Chapter 1: Introduction

This chapter will introduce you to some basic concepts and should make you familiar with your programming environment. The goal of this chapter is to set everything up so that you will be ready to start working on *Baby-Bayes* from the next chapter on. Thus, here I will provide you with some background information and recommendations before you actually start programming.

Section 1.1: Getting Started with Writing a Software

Subsection 1.1.1: The academic software lifecycle

Writing a software in academia is slightly different from writing a commercial software. Academic software is more experimental and the algorithms and designs used are often not clear from the start. This makes it inherently difficult to design and plan well from the beginning. Furthermore, the algorithms needed might be entirely new or only new to the developer, so expertise might be limited. There is a reason why we call it *research*.

Software engineers have developed different paradigms of developing a software, which include, amongst others, the Waterfall model, iterative development and prototyping. My own impression is the many biologists are afraid that a software needs to be planed exactly from the beginning, as would be done in the *waterfall model*. From my own experience this is rarely the case and often does not lead to a successful software. The major problem is that we do not know now what algorithm will work, which data and models we will use in 5 or 10 years from now. A good software should be reusable also for the future, but it is impossible to plan ahead precisely.

You should do your best to have an outline for the software that you want to write; which tasks it will perform; which methods, data structures and algorithms are needed. A lot of computational biologists that I know write first a prototype in a programming language that will produce a result quickly, for example in *R* or *python*, and afterwards implement an efficient and clean version in *c* or *c++*. Writing a prototype can also be useful if you work on single feature that will be added later to a larger software. The additional

advantage of prototypes are that they can be used to test and validate your implementation; it's simply unlikely that you made the same implementation mistake twice, although mistakes in the model can not be caught. In the end, software development in academia is rather an *iterative process* with continuous additions and refinement over the years to come. Don't be afraid of not foreseeing the potential use cases of your software, just try not to program yourself into a corner.

Our software *RevBayes* originated because we could not extend *MrBayes* to the new types of models and data structures needed (mostly models for time trees). *MrBayes* was simply not flexible enough to accommodate the huge variety of phylogenetic models although it is still one of the most important software in evolutionary biology and marks a cornerstone in this research area. In the present book we have the slightly artificial situation that I already know exactly how the software *BabyBayes* is going look like: I have the unusual advantage of having written a very similar software before.

The incremental nature of academic software is reflected in the versioning process (although the same can be said for commercial software). You will often find that the third position of a version is updated for bug fixes (from version 1.2.0 to 1.2.1), the second position is updated when additional features are available (going from version 1.2.1 to 1.3.0), and the first position is only changed for major, often structural changes of the software. Note that these are only guidelines some people follow and other do not.

What I find important is that the version and updates can be followed using a version control system. A version control system, such as *git*, helps you to backup your code, create new release versions, and work on your code together with collaborators. I will talk more about *git* repositories below.

Subsection 1.1.2: Choosing a programming language

Choosing the right programming language for a specific task is not as easy as it sounds. You will notice this predicament when you ask different computer scientists or computational biologist: they will give you different answers on what they think is the best programming language for your problem. There are several programming languages around that can be recommended, for example, c++, Java, python, perl and R. All of these have their advantages

and disadvantages. In most situations, you should or will pick the one that (a) is familiar to you, and (b) most people in your research community are using. Only if there are good reasons for not sticking with a programming language that you already know, then you should learn another programming language. Furthermore, once you know a few programming languages it will be easy for you to learn a new one. Basically all programming languages follow a similar set of rules and structures.

There are a few ways how to classify programming languages. Some programming languages, such as *R*, *python* and *pearl*, are interpreted. That means, another software, the *interpreter*, interacts with you and translates your commands to actually perform some tasks. This interpretation or translation happens in *real-time*. Each command is interpreted after the previous one and you can interact with the computer program, for example, to query to value of a variable.

Other programming languages, such as *c*, *c++* and *Java*, are compiled programming languages. That means, that your source code will be translated by the *compiler* into machine code. Each CPU (the Central Processing Unit in your computer) works with a vendor (e.g., Intel or AMD) specific machine code language which is called the *assembly* programming language. The compiler translates code written in your higher level programming language into assembly code. On your computer you will find most software in assembly code. If you would open the software in a text editor you would not be able to read and understand the code. So you are lucky that today you do not need to write the commands in assembly anymore. Common programming language are high level languages and are translated into machine readable code using the the compiler. The important piece for you is that you need to learn how to use the compiler (more below) and this short paragraph tried to explain you why.

There are other classifications of programming languages. For example, *Java* and *c++* are object oriented programming languages whereas other programming languages are *functional* programming languages. The specific details are beyond the scope of this introductory book.

My own classification between these programming languages is: - **c**: Was used extensively before but should be replaced by c++ for larger software projects because of the object oriented programming design. - **c++**: One of the fastest and most powerful programming language. The big problem

with the powerfulness is the complexity which is a burden for novice programmers. - **Java**: Similar to c++ but much more restrictive which helps novice programmers to learn fundamental programming design principles such a objected oriented programming. - **python**: An increasingly popