanStrt}_0 &= \text{meanStrt}_0 / \text{dNgens}; \\
\text{meanStrt}_1 &= \text{meanStrt}_1 / \text{dNgens}; \\
\text{meancLmbd}_1 &= \text{meancLmbd}_1 / \text{dNgens}; \\
\text{meancLmbd}_2 &= \text{meancLmbd}_2 / \text{dNgens}; \\
\text{meancAng} &= \text{meancAng} / \text{dNgens}; \\
\text{meancSize} &= \text{meancSize} / \text{dNgens}; \\
\text{meancEcc} &= \text{meancEcc} / \text{dNgens}; \\
\text{meancG}_0 &= \text{meancG}_0 / \text{dNgens}; \\
\text{meancG}_1 &= \text{meancG}_1 / \text{dNgens}; \\
\text{meancRg} &= \text{meancRg} / \text{dNgens}; \\
\text{meanASR} &= \text{meanASR} / \text{dNgens}; \\
\text{meanIm} &= \text{meanIm} / \text{dNgens}; \\
\text{meanIf} &= \text{meanIf} / \text{dNgens}; \\
\text{meanMdifM} &= \text{meanMdifM} / \text{dNgens}; \\
\text{meanMdifF} &= \text{meanMdifF} / \text{dNgens}; \\
\end{align*}
\]

The code for this function is pretty tedious and repetitive, but hopefully its utility is obvious. The only new idea here is that we use the function \texttt{size()} to access the number of items currently stored in the vector \texttt{vN}. None of our code puts anything in our vectors yet, but the plan is to have one entry in each data vector for each generation. Thus, all of these vectors will be the same length, and the length should be equal to the number of generations. Our loop iterates enough times to step through all of the entries in \texttt{vN} (and all of the other vectors, which are the same length). Note that a specific element of a vector can be accessed using square brackets in the same way we access the elements of an array. The rest of this code just adds up all of the values across generations and uses the variable \texttt{dNgens} to divide these sums by the number of generations. The result, of course, is the mean across generations.

In the \texttt{private}: section of the simulation\_engine class, declare an object of \texttt{simulation\_data} by adding the following line:

\[
\text{simulation\_data \ sim\_data;}
\]

At this point, we have to find a way to get our data from each generation into the \texttt{sim\_data} object. It should come as no surprise that our preferred way of accomplishing this feat is to add another function to a class. In this case, we need to add it to the class that has access to the variables we need. All of the variables are calculated in the simulation\_engine class, so we need to populate our vectors from within this class. Add the following function to simulation\_engine:

\[
\text{void store\_variables\_in\_memory()} \\
{} \\
\]
Here we see a new function, `push_back()`, which is a member of `vector`. When we call `push_back()`, it takes a single argument, and this value is added to the end of the list. After each call to `push_back()`, the length of the vector has increased by one element, of course, and you will recall that we can obtain the length whenever we want by using the member function `size()`. Our new function, `store_variables_in_memory()`, possesses many similarities to our `save_population_variables()` function, except that now, instead of dumping our values into a file on our hard drive, we dump them into a bunch of vectors stored in memory.

At the end of the simulation run, we will still want to save the variables to a file, so we should create a function to handle this eventuality as well. Copy the entire `save_population_variables()` function and paste a duplicate of it in your `simulation_engine` class. Rename this duplicate to `save_stored_variables()` and change it to look like the code pasted below.

```cpp
void save_stored_variables()
{
    size_t i;
    sim_data.vN.push_back(PopulationSize);
    sim_data.vZbar0.push_back(phenotypic_mean[0]);
    sim_data.vZbar1.push_back(phenotypic_mean[1]);
    sim_data.vP00.push_back(phenotypic_variance[0]);
    sim_data.vP11.push_back(phenotypic_variance[1]);
    sim_data.vP01.push_back(phenotypic_covariance);
    sim_data.vRp.push_back(phenotypic_correlation);
    sim_data.vGBar0.push_back(genotypic_mean[0]);
    sim_data.vGBar1.push_back(genotypic_mean[1]);
    sim_data.vG00.push_back(genotypic_variance[0]);
    sim_data.vG11.push_back(genotypic_variance[1]);
    sim_data.vG01.push_back(genotypic_covariance);
    sim_data.vRg.push_back(genotypic_correlation);
    sim_data.vLambda1.push_back(EigenValue[0]);
    sim_data.vLambda2.push_back(EigenValue[1]);
    sim_data.vEvecX.push_back(EigenVector1[0]);
    sim_data.vEvecY.push_back(EigenVector1[1]);
    sim_data.vAngle.push_back(LeadAngle);
    sim_data.vSize.push_back(Sigma);
    sim_data.vEccen.push_back(Epsilon);
    sim_data.vStrt0.push_back(sel_diff_trt_0);
    sim_data.vStrt1.push_back(sel_diff_trt_1);
    sim_data.vcLmbd1.push_back(cPrevEval[0]);
    sim_data.vcLmbd2.push_back(cPrevEval[1]);
    sim_data.vcAng.push_back(cPrevAngle);
    sim_data.vcSize.push_back(cPrevSigma);
    sim_data.vcEcc.push_back(cPrevEpsilon);
    sim_data.vcG00.push_back(cPrevG00);
    sim_data.vcG11.push_back(cPrevG11);
    sim_data.vcG01.push_back(cPrevG01);
    sim_data.vcRg.push_back(cPrevRg);
    sim_data.vASR.push_back(SexRatio);
    sim_data.vIm.push_back(Im);
    sim_data.vIf.push_back(If);
    sim_data.vMdifM.push_back(MdiffMales);
    sim_data.vMdifF.push_back(MdiffFemales);
}
```
This function should be used at the very end of the simulation run, and it iterates through every vector containing population-level variables and saves the values we have stored in our vectors, for every generation, to a file. We can replace `output_population_variables()` with the two new functions we just created. We should call `store_variables_in_memory()` exactly where we called `output_population_variables()`, and we should call `save_stored_variables()` at the very
end of the simulation run. After making these changes, your “.cpp” file should look something like this (the only changes are near the end):

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations;

    simulation_engine my_sim;
    my_sim.display_parameters();

    bool initialization_success;
    initialization_success = my_sim.initialize_population();
    if (!initialization_success)
    {
        std::cout << "\nSimulation Initialization Failure!\n"
        return 0;
    }

    int ig;
    for (ig = 0; ig < my_sim.getNumberOfInitialGenerations(); ig++)
    {
        my_sim.monogamy();
        my_sim.mutation();
        my_sim.selection(true);
        my_sim.population_regulation();
    }

    // Turn the life cycle halfway, so the
    // experimental generations start with progeny
    my_sim.calculate_values_progeny();
    my_sim.monogamy();
    my_sim.mutation();

    for (generations = 0; generations < my_sim.getNumberOfGenerations(); generations++)
    {
        if (!my_sim.is_extinct())
        {
            my_sim.selection(false);
            my_sim.calculate_values_progeny();
            my_sim.population_regulation();
            my_sim.monogamy();
            my_sim.mutation();
            my_sim.calculate_values_adults();
            my_sim.store_variables_in_memory();
        }
    }

    my_sim.save_stored_variables();

    char end_it;
    std::cout << "\n\nEnter any character to exit...";
    std::cin >> end_it;
```
return 0;
}

These changes do not have a substantial effect on the program’s functionality, as far as the user is concerned, but they will be very important as we move forward. We have changed our program to store the values of variables in memory and to save everything to a file at the very end of the simulation run, and to the user these changes probably would not be noticeable. You should compile and run the program just to make sure that nothing is broken.

A small change we are prepared to make, that might actually be at least slightly noticeable, is to append the means to the last row of the output. We should add some simple functions to the simulation_engine class to perform both of these steps. Add the following function to calculate the means:

```cpp
void calc_run_means()
{
    sim_data.calculate_means();
}
```

This function just calls the function that we already wrote as a member of our simulation_data class. We need another function to append the means to the end of our output file. This function will do the trick (also add it to the simulation_engine class):

```cpp
void append_means_to_output_file()
{
    ofstream outfile;
    outfile.open("output.csv", ofstream::app);
    outfile << "Mean:"
        << sim_data.meanN;
    outfile <<","
        << sim_data.meanZbar0;
    outfile <<","
        << sim_data.meanZbar1;
    outfile <<","
        << sim_data.meanP00;
    outfile <<","
        << sim_data.meanP11;
    outfile <<","
        << sim_data.meanP01;
    outfile <<","
        << sim_data.meanRp;
    outfile <<","
        << sim_data.meanGbar0;
    outfile <<","
        << sim_data.meanGbar1;
    outfile <<","
        << sim_data.meanG00;
    outfile <<","
        << sim_data.meanG11;
    outfile <<","
        << sim_data.meanG01;
    outfile <<","
        << sim_data.meanRg;
    outfile <<","
        << sim_data.meanLambda1;
    outfile <<","
        << sim_data.meanLambda2;
    outfile <<","
        << sim_data.meanEvecX;
    outfile <<","
        << sim_data.meanEvecY;
    outfile <<","
        << sim_data.meanAngle;
    outfile <<","
        << sim_data.meanSize;
    outfile <<","
        << sim_data.meanEccen;
    outfile <<","
        << sim_data.meanStrt0;
    outfile <<","
        << sim_data.meanStrt1;
    outfile <<","
        << sim_data.meancLmbd1;
    outfile <<","
        << sim_data.meancLmbd2;
    outfile <<","
        << sim_data.meancAng;
    outfile <<","
        << sim_data.meancSize;
    outfile <<","
        << sim_data.meancEcc;
    outfile <<","
        << sim_data.meancG00;
```
VALIDATING THE SIMULATION

The only remaining step here is to call these two functions at the very end of our “.cpp” file, right after the command that reads `my_sim.save_stored_variables();`. Add these two lines:

```cpp
my_sim.calc_run_means();
my_sim.append_means_to_output_file();
```

Compile and run the program. The output should be essentially the same as before, except that now an additional row has been added reporting the mean across generations for each variable. Go ahead and check the values with your spreadsheet program. You should find that they are in exact agreement (except perhaps in the number of reported digits).

**Multiple Runs and the Mean of Means**

With these changes to the simulation_engine in place, we are prepared to alter our “.cpp” file to run multiple rounds of the simulation. We can just put everything in a loop and run it for the number of iterations we desire. We should rearrange things a bit and add a new variable to specify the number of replications. For instance, we might as well declare all of our iterator variables at the beginning. We also need to embed everything in a loop and change how we handle the `my_sim` object so we can have more than one. Make changes to your “.cpp” file until it looks like this:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations, ig, rep, number_of_replicates;
    number_of_replicates = 1;

    simulation_engine *my_sim = new simulation_engine[number_of_replicates];

    for (rep = 0; rep < number_of_replicates; rep++)
    {
        my_sim[rep].display_parameters();

        bool initialization_success;
        initialization_success = my_sim[rep].initialize_population();
        if (!initialization_success)
        {
            std::cout << "\nSimulation Initialization Failure!\n"
            return 0;
        }

        for (ig = 0; ig < my_sim[rep].getNumberOfInitialGenerations(); ig++)
```
{  
    my_sim[rep].monogamy();  
    my_sim[rep].mutation();  
    my_sim[rep].selection(true);  
    my_sim[rep].population_regulation();  
}

// Turn the life cycle halfway, so the experimental generations start with progeny
my_sim[rep].calculate_values_progeny();
my_sim[rep].monogamy();
my_sim[rep].mutation();

for (generations = 0; generations < my_sim[rep].getNumberOfGenerations(); generations++)
{
    if (!my_sim[rep].is_extinct())
    {
        my_sim[rep].selection(false);
        my_sim[rep].calculate_values_progeny();
        my_sim[rep].population_regulation();
        my_sim[rep].monogamy();
        my_sim[rep].mutation();
        my_sim[rep].calculate_values_adults();
        my_sim[rep].store_variables_in_memory();
    }
}

my_sim[rep].save_stored_variables();

my_sim[rep].calc_run_means();
my_sim[rep].append_means_to_output_file();

delete[] my_sim;

char end_it;
std::cout << "\n\nEnter any character to exit...";  
std::cin >> end_it;
return 0;

One big changes is that we declare a pointer to an array of simulation_engine objects and use the new operator to allocate memory for this array. The size of the array is specified by number_of_replicates, a new parameter of our model. Another way to accomplish the same thing would be to use a vector of simulation_engine objects. At the end of the program, we use delete[] to free up the memory associated with this new array. For now, the number of replicates is set to one, so the program will still run only one replicate of the simulation under the current parameter values.

The second big change is that we have added a new loop, which uses the variable rep as its iterator and repeats a number of times equal to number_of_replicates. Almost everything in our original “.cpp” file is now within this loop, so we basically end up repeating everything we were already doing for an arbitrary number of times set by number_of_replicates. Every occurrence of my_sim now has to have closed brackets (i.e., [rep]) specifying which element of the array is being accessed. Run the program with number_of_replicates set to one to verify that its functionality remains unchanged.
Now set the number of replications to two and see what happens. You should notice that one set of parameter values appears in the console window, and then a little while later, a second set appears below the first. Then eventually the program stops. If you have noticed the “Diagnostic Tools” window, which usually covers the Solution Explorer window when the program is running, you might have realized that the memory usage of the program doubled. All of these clues indicate that the program did indeed run two complete replicates of the simulation. In addition, it apparently runs them sequentially, using only one computing core on your machine. Parallel programming is beyond the scope of this tutorial, but this program does lend itself to parallelization, which could speed up your simulation runs considerably if you have access to a computer with a large number of cores.

Look at the output from the latest simulation run. Unfortunately, there is only one file, and it only contains data from the last run. Every simulation object uses the same file name, so each replicate overwrites the previous replicate. This attribute actually makes it pointless to run multiple replicates, so we will have to do something about it. The easiest solution is probably to make the output file name a variable that we can set at the beginning of the program and modify as the replicates progress. Just below your other variable declarations at the beginning of `main()`, add:

```cpp
std::string outfile_name, extended_outfile_name;
outfile_name = "outfile";
```

We also need to change our functions within the simulation_engine class that save to the hard drive so that they can accept a string as an argument. Change the declarations for `save_stored_variables()` and `append_means_to_output_file()` to:

```cpp
void save_stored_variables(std::string filename)
and:
void append_means_to_output_file(std::string filename)
```

Also, within each of these functions, find the `outfile.open()` commands and change them. For the `save_stored_variables()` function, change it from `outfile.open(“output.csv”);` to:

```cpp
outfile.open(filename);
```

And for the `append_means_to_output_file()` function, change it from `outfile.open(“output.csv”, std::fstream::app);` to:

```cpp
outfile.open(filename, std::fstream::app);
```

Now return to your “.cpp” file and make your way to the space right below the statement `my_sim[rep].display_parameters()`, which is inside your `rep` loop. Add the following line of code here:

```cpp
extended_outfile_name = outfile_name + "_run_" + std::to_string(rep + 1) + ".csv";
```

This statement will create a new `extended_outfile_name` for each iteration through the `rep` loop, and the files will have names like “outfile_run_1.csv”, “outfile_run_2.csv”, and so forth. Then find your calls to `save_stored_variables()` and `append_means_to_output_file()` within the `rep` loop, and change them to include `extended_outfile_name` as an argument. This section of code will now look like this:
my_sim[rep].save_stored_variables(extended_outfile_name);
my_sim[rep].calc_run_means();
my_sim[rep].append_means_to_output_file(extended_outfile_name);

Compile and run the program. You should obtain two output files, one for each replicate of the simulation.

We are very close to where we wish to be with our simulation model’s functionality, but we still do not have results that are directly comparable to those reported by Jones et al. (2003). We do have the capacity to conduct 20 replicate runs for each parameter combination, by setting `number_of_replicates` to 20, and each replicate also calculates a mean across generations for each variable. However, we do not obtain a mean of these means across replicates yet, and we also may wish to calculate a standard deviation of the means across replicates as an indication of how much variability we have across runs. At the end of each replicate, we do know the means are stored using the `simulation_data` class but we do not have direct access to them from the “.cpp” file, because we declared our `simulation_data` objects as private in the `simulation_engine`

The next step is to find a way to grab those means and put them in some sort of accessible place so that we can calculate the mean of these means. We will also want to go ahead and calculate the standard deviations of these means. Then we will want to write all of that to a new file of summary statistics. We will start by declaring a new class. Declare the following class in “simulation_engine.h”, before the declaration of the `simulation_engine` class:

```cpp
class mean_recorder
{
public:
    std::vector<double> mean_list;
};
```

Then in the private section of the `simulation_engine` class, declare an object of `mean_recorder` with the following line of code:

```cpp
mean_recorder m_rec;
```

Finally, add a new function to report the means to the main program. We will have the function return an object of the `mean_recorder` class after we populate the object with all of the means from the simulation. Here is what the function should look like, and it should go in the `simulation_engine` class:

```cpp
mean_recorder report_means()
{
    m_rec.mean_list.push_back(sim_data.meanN);
    m_rec.mean_list.push_back(sim_data.meanZbar0);
    m_rec.mean_list.push_back(sim_data.meanZbar1);
    m_rec.mean_list.push_back(sim_data.meanP00);
    m_rec.mean_list.push_back(sim_data.meanP11);
    m_rec.mean_list.push_back(sim_data.meanP01);
    m_rec.mean_list.push_back(sim_data.meanRp);
    m_rec.mean_list.push_back(sim_data.meanGbar0);
    m_rec.mean_list.push_back(sim_data.meanGbar1);
    m_rec.mean_list.push_back(sim_data.meanG00);
    m_rec.mean_list.push_back(sim_data.meanG11);
    m_rec.mean_list.push_back(sim_data.meanG01);
```
m_rec.mean_list.push_back(sim_data.meanRg);  
m_rec.mean_list.push_back(sim_data.meanLambda1);  
m_rec.mean_list.push_back(sim_data.meanLambda2);  
m_rec.mean_list.push_back(sim_data.meanEvecX);  
m_rec.mean_list.push_back(sim_data.meanEvecY);  
m_rec.mean_list.push_back(sim_data.meanAngle);  
m_rec.mean_list.push_back(sim_data.meanSize);  
m_rec.mean_list.push_back(sim_data.meanEccen);  
m_rec.mean_list.push_back(sim_data.meanStrt0);  
m_rec.mean_list.push_back(sim_data.meanStrt1);  
m_rec.mean_list.push_back(sim_data.meanCmLmbd1);  
m_rec.mean_list.push_back(sim_data.meanCmLmbd2);  
m_rec.mean_list.push_back(sim_data.meanCmAng);  
m_rec.mean_list.push_back(sim_data.meanCmSize);  
m_rec.mean_list.push_back(sim_data.meanCmEcc);  
m_rec.mean_list.push_back(sim_data.meanCmG00);  
m_rec.mean_list.push_back(sim_data.meanCmG11);  
m_rec.mean_list.push_back(sim_data.meanCmG01);  
m_rec.mean_list.push_back(sim_data.meanCmRg);  
m_rec.mean_list.push_back(sim_data.meanASR);  
m_rec.mean_list.push_back(sim_data.meanIm);  
m_rec.mean_list.push_back(sim_data.meanIf);  
m_rec.mean_list.push_back(sim_data.meanMdifM);  
m_rec.mean_list.push_back(sim_data.meanMdifF);  
return m_rec;
}

We have to head over to our “.cpp” file and set up the program to receive these means. We also need to keep them separate from run to run, so we should declare an array of mean_recorders. We do not know ahead of time how many replicates a user might request, so we will want to allocate memory for this array at run time. A convenient way to accomplish this sort of allocation is to use the new and delete operators, as we did for our simulation_engine objects. Near the declaration of the my_sim array, add the following line of code:

```
mean_recorder *mean_array = new mean_recorder[number_of_replicates];
```

Also, near your other delete[] statement near the end of the program, add:

```
delete[] mean_array;
```

Within the rep loop, we need to fill our new mean_array with values, so add the following line of code before the closing bracket of this loop:

```
mean_array[rep] = my_sim[rep].report_means();
```

After the rep loop is over, we can calculate the mean of the means across replicates, as well as the standard deviations of these means. Add the following code to your “.cpp” file after the closing brace of the rep loop:

```
int i, j, iNmeans;
double dNreps = number_of_replicates;
std::vector<double> mean_of_means;
std::vector<double> std_dev_of_means;
```
CHAPTER 13

\[
\text{iNmeans} = \text{static\_cast<int>(mean\_array[0].mean\_list.size());}
\]

for (i = 0; i < iNmeans; i++)
{
    mean_of_means.push\_back(0);
    std\_dev\_of\_means.push\_back(0);
}

// Calculate the means of the means
for (i = 0; i < number\_of\_replicates; i++)
{
    for (j = 0; j < iNmeans; j++)
    {
        mean\_of\_means[j] = mean\_of\_means[j] + mean\_array[i].mean\_list[j];
    }
}

for (j = 0; j < iNmeans; j++)
    mean\_of\_means[j] = mean\_of\_means[j] / dNreps;

// Calculate the standard deviations of the means
for (i = 0; i < number\_of\_replicates; i++)
{
    for (j = 0; j < iNmeans; j++)
    {
        std\_dev\_of\_means[j] = std\_dev\_of\_means[j] +
            (mean\_array[i].mean\_list[j] - mean\_of\_means[j]) *
            (mean\_array[i].mean\_list[j] - mean\_of\_means[j]);
    }
}

for (j = 0; j < iNmeans; j++)
{
    std\_dev\_of\_means[j] = std\_dev\_of\_means[j] / dNreps;
    std\_dev\_of\_means[j] = sqrt(std\_dev\_of\_means[j]);
}

// Save to a file
std::string summary\_filename;
summary\_filename = outfile\_name + "\_summary.csv";
std::ofstream outfile;
outfile.open(summary\_filename);
outfile << "Rep,N,zbar0,zbar1,P00,P11,P12,r(P),gbar0,gbar1,G00,G11,G01,r(G)";
outfile << ",\Lambda_1,\Lambda_2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1";
outfile << ",\Lambda_1,\Lambda_2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1";
outfile << ",\Lambda_1,\Lambda_2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1";
outfile << ",ASR,Im,If,MdifM,MdifF";
for (i = 0; i < number\_of\_replicates; i++)
{
    outfile << "\nRep" << i+1;
    for (j = 0; j < iNmeans; j++)
    {
        outfile << "," << mean\_array[i].mean\_list[j];
    }
}

outfile << "\nMean\_of\_means";
for (j = 0; j < iNmeans; j++)
VALIDATING THE SIMULATION

This new section of code starts by declaring some needed variables, including a couple of vectors in which to store the new means and standard deviations we are about to calculate. We store the total number of means (that is, the number of columns in our spreadsheet) in `iNmeans`, and the `static_cast<int>()` function just ensures that the number is properly converted to an integer before it is stored in `iNmeans`. We then use `push_back()` to fill each vector with a zero for each mean that we will calculate. By starting with zeros, we can iteratively add the means across replicates and then divide by the number of replicates. Note that we have also declared a double in which to store the number of replicates so that we will not have to divide by an integer. Then we have two loops that sum up values for the means and the standard deviations. For the means, we just sum all of the observations and divide by the number of replicates. For the standard deviations, we create a sum related to the variance (the sum of squared deviations from the mean), divide by the number of observations (which gives the variance: the average squared deviation from the mean), and take the square root to get the standard deviation (the standard deviation is the square root of the variance, by definition). This section of code ends with a short routine that saves the results to a file on the hard drive. The filename has “_summary.csv” appended to it, which identifies it as the summary file containing the means from each replicate, the means of these means across replicates, and the standard deviations of these means across replicates.

**Saving the Parameter Values**

We should also save a file with the parameter combinations associated with this particular simulation run. Without the parameter values, a particular run of the simulation is useless, because we are interested in how the values of various parameters affect evolutionary processes acting on the population. Consequently, any simulation or model should take care to somehow associate a list of parameter values with the results. In our case, we will just save an additional file, with the same core filename, but with “_parameters.csv” appended. We have already basically written the code for the function we need, as we already output the parameter values to the screen. We just have to copy this code into a new function and modify it slightly to save to a file instead of outputting to the screen. Go to your “simulation_engine.h” header file, copy `display_parameters()`, and paste it near the end of your simulation_engine class. Change the name of this function to the following:

```cpp
void save_parameter_values(std::string filename)
```

We have also added an argument so that the function accepts a string holding the filename. Add code at the beginning to declare an `ofstream` object and to open the file for writing. At the end of the function, be sure to `close()` the file. Then, throughout the function, everywhere you see `std::cout`, change it to `outfile` (or whatever you named your `ofstream` object), and also change every `\t` to a comma to conform with the “.csv” format. After you make these changes, the function will look like this:

```cpp
void save_parameter_values(std::string filename)
```
{
std::ofstream outfile;
outfile.open(filename);
outfile << "Parameter_Values:\n";
// Demographic Parameters
outfile << "Demographic_Parameters:\n";
outfile << "No_Generations:\n" << NumberOfGenerations << "\n";
outfile << "Initial_Pop_Size:\n" << PopulationSize << "\n";
outfile << "Carrying_Capacity:\n" << CarryingCapacity << "\n";
outfile << "Female_Fecundity:\n" << Fecundity << "\n";

// Mating Parameters
outfile << "Mating_Parameters:\n";
outfile << "Max_Mating_Enc.:\n" << MaxMatingEncounters << "\n";
outfile << "Gaussian_Pref_Var:\n" << GaussianPreferenceVariance << "\n";

// Quantitative Genetic Parameters
outfile << "Quantitative_Genetic_Parameters:\n";
outfile << "No_Loci_Trait0:\n" << NumLociTrait0 << "\n";
outfile << "No_Loci_Trait1:\n" << NumLociTrait1 << "\n";
outfile << "No_Loci_Pleiotrop:\n" << NumLociBoth << "\n";
outfile << "Env_Variance_Trt0:\n" << EnvironmentalVariance[0] << "\n";
outfile << "Env_Variance_Trt1:\n" << EnvironmentalVariance[1] << "\n";

// Mutational Parameters
outfile << "Mutational_Parameters:\n";
outfile << "Mut_Var_Trait0:\n" << MutationalVariance[0] << "\n";
outfile << "Mut_Var_Trait1:\n" << MutationalVariance[1] << "\n";
outfile << "Mut_Correlation:\n" << MutationalCorrelation << "\n";
outfile << "Mutation_Rate:\n" << MutationRatePerLocus << "\n";

// Selection Parameters
outfile << "Selection_Parameters:\n";
outfile << "Omega_Trait0:\n" << SelectionStrength[0] << "\n";
outfile << "Omega_Trait1:\n" << SelectionStrength[1] << "\n";
outfile << "Selection_Corr:\n" << SelectionalCorrelation << "\n";
outfile << "Optimum_Trait0:\n" << Optimum[0] << "\n";
outfile << "Optimum_Trait1:\n" << Optimum[1] << "\n";
if (ExperimentalSelectionSexLimited)
    outfile << "Sex_Limited_Sel:,true\n";
else
    outfile << "Sex_Limited_Sel:,false\n";

// Initial Generations
outfile << "Initial_Generations_Parameters:\n";
outfile << "No_Initial_Gens:\n" << NumberOfInitialGenerations << "\n";
outfile << "Initial_Omega_Trait0:\n" << InitialSelectionStrength[0] << "\n";
outfile << "Initial_Omega_Trait1:\n" << InitialSelectionStrength[1] << "\n";
outfile << "Initial_Sel_Corr:\n" << InitialSelectionalCorrelation << "\n";
outfile << "Init_Opt_Trt0:\n" << InitialOptimum[0] << "\n";
outfile << "Init_Opt_Trt1:\n" << InitialOptimum[1] << "\n";
if (InitialSelectionSexLimited)
    outfile << "Init_Sex_Lim_Sel:,true\n";
else
    outfile << "Init_Sex_Lim_Sel:,false\n";
outfile.close();
}
Return to your “.cpp” file, where we will have to add a call to this new function. We need to establish a filename. At the beginning of the program, immediately after the line that reads `outfile_name = "outfile";`, add:

```cpp
std::string parameter_filename;
parameter_filename = outfile_name + "_parameters.csv";
```

While we could call the function that saves the parameter values every time we loop through the replicates of the simulation, such an approach would be a waste of time because we would rewrite the parameter file during each replicate. Since the replicates are identical with respect to parameter values, we should actually only write one parameter file. The question is where we should put the call to our new function. We have to put it after the creation of the my_sim array and before its destruction. We do not want it inside any loop, so anywhere not in a loop between the new and delete commands associated with my_sim should work. We will put it immediately before the start of the rep loop. Add the following command:

```cpp
my_sim[0].save_parameter_values(parameter_filename);
```

Also remove the line of code within the rep loop that calls the `display_parameters()` function, as it is no longer needed. Your “.cpp” file should look something like this:

```cpp
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"

int main()
{
    int generations, ig, rep, number_of_replicates;
    number_of_replicates = 20;

    std::string outfile_name, extended_outfile_name;
    outfile_name = "outfile";

    std::string parameter_filename;
    parameter_filename = outfile_name + "_parameters.csv";

    simulation_engine *my_sim = new simulation_engine[number_of_replicates];
    mean_recorder *mean_array = new mean_recorder[number_of_replicates];

    my_sim[0].save_parameter_values(parameter_filename);

    for (rep = 0; rep < number_of_replicates; rep++)
    {
        extended_outfile_name = outfile_name + "_run_" + std::to_string(rep) + ".csv";

        bool initialization_success;
        initialization_success = my_sim[rep].initialize_population();
        if (!initialization_success)
        {
            std::cout << "\nSimulation Initialization Failure!\n";
            return 0;
        }
    }
```
for (ig = 0; ig < my_sim[rep].getNumberOfInitialGenerations(); ig++)
{
    my_sim[rep].monogamy();
    my_sim[rep].mutation();
    my_sim[rep].selection(true);
    my_sim[rep].population_regulation();
}

// Turn the life cycle halfway, so the experimental generations start with progeny
my_sim[rep].calculate_values_progeny();
my_sim[rep].monogamy();
my_sim[rep].mutation();
for (generations = 0; generations < my_sim[rep].getNumberOfGenerations(); generations++)
{
    if (!my_sim[rep].is_extinct())
    {
        my_sim[rep].selection(false);
        my_sim[rep].calculate_values_progeny();
        my_sim[rep].population_regulation();
        my_sim[rep].monogamy();
        my_sim[rep].mutation();
        my_sim[rep].calculate_values_adults();
        my_sim[rep].store_variables_in_memory();
    }
}
my_sim[rep].save_stored_variables(extended_outfile_name);
my_sim[rep].calc_run_means();
my_sim[rep].append_means_to_output_file(extended_outfile_name);
mean_array[rep] = my_sim[rep].report_means();

int i, j, iNmeans;
double dNreps = number_of_replicates;
std::vector<double> mean_of_means;
std::vector<double> std_dev_of_means;

iNmeans = static_cast<int>(mean_array[0].mean_list.size());
for (i = 0; i < iNmeans; i++)
{
    mean_of_means.push_back(0);
    std_dev_of_means.push_back(0);
}

// Calculate the means of the means
for (i = 0; i < number_of_replicates; i++)
{
    for (j = 0; j < iNmeans; j++)
    {
        mean_of_means[j] = mean_of_means[j] + mean_array[i].mean_list[j];
    }
}
for (j = 0; j < iNmeans; j++)

// Calculate the standard deviations of the means
for (i = 0; i < number_of_replicates; i++)
{
    for (j = 0; j < iNmeans; j++)
    {
            (mean_array[i].mean_list[j] - mean_of_means[j]) *
            (mean_array[i].mean_list[j] - mean_of_means[j]);
    }

    for (j = 0; j < iNmeans; j++)
    {
        std_dev_of_means[j] = sqrt(std_dev_of_means[j]);
    }
}

// Save to a file
std::string summary_filename;
summary_filename = outfile_name + "_summary.csv";
std::ofstream outfile;
outfile.open(summary_filename);
outfile << "Rep,N,zbar0,zbar1,P00,P11,P12,r(P),gbar0,gbar1,G00,G11,G01,r(G)";
outfile << ",,Lambda1,Lambda2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1";
outfile << ",,~Lmbd1,~Lmbd2,~Ang,~Size,~Ecc,~G00,~G11,~G01,~r(g)";
outfile << ",,ASR,Im,If,MdifM,MdifF";
for (i = 0; i < number_of_replicates; i++)
{
    outfile << "\nRep" << i+1;
    for (j = 0; j < iNmeans; j++)
    {
        outfile << "," << mean_array[i].mean_list[j];
    }
}

outfile << "\nMean_of_means";
for (j = 0; j < iNmeans; j++)
{
    outfile << "," << mean_of_means[j];
}

outfile << "\nStddev_of_means";
for (j = 0; j < iNmeans; j++)
{
    outfile << "," << std_dev_of_means[j];
}

outfile.close();
delete[] my_sim;
Validating the Simulation

After all of that work, we are finally in a position to check the results of our simulation model against the results reported in the Jones et al. (2003) paper. Feel free to replicate as much of the results of the paper as you wish, but for our current purposes, we will focus on Table 1. The simulations for the first line of this table used the following parameter values:

```
// Initialize the Parameters

// Initial Generations Parameters
NumberOfInitialGenerations = 10000;
InitialSelectionStrength[0] = 9;
InitialSelectionStrength[1] = 9;
InitialSelectionCorrelation = 0;
InitialOptimum[0] = 0;
InitialOptimum[1] = 0;
InitialSelectionSexLimited = false;

// Demographic Parameters
NumberOfGenerations = 2000;
PopulationSize = 256;
CarryingCapacity = PopulationSize;
Fecundity = 4;
PopulationExtinct = false;

// Mating Parameters
MaxMatingEncounters = 50;
GaussianPreferenceVariance = 0;

// Quantitative Genetic Parameters
NumLociTrait0 = 0;
NumLociTrait1 = 0;
NumLociBoth = 50;
EnvironmentalVariance[0] = 1;
EnvironmentalVariance[1] = 1;
EnvironmentalStDev[0] = sqrt(EnvironmentalVariance[0]);
EnvironmentalStDev[1] = sqrt(EnvironmentalVariance[1]);

// Mutational Parameters
MutationalVariance[0] = 0.05;
MutationalVariance[1] = 0.05;
MutationalCorrelation = 0;
MutationRatePerLocus = 0.0002;

// Selection Parameters
SelectionStrength[0] = 9;
SelectionStrength[1] = 9;
SelectionalCorrelation = 0;
Optimum[0] = 0;
```
Optimum[1] = 0;
ExperimentalSelectionSexLimited = false;

Change the parameter values in your model to match those listed above. Also ensure that the number of replicates is set to twenty (i.e., number_of_replicates = 20; in the “.cpp” file). Run the simulation and inspect the results. You might notice that the simulation takes a while to run. On my computer, the 20 replicates took about 13 minutes to complete. The slowness is due to the huge number of calculations being conducted in the process of keeping track of hundreds of individuals for tens of thousands of generations. We can speed the simulation up a bit by changing our compiling profile. Up until now, we have been compiling the program in “debug” mode, which allows Visual Studio to keep track of additional data relevant to problems that might arise as the program runs. We can also compile in “Release” mode, which is a little more dangerous as we develop a program but perfectly safe once we know the program is working properly. To compile in release mode, find the little text window in the toolbar at the top of the screen where “Debug” is displayed. Press the arrow to reveal a dropdown menu, and select “Release” (Figure 13.1).

![Figure 13.1: The upper left of the Visual Studio window showing the location of the menu that toggles between debug and release compiling options.](image)

The “Release” version of the program took about three minutes to run the same simulations that previously took about thirteen minutes for the “Debug” version on my computer. While it might be tempting to always use the “Release” option, stick with “Debug” until your program is running smoothly and verifiably doing what you think it should.

Now I can compare my results to those from Jones et al. (2003). Table 1 from that paper reports numerous means, and our program calculates the same values. A thorough investigator would compare results from every column to be sure our program does not contain an undetected bug. If you undertake this comparison, note that some of the variables related to single-generation changes in Jones et al. (2003) are standardized by dividing by the mean. Thus, to compare the results of our simulation to the results in the table, we would have to perform the same operation (e.g., \( \Delta G_{11} \) in our program corresponds to \( \Delta G_{11} \) in the table). Here is a partial comparison of results from our simulation to those reported by Jones et al. (2003):
Table 13.1: A comparison of our output with the output reported in the Jones et al. (2003) paper. The first three variables are parameters describing the selection surface: \( \Omega_{\text{Trait}0} = \omega_{11}, \Omega_{\text{Trait}1} = \omega_{22}, \) and \( \text{Selection\_Corr} = r_{g}. \) The last six columns show means for some of the important population variables: \( G_{00} = G_{11}, G_{11} = G_{22}, r(G) = r_g, \) \( \Sigma, \) \( \text{Eccen} = \varepsilon, \) and \( \text{Angle} = \phi. \) Note that traits 0 and 1 in our simulation correspond to traits 1 and 2 in Jones et al. (2003).

<table>
<thead>
<tr>
<th>Source</th>
<th>( \omega_{11} )</th>
<th>( \omega_{22} )</th>
<th>( r_{\omega} )</th>
<th>( G_{11} )</th>
<th>( G_{22} )</th>
<th>( r_g )</th>
<th>( \Sigma )</th>
<th>( \varepsilon )</th>
<th>( \phi )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones et al. (2003)</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0.201</td>
<td>0.192</td>
<td>0.044</td>
<td>0.39</td>
<td>0.64</td>
<td>8.7</td>
</tr>
<tr>
<td>Our Simulation</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>0.191</td>
<td>0.209</td>
<td>-0.004</td>
<td>0.40</td>
<td>0.64</td>
<td>0.41</td>
</tr>
<tr>
<td>Jones et al. (2003)</td>
<td>9</td>
<td>9</td>
<td>0.9</td>
<td>0.113</td>
<td>0.111</td>
<td>0.416</td>
<td>0.22</td>
<td>0.40</td>
<td>43.0</td>
</tr>
<tr>
<td>Our Simulation</td>
<td>9</td>
<td>9</td>
<td>0.9</td>
<td>0.111</td>
<td>0.111</td>
<td>0.424</td>
<td>0.22</td>
<td>0.39</td>
<td>43.2</td>
</tr>
<tr>
<td>Jones et al. (2003)</td>
<td>49</td>
<td>49</td>
<td>0</td>
<td>0.432</td>
<td>0.443</td>
<td>0.028</td>
<td>0.88</td>
<td>0.63</td>
<td>5.9</td>
</tr>
<tr>
<td>Our Simulation</td>
<td>49</td>
<td>49</td>
<td>0</td>
<td>0.460</td>
<td>0.471</td>
<td>0.020</td>
<td>0.93</td>
<td>0.63</td>
<td>3.8</td>
</tr>
<tr>
<td>Jones et al. (2003)</td>
<td>49</td>
<td>49</td>
<td>0.9</td>
<td>0.280</td>
<td>0.267</td>
<td>0.364</td>
<td>0.55</td>
<td>0.44</td>
<td>41.5</td>
</tr>
<tr>
<td>Our Simulation</td>
<td>49</td>
<td>49</td>
<td>0.9</td>
<td>0.263</td>
<td>0.272</td>
<td>0.340</td>
<td>0.54</td>
<td>0.47</td>
<td>43.7</td>
</tr>
</tbody>
</table>

Our results are pretty close to those reported by Jones et al. (2003). They are not exact, however, because the model includes a substantial stochastic element due to genetic drift (arising from the finite population size). We can get an idea of whether our results are close enough by examining our standard deviations. For instance, the values of \( G_{11} \) in the third set of results in Table 13.1 are 0.432 and 0.460 for the Jones et al. (2003) simulations and our present simulations, respectively. The standard deviation for this number across runs in my simulations is 0.08, so the numbers are well within a single standard deviation of one another. Thus, we can conclude that they are the same, while acknowledging some uncertainty stemming from the stochastic nature of the simulations. If we wished to show for sure that the two simulations produced identical values, we could run simulation with much larger population sizes with many more replicates. Regardless, I am now prepared to declare that our program is working as intended.

Chapter Summary

This chapter tackled a wide-ranging array of topics to better align our program with previously published work. For instance, we implemented a monogamous mating system, and this change actually turned out to be more work than one might have predicted. Next, we altered the program so that it would be able to run multiple independent replicates under the same set of parameter combinations. We also created a new class in which to store all the data generated by the simulation, which allowed us to output a file containing the means of each variable from each simulation replicate as well as the means of these means across replicates. We calculated the standard deviations of these means across replicates as an indicator of how much variability we saw across independent runs of the simulation. Finally, we were able to compare our results to those published by Jones et al. (2003) to verify that our simulation produced similar results under the same parameter combinations. This chapter did introduce a new programming concept. We learned how to use the Standard Template Library (STL), which is a family of classes designed to hold data. The class that we used in this chapter is known as a vector, and it is basically a one-dimensional array, or list. The nice feature of the STL is that the containers manage their own memory, such that we do not need to worry about allocating and freeing memory with the `new` and `delete` operators. In other words, when we use the STL, we do not need to be concerned about memory leaks. At the end of the chapter, we proudly declared that we have a simulation program that is working as intended and capable of reproducing the results of papers in the published evolutionary literature.
Chapter 14. Running on Linux with Shell Scripting

Our program is now fully functional, and it reproduces results of prior work. We have the skills now to modify it to produce the results of other published work, as well as to drill into novel topics. However, if we wished to produce all of the results of, say, the Jones et al. (2003) paper, such an endeavor would be extremely tedious. Each time we wished to produce a row of Table 1, for instance, we modified the parameter values, which are hard-coded into the program, recompiled it, and ran it. The 20 runs took a few minutes to run, and then we inspected the results and started the whole process over again. This way of producing results requires you to sit at your computer and tend your program every few minutes. While you would probably get a lot of web surfing done, you would not be able accomplish much else during the short simulation runs. What we need is a way to automate a large number of runs. We also need the ability to change parameter combinations without recompiling the program every time.

These goals can be accomplished a number of different ways. For instance, we could add yet another, more inclusive loop that cycles through multiple different parameter combinations. We could even put most of the commands in our main “.cpp” program in a more inclusive class to keep the program organized and object-oriented. We could even imagine a graphical user interface (as Jones et al. 2003 actually used) that allows the user to set the parameter combinations. Here, however, we are going to take a different approach. Much of the scientific computing in the world takes place on supercomputers, and most supercomputers use a Linux-type operating system. These operating systems provide some conveniences, and they have versions that can run on an ordinary desktop or laptop computer. If we make our program compatible with the Linux way of doing things, then we will have a program that can run on anything from a cruddy laptop to the world’s most sophisticated supercomputer.

One of the most useful features of a Linux-based operating system is the so-called shell, which is a command-line interpreter that also functions as a sort of scripting programming language. Thus, you can either type a single command at the command prompt and hit enter, or you can construct a text file with lines of code that can be executed like any other program. If you type a single command, the shell just interprets it and executes it immediately. If you set up a text file for the shell to execute, it runs the commands line after line automatically. This way of using the shell is called “shell scripting”, and these “shell scripts” provide a way for you to automate your computer to invoke a wide variety of programs and commands. Shell scripting is one of the most useful tools around for setting up automated workflows.

Making Sure our Program is Linux Compatible

Before we can implement shell scripting in a Linux environment, we need to ensure that our program can actually run on Linux. Unfortunately, not all C++ compilers are identical, so certain commands we used in Visual Studio (which uses MSCV – the Microsoft C++ compiler) may not be supported in the Linux compiler (typically the GCC – GNU Compiler Collection – C++ compiler). You will need access to a machine running some version of Linux. If you have an old computer lying around not doing anything, you could consider installing Ubuntu on it. The Ubuntu operating system is free. You can download the desktop version, burn it to a DVD, and insert it into your old computer’s DVD drive to install it. Alternatively, you can follow the steps in Appendix 2 and set up a virtual machine with Ubuntu installed on it. Regardless of which approach you choose, scan through Appendix 2 and ensure that your Linux machine has the g++ compiler installed.
We can attempt to compile our program on Linux by copying our source and header files over to our Linux computer. If you are using a virtual machine, as described in Appendix 2, just create a new folder (named “gmatrix” or some other simple name) in your shared folder on the Windows side. Copy “Chapter13example.cpp”, “MTwisterFunctions.h”, and “simulation_engine.h” from the folder corresponding to your Visual Studio project into this new folder you just created. If you are not using a virtual machine, you can transfer these files to your Linux machine on any USB drive.

Open a terminal window in Ubuntu, and navigate to the folder containing these source and header files (i.e., the shared folder). On my machine, the following command gets me there, regardless of my starting folder:

```bash
cd ~/sf_ushare/gmatrix
```

My shared folder is named “ushare” on the Windows side and “sf_ushare” on the Ubuntu side. Now that I am here, I can attempt to compile the program by invoking the g++ compiler, as so:

```bash
g++ Chapter13example.cpp -o gmatrix_simulator
```

The g++ compiler requires, at minimum, the name of the “.cpp” source file and a name for the compiled program. The argument after “-o” is an arbitrary name for the executable file. That is a lowercase letter “oh”, not a zero. When you run this command, you will be rewarded with a screen full of errors. Clearly, we have a problem. Some of the commands that work just fine in the Microsoft compiler do not work at all in the Linux compiler. If you scan through the errors, you will see a couple of recurring themes. A lot of the errors say something about “open(const *char, …)”. One error mentions “to_string” and claims it is not a member of “std”. A couple of errors seem bent out of shape about “rand” and “srand”.

Fixing the errors will take a bit of sleuthing, but fortunately I have already done the sleuthing for these particular errors for you. Go ahead and delete the copies of your various files that you placed in your shared folder. You will want to make the changes that fix the errors in Visual Studio, so that the versions you compile for Windows and Linux are based on exactly the same source code. Anything that will compile on the GNU compiler will also compile using the Microsoft compiler, even though the reverse is not true.

The easiest errors to fix are the “srand” and “rand” ones. Apparently the Microsoft compiler automatically includes the header containing these functions without the need for an `include` statement, but the g++ compiler does not. A quick internet search shows that “srand” and “rand” are in the “stdlib.h” header. We can fix this error by adding the following statement to the beginning of our “simulation_engine.h” file:

```c
#include <stdlib.h>
```

The next error seems to involve our call to `to_string()`, which we use to convert an integer into a string for some of our filenames (in the main “.cpp” file). Some research shows that this function is only available in recent C++ compilers. The version of the C++ compiler on our Linux machine must not be as up-to-date as the Microsoft compiler. We can use the internet to find an alternative, and one option is to use the ancient C command `itoa()`. We find all sorts of warnings that this function is not part of standard C++, so we should probably steer clear of it. Another alternative is to use a `stringstream`, which provides a general solution for lots of different text manipulations. The problem seems so small, but we will need to make a number of changes to solve it correctly without using `to_string()`, which is
actually the simplest way to convert an integer to a string. To use a stringstream, first add the following include statement to your “.cpp” file:

```cpp
#include <sstream>
```

Just after the opening brace of `main()`, add the following declaration for a stringstream object named “ss”:

```cpp
std::stringstream ss;
```

Now find the statement that includes the call to `to_string()`, which is at the beginning of the `rep` loop, and replace this single line with the following three lines of code:

```cpp
ss.str("";
ss << outfile_name << "_run_" << rep + 1 << ".csv";
extended_outfile_name = ss.str();
```

The first statement sets the contents of the stringstream to nothing, so that it will be empty when we start adding to it. The second line uses the `<<` operator to build our string. We can push strings and numbers into the stringstream with this operator, without worrying about any kinds of conversions. Each new item is just appended to the end. Finally, the call to `ss.str()` (without arguments) returns the characters contained in `ss` as a string. Ultimately, we end up doing exactly what we did using the function `to_string()`, but with more commands and variables. Normally, we would be disappointed in the added code and complexity, but in this case at least we managed to learn a little bit about the stringstream class. Set the `number_of_replicates` to a small value, such as two or three, and compile the program in “Debug” mode. If it compiles without errors or warnings (it should), run the program to make sure it still outputs files with the correct names.

The final error, which we committed many times in our program, involves the filenames we used to write to the hard drive. In older versions of C++, the `fstream` member function `open()` expects a null-terminated character array rather than a string. In the GNU compiler, we are getting an error every time we try to save a file because our file names are strings. We need to convert them all to null-terminated character arrays. These null-terminated arrays were the old-school way to handle strings before the string class was invented. The easiest way to fix this problem is probably to write a function that accepts a string and returns a character array. Add the following code to the very beginning of “simulation_engine.h”, immediately below the `#include` statements:

```cpp
void convert_string_to_char(std::string my_string, char *my_char)
{
    size_t i;
    size_t str_length;

    str_length = my_string.size();

    if (str_length > 255)
        str_length = 255;

    for (i = 0; i < str_length; i++)
    {
        my_char[i] = my_string[i];
    }
```
my_char[i] = '0';
}

This function accepts a string and a pointer to a character array. It then puts up to the first 255 characters of the string into the character array. If the string is less than 255 characters, then the entire string ends up being copied into the character array. Finally, it appends a \0 character to the end, which is the null character in C++. Thus, after this function is called, the pointer points to a null-terminated character array. Now we just need to change the code everywhere we save a file so that it calls this function. For example, in our “.cpp” file, we save the summary file. Change the code to look like this:

```
// Save to a file
std::string summary_filename;
summary_filename = outfile_name + "_summary.csv";
std::ofstream outfile;
char temp_fn[256];
convert_string_to_char(summary_filename, temp_fn);
outfile.open(temp_fn);
outfile << "Rep,N,bar0,zbar1,P00,P11,P12,r(P),gbar0,gbar1,G00,G11,G01,r(G)"
outfile << ",,Lambda1,Lambda2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1"
outfile << ",,~Lmbd1,~Lmbd2,~Ang,~Size,~Ecc,~G00,~G11,~G01,~r(g)"
outfile << ",,ASR,Im,If,MdifM,MdifF"
```

The changes affect the fourth to sixth lines here. We need to make similar changes to `save_parameter_values()`, `append_means_to_output_file()`, and `save_stored_variables()`. In each case, change the statement that reads `outfile.open(filename);` to the following three lines:

```
char temp_fn[256];
convert_string_to_char(filename, temp_fn);
outfile.open(temp_fn);
```

In `append_means_to_output_file()`, be sure to keep the argument `std::fstream::app` in your call to `open()`. After you make this change everywhere we fed a string to `open()`, so that now `open()` just receives null-terminated character arrays, all of the bugs should be fixed. Copy your source and header files back over to your Linux operating system. Use the `cd` command to navigate to the folder containing your source and header files, and compile the program with the following command:

```
g++ Chapter13example.cpp -o gmatrix_simulator
```

The program should compile without any errors. To run the program, type:

```
./gmatrix_simulator
```

The ./ means the current directory to Ubuntu, and you just type the name of a program to execute it. There is a possibility that you will not have permission to execute the program, depending on where you compiled it. If you do not have permission, you have to change the permissions. Use the following command:

```
chmod u+x gmatrix_simulator
```
If you could not run the program before, you should be able to now. You can open your output files within Ubuntu. Use the file cabinet icon on the toolbar to navigate to the folder containing your results (i.e., your folder shared with Windows). If you double-click on the files, they will open in the free spreadsheet program, LibreOffice Calc, which is included with Ubuntu. You will see that these files look very much like the ones produced by the Windows version of the program. In fact, the two versions should run identically.

Table 14.1: Simulation parameters that the user should be able to access.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Program Variable</th>
<th>Default</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Replicates</td>
<td>number_of_replicates</td>
<td>5</td>
<td>--reps</td>
</tr>
<tr>
<td>Number of Initial Generations</td>
<td>NumberOfInitialGenerations</td>
<td>1000</td>
<td>--init_gens</td>
</tr>
<tr>
<td>Initial $\omega_{00}$</td>
<td>InitialSelectionStrength[0]</td>
<td>49</td>
<td>--init_w00</td>
</tr>
<tr>
<td>Initial $\omega_{11}$</td>
<td>InitialSelectionStrength[1]</td>
<td>49</td>
<td>--init_w11</td>
</tr>
<tr>
<td>Initial $r_{00}$</td>
<td>InitialSelectionCorrelation</td>
<td>0</td>
<td>--init_sel_corr</td>
</tr>
<tr>
<td>Initial $\theta_{0}$</td>
<td>InitialOptimum[0]</td>
<td>0</td>
<td>--init_opt0</td>
</tr>
<tr>
<td>Initial $\theta_{1}$</td>
<td>InitialOptimum[1]</td>
<td>0</td>
<td>--init_opt1</td>
</tr>
<tr>
<td>Initial Selection Sex-Limited?</td>
<td>InitialSelectionSexLimited</td>
<td>false</td>
<td>--init_sex_limit</td>
</tr>
<tr>
<td>Number of Exp. Generations</td>
<td>NumberOfGenerations</td>
<td>1000</td>
<td>--gens</td>
</tr>
<tr>
<td>Carrying Capacity/Pop. Size</td>
<td>PopulationSize</td>
<td>256</td>
<td>--carry_cap</td>
</tr>
<tr>
<td>Fecundity per Female</td>
<td>Fecundity</td>
<td>4</td>
<td>--fecund</td>
</tr>
<tr>
<td>Maximum No. Mating Encounters</td>
<td>MaxMatingEncounters</td>
<td>50</td>
<td>--mate_enc</td>
</tr>
<tr>
<td>Gaussian Preference Variance</td>
<td>GaussianPreferenceVariance</td>
<td>0</td>
<td>--pref_var</td>
</tr>
<tr>
<td>Number of Trait 0 Loci</td>
<td>NumLocTrait0</td>
<td>0</td>
<td>--loci_trt0</td>
</tr>
<tr>
<td>Number of Trait 1 Loci</td>
<td>NumLocTrait1</td>
<td>0</td>
<td>--loci_trt1</td>
</tr>
<tr>
<td>Number of Pleiotropic Loci</td>
<td>NumLocBoth</td>
<td>50</td>
<td>--loci_pleio</td>
</tr>
<tr>
<td>Trait 0 Environmental Variance</td>
<td>EnvironmentalVariance[0]</td>
<td>1</td>
<td>--env_var0</td>
</tr>
<tr>
<td>Trait 1 Environmental Variance</td>
<td>EnvironmentalVariance[1]</td>
<td>1</td>
<td>--env_var1</td>
</tr>
<tr>
<td>Mutational Variance Trait 0</td>
<td>MutationalVariance[0]</td>
<td>0.05</td>
<td>--mut_var0</td>
</tr>
<tr>
<td>Mutational Variance Trait 1</td>
<td>MutationalVariance[1]</td>
<td>0.05</td>
<td>--mut_var1</td>
</tr>
<tr>
<td>Mutational Correlation</td>
<td>MutationalCorrelation</td>
<td>0</td>
<td>--mut_corr</td>
</tr>
<tr>
<td>Per Locus Mutation Rate</td>
<td>MutationRatePerLocus</td>
<td>0.0002</td>
<td>--mut_rate</td>
</tr>
<tr>
<td>Experimental $\omega_{00}$</td>
<td>SelectionStrength[0]</td>
<td>49</td>
<td>--exp_w00</td>
</tr>
<tr>
<td>Experimental $\omega_{11}$</td>
<td>SelectionStrength[1]</td>
<td>49</td>
<td>--exp_w11</td>
</tr>
<tr>
<td>Experimental $r_{00}$</td>
<td>SelectionCorrelation</td>
<td>0</td>
<td>--exp_sel_corr</td>
</tr>
<tr>
<td>Experimental $\theta_{0}$</td>
<td>Optimum[0]</td>
<td>0</td>
<td>--exp_opt0</td>
</tr>
<tr>
<td>Experimental $\theta_{1}$</td>
<td>Optimum[1]</td>
<td>0</td>
<td>--exp_opt1</td>
</tr>
<tr>
<td>Exp. Selection Sex Limited?</td>
<td>ExperimentalSelectionSexLimited</td>
<td>false</td>
<td>--exp_sex_limit</td>
</tr>
<tr>
<td>Output File Name Nucleus</td>
<td>outfile_name</td>
<td>outfile</td>
<td>--filename</td>
</tr>
</tbody>
</table>

**Accepting Command-Line Arguments**

In the interest of automating our program, we will need the user to have the ability to change parameter values, without having to change the code and recompile the program every time. For a command-line program like ours, the easiest way to set the parameters is to set up the program to accept arguments from the command line. The concept behind this modification is simple, because a program compiled with C++ has the capacity to accept arguments and place them in arrays. The tedious part will occur as we design code to parse the arguments, that is, to interpret the meaning of the arguments and to change parameter values accordingly. If we want our user to be able to change every parameter value, then our program will have to accept many arguments. We will have to be careful to keep everything as organized as possible, because dealing with all these arguments could become quite a mess. To modify our program to accept arguments, all we have to do is make a slight change to `int main()`. Add text within the parentheses to make it look like this:

```
int main(int argc, char* argv[])
```
The integer $\texttt{argc}$ counts the number of arguments, and the character array $\texttt{argv}$ keeps track of the text of each argument. It is actually an array of null-terminated strings, each of which can be multiple characters in length. The minimum number of arguments is one, the name of the program. Thus, the first argument ($\texttt{argv[0]}$) is not very interesting to us. The other arguments after the first, however, can be used to change parameter values if we design some code to parse them.

Usually, arguments for these types of command-line programs take the form of a letter after a minus sign indicating a parameter to change, followed by the value. We actually encountered this format a while back, with the g++ compiler, which used “-o” followed by a string to indicate the name of the output file. Our program has so many parameters, however, that a single-letter abbreviation is not practical. Many programs take the approach of having longer abbreviations preceded by two hyphens instead of one. We will use that approach for our program. Table 14.1 shows a list of the parameters we might want to be able to change, a suggested default value, a suggested abbreviation for the command line, and the name of the variable in our program that holds the parameter value.

My proposal here is to use a function, which we can define in the simulation_engine header, to parse the arguments. Then we will pair it with a function in the simulation_engine class that actually changes the parameter values for our simulations. The easiest way to keep track of all the parameter values will be to create a new class, so add the following code to your simulation_engine header, immediately after the existing #include statements:

```cpp
#include <sstream>

class parameter_value_set
{
  public:
    int p_reps, p_init_gens;
    double p_init_w00, p_init_w11, p_init_sel_corr, p_init_opt0, p_init_opt1;
    bool p_init_sex_lim;
    int p_gens, p_carry_cap, p_fecund, p_mate_enc;
    double p_pref_var;
    int p_loci_trt0, p_loci_trt1, p_loci_pleio;
    double p_env_var0, p_env_var1, p_mut_var0, p_mut_var1, p_mut_corr, p_mut_rate;
    double p_exp_w00, p_exp_w11, p_exp_sel_corr, p_exp_opt0, p_exp_opt1;
    bool p_exp_sex_lim;
    std::string file_name;
};

int parse_command_line_arguments(int arg_c, char* arg_v[], parameter_value_set &parm_set)
{
  // First set default values
  parm_set.p_reps = 5;
  parm_set.p_init_gens = 1000;
  parm_set.p_init_w00 = 49;
  parm_set.p_init_w11 = 49;
  parm_set.p_init_sel_corr = 0;
  parm_set.p_init_opt0 = 0;
  parm_set.p_init_opt1 = 0;
  parm_set.p_init_sex_lim = false;
  parm_set.p_gens = 1000;
  parm_set.p_carry_cap = 256;
  parm_set.p_fecund = 4;
  parm_set.p_mate_enc = 50;
  parm_set.p_pref_var = 0;
  // Rest of function...
```
parm_set.p_loci_trt0 = 0;
parm_set.p_loci_trt1 = 0;
parm_set.p_loci_pleio = 50;
parm_set.p_env_var0 = 1;
parm_set.p_env_var1 = 1;
parm_set.p_mut_var0 = 0.05;
parm_set.p_mut_var1 = 0.05;
parm_set.p_mut_corr = 0;
parm_set.p_mut_rate = 0.0002;
parm_set.p_exp_w00 = 49;
parm_set.p_exp_w11 = 49;
parm_set.p_exp_sel_corr = 0;
parm_set.p_exp_opt0 = 0;
parm_set.p_exp_opt1 = 0;
p parm_set.p_exp_sex_lim = false;
p parm_set.file_name = "outfile";

if (arg_c < 2) // There are no arguments so defaults are in use
  return 0;

  return 1;

int i;
std::string tstr1, tstr2;
for (i = 1; i < arg_c - 1; i++)
{
  tstr1 = arg_v[i];
  tstr2 = arg_v[i + 1];
  if (tstr1 == "--reps")
  {
    std::stringstream ss(tstr2);
    ss >> parm_set.p_reps;
  }
  if (tstr1 == "--init_gens")
  {
    std::stringstream ss(tstr2);
    ss >> parm_set.p_init_gens;
  }
  if (tstr1 == "--init_w00")
  {
    std::stringstream ss(tstr2);
    ss >> parm_set.p_init_w00;
  }
  if (tstr1 == "--init_w11")
  {
    std::stringstream ss(tstr2);
    ss >> parm_set.p_init_w11;
  }
  if (tstr1 == "--init_sel_corr")
  {
    std::stringstream ss(tstr2);
    ss >> parm_set.p_init_sel_corr;
  }
  if (tstr1 == "--init_opt0")
  {
    std::stringstream ss(tstr2);
  }
ss >> parm_set.p_init_opt0;
}
if (tstr1 == "--init_opt1")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_init_opt1;
}
if (tstr1 == "--init_sex_lim")
{
    if (tstr2[0] == 't' || tstr2[0] == 'T')
        parm_set.p_init_sex_lim = true;
}
if (tstr1 == "--gens")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_gens;
}
if (tstr1 == "--carry_cap")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_carry_cap;
}
if (tstr1 == "--fecund")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_fecund;
}
if (tstr1 == "--mate_enc")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_mate_enc;
}
if (tstr1 == "--pref_var")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_pref_var;
}
if (tstr1 == "--loci_trt0")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_loci_trt0;
}
if (tstr1 == "--loci_trt1")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_loci_trt1;
}
if (tstr1 == "--loci_pleio")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_loci_pleio;
}
if (tstr1 == "--env_var0")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_env_var0;
}


```cpp
if (tstr1 == "--env_var1")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_env_var1;
}
if (tstr1 == "--mut_var0")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_mut_var0;
}
if (tstr1 == "--mut_var1")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_mut_var1;
}
if (tstr1 == "--mut_corr")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_mut_corr;
}
if (tstr1 == "--mut_rate")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_mut_rate;
}
if (tstr1 == "--exp_w00")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_exp_w00;
}
if (tstr1 == "--exp_w11")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_exp_w11;
}
if (tstr1 == "--exp_sel_corr")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_exp_sel_corr;
}
if (tstr1 == "--exp_opt0")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_exp_opt0;
}
if (tstr1 == "--exp_opt1")
{
    std::stringstream ss(tstr2);
    ss >> parm_set.p_exp_opt1;
}
if (tstr1 == "--exp_sex_lim")
{
    if (tstr2[0] == 't' || tstr2[0] == 'T')
        parm_set.p_exp_sex_lim = true;
}
if (tstr1 == "--filename")
{

```
The function is long and tedious, but it is also pretty straightforward. If there are no arguments (i.e., \texttt{arg_c < 2}), then it exits without doing anything. It also checks to see if the first argument (\texttt{arg_v[1]}) is \texttt{--h} or \texttt{--help}, in which case it returns the value 1 to signal to the program to output help (something we will not implement but that would be a nice feature for most pieces of software). If there are other arguments, then the \texttt{i} loop parses them. Note that we expect an argument indicating the parameter followed by a value for that parameter, so our loop has to stop one short of the number of arguments. Otherwise it would run off the end of the array and access memory that is not available.

We need a complementary function in our \texttt{simulation_engine} class that uses a \texttt{parameter_value_set} to update the parameter values used by the program. Add the following new function to \texttt{simulation_engine}:

void update_parameter_values\( (\texttt{parameter_value_set \&parm_set}) \)
{
// Set Parameter Values

// Initial Generations Parameters
NumberOfInitialGenerations = parm_set.p_init_gens;
InitialSelectionStrength[0] = parm_set.p_init_w00;
InitialSelectionStrength[1] = parm_set.p_init_w11;
InitialSelectionalCorrelation = parm_set.p_init_sel_corr;
InitialOptimum[0] = parm_set.p_init_opt0;
InitialOptimum[1] = parm_set.p_init_opt1;
InitialSelectionSexLimited = parm_set.p_init_sex_lim;

// Demographic Parameters
NumberOfGenerations = parm_set.p_gens;
CarryingCapacity = parm_set.p_carry_cap;
PopulationSize = CarryingCapacity;
Fecundity = parm_set.p_fecund;
PopulationExtinct = false;

// Mating Parameters
MaxMatingEncounters = parm_set.p_mate_enc;
GaussianPreferenceVariance = parm_set.p_pref_var;

// Quantitative Genetic Parameters
NumLociTrait0 = parm_set.p_loci_trt0;
NumLociTrait1 = parm_set.p_loci_trt1;
NumLociBoth = parm_set.p_loci_pleio;
EnvironmentalVariance[0] = parm_set.p_env_var0;
EnvironmentalVariance[1] = parm_set.p_env_var1;
EnvironmentalStDev[0] = sqrt(EnvironmentalVariance[0]);
EnvironmentalStDev[1] = sqrt(EnvironmentalVariance[1]);

// Mutational Parameters
MutationalVariance[0] = parm_set.p_mut_var0;
MutationalVariance[1] = parm_set.p_mut_var1;
MutationalCorrelation = parm_set.p_mut_corr;
}
MutationRatePerLocus = parm_set.p_mut_rate;

// Selection Parameters
SelectionStrength[0] = parm_set.p_exp_w00;
SelectionStrength[1] = parm_set.p_exp_w11;
SelectionalCorrelation = parm_set.p_exp_sel_corr;
Optimum[0] = parm_set.p_exp_opt0;
Optimum[1] = parm_set.p_exp_opt1;
ExperimentalSelectionSexLimited = parm_set.p_exp_sex_lim;

This simple function merely accepts a parameter_value_set object, passed by reference, and sets the parameter values of the simulation_engine accordingly. All that remains is to call these functions in the appropriate places. We might as well parse the arguments right at the beginning of the “.cpp” file, so change the first few lines of int main() to look like this:

```cpp
int main(int argc, char* argv[]) {
    int flag;
    parameter_value_set parameter_values;
    flag = parse_command_line_arguments(argc, argv, parameter_values);
    if (flag == 1)
    {
        // output help file and exit
        return 0;
    }

    std::stringstream ss;
    int generations, ig, rep, number_of_replicates;
    number_of_replicates = parameter_values.p_reps;

    std::string outfile_name, extended_outfile_name;
    outfile_name = parameter_values.file_name;
    outfile_name = parameter_values.file_name;

    The simulation should proceed unless parse_command_line_arguments() returns a value of one. In that case, the program should output help text to the screen and exit. For now, we will have the program exit. In principle, we could come back and add some help text later. So far, we have used two of our command-line parameter values, --reps and --filename. To use the rest, we have to invoke the function in the simulation_engine class. This function should be invoked after the simulation_engine array is created, and we need to invoke it for each object in the array. Immediately before the line that reads my_sim[0].save_parameter_values(paramer_filename), add the following loop:

```cpp
    int m;
    for (m = 0; m < number_of_replicates; m++)
        my_sim[m].update_parameter_values(parameter_values);
```
Configuration Properties->Debugging. You will see an entry for “Command Arguments”, which is where you should type your command-line arguments. Type the following into the little window:

```
--reps 3 --carry_cap 128 --mut_var0 0.10
```

Run the program and confirm that your output reflects these changes. Your changes to the parameter values should be reflected in your output files. For example, you should get three replicates instead of the default five, and each replicate should have a carrying capacity (and population size) of 128 and a mutational variance for trait 0 of 0.1.

We can (and should) do further testing to make sure that every argument works as intended, but this testing is more easily carried out on our Linux machine, which is more at peace with its command line than our Windows machine. Open the dialog where you entered the command-line arguments (Project->Properties->Debugging->Command Arguments) and delete the text you placed there. We want to go ahead and get rid of the command-line arguments, so we do not forget about them later.

Copy your “.cpp” and “.h” files from your Visual Studio project directory into the folder (or a subfolder) in the directory that is shared with your Ubuntu virtual machine. Compile the program the following statement (using whatever name your “.cpp” file actually has):

```
g++ Chapter13example.cpp -o gmatrix_simulator
```

Then run the program with no command-line arguments (i.e., type `./gmatrix_simulator` at the command prompt. It should run and give you the usual output under the default parameter combinations. Now let us run it with some arguments. We can use the same ones as above to show that the functionality is the same on Ubuntu as it was on Windows. At the command prompt, type:

```
./gmatrix_simulator --reps 3 --carry_cap 128 --mut_var0 0.10 --filename ubuntu_test
```

We also added one additional argument to change the filename. After these runs, you should be able to check the files and verify that the arguments are working as intended. We have accomplished another step in our mission – our program now accepts command-line arguments that can be used to change the parameter values.

Shell Scripting

Now that we have an Ubuntu version of our program that can accept arguments from the command line, we are ready to learn about shell scripting. As noted above, the “shell” is the command-line interpreter that makes it possible for you to communicate with your machine through the command line. Ubuntu and the Mac OS use BASH (Bourne-Again SHell) as their command-line language. In addition to being a command-line interpreter, BASH functions as a scripting language, which allows the construction of simple programs that automate consecutive tasks. We will only scratch the surface of what is possible in the BASH environment, but it will be enough to automate our simulation runs.

We can start with a simple script that just invokes a few standard commands. Open JEdit (if you do not yet have it, use `sudo apt-get install jedit` to install it). Create a new text file (an empty one should open by default), and type the following text into it:

```
#!/bin/bash

pwd
ls
cd /media
pwd
```

206
Save the file in your shared drive with a name such as “test.sh”. Use the cd command to navigate to the folder where you saved the text file, and type \texttt{./test.sh} to run the script. You should get output that looks something like the output in Figure 14.1 (if you do not have permission to execute it, use \texttt{chmod u+x test.sh} to change the permissions).

![Figure 14.1: The results of a simple shell script.](image)

The script prints the path to a directory and the contents of the directory, starting with the current directory, and proceeding through two additional directories. Our script automated these tasks. The first line of the script, \texttt{#!/bin/bash}, identifies the text file as a BASH shell script. Every other line is a command that will be executed in order, starting at the top of the file. The command \texttt{pwd} (print working directory) simply outputs the path to the current directory. The command \texttt{ls} lists the contents of the current directory, and \texttt{cd} allows us to move through the file system. If you call \texttt{cd} without any arguments (as we do in the script), you end up in your home directory.

Believe it or not, we know enough now to write a script that will invoke our program. If “test.sh” is no longer open in JEdit, go ahead and open it again. Delete all of the text, except for the first line (\texttt{#!/bin/bash}). Add text to “test.sh” until it looks something like this:

\begin{verbatim}
#!/bin/bash
./gmatrix_simulator \
   --filename test_output \
   --reps 4 \
   --carry_cap 200 \
   --gens 500
\end{verbatim}

The \ character indicates that the command spills onto the next line, so the \ will not be included in your command that ends up being executed, but it does allow you to organize exceptionally long commands by putting different parts on different lines in your script. Thus, we could either put our entire command, with all of the arguments, on one huge line, or we could choose to put one argument per line, or anything in between. Save the “test.sh” script into the same directory where your compiled program resides, and type \texttt{./test.sh} to run it. The program should run with the appropriate changes to the parameter values. You can leave “test.sh” where it is if you wish, but trash everything else.
shared between a host (i.e., Windows in our case) and a virtual machine (i.e., Ubuntu), delete files from
the host side to ensure they really get deleted.

**Two More Minor Modifications**

Our program still has two features that are less than ideal. One is that it prompts the user to hit enter at
the end. That feature was nice when we were debugging and viewing output on the screen, but the
program no longer outputs anything to the screen. In addition, if we have to hit enter after each time the
program runs, what is the point of stringing together multiple runs in a script? After each run of the
program, we would have to return to our computer and hit enter. The second problem has to do with
the allocation of memory, and we will return to it after we fix the first problem. Start by discarding all of
the source files and compiled executables on your Ubuntu machine. We should make the changes back in
Windows in Visual Studio, and we do not want to accumulate lots of different versions of our source and
header files all over our computer. Back in Visual Studio, scroll down to the end of your “.cpp” file.
Delete the following section of code:

```cpp
char end_it;
std::cout << "\n\nEnter any character to exit...";
std::cin >> end_it;
```

That solves problem one.

Problem two is that we allocate all of the memory for every simulation run simultaneously when the
simulation_engine objects are created. We also always allocate arrays that can hold 2000 adults and
10,000 progeny, regardless of the true population size. Once we know the parameter values for the run,
however, we also know the maximum number of adults and progeny, so we should allocate these arrays
in a way that uses a minimal amount of memory. With the values of 2000 and 10,000 hard-coded into the
model, the program will crash if the parameter values exceed these values, and the program will waste
memory if the parameter values are smaller than these values. We can fix this problem by taking the
memory allocation out of the constructor and moving it to the `initialize_population()` function.
Proceed to the constructor, `simulation_engine()`, and find this section of code:

```cpp
// Set the maximum number of adults and progeny in the population
// Allocate memory for the pointers for the adults and progeny
NadultMax = 2000;
NprogMax = 10000;
adult = new individual[NadultMax];
progeny = new individual[NprogMax];
```

Delete it from the constructor and add it to `initialize_population()`, with the following changes:

```cpp
// Set the maximum number of adults and progeny in the population
// Allocate memory for the pointers for the adults and progeny
NadultMax = CarryingCapacity;
NprogMax = CarryingCapacity * Fecundity;
adult = new individual[NadultMax];
progeny = new individual[NprogMax];
```

The maximum number of adults we ever have is equal to the carrying capacity, and the maximum
number of progeny we could ever have is equal to the carrying capacity times the fecundity per female.
Usually, we will be far below the maximum number of progeny, but we could theoretically approach it in
a population with mostly females.
Right now, we free up the memory associated with these arrays in the destructor for our class, but if we want to save memory, we need a function to free it up. Add this function to your class:

```c++
void deinitialize_population()
{
    delete[] adult;
    delete[] progeny;
}
```

Also delete the destructor, `~simulation_engine()`, because it is no longer needed. Back in your “.cpp” file, invoke `deinitialize_population()` right before the closing brace of the `rep` loop. After the `my_sim[rep]` object reports its means, it is no longer needed and its arrays can be decommissioned. This line of code invokes the function:

```c++
my_sim[rep].deinitialize_population();
```

Here is what the “.cpp” file looks like now, after all of the recent changes:

```c++
#include <iostream>
#include "MTwisterFunctions.h"
#include "simulation_engine.h"
#include <sstream>

int main(int argc, char* argv[])
{
    int flag;
    parameter_value_set parameter_values;
    flag = parse_command_line_arguments(argc, argv, parameter_values);
    if (flag == 1)
    {
        // output help file and exit
        return 0;
    }

    std::stringstream ss;
    int generations, ig, rep, number_of_replicates;
    number_of_replicates = parameter_values.p_reps;

    std::string outfile_name, extended_outfile_name;
    outfile_name = parameter_values.file_name;

    std::string parameter_filename;
    parameter_filename = outfile_name + "_parameters.csv";

    simulation_engine *my_sim = new simulation_engine[number_of_replicates];
    mean_recorder *mean_array = new mean_recorder[number_of_replicates];

    int m;
    for (m = 0; m < number_of_replicates; m++)
    {
        my_sim[m].update_parameter_values(parameter_values);
        my_sim[0].save_parameter_values(parameter_filename);
    }

    for (rep = 0; rep < number_of_replicates; rep++)
    {
```
ss.str("");
ss << outfile_name << "_run_" << rep + 1 << ".csv";
extended_outfile_name = ss.str();

bool initialization_success;
initialization_success = my_sim[rep].initialize_population();
if (!initialization_success)
{
    std::cout << "\nSimulation Initialization Failure!\n"
    return 0;
}

for (ig = 0; ig < my_sim[rep].getNumberOfInitialGenerations(); ig++)
{
    my_sim[rep].monogamy();
    my_sim[rep].mutation();
    my_sim[rep].selection(true);
    my_sim[rep].population_regulation();
}

    // Turn the life cycle halfway, so the
    // experimental generations start with progeny
    my_sim[rep].calculate_values_progeny();
    my_sim[rep].monogamy();
    my_sim[rep].mutation();

for (generations = 0; generations < my_sim[rep].getNumberOfGenerations(); generations++)
{
    if (!my_sim[rep].is_extinct())
    {
        my_sim[rep].selection(false);
        my_sim[rep].calculate_values_progeny();
        my_sim[rep].population_regulation();
        my_sim[rep].monogamy();
        my_sim[rep].mutation();
        my_sim[rep].calculate_values_adults();
        my_sim[rep].store_variables_in_memory();
    }
}

my_sim[rep].save_stored_variables(extended_outfile_name);

my_sim[rep].calc_run_means();
my_sim[rep].append_means_to_output_file(extended_outfile_name);

mean_array[rep] = my_sim[rep].report_means();
my_sim[rep].deinitialize_population();

int i, j, iNmeans;
double dNreps = number_of_replicates;
std::vector<double> mean_of_means;
std::vector<double> std_dev_of_means;

iNmeans = static_cast<int>(mean_array[0].mean_list.size());
for (i = 0; i < iNmeans; i++)
{  
    mean_of_means.push_back(0);
    std_dev_of_means.push_back(0);
}

// Calculate the means of the means
for (i = 0; i < number_of_replicates; i++)
{
    for (j = 0; j < iNmeans; j++)
    {
        mean_of_means[j] = mean_of_means[j] + mean_array[i].mean_list[j];
    }
}

for (j = 0; j < iNmeans; j++)

// Calculate the standard deviations of the means
for (i = 0; i < number_of_replicates; i++)
{
    for (j = 0; j < iNmeans; j++)
    {
            (mean_array[i].mean_list[j] - mean_of_means[j]) *
            (mean_array[i].mean_list[j] - mean_of_means[j]);
    }
}

for (j = 0; j < iNmeans; j++)
{
    std_dev_of_means[j] = sqrt(std_dev_of_means[j]);
}

// Save to a file
std::string summary_filename;
summary_filename = outfile_name + "_summary.csv";
std::ofstream outfile;
char temp_fn[256];
convert_string_to_char(summary_filename, temp_fn);
outfile.open(temp_fn);
outfile << "Rep,N,zbar0,zbar1,P00,P11,P12,r(P),gbar0,gbar1,G00,G11,G01,r(G)"
        << ",Lambda1,Lambda2,EvecX,EvecY,Angle,Size,Eccen,Strt0,Strt1"
        << ",~Lmbd1,~Lmbd2,~Ang,~Size,~Ecc,~G00,~G11,~G01,~r(g)"
        << ",ASR,Im,If,MdifM,MdifF";
for (i = 0; i < number_of_replicates; i++)
{
    outfile << "\nRep" " << i+1;
    for (j = 0; j < iNmeans; j++)
    {
        outfile << "," " << mean_array[i].mean_list[j];
    }
}

outfile << "\nMean_of_means";
for (j = 0; j < iNmeans; j++)
CHAPTER 14

```cpp
{
    outfile << "," << mean_of_means[j];
}

outfile << "\nStdev_of_means";
for (j = 0; j < iNmeans; j++)
{
    outfile << "," << std_dev_of_means[j];
}

outfile.close();
delete[] my_sim;
delete[] mean_array;
return 0;
}
```

Run the program in the Visual Studio debugger, and you should see that it uses a lot less memory than it did before, because now it only allocates the memory it needs for each individual simulation run. Before, it allocated all of the memory it could possibly need for all planned replicates simultaneously. It also automatically exits without the need to enter a character.

**Using Shell Scripting for a Massive Run**

Our simulation program is completely done (except for the help output), so go ahead and copy the header and source files over to your Linux machine. Compile them using g++. Let us put together a script that includes a line for every parameter value. We want to keep our scripts easy to understand, even if it means they might be quite long. Use JEdit to put together the script pasted below, which will produce one run of the simulation with 20 replicates, under the parameter values used by Jones et al. (2003) for the first row of Table 1.

```bash
#!/bin/bash

./gmatrix_simulator
    --filename jones_table1_row1
    --reps 20
    --init_gens 10000
    --init_w00 9
    --init_w11 9
    --init_sel_corr 0
    --init_opt0 0
    --init_opt1 0
    --init_sex_lim false
    --gens 2000
    --carry_cap 256
    --fecund 4
    --mate_enc 50
    --pref_var 0
    --loci_trt0 0
    --loci_trt1 0
    --loci_pleio 50
    --env_var0 1
    --env_var1 1
    --mut_var0 0.05
    --mut_var1 0.05
    --mut_corr 0
```
This script sets every parameter that is accessible to the user. Some of the arguments are not necessary, of course, because they set the parameter in question to the default value. However, if we start with this script, then we can just copy the text and modify it to alter the parameter values for subsequent runs. For instance, we could write a script to run the simulations for the first two rows of Table 1 in Jones et al. (2003), and it would look like the text pasted below.

```bash
#!/bin/bash
./gmatrix_simulator \ 
--filename jones_table1_row1 \ 
--reps 20 \ 
--init_gens 10000 \ 
--init_w00 9 \ 
--init_w11 9 \ 
--init_sel_corr 0 \ 
--init_opt0 0 \ 
--init_opt1 0 \ 
--init_sex_lim false \ 
--gens 2000 \ 
--carry_cap 256 \ 
--fecund 4 \ 
--mate_enc 50 \ 
--pref_var 0 \ 
--loci_trt0 0 \ 
--loci_trt1 0 \ 
--loci_pleio 50 \ 
--env_var0 1 \ 
--env_var1 1 \ 
--mut_var0 0.05 \ 
--mut_var1 0.05 \ 
--mut_corr 0 \ 
--mut_rate 0.0002 \ 
--exp_w00 9 \ 
--exp_w11 9 \ 
--exp_sel_corr 0 \ 
--exp_opt0 0 \ 
--exp_opt1 0 \ 
--exp_sex_lim false

./gmatrix_simulator \ 
--filename jones_table1_row2 \ 
--reps 20 \ 
--init_gens 10000 \ 
--init_w00 9 \ 
--init_w11 9 \ 
--init_sel_corr 0.25 \ 
--init_opt0 0 \ 
--init_opt1 0 \ 
```
This script invokes the program twice, and the second run will start as soon as the first one ends. The second command invoking the program is just like the first, except we have changed the name of the output file, so the results from the first run do not get overwritten, and we have also changed the value of the selectional correlation, which was the parameter varying in the first set of rows in the table. Adding the second command was easy, of course, because we can copy the first command, paste it into our script, and edit it a bit to make sure it will do what we want it to. For a script that tackles all of the runs making up Table 1 in Jones et al. (2003), see the GitHub page associated with this book.

We are using the simplest form of shell script here, where we type the commands that we would normally type into the command line into the text file that serves as our script. This approach works well for our purposes, because much of the functionality of our program is already automated within our C++ code. Each time we invoke a command, it takes a while to run, and our script does not need to include a huge number of commands. However, the BASH shell has an entire programming language built into it, and it supports variables, if statements, for loops, and so forth. This topic is beyond the scope of our book, but you should be aware that shell scripting can be used to write much more complex and useful scripts than the ones we have developed here.

Chapter Summary

This chapter saw some important changes to our program, along with some key programming concepts that made these changes possible. In terms of programming concepts, we learned a bit about the Linux operating system and modified our program so that it would successfully compile using the GNU g++ compiler that comes with Linux. In the process, we learned about the “stringstream” class, which provides a convenient way of converting text to numbers and vice versa. We also learned a little about null-terminated character arrays, which were used to handle strings in the days before the string class was invented. Next, we modified our program to accept arguments from the user from the command-line, and we learned how to parse these arguments. Finally, we delved into shell scripting in Linux as a way to automate multiple runs of our program. With these last few steps, we have completed the program, and we now possess the skills to write programs and run them in a convenient way on a wide range of operating systems.
Chapter 15. Taking Mate Choice for a Spin

After Chapter 14, we should feel very pleased with our program. It runs without crashing, it can be automated to run many replications under many different parameter combinations, and it reproduces the results of previous studies based on the same model. If our goal was to reinvent the wheel, then we should congratulate ourselves. We did it! However, we no doubt have loftier ambitions, and we would like to explore new avenues of research. One such avenue, which we built into our model but have not yet used much, is the evolution of mating preferences and sexually selected traits. As of this writing, the only published study to use individual-based quantitative genetics simulations to study sexual selection appears to be an article entitled, “Individual-based simulation of sexual selection: a quantitative genetic approach” by D. van Dijk, P. M. A. Sloot, J. C. Tay, and M. C. Schut (2012, Procedia Computer Science 1:2003-2011). This study actually differs from our model in many ways and explores only a few aspects of the sexual selection process, so many issues remain to be addressed with our model. We will start by asking whether or not mate choice affects the evolution of the male trait. The answer to this question is known, of course, but it serves as a nice starting point.

Bringing Back Non-Monogamous Mating

We took great pains in a previous chapter to implement a monogamous mating system to make our model directly comparable to the Jones et al. (2003) model, but for the sexual selection model, we want the mating system to be polygynous with the potential for mate choice. When mating is polygynous, we will impose random mating during the initial generations and mate choice during the experimental generations. In this way, we can build up a population with genetic variation in traits and preferences before adding mate choice.

To implement this change, we could take several routes. One route would be to add another variable for the parameter specifying the nature of the mating system. Our program is now a bit cumbersome, with many moving parts, so this change would require several small changes in many places. For instance, we would need to add an argument to allow the mating system to be set from the command-line, we would need to add a new parameter variable, and we would need to change the source code to respond to the parameter value. All of these steps might be worth doing if we want to keep a general model handy that can do almost anything. However, as we get more interested in sexual selection, our model may diverge further and further from a model aimed at understanding the genetic architecture under natural selection.

Another alternative is to create a new project with a model specifically focused on sexual selection, and just let it diverge from the basic model we have already developed. This approach is a little simpler from the perspective of describing it in a book, so we will take this route. You will want to create a new project in Visual Studio with a name like “sexual_selection_sim”. Then import the source and header files from your existing project. Here is a quick synopsis of the steps: (1) Select File->New->Project…; (2) Navigate to the Visual C++ menu and choose “Empty Project”; (3) Type a name into the “Name” dialog box; (4) Press “Okay”; (5) Go to Project->Properties->Configuration Properties and set “Character Set” to “Not Set”; (5) Return to Windows, find the folder containing your old project (from the previous chapter), copy the source and header files, and paste them into the appropriate folder associated with your new project (the folder with the project’s name nested within a folder with the project’s name); (6) Rename the source file to something that is more informative; and (7) Return to
Visual Studio and use the Solution Explorer to add the two header files and source file (i.e., right click on “Header Files” and “Source Files” and use Add->Existing Item). Finally, compile the program and run it to make sure it runs just like the old version.

Luckily, we have already implemented everything we need for a sexual selection model, so some slight alterations to the source code will put us in a position to run some relevant simulations. In the source (“.cpp”) file, find the initial generations loop (the iterator should be `ig`). In this loop, the first line of code invokes `monogamy()`. Change it to:

```cpp
my_sim[rep].polygynous_mating();
```

Make the same change in the call to `monogamy()` that occurs between the initial generations and experimental generations. Finally, change the line of code that invokes `monogamy()` in the `generations` loop to:

```cpp
my_sim[rep].gaussian_mating();
```

Compile the program and run it to make sure that it still works. The results should be similar to runs under monogamy, except for some subtle differences. For instance, the genetic variance should be smaller under polygyny than monogamy, because polygynous mating reduces the effective population size relative to monogamy. You might also notice that the male opportunity for sexual selection (“Im” in our output) and the mating differential on males (“MdifM”) are no longer exactly zero, as they were under monogamy. Thus, with polygyny we have the potential for sexual selection on males, whereas our monogamous mating system with a perfectly equal sex ratio provided no room for the sexual selection process.

**Conducting an Experiment**

A scientific study based on a simulation model is an interesting beast, sandwiched awkwardly between actual empirical studies and mathematical analytical studies. The simulation has the problem that it does not involve real organisms, so the insights are only as good as the parameter values plugged into the model. On the other hand, the simulation differs from analytical theory in that it cannot be completely solved, as systems of equations often can, so we may not be able to tell easily how every individual parameter affects the behavior of the system. Usually, individual-based simulations have more parameters and fewer simplifying assumptions than mathematical theory, so the simulations can be more realistic but are also harder to understand. With many parameters, an exhaustive treatment of all combinations of possible parameter values is almost always impossible. Consequently, the analysis of a simulation-based model is best approached in the same way an empirical problem would be tackled in a living system.

To analyze the simulation, then, we need a hypothesis. We will need control runs and experimental runs, and we will need to decide which variables are important to measure to test our hypothesis. Luckily, we have access to every possible variable in our simulated system if we want it. This feature is both a blessing and a curse. On the one hand, we can measure aspects of our simulation that we would never be able to measure in living systems. On the other hand, the sheer volume of variables at our disposal can be overwhelming. We will narrow the scope of the problem by choosing a couple of straightforward predictions from sexual selection theory and testing them using our model. All of the predictions here are summarized and discussed in the excellent book *Sexual Selection* by Malte Andersson.

**Prediction 1: Mate Choice Drives the Evolution of the Male Phenotype**

We can easily arrange a situation in which females prefer males who are larger or smaller than the mean male trait size. When females prefer such males, the male trait should respond to this sexual
TAKING MATE CHOICE FOR A SPIN

selection pressure and become larger or smaller. In addition, we can predict that stronger mate choice should result in a more rapid response to selection in the male trait. These simulations are most easily run in our Linux machine, so copy your header files and your source files (modified to use polygyny and Gaussian mating) over to your Linux system and compile them with an informative name (in a folder with an equally informative name). Start with the script that we used before to test the simulation relative to the Jones et al. (2003) paper. The control run will start with the following parameter values (note that my compiled executable is named “Gmatrix_ss_simulator” – change this name to match yours):

```bash
#!/bin/bash
./Gmatrix_ss_simulator \
--filename mate_choice_control \ 
--reps 20 \ 
--init_gens 10000 \ 
--init_w00 9 \ 
--init_w11 9 \ 
--init_sel_corr 0 \ 
--init_opt0 0 \ 
--init_opt1 1 \ 
--init_sex_lim true \ 
--gens 2000 \ 
--carry_cap 256 \ 
--fecund 4 \ 
--mate_enc 50 \ 
--pref_var 0 \ 
--loci_trt0 0 \ 
--loci_trt1 0 \ 
--loci_pleio 50 \ 
--env_var0 1 \ 
--env_var1 1 \ 
--mut_var0 0.05 \ 
--mut_var1 0.05 \ 
--mut_corr 0 \ 
--mut_rate 0.0002 \ 
--exp_w00 0 \ 
--exp_w11 9 \ 
--exp_sel_corr 0 \ 
--exp_opt0 0 \ 
--exp_opt1 1 \ 
--exp_sex_lim true
```

These parameter values have some important changes relative to the values we used in the previous chapter. For instance, now we are using sex-limited selection. In the initial generations, we have selection on both the trait and the preference, but the optimum for the preference is one instead of zero, which will result in a mean preference in the females that is a little less than one standard deviation above the mean of the male trait. In the experimental generations, we remove natural selection from the trait, to see what sexual selection will do in the absence of natural selection. We also keep selection on the preference, so that females will continue to prefer males with trait values slightly above the mean. In the control runs, we actually have random mating (**pref_var** is set to zero), so the female preference will be constrained by natural selection and the male trait will be affected by only genetic drift.

For the experimental runs, we want to vary one parameter relative to the control, much like you would do in an experimental study. If we want to know how the female mating preference affects male trait
evolution, we need to hold everything else constant and just vary female mating preferences. The next command in your script, then, should look like this:

```bash
./Gmatrix_ss_simulator \
--filename mate_choice_pref_var_199 \
--reps 20 \
--init_gens 10000 \
--init_w00 9 \
--init_w11 9 \
--init_sel_corr 0 \
--init_opt0 0 \
--init_opt1 1 \
--init_sex_lim true \
--gens 2000 \
--carry_cap 256 \
--fecund 4 \
--pref_var 199 \
--loci_trt0 0 \
--loci_trt1 0 \
--loci_pleio 50 \
--env_var0 1 \
--env_var1 1 \
--mut_var0 0.05 \
--mut_var1 0.05 \
--mut_corr 0 \
--mut_rate 0.0002 \
--exp_w00 0 \ 
--exp_w11 9 \
--exp_sel_corr 0 \
--exp_opt0 0 \
--exp_opt1 1 \
--exp_sex_lim true
```

Everything here is the same as in the first control run, except now `pref_var` is set to 199, which imposes weak mate choice. Recall that larger values result in weaker preferences and smaller values result in stronger preferences (except for zero, which we have reserved as a flag to indicate random mating). Add additional blocks of commands to your script with `pref_var` values of 99, 49, 29, 19, 9, and 4. Make sure the name of the executable (“Gmatrix_ss_simulator” in my script) is the same name as the one you gave your program when you compiled it. Also, ensure that each command in your script has a different “--filename” argument so that the runs do not write over one another. Run your script and wait for the output to populate your folder. These runs will take a while to complete, so you might want to let them run overnight or during your lunch break.

After the runs complete, inspect some of the output. In my results, the summary file for the control runs shows a mean male trait (zbar0) of -0.074 and a mean female preference of 0.998, so these means are pretty close to expectations. Recall that the male trait started with a value around zero before we removed all selection, and the female trait consistently experienced stabilizing selection with an optimum of one. If we look at the standard deviations of these means, we see that the standard deviation for the male trait (1.20 in my runs) is much larger than the standard deviation for the female preference (0.06 in my simulations). This result is consistent with our understanding of evolution, because we expect the male trait values to diverge in different runs due to genetic drift, whereas the female preference values will be held close to their initial values by the action of stabilizing selection.
Figure 15.1: The evolution of the female preference and the male ornament in a control simulation with random mating. The female preference is under stabilizing selection with an optimum of 1, whereas the male ornament is under no selection in the experimental generations. Consequently, the male ornament is free to change in value as a result of genetic drift. Because mating is random, the female preference has no effect in these control simulations.

Inspection of the results from an individual run will reveal these dynamics. Figure 15.1 shows the dynamics of the mean male trait value (the ornament) and the mean female preference over 2000 generations of a control simulation run. The female preference stays relatively close to its naturally selected optimum of one, with small departures due to genetic drift. Each departure is followed by a return to the optimum. With a larger population size or stronger selection, these departures would be even smaller.

Table 15.1: The effects of mating preferences on male trait evolution and the strength of sexual selection.

<table>
<thead>
<tr>
<th>Mating Preference Parameter</th>
<th>Mean Male Trait Value</th>
<th>Mean Female Preference</th>
<th>Opportunity for Sexual Sel. ($I_M$)</th>
<th>Mating Differential on Male Trait</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Mating</td>
<td>-0.074</td>
<td>0.998</td>
<td>1.072</td>
<td>0.0003</td>
</tr>
<tr>
<td>199</td>
<td>0.601</td>
<td>1.011</td>
<td>1.070</td>
<td>0.0063</td>
</tr>
<tr>
<td>99</td>
<td>1.504</td>
<td>0.990</td>
<td>1.070</td>
<td>0.0128</td>
</tr>
<tr>
<td>49</td>
<td>2.934</td>
<td>1.004</td>
<td>1.074</td>
<td>0.0272</td>
</tr>
<tr>
<td>29</td>
<td>4.694</td>
<td>1.007</td>
<td>1.072</td>
<td>0.0428</td>
</tr>
<tr>
<td>19</td>
<td>6.756</td>
<td>0.991</td>
<td>1.079</td>
<td>0.0625</td>
</tr>
<tr>
<td>9</td>
<td>15.302</td>
<td>1.055</td>
<td>1.095</td>
<td>0.1335</td>
</tr>
<tr>
<td>4</td>
<td>28.264</td>
<td>1.008</td>
<td>1.145</td>
<td>0.2479</td>
</tr>
</tbody>
</table>

The male ornament, however, appears to evolve randomly, with large departures from its starting values and what appears to be a random walk of ornament values over time. Inspection and comparison of multiple independent control runs will show that the female preference always has similar dynamics, whereas the male ornament’s evolution is unpredictable.
Inspection of the experimental results, in which we test different values of the strength of the mating preference, show very little effect of mate choice for a value of 199 for the Gaussian mating preference parameter but a very strong effect for a value of 4. Table 15.1 summarizes the mean male trait values and female preference values under different strengths of mate choice.

Our results show that the mating preference parameter does affect male trait evolution, and the effects are most pronounced when the value of the mating preference parameter is smaller. A smaller value corresponds to a narrower window of female preference, resulting in stronger mate choice. As expected, the female preference value is always near 1, its naturally selected optimum. Interestingly, mate choice has a relatively small effect on the opportunity for sexual selection, which turns out to describe the mating system’s potential to produce sexual selection. This effect is small, because all of these simulation runs have a strictly polygynous mating system, the most favorable type for sexual selection on males to occur. Finally, the mating differential does increase as mating preferences get stronger, with particularly striking increases when the mating preference parameter reaches a value below 10.

Unfortunately, this table does not fully capture the true dynamics of the system, because the male trait value actually increases throughout the run for the parameter values that produce effective sexual selection. For example, Figure 15.2 shows the dynamics of the ornament and preference for a sample run of the simulation with a Gaussian preference parameter of 19. This figure shows clearly that the value of the male ornament tends to increase throughout the simulation run, although it sometimes decreases for short periods due to genetic drift. From what we know about evolutionary biology, we can surmise that the male ornament would actually continue to increase for as long as we ran the simulation to some arbitrarily large value. This result would, of course, be impossible in the real world, where a plethora of constraints would prevent the evolution of an infinitely large trait. Under the present parameter combinations, our model has no constraints, because we have intentionally eliminated natural selection on the male trait.
Despite several unrealistic aspects of this set of parameters, we can conclude that our model works and that female preferences for males possessing ornaments larger than the mean do indeed result in the evolution of larger male traits over evolutionary time.

**Prediction 2: Mate Choice Causes Genetic Correlations Between Traits and Preferences**

One important feature of sexual selection, especially when we talk about the evolution of mating preferences, is that mate choice results in a genetic correlation between the ornament and the preference. This genetic correlation arises because the choosiest females will mate with only the most attractive males, and their offspring will have genes for extreme choosiness and extremely attractive traits. The less choosy females, on the other hand, will mate will less attractive males on average, so their offspring will have the less choosy genes and the less attractive ornament genes. This situation leads to a genetic correlation in the population between ornament and preference values.

We can use our model to ascertain the extent to which genetic correlations between ornaments and preferences arise as a function of different parameter values. For the purposes of demonstration, we will use the following set of starting parameter values:

```bash
#!/bin/bash
./Gmatrix_ss_simulator \
--filename gen_corr_control \
--reps 20 \
--init_gens 10000 \
--init_w00 99 \
--init_w11 99 \
--init_sel_corr 0 \
--init_opt0 0 \
--init_opt1 0 \
--init_sex_lim true \
--gens 2000 \
--carry_cap 2048 \
--fecund 4 \
--mate_enc 200 \
--pref_var 0 \
--loci_trt0 50 \
--loci_trt1 50 \
--loci_pleio 0 \
--env_var0 0 \
--env_var1 0 \
--mut_var0 0.05 \
--mut_var1 0.05 \
--mut_corr 0 \
--mut_rate 0.0002 \
--exp_w00 99 \
--exp_w11 99 \
--exp_sel_corr 0 \
--exp_opt0 0 \
--exp_opt1 0 \
--exp_sex_lim true
```

These parameter values have some key differences compared to the set we used in the previous section. For instance, we now have natural selection on the trait and preference during the initial and experimental generations, but the selection is weak. We also set the optima for both traits to 0, as we have no reason to believe that the preference should have an optimum different than that of the trait. In the
previous example we used all pleiotropic loci, and now we are using separate loci for the trait and preference. In most systems, it seems reasonable that the preferences, which are encoded in the sensory systems and the brain, would be determined by different loci than the ornaments, which tend to be morphological traits. Finally, we are increasing the carrying capacity and removing environmental variance to make the effects of mate choice and selection more obvious. The parameter values above are for the control run, with random mating. For the experimental runs, use the same preference parameters as we did in the previous section: 199, 99, 49, 29, 19, 9, and 4.

After the runs complete, assuming we correctly changed the name of the output file for each parameter set, we can examine the output by looking at the summary files. Table 15.2 shows the genetic correlation as a function of the value of the Gaussian preference parameter. The table also shows the mean male trait value, the standard deviation in mean male trait values across runs, and the mean opportunity for sexual selection.

Table 15.2: The effects of mating preferences on the evolution of the genetic correlation between the trait and preference.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Mating</td>
<td>-0.016</td>
<td>0.112</td>
<td>1.000</td>
<td>0.0001</td>
</tr>
<tr>
<td>199</td>
<td>-0.014</td>
<td>0.103</td>
<td>1.001</td>
<td>0.0043</td>
</tr>
<tr>
<td>99</td>
<td>-0.029</td>
<td>0.113</td>
<td>1.001</td>
<td>0.0080</td>
</tr>
<tr>
<td>49</td>
<td>0.041</td>
<td>0.168</td>
<td>1.002</td>
<td>0.0154</td>
</tr>
<tr>
<td>29</td>
<td>0.026</td>
<td>0.250</td>
<td>1.002</td>
<td>0.0236</td>
</tr>
<tr>
<td>19</td>
<td>0.091</td>
<td>0.474</td>
<td>1.003</td>
<td>0.0349</td>
</tr>
<tr>
<td>9</td>
<td>-0.316</td>
<td>0.784</td>
<td>0.994</td>
<td>0.0602</td>
</tr>
<tr>
<td>4</td>
<td>-2.435</td>
<td>9.358</td>
<td>0.539</td>
<td>0.0950</td>
</tr>
</tbody>
</table>

From Table 15.2, we can see that stronger sexual selection does result in a larger genetic correlation between the trait and the preference. The value of the genetic correlation is indistinguishable from zero in the control, and it increases to 0.095 in the run with a Gaussian preference parameter of 4. Intermediate strengths of preference produce intermediate values for the genetic correlation. A genetic correlation less than 0.1 does not sound very impressive, however. We might have expected a larger value, especially when mate choice is very strong. No one would blame us for wondering if this value of a genetic correlation is large enough to have evolutionary effects.

If we examine the mean trait values in Table 15.2, we see that they stay pretty close to zero, except when the Gaussian preference parameter has a value of 4. For some reason, the standard deviation across runs in the male trait value increases as mate choice becomes stronger, with a very dramatic increase for the strongest mate choice. This pattern seems strange at first, but inspection of the means from individual runs (in the summary file) shows that the control means are almost always near zero, whereas the means under strong mate choice sometimes differ dramatically from zero. This pattern is most evident in the runs with a Gaussian preference parameter of 4, where male trait means were sometimes a large positive value (up to 17.3 in my runs) and sometimes substantially negative (-26.2 in my most extreme run). Clearly, the mean trait value is not capturing the entire essence of what is happening in our runs.

We can obtain a clearer picture of the dynamics of the trait and preference by examining individual runs of the simulation. We can start by taking a look at some control runs to get an idea of how the mean and preference behave in the absence of sexual selection. Figure 15.3 shows the results from a typical control run. In these runs, stabilizing selection keeps the trait and preference near their optima, which are both zero in this case. Small departures are caused by genetic drift, but the trait values never exceed an
absolute value of more than about 0.5, which is generally much less than a single phenotypic standard deviation.

When we examine the runs with a Gaussian preference parameter of 4, we see a very different picture. The values of the trait and preference for a sample run are shown in Figure 15.4. In this particular run, the male trait value oscillates between very small and very large values. Interestingly, the female preference mirrors the evolution of the male trait, albeit weakly. This phenomenon is caused by the genetic correlation. As the male trait evolves to a large positive value, for instance, the female preference also increases in value slightly because the preference is positively correlated with the male trait. A similar effect occurs as the trait evolves to extreme negative values. The genetic correlation, although small in magnitude on average, results in very dramatic trait evolution (note that the range of values shown in Figure 15.4 is much, much larger than that in Figure 15.3). Your individual runs might look considerably different than the one shown in Figure 15.4. Indeed, each simulation run looked quite a bit different from the previous one in my simulations, so the dynamics include a fair bit of stochasticity. Most of the runs involve some oscillatory behavior, but in some runs the male trait mean seems to stabilize at a very extreme positive or negative value. In some runs, the population even goes extinct before 2000 generation elapse, probably due to very strong selection that essentially kills almost all of the males in the population.

Figure 15.3: A control run from our simulations aimed at examining the evolution of the genetic correlation during mate choice. The values of the ornament and preference remain near their optima, with small departures caused by genetic drift.

The dynamics under a Gaussian preference parameter value of 4 provide an example of Fisherian sexual selection. In our model, we know that the male trait signifies nothing. Males with a value of zero have the highest probability of surviving due to stabilizing selection with an optimum of zero, but traits that depart from zero have reduced fitness. We should find it interesting, then, that the male trait evolves away from this naturally selected optimum even though males carrying these extreme traits have reduced fitness from a natural selection standpoint. An explanation for this process was provided by R. A. Fisher,
one of the architects of modern evolutionary biology, in his book, *The Genetical Theory of Natural Selection* (1930, Clarendon Press). Mate choice results in the evolution of a genetic correlation, for reasons we have already discussed, and this genetic correlation means that mating preferences evolve when the male trait evolves. Thus, if females prefer males with larger traits, then female preferences for large-trait males will also become stronger over evolutionary time, because males with large traits will tend to produce daughters with strong preferences. Males with large traits, being preferred, will also leave more offspring than other males, so the strong preferences will increase in frequency in the population. Ultimately, we see a positive feedback loop in which stronger preferences result in larger traits, which result in stronger preferences causing even larger traits, and so forth. Fisherian sexual selection is a rare example of a positive feedback loop in biology and has been suggested as a mechanism for the evolution of arbitrarily large, useless ornaments, such as the peacock’s tail.

**Figure 15.4:** The evolution of the male trait and female preference during a single run of the simulation with a Gaussian preference parameter of 4, which corresponds to a strong preference. Here, we see dramatic trait evolution as a consequence of sexual selection, and in this particular example, the male trait alternates between extreme negative and extreme positive values.

Inspection of runs with weaker mate choice, or larger values of the Gaussian preference parameter, shows a similar pattern, but with more noise and less extreme trait values. Regardless, mate choice and the associated genetic correlation appear to affect evolutionary trajectories, except when preferences are weak. A couple of questions probably remain after this analysis, and we would have to do more work to get answers to these questions. For instance, what starts the Fisherian process? From an initial inspection of runs, it looks like genetic drift of the preference is enough to jump start the whole affair, but we could probably envision additional simulations to address this hypothesis more directly. Also, why does the evolution of the trait sometimes stop, as in Figure 15.4, and then reverse direction and evolve continuously toward an equally extreme value of opposite sign? The answer to this question is even less clear, and again we would have to do additional work to solve it.
Our goal here was not to answer every question about sexual selection. Instead, the goal was to provide an introduction to the programming skills and logic necessary to construct evolutionary models. We have now accomplished this goal. Even though we will leave our sexual selection questions hanging, we feel good that we have developed a skill set capable of addressing these and many other questions by modifying our simulation model or creating entirely new models in the future.

Chapter Summary

In this chapter, we made a copy of our project and switched it back to non-monogamous mating to investigate some aspects of sexual selection. We saw that the analysis of a simulation-based model requires an approach much like that of an empirical study. We need to formulate hypotheses with specific predictions and then manipulate specific parameters of our model to test these predictions. In particular, we tested the hypothesis that mate choice would result in the evolution of elaborate traits. This test was fairly trivial, as we knew that we would find a positive outcome. However, it provided a good introduction to the logic underlying the approach and the setup of a script to address the hypothesis. Our second experiment addressed the hypothesis that mate choice would result in the evolution of a genetic correlation between male ornaments and female preferences. We found that strong mate choice did result in a genetic correlation, but the correlation was surprisingly weak, with a value of less than 0.1 in all cases. Nevertheless, these small genetic correlations were enough to start a Fisherian sexual selection process, which resulted in extreme trait evolution and even caused population extinction in some cases. Our goal here was not to fully analyze sexual selection, as that could be a topic for another entire book, but to see the logic underlying the analysis of simulation-based models. With this last piece of the puzzle in place, our mission has been accomplished.
Appendix 1. Random Number Generator Code

#pragma once
#include <cmath>

// The following code has to do with the random number
// generator. This is the Mersenne Twister prng.
// The reference is:
// M. Matsumoto and T. Nishimura,
// Mersenne Twister: A 623-Dimensionally Equidistributed
// Uniform Pseudo-Random Number Generator,
// ACM Transactions on Modeling and Computer Simulation,
// Vol. 8, No. 1, January 1998, pp. 3-30

// Copyright corresponding to Mersenne Twister Code:

// Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
// All rights reserved.
// Redistribution and use in source and binary forms, with or without
// modification, are permitted provided that the following conditions
// are met:
// 1. Redistributions of source code must retain the above copyright
// notice, this list of conditions and the following disclaimer.
// 2. Redistributions in binary form must reproduce the above copyright
// notice, this list of conditions and the following disclaimer in the
// documentation and/or other materials provided with the distribution.
// 3. The names of its contributors may not be used to endorse or promote
// products derived from this software without specific prior written
// permission.
// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
// "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
// LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
// A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER
// OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
// EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
// PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
// PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
// LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
// NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
// SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

/* Period parameters */

#define N 624
#define M 397
#define MATRIX_A 0x9908b0df /* constant vector a */
#define UPPER_MASK 0x80000000 /* most significant w-r bits */
#define LOWER_MASK 0x7fffffff /* least significant r bits */
#define My_PI 3.14159265358979323846 /* PI */
/* Tempering parameters */
#define TEMPERING_MASK_B 0x9d2c5680
#define TEMPERING_MASK_C 0xefc60000
#define TEMPERING_SHIFT_U(y) (y >> 11)
#define TEMPERING_SHIFT_S(y) (y << 7)
#define TEMPERING_SHIFT_T(y) (y << 15)
#define TEMPERING_SHIFT_L(y) (y >> 18)

static unsigned long mt[N]; /* the array for the state vector */
static int mti = N + 1; /* mti==N+1 means mt[N] is not initialized */
double Two2the36 = 4294967296.0;
static double rnZ = 0;

/* initializing the array with a NONZERO seed */
void sgenrand(unsigned long int seed)
{
    /* setting initial seeds to mt[N] using */
    /* the generator Line 25 of Table 1 in */
    /* [KNUTH 1981, The Art of Computer Programming */
    /* Vol. 2 (2nd Ed.), pp102] */
    mt[0] = seed & 0xffffffff;
    for (mti = 1; mti<N; mti++)
        mt[mti] = (69069 * mt[mti - 1]) & 0xffffffff;
}

inline double genrand()
{
    unsigned long y;
    static unsigned long mag01[2] = { 0x0, MATRIX_A };
    /* mag01[x] = x * MATRIX_A for x=0,1 */
    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N + 1) /* if sgenrand() has not been called, */
            sgenrand(4357); /* a default initial seed is used */

        for (kk = 0; kk<N - M; kk++) {
            y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
            mt[kk] = mt[kk + M] ^ (y >> 1) ^ mag01[y & 0x1];
        }
        for (; kk<N - 1; kk++) {
            y = (mt[kk] & UPPER_MASK) | (mt[kk + 1] & LOWER_MASK);
            mt[kk] = mt[kk + (M - N)] ^ (y >> 1) ^ mag01[y & 0x1];
        }

        y = (mt[N - 1] & UPPER_MASK) | (mt[0] & LOWER_MASK);
        mt[N - 1] = mt[M - 1] ^ (y >> 1) ^ mag01[y & 0x1];

        mti = 0;
    }

    y = mt[mti++];
    y ^= TEMPERING_SHIFT_U(y);
    y ^= TEMPERING_SHIFT_S(y) & TEMPERING_MASK_B;
    y ^= TEMPERING_SHIFT_T(y) & TEMPERING_MASK_C;
    y ^= TEMPERING_SHIFT_L(y);
}
APPENDIX I

    return ((double)y / Two2the36); /* reals */
    //return y; /* for integer generation */
    // This should give a range of [0,1); for [0,1]
    // use Two2the36-1.
}

inline int randnum(int Max) // returns a value between 0 and Max-1.
{
    double rnms;
    double dbrnum;
    int irnum;
    rnms = genrand();
    dbrnum = floor(rnms * Max);
    irnum = static_cast<int>(dbrnum);
    return irnum;
}

// The following function is a standard routine for generating a random number from
// a normal distribution.
inline double randnorm(double rnmean, double rnsd)
{
    double S, v1, v2, X2;
    if (rnZ != 0)
    {
        X2 = rnZ;
        rnZ = 0;
    }
    else
    {
        do {
            v1 = 2.0 * genrand() - 1.0;
            v2 = 2.0 * genrand() - 1.0;
            S = v1 * v1 + v2 * v2;
        } while (S >= 1.0);
    S = sqrt((-2.0 * log(S)) / S);
    X2 = v1 * S;
    rnZ = v2 * S;
    }

    return rnmean + X2 * rnsd;
}

// The function below will generate random numbers from a bivariate
// normal distribution. The std deviations of the two
// distributions
// are given by sigmaX and sigmaY, and the correlation is rho.
// These bivariate normal routines are from the GNU free software database
// (C) 2000 James Theiler and Brian Gough.
inline void randbivnorm(double sigmaX, double sigmaY, double rho, double &bivN1, double &bivN2)
{
    double bivU, bivV, bivR2, bivScale;
do
{
    // choose x and y from a uniform square (-1, -1) to (1, 1)
    bivU = 2 * genrand() - 1;
    bivV = 2 * genrand() - 1;

    // see if it is in the unit circle
    bivR2 = bivU * bivU + bivV * bivV;
    } while (bivR2 > 1.0 || bivR2 == 0);

    bivScale = sqrt(-2.0 * log(bivR2) / bivR2);
    bivN1 = sigmaX * bivU * bivScale;
    bivN2 = sigmaY * (rho * bivU + sqrt(1 - rho*rho) * bivV) * bivScale;
} // end of randbivnorm

inline double bivnormpdf(const double bnX, const double bnY, const double sigmaX, const double sigmaY, const double rho)
{
    double bnU = bnX / sigmaX;
    double bnV = bnY / sigmaY;
    double bnC = 1 - rho*rho;
    double bnP = (1 / (2 * My_PI * sigmaX * sigmaY * sqrt(bnC))) * 
        exp(-((bnU * bnU - 2 * rho * bnU * bnV + bnV * bnV) / (2 * bnC)));
    return bnP;
} // end of bivnormpdf
Appendix 2. Setting Up a Linux Virtual Machine

Given that you are reading this book, your main operating system is probably Windows 10. If you are using a Mac, then your operating system already has most of the functionality of Linux built in, so you are already prepared to use shell scripting to automate your program runs. If you are using Windows 10, it actually has a Linux subsystem built in as well, so in principle you could activate the Linux subsystem and use it to automate your workflow. However, another general solution is to set up a “virtual machine”, which allows you to run another entire computer within your existing computer. Your new virtual computer can be configured to run almost any operating system, and will behave as if it is an actual computer. Because most distributions of Linux are free, we can download the version we prefer and install it on a virtual machine, which will give us Linux functionality within our Windows machine. Follow the steps below to set up a virtual machine with the Ubuntu version of Linux installed. As you perform these steps, keep in mind that virtual machines and Linux operating systems are constantly evolving, so you may need to troubleshoot some of the steps due to recent changes. Your best resource for troubleshooting is your internet search engine, but you will have to take some care to separate the good advice from the bad.

**Step 1 – Install VirtualBox:** We will use VirtualBox by Oracle, which can be downloaded at “www.virtualbox.org”. Like everything else in this book, VirtualBox is free. Download the installer and follow the directions to install VirtualBox on your machine.

**Step 2 – Download Ubuntu:** Before you start messing around with VirtualBox, you will also need to download a Linux operating system. There are various distributions (i.e., versions) of Linux, one of which is known as Ubuntu. Ubuntu is very popular and is widely used in computational biology circles. Go to “www.ubuntu.com/desktop” and hit the “Download Ubuntu” button. You will want the LTS version, which is the “long-term stability” release. The LTS versions are supported and updated for five years after their release, so the 16.04.3 LTS version, which is the current one as of this writing, will be supported until 2021. Click through the pages until the download begins. You can donate money if you wish, and I encourage you to support Ubuntu, but you can also set all the sliders to zero and get it for free. If you use it for a while and are happy with it, you can always donate later. This download might take a while, and it will save an “iso” file to your computer, which is a CD or DVD image. You will be able to install directly from this image onto your VirtualBox, so keep the Ubuntu iso file in a convenient location (like your desktop).

**Step 3 – Create a Virtual Machine and Install Ubuntu:** The next step is to create a virtual machine in VirtualBox and install Ubuntu on it. Open VirtualBox and hit the New button in the toolbar just below the File menu. Type a name for your new machine – something like “Ubuntu1604” would be fine. Make sure that Linux and Ubuntu (64-bit) are selected from the drop down menus. Hit the Next button to go to the next page, where you will select the amount of RAM for your machine. You want your virtual machine to have as much RAM as you can afford to give it, so that it is powerful enough to actually perform analyses. Use the slider to move the amount of RAM toward the rightmost end of the section highlighted in green. My laptop computer only has 8 GB of RAM, so on that computer, I would give my virtual machine half of it, which is 4096 MB (or 4 GB). When the virtual machine is running, my host operating system (Windows 10) will still have 4 GB left to run. Here we see one of the main problems with a virtual machine – we are splitting our system resources because the computer has to run both the host operating system and the virtual operating system at the same time. Once you are satisfied with your choice, hit the button to move to the next step.
The next decision is also an important one, and it involves the allocation of storage. Your guest operating system will run from a virtual hard disk created by VirtualBox. This virtual drive will take up space on an actual hard drive, so you need to think about how large you want it to be and where to put it. My laptop computer only has a 256 GB solid-state drive, which limits me pretty severely. My only options are to either give my virtual machine a small virtual hard disk on my computer’s solid-state drive or to give my virtual machine a large virtual hard disk on an external drive. If you have tons of hard drive space, I recommend giving your virtual machine a lot of storage, maybe 1 TB or more, but on a drive other than your main boot drive. In fact, if you have a desktop computer, you might want to think about installing another internal drive and allocating almost the entire drive to the virtual machine. However, I do not have this option on my laptop, so I would install VirtualBox on my solid-state drive and give it a relatively small virtual hard disk of 64 GB. Consequently, I would have to be careful with that space and run most of my analyses involving large files on an external drive, but I would still be able to use my virtual machine without an external drive as long as I stuck with relatively small analyses.

The next step is to select “Create a virtual hard disk now” from the menu and press the Create button. On the next page, leave “VDI (VirtualBox Disk Image)” selected and press Next. Leave “Dynamically allocated” checked on the next page and press Next. Now enter the maximum size you would like your virtual hard disk to be in the text box (or use the slider) and hit the Create button. The value you enter must be smaller than the amount of space available on your hard drive. If you are installing on your boot drive, where your host operating system resides, be sure to leave an extra 30-60 GB free for your host operating system. If you completely fill your hard drive, your computer will probably stop working. The creation of the virtual hard disk is almost instant, because VirtualBox will expand it only as needed. Right now it is very small. You should be back at the VirtualBox starting screen now, and your virtual machine should be listed in the window on the left. To the right of that window, you will see a variety of settings and details. Most of these can be left as defaults, but we do need to change one or two. Click on System, and go to the Processor tab. Increase the number of processors to half the number of CPUs contained in your computer. My laptop has eight, so I would let my virtual machine use four of them. Hit Okay. Now scroll down and click on USB. The USB settings will come up, and you will see that USB 1.1 is enabled. We would like to be able to use the fastest USB drives possible, so click on USB 3.0. A watchful eye will see a new message at the bottom of the window, which says “Invalid settings detected”. A mouse over the warning will show that we need to install the “Oracle VM VirtualBox Extension Pack”, so let us do that. Go to “https://www.virtualbox.org/wiki/Downloads” and download the extension pack for all supported platforms by clicking the appropriate link. An installer will download, and you can just double click it to begin the installation. Now go back to the USB settings and make sure USB 3.0 is still selected. You should see that the warning is gone.

Now we are ready to actually install Ubuntu onto the virtual machine. Ensure that you know where you put the “iso” file containing the Ubuntu operating system, and open VirtualBox if it is not already open. Make sure your new virtual machine is selected in the left-hand window, and then press Start. A new window will open and a dialog will immediately ask you to provide a start-up disk. The start-up disk is your Ubuntu “iso” file, so navigate to it and select it. Press Start. Now Ubuntu 16.04 will boot from the “.iso” file as if it is booting from a dvd. You will have a choice to either try Ubuntu or install Ubuntu. You want to install Ubuntu, so press the appropriate button. Check the box to “Download updates while installing Ubuntu” and press Continue. The next step is scary because the default selection is to “Erase disk and install Ubuntu”. Fortunately, it is talking about the virtual hard disk you created earlier, so your computer’s hard drive is not at risk. Leave “Erase disk and install Ubuntu” checked and click the Install.
Now button. Answer the next several questions, which are self-explanatory. Eventually, you will have to pick a username and a password. Keep both of these short, as you will have to type them a lot. I would just use my first name and a short password of six lowercase letters. Remember that your virtual machine does not have to be extremely secure because your computer is already protected by your host operating system (which should have a strong password already). Now the actual installation will start, and it might take a while. Once the installation is complete, it will probably prompt you for a restart, so go ahead and do that.

**Step 4 – Install the Guest Additions CD:** A few more steps are required to get our virtual machine set up the way we want it, and we might as well perform these steps now. First, install the “Guest Additions CD” by going to the menu at the top of your virtual machine window (while the virtual machine is running) – this menu is the one provided by VirtualBox, not the one inside the Ubuntu operating system. Under Devices, select “Insert Guest Additions CD Image…” and follow the directions. The Guest Additions CD (it is a virtual CD, not a real one) adds some functionality and will have to be reinstalled any time Ubuntu undergoes a major update. Restart your virtual machine when this installation is complete. If you encounter errors installing the Guest Additions CD, you might need to run the following commands (open a terminal window and type them at the command prompt, hitting “enter” after each command):

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install build-essential module-assistant
sudo m-a prepare
```

After running these four commands, repeat the instructions above and they should work. If you are not sure how to open a terminal window, find the directions about half way through Step 5.

**Step 5 – Set Up a Shared Folder:** Next, we need a convenient way of communicating with our host operating system, and one option is to use a shared folder. This folder will be visible to both the host and guest operating systems, and this step will only work if you have already installed the Guest Additions CD successfully. Keep in mind that your virtual machine has a large potential for disaster, so important files should be kept in the shared folder or on an external USB drive rather than solely on the virtual hard disk. To add a shared folder, start by closing your virtual machine. You should always formally shut it down using the menu in the upper right corner of the Ubuntu operating system. Do not just close the window, because that is like simply pulling the plug and will eventually result in a nonfunctional machine. Now open the VirtualBox manager, which might already be open. In the list of settings to the right, you will see there is one for shared folders. Click on “Shared folders” to open the settings dialog. You will see “Machine Folders” in a window, and to the right of the window is a little folder icon with a green “+” on it. Press that icon, and another dialog will pop up. Use the pull down menu labeled “Folder Path” and select “Other…”, which will bring up yet another dialog. Navigate to a convenient place, such as your desktop and select “New folder” from the dialog’s task bar or right click on an empty space in the window and select New->Folder. Name this folder something like “ubuntu_share”. In general, do not use spaces; if you want a space, use the underscore (“_”) instead. Back in the dialog, select your new folder and hit the Select Folder button. Now you will be back in the “Add Share” dialog. Leave “Read-only” unchecked and check the “Auto-mount” box. Press “OK” in both open dialogs to shut them. Still in your Windows operating system, navigate to your “ubuntu_share” folder and create a new text file with Notepad. Name this new file something like “test.txt”, type something into it and save it. We will use it to make sure the shared drive is working.
Now start your Ubuntu virtual machine. We need to figure out where the shared folder is located and how to access it. The toolbar on the left of the Ubuntu user interface has an icon that looks like an open file drawer. Double click this icon to open a file browser. Your shared drive will probably be located in a list to the left. Try to click on it, and you will find that you do not have the proper permissions to access it. Obviously, this situation has to be remedied for the folder to be useful to you. Shut the file browser by moving your mouse up and to the left of the information bar at the top of the Ubuntu user interface. When you mouse over the toolbar, a small “x” icon should appear to the left of a bunch of menus. This icon allows you to shut the window.

The task of finding our folder and giving ourselves access is best accomplished from a terminal window, using the command line. Click on the icon at the top of your left toolbar (“Search your computer”). Type terminal into the search box. An icon named “Terminal” should appear in the search results. Drag this icon onto the left toolbar, because you will need to use it a lot. Shut the search interface, and click on the Terminal icon. Type `ls` (that first character is a lower-case L) at the command prompt and hit enter. In general, you always hit enter after typing a command. This command lists the files in the current directory. You will see that your shared folder is not listed. This directory is your home directory, and Ubuntu uses the shortcut `~` for it. Ubuntu has another special directory known as the root directory, which is at the base of the file system, and its shortcut is just a plain `/`. To move between directories you can use the command `cd`, so try typing `cd /` to go to the root directory. Then type `ls` to see what files are there. Lots of directories should be listed, but we are interested in the one called “media”, so type `cd media` followed by `ls`. Now type `ls -l` [enter] (lower-case L followed by s, space, minus, lower-case L, and hit enter). The `-l` argument instructs `ls` to provide more information, including the permissions and owners of the files and folders. First you will see that your shared folder is here, but renamed by Ubuntu to “sf_ubuntu_share”. You will also see the word “vboxsf” in the row corresponding to your shared folder. You must be a member of this group to access the folder’s contents. Type `groups` [enter] to see if you are a member of vboxsf. If you are not, type `sudo usermod -a -G vboxsf adam` and hit enter, where “adam” is your username. You will have to type the password you selected when you set up Ubuntu. This setting change will not take effect until you restart your virtual machine, so go ahead and restart it.

You will not want to dig through all the folders to get to your shared folder every time, so let us also create a link to it from the home directory. This link is like a shortcut, and you will be able to use it just like the actual directory. After Ubuntu restarts, open a new terminal. Make sure you are in the home directory (which you should be by default). Type `ln -s /media/sf_ubuntu_share` and hit enter (the “l” is a lower-case “L”). This command makes a symbolic link to your sf_ubuntu_share folder, which you can use just like the real folder. Type `cd sf_ubuntu_share` followed by `ls` to list the contents. You should see the text file you created earlier in Windows and placed in the folder. If you want to see its contents, you can type `less test.txt`, where “test.txt” is the filename.

**Step 6 – Attach a USB External Drive:** If you have a computer with massive amounts of storage, this step might not be necessary, as you can put your shared drive on a huge hard drive and just pile all of your files in there. However, a USB drive can be a nice convenience for backing up files or analyzing large datasets. An external USB 3.0 hard drive can also potentially be very fast, so its use may not result in a large performance cost. With the virtual machine set up to use USB 3.0, it can easily recognize and access an external drive. Start by launching your virtual machine. Plug an external hard drive into your computer. Examine the lower right-hand corner of your Ubuntu virtual machine. You will see a series of
icons that look like DVDs, computer screens, folders, and so on. One of them looks like a little USB plug. A mouse-over shows that this icon indicates the activity of attached USB devices, and probably no devices are currently attached. Right click the icon and select your external hard drive from the list of devices. This action will detach it from Windows and attach it to your Ubuntu machine. This action will only work if the device is not being used by Windows, so all files and folders on the drive must be shut. After the device is attached to Ubuntu, the virtual machine will be able to work with the files on it until you detach it, again by right-clicking the appropriate icon. Once detached, the drive should be available to Windows once more. An external drive provides another way to transfer files between your host and guest operating systems. Within Ubuntu, the path to the external drive should be 

/media/adam/TOSHIBA EXT/, where adam is your username and TOSHIBA EXT is the name of the external hard drive (if yours has spaces in its name, replace them with underscores).

**Step 7 – Install JEdit:** The program JEdit is a free text editor aimed at computer programmers. It has lots of functionality that makes it more useful than regular everyday Notepad. In Ubuntu, the easiest way to install programs is usually to use the command *apt-get*. The command *sudo* allows Ubuntu to run a command as the super user, who can make changes to literally anything on the computer. Always think about unintended consequences if you do anything as the super user. To install JEdit, you can use the following commands:

```
sudo apt-get update
sudo apt-get install jedit
```

The first command just updates the apt-get repositories to ensure that you are getting the latest version of whatever you are trying to install. Always invoke this command before installing anything. The second command gets the JEdit program and installs it on your computer. After installation is complete, you can find JEdit by using the “Search your computer” button. You might also want to drag the JEdit icon over to your toolbar so you have easy access to it.

**Step 8 – The G++ Compiler:** You also should ensure that you have the g++ compiler, which is the GNU version of the C++ compiler, installed. It should be installed automatically as part of the Ubuntu installation. At a terminal window, type *man g++* to see if it is installed already. If it is installed, you will see a help file pop up. Otherwise, you will get an error indicating that there is no manual entry. If g++ is not installed, type the following to install it:

```
sudo apt-get update
sudo apt-get install g++
```

If you just ran sudo apt-get update a few minutes ago, you do not need to run the update again. Now your Ubuntu virtual machine is ready to go. If you get stuck, consult the list of Linux commands below or search the internet for help.

**Some Useful Linux Commands**
In a Linux desktop environment, much of the interaction with the operating system can be accomplished through the graphical user interface. On a supercomputer, however, almost everything is handled from the command line, so knowledge of a few commands can be helpful. Here is a list of some of the most commonly used commands in Linux. For more information, get an introductory book on Linux or search the web. Many great websites provide excellent examples of how to survive the Linux command line. A
lot of powerful commands are left out here, so if you would like to be considered proficient with Linux, you will need to get a book and do some additional reading.

Here are some important commands in Ubuntu:

> Redirects the standard output into a text file with the specified name. In other words, if the output would usually appear on the screen, now it will be written to a text file. Examples:

```bash
grep include simulation_engine.h > include_statements.txt
ls -a > files.txt
```

cat Concatenate any number of text files and output the results to the screen. Use the > operator to redirect the output to a file (see the entry for > above).

```bash
cat text_file1.txt text_file2.txt
```

cd Change directory. Use this command to navigate through the computer’s file structure. Note that cd with no arguments will send you to the home folder, which is represented by a ~. If you use cd /, you will end up in the root directory, and cd .. moves you up one level in the directory hierarchy.

```bash
cd ~/sf_ushare/Gmatrix
```

chmod Modify the permissions for a file.

```bash
chmod u+x myscript.h
```

cp Copy a file to a new location.

```bash
cp ~/sf_ushare/test.sh ~/test_moved.sh
```

grep A powerful utility that prints lines from a text file matching a specified pattern. It is beyond the scope of this book, but combining grep with “regular expressions” can be a very powerful way to search and parse text files. If you are interested, use a search engine to find tutorials on “regular expressions”.

```bash
grep Mean_of_means simrun_summary.csv
grep Mean_of_means *.csv
```

head Display the first ten lines of a text file. Use the argument -n to change the number of lines displayed.

```bash
head -n 50 text_file.txt
```

less View the contents of a text file.

```bash
less text_file.txt
```
APPENDIX 2

**ls** List the files in a directory. Use the argument `-a` to show all the files in a directory, including hidden ones. Use the argument `-l` to show additional information about the files, including the permissions, owner, and size.

```bash
ls -a
```

**man** Show the help manual for the relevant program or command.

```bash
man grep
```

**mkdir** Create a new directory.

```bash
mkdir my_new_directory
```

**mv** Move a file to a new location. This command deletes the copy of the file in the old location. If you use move within the same directory and specify a new name, this command has the effect of renaming the file.

```bash
mv ~/test.sh ~/sf_ushare
mv ~/test.sh ~/sf_ushare/test_moved.sh
```

**nano** Open a text file with the **nano** command-line text editor.

```bash
nano test.sh
```

**pwd** Display the path of the current working directory.

```bash
pwd
```

**rm** Delete a file or directory. Be very careful with this one.

```bash
rm test.sh
```

**sudo** If you precede a command with **sudo**, you will execute the command as a super user, capable of making any change to the computer. Use this command with caution, as a super user can literally destroy your operating system (by, for example, deleting every file in the root directory accidentally).

```bash
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install jedit
```

**tail** Display the final 10 lines from a text file. Use the argument `-n` to change the number of lines displayed.

```bash
tail -n 50 test.sh
```
Appendix 3. Useful C++ Commands

The C++ programming language consists of many commands, and most of them were not covered in this book. However, we covered enough to write a functional individual-based simulation program. To learn more about the C++ programming language, consult the many books that are available on the subject. I mention two good ones in Chapter 1, but there are many others. Also be aware that entire websites are devoted to the C++ language, so a savvy internet searcher can find the answer to almost any problem on the web. This appendix provides a summary of the most important commands we covered in this book, plus a few we did not cover. In addition to commands, some concepts, such as functions and classes, are included in this appendix.

array

An array can be thought of as a list or table of things. The things can be anything from a single character to entire classes. Arrays provide a simple way of iterating through a group of entities. Arrays can be declared in multiple different ways. A small array whose size is known at design time can be declared using [] after the variable name. For instance, int numbers[10] would declare an array of 10 integers (with index values from 0 to 9). Larger arrays, whose size may or may not be known at design time, can be declared using pointers (see below).

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string name[3];
    name[0] = "George";
    name[1] = "Martha";
    name[2] = "Kelly";

    for (size_t i = 0; i < 3; i++)
        std::cout << name[i] << "\n";
}
```

bool

Use this keyword to declare a variable of type Boolean, which can have the value of true or false.

```cpp
#include <iostream>

int main()
{
    bool is_it = false;

    if (!is_it)
        std::cout << "It isn't!";
    else
        std::cout << "It is!";
}
```

char

A variable type that holds a single text character, such as “a” or “K” or “1”. Note that a numerical character, like “1” or “2”, will be represented as text, not as a numerical value. Each character does have an underlying numerical value (that the program uses to look
up the character), and you can find out what the hidden numerical value is by looking it up on an ASCII table or using code like that shown in the example below. Note that C++ expects a single quote around a character, and a double quote around a string. Thus, you would refer to ‘A’ or “A banana” using different quotation marks.

```
#include <iostream>

int main()
{

    char char_a = 'a';
    char char_6 = '6';

    std::cout << "The character a:\t" << char_a << "\n";
    std::cout << "The character 6:\t" << char_6 << "\n";
    std::cout << "ASCII code for a:\t" << static_cast<int>(char_a) << "\n";
    std::cout << "ASCII code for 6:\t" << static_cast<int>(char_6) << "\n";
}
```

**cin**

This function retrieves information from the standard input (the keyboard) and deposits it in a variable. The extraction operator, `>>`, is used with **cin** to direct the input into the variable of interest. Usually this function would be used to get a number, character or string from the user in an interactive program. See the entry for **cout** for an example of how to use **cin**.

**class**

The class is the special variable type that is at the heart of object-oriented programming. Classes provide flexible containers that can hold and organize related variables and functions that act on those variables. Throughout this book, we developed the simulation_engine class, which contains all of the variables needed to construct a successful evolutionary simulation. A class has member variables and member functions that act on those variables. You declare an instance of a class in the same way you would declare any kind of variable, and an instance of a class is known as an object. The example below shows how to declare a class. However, the usage of a class is a huge part of this book and cannot be summarized concisely enough for this appendix. See Chapter 3 for a more detailed introduction.

```
#include <iostream>
#include <string>

class phone_book
{
private:
    std::string first_name;
    std::string last_name;
    std::string phone_number;

public:
    void set_name(std::string first, std::string last)
    {
        first_name = first;
        last_name = last;
    }

    void set_number(std::string number)
```
USEFUL C++ COMMANDS

```cpp
{
    phone_number = number;
}

void show_record()
{
    std::cout << last_name << ", " << first_name
              << ":\t" << phone_number;
}
};

int main()
{
    phone_book my_records[3];

    my_records[0].set_name("Charles", "Darwin");
    my_records[0].set_number("555-123-4567");
    my_records[1].set_name("Thomas", "Huxley");
    my_records[1].set_number("303-404-5055");
    my_records[2].set_name("Barbara", "McClintock");
    my_records[2].set_number("208-882-2288");

    for (size_t i = 0; i < 3; i++)
    {
        std::cout << "\n";
        my_records[i].show_record();
    }
}

const

This keyword indicates that the variable’s value cannot be changed after its initial declaration.

```cpp
#include <iostream>

int main()
{
    const double pi = 3.14159265;
    std::cout << pi;
}

```cpp
```
cout

Use this function with the insertion operator, <<, to output text to the screen. Almost anything can be output, including characters, strings, and numbers. The example below also shows the usage of cin.

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string your_name;
    std::cout << "Enter your name:\t" << your_name << "\n";
    std::cin >> your_name;
    std::cout << "\nHowdy, " << your_name << "!\n";
    std::cout << "What a great name!\n";
    std::cout << "I once had a dog named Big " << your_name << "!";
```
**APPENDIX 3**

}  

**delete**  
This command is used to free up the memory dynamically allocated using the `new` command. Any time you use `new`, it should be paired with `delete` somewhere else in your program to release the memory that was allocated. A failure to use `delete` will result in a memory leak, which will continue to fill up your system’s memory until your machine crashes. See the entry for `new` for more details.

**double**  
Declares a new variable of type double. A double is a floating-point number, that is, a real number with a decimal point. A double variable takes up 8 bytes of memory and can hold a value ranging from $1.7 \times 10^{-308}$ to $1.7 \times 10^{308}$ with 15 digits of precision. These values may differ for different compilers.

```cpp
#include <iostream>

int main()
{
    const double pi = 3.14159265;
    double radius = 6;
    double area;

    area = pi * radius * radius;

    std::cout << "A medium pizza has an area of ";
    std::cout << area << " square inches."
}
```

**for**  
This keyword specifies the start of a `for` loop. A `for` loop uses an iterator to conduct a specified number of repetitions.

```cpp
#include <iostream>

int main()
{
    int i;
    for (i = 0; i < 10; i++)
    {
        std::cout << "\nThe value of i is: \t" << i;
    }

    std::cout << "\nThe loop is over."
    std::cout << "\nThe value of i is: \t" << i;
}
```

**fstream**  
A family of classes used for input and output involving files on the hard drive. Objects of type `ofstream` are used to write files to the hard drive, and `ifstream` objects are used to read from the hard drive. The member function `open()`, which takes a filename as an argument, is used to open the file for reading or writing. The function `close()` is used to close the file when you are finished with it.

```cpp
#include <iostream>

#include <fstream>
```
int main()
{
    std::ofstream out_file;
    std::cout << "Writing to the file...\n";
    out_file.open("test.txt");
    out_file << "Cool text file contents!\n";
    out_file.close();
    std::cout << "Done!\n\n";

    std::cout << "Appending to the file...\n";
    out_file.open("test.txt", std::ofstream::app);
    out_file << "\nMore cool contents!\n";
    out_file.close();
    std::cout << "Done!\n\n";

    std::ifstream in_file;
    std::cout << "Reading from the file...\n";
    in_file.open("test.txt");
    std::cout << "File contents:\n\n";
    char next_char;
    while (!in_file.eof())
    {
        in_file.get(next_char);
        std::cout << next_char;
    }
    in_file.close();

    std::cout << "\n\nDone!";
}

function
A function is a self-contained snippet of code that performs a specified task. Many of the classes built into C++ have functions that go with them. You can also design your own custom functions. A function can accept any number of arguments, but only returns one value. However, there are other ways to get information into and out of a function. You can pass variables by reference or using pointers. To learn more about those topics, see the text or search the internet.

#include <iostream>

double cubed(double number)
{
    double result;
    result = number*number*number;
    return result;
}

int main()
{
    double my_number;
    std::cout << "Enter a number:";
    std::cin >> my_number;

    std::cout << "\n\nYour number cubed:\n\n"
    std::cout << cubed(my_number);
}
if-else  One of the most important commands in any programming language, the if statement evaluates an expression and only executes the associated code if the expression evaluates to true. Use the else statement to specify what happens when the expression evaluates to false.

```cpp
#include <iostream>

int main()
{
    double my_number;
    std::cout << "Enter a number:";
    std::cin >> my_number;

    if (my_number == 3)
    {
        std::cout << "Your number is 3!";
    }
    else
    {
        std::cout << "Your number is not 3!";
    }
}
```

ifstream  See fstream.

int  Declares a new variable of the integer type. An int takes up 4 bytes of memory and must hold an integer, so no decimal point is allowed. The value of an int can typically range from -2147483648 to 2147483647. Depending on the compiler, exceeding the value can either generate an error or roll over to the other end of the range. The latter type of error would cause serious, potentially undetectable problems in a program.

```cpp
#include <iostream>

int main()
{
    int i;
    i = 2147483646;

    std::cout << i << 
    std::cout << i+1 << 
    std::cout << i+2 << 
}
```

new  Use new to dynamically allocate memory to which a pointer points. When memory is allocated using new, the memory comes from the free store, whose size is limited only by the physical memory in your computer. Thus, large arrays, or even small arrays of large objects, should always be on the free store. Any time you use new, you must also use delete when you are finished with your variables or objects to release the memory.

```cpp
#include <iostream>

int main()
```
USEFUL C++ COMMANDS

```cpp
{
    int count = 10;

    double *number_list = new double[count];

    for (size_t i = 0; i < count; i++)
    {
        number_list[i] = (static_cast<double>(i) + 0.03)*0.95;
    }

    std::cout << "Number list:\n";
    for (size_t i = 0; i < count; i++)
        std::cout << number_list[i] << "\n";

    delete[] number_list;
}
```

ofstream See `fstream`

pointer A pointer is a variable that contains the memory address of a variable or array of variables, and the pointer gives you direct access to the value of the variable. Pointers are conveniently used with the `new` and `delete` operators to allocate and manage variables on the free store. Pointers allow the dynamic allocation of memory, which permits a programmer to ensure that arrays are always precisely the size needed for the task at hand. See the entry for `new` or Chapter 6 for an example of how to use a pointer.

rand() The built-in C++ pseudorandom number generator. It is neither efficient nor good. The Microsoft compiler includes it automatically. For the GNU compiler, `rand()` is in “stdlib.h”. This random number generator returns an integer between 0 and `RAND_MAX`, a number specified by the compiler (use `std::cout << RAND_MAX` to see the value – for my version of Visual Studio, the value of `RAND_MAX` is only 32767). In our simulation, we use `rand()` only to provide a random seed for our commercial grade random number generator, the Mersenne Twister.

```cpp
#include <iostream>
#include <time.h>
#include <stdlib.h>

int main()
{
    srand(static_cast<int>(time(NULL)));
    std::cout << "Random Numbers:\n";
    for (size_t i = 0; i < 10; i++)
        std::cout << rand() << "\n";
}
```

reference A reference serves as an alias to a specific variable or object. In this book, we mainly use references to allow functions to access specific variables. Basically, the reference to a variable contains the memory address of the variable, so if we pass a variable by reference to a function, the function has the ability to manipulate the memory associated with the original variable.
#include <iostream>

void square_my_number(double &number)
{
    number = number*number;
}

int main()
{
    double my_number;

    std::cout << "Enter a Number:\t";
    std::cin >> my_number;
    std::cout << "\n\nOriginal Number: \t" << my_number;
    std::cout << "\nSquared Number: \t";
    square_my_number(my_number);
    std::cout << my_number << "\n";
}

return This statement is used at the end of a function to specify the value that the function will return. The value can be any type of variable, but a function can only return one value. See the entry for function to see an example of its use.

setprecision This function can be used to alter the precision of variables that are output using cout.

#include <iostream>
#include <iomanip>

int main()
{
    double pi = 3.14159265359;
    std::cout << "Default Precision: \t" << pi << "\n";
    std::cout << std::setprecision(3) << "Precision of 3: \t" << pi << "\n";
    std::cout << std::setprecision(10) << "Precision of 10: \t" << pi << "\n";
}

size_t A variable of an unsigned integer type, often used as an iterator. The size_t variable type has the feature that it can always represent the size of any object, which makes it a safe choice for the iterator in, for example, a for loop.

#include <iostream>

int main()
{
    size_t j;
    for (j = 0; j < 10; j++)
        std::cout << j << ", ";
}

srand() A built-in function (included automatically in the Microsoft C++ compiler or in the header "stdlib.h" in the GNU g++ compiler) to seed the built in random number generator. The srand() function accepts one argument, an integer that serves as the
random number generator seed. For a given seed, the random number generator will always produce an identical sequence of numbers. See \texttt{rand()} for an example.

\textbf{\texttt{static\_cast}} This function tells the compiler to convert a variable to a new type. For instance, it could be used to covert a \texttt{double} to an \texttt{int}. However, you are responsible for ensuring that no important information is lost during the conversion.

```cpp
#include <iostream>

int main()
{
    double dNum;
    int iNum;

    dNum = 1.322;
    iNum = \texttt{static\_cast<int>}(dNum);
    std::cout << dNum << " ~ " << iNum << \"\n\n";

    iNum = 5;
    dNum = \texttt{static\_cast<double>}(iNum);
    std::cout << iNum << " = " << dNum;
}
```

\textbf{\texttt{string}} The string class in C++ provides a convenient and powerful way of storing and manipulating strings (i.e., many alphanumeric characters strung together as a word or a sentence). We used only a small subset of the functions associated with strings in this book, so see the web for additional tips.

```cpp
#include <iostream>
#include <string>

int main()
{
    std::string test_string;
    test_string = "This is a string!";

    std::cout << "A string:\n" << test_string;
    std::cout << \"\nThe first four letters:\n";
    std::cout << test_string.substr(0, 4);
}
```

\textbf{\texttt{time()}} This function returns an object of \texttt{time\_t}, and also accepts a pointer to an object of \texttt{time\_t}. The \texttt{time\_t} object returned usually contains the number of seconds elapsed since 12:00 a.m. of January 1, 1970. If a pointer to a \texttt{time\_t} object is provided, then that \texttt{time\_t} object will be updated to contain the same value. For our purposes, these details do not matter. For an argument, we can use \texttt{NULL}, which tells \texttt{time()} not to use the argument. We can also use \texttt{static\_cast} (see above) to convert the \texttt{time\_t} object into an integer. In this book, we use \texttt{time()} to generate a unique seed for our random number generator each time our program runs.

```cpp
#include <iostream>
#include <time.h>
```
int main()
{
    std::cout << time(NULL);
    std::cout << " seconds elapsed since 1970.\n";
}

vector
A container from the standard template library (STL). A vector is a list similar to a one-dimensional array. The containers from the STL have a variety of functions implemented to facilitate manipulation of their contents. Memory allocation for STL containers is handled automatically, so there is no need to use new or delete with them and they will grow as necessary to accommodate additional elements added to them. A vector can be a list of any type of variable or object, so this container can be quite convenient for many purposes. Some key member functions for a vector include: push_back(), which adds an element to the end of the vector, increasing its size by one; size(), which reports the number of elements in the vector; and clear(), which removes all elements from a vector, reducing its size to 0. Individual elements of a vector can be accessed by using a numeral in square brackets (e.g., [0] or [3]), in the same way elements are accessed for an ordinary array. Containers from the STL come in different varieties and are suited to different tasks. They also have many additional convenient member functions in addition to the ones mentioned here. Many detailed descriptions of the STL are available on the internet.

#include <iostream>
#include <vector>
#include <algorithm> // for sort function

int main()
{
    std::vector<int> somenumbers;
    somenumbers.push_back(4);
    somenumbers.push_back(12);
    somenumbers.push_back(122);
    somenumbers.push_back(9);
    std::cout << "Our Vector of Integers:\n";
    for (size_t i = 0; i < somenumbers.size(); i++)
        std::cout << somenumbers[i] << "\n";
    std::cout << "\nThe third element of our vector:\n";
    std::cout << somenumbers[2] << "\n";
    std::sort(somenumbers.begin(), somenumbers.end());
    std::cout << "\nOur Vector of Integers, Sorted:\n";
    for (size_t i = 0; i < somenumbers.size(); i++)
        std::cout << somenumbers[i] << "\n";
}

while
This keyword specifies another type of loop, which will run until the expression in parentheses evaluates as false. The expression is evaluated before the loop ever runs, so if the expression is false to begin with, the code within the loop will never run. If you need a loop that always runs once before the expression is evaluated, consider using a do-while loop (use the internet to look up the syntax).
```cpp
#include <iostream>
#include <time.h>

int main()
{
    int roll_1, roll_2, counter;
    srand(static_cast<int>(time(NULL)));

    // How many dice rolls does it take to get // double sixes?
    roll_1 = 0;
    roll_2 = 0;
    counter = 0;

    while ((roll_1 + roll_2) != 12)
    {
        roll_1 = rand() % 6 + 1;
        roll_2 = rand() % 6 + 1;
        counter++;

        std::cout << "\nRoll " << counter << "::t;"
        std::cout << roll_1 << ", " << roll_2;
    }

    std::cout << "\n\nSuccess!";
    std::cout << "\nDouble sixes on the ";
    std::cout << counter << "th roll!\n";
}
```
# Index

<table>
<thead>
<tr>
<th>%</th>
<th>39</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>63</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>7</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>7</td>
</tr>
<tr>
<td>!=</td>
<td>74</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>75</td>
</tr>
<tr>
<td>\0</td>
<td>198</td>
</tr>
<tr>
<td>\n</td>
<td>10</td>
</tr>
<tr>
<td>\t</td>
<td>11</td>
</tr>
<tr>
<td>#pragma once</td>
<td>8</td>
</tr>
</tbody>
</table>

## A
- absolute fitness, 142
- absolute value, 149
- additive genetic effects, 44
- alleles, 44
- appending to a file, 111
- arguments, 9
- array, 36

## B
- BASH, 206
- bivariate Gaussian distribution, 94
- bivariate normal distribution, 11, 94
- bool, 46
- Boolean, 46

## C
- carrying capacity, 99
- character set, 5
- chmod, 198
- cin, 7
- class, 13, 31
- close(), 27
- command-line arguments, 199, 206
- comments, 8
- constructor, 15
- correlation, 105
- covariance, 104, 142
- cout, 7

## D
- delete, 47
- debug mode compiling, 193
- destructor, 15, 48
- directional selection, 92
- double, 9

## E
- eigenvalues, 138
- eigenvectors, 138
- else statement, 61
- environmental variation, 44
- extraction operator, 7

## F
- fabs(), 149
- Fisherian sexual selection, 223
- fitness, 142
- for loop, 21
- free store, 46
- fstream, 26, 110
- functions, 9

## G
- g++ compiler, 195
- G-matrix, 137
- Gaussian distribution, 92
- genrand(), 11
- global variables, 42
- GNU compiler collection, 195

## H
- Hardy-Weinberg Equilibrium, 19
- header file, 8
- hello world program, 6

## I
- if statement, 57
- ifstream, 26, 111
include statements, 6
insertion operator, 7
installing Visual Studio, 3
int, 7
iostream, 6
Linux, 195
main(), 7
Marsenne Twister, 8
mating system, 72
memory leak, 47
modulus operator, 39
monogamy, 165
mutation, 85
namespace, 27
natural selection, 92
natural selection model, 19
new, 47
not, 63
null-terminated char array, 198
numerical analysis, 19
objects, 31
ofstream, 26, 111
open(), 27
overdominance, 29
parameters versus variables, 14
passing by reference, 12, 36
pointers, 46
population regulation, 99
private:, 13
public:, 13
push_back(), 177
rand(), 16
randbivnorm(), 11
randnorm(), 60
randnum(), 87
random number generator, 8, 16
random number seed, 8, 16
relative fitness, 142
release mode compiling, 193
return, 7
saving to a file, 26-29
scope, 42
selection surface, 94
setprecision, 25, 66
sex-limited selection, 126
sexual selection, 114
shell, 195
shell scripting, 206
single-locus selection model, 19
size_t, 38
sgenrand(), 16
sqrt(), 71
srand(), 16
stabilizing selection, 92
stack, the, 46
standard template library, 173
static_cast(), 16
std::, 27
stringstream, 196
time(), 16
variables, 14
variance, 104
vector, 173
Visual Studio, 3
void, 11
while, 75
C++ for Biologists: Evolutionary Models

This tutorial-based book provides an introduction to the development and analysis of agent-based evolutionary models using the C++ programming language. No prior programming skills are required, so this book is aimed at biologists interested in adding sophisticated computer models to their research.

Dr. Adam G. Jones is a Professor of Biological Sciences at the University of Idaho, where his work specializes in evolutionary biology, genomics, bioinformatics and computational biology.

```cpp
  }
}

for (j = 0; j < iNmeans; j++)

// Calculate the standard deviations of the means
for (i = 0; i < number_of_replicates; i++)
{
  for (j = 0; j < iNmeans; j++)
  {
  
  std dev of means[j] = std dev of means[j] +
    (mean_array[i].mean_list[j] - mean_of_means[j]) +
    (mean_array[i].mean_list[j] - mean_of_means[j]);
  }
}

for (j = 0; j < iNmeans; j++)
{
  std dev of means[j] = std dev of means[j] / dNreps;
  std dev of means[j] = sqrt(std dev of means[j]);
}
```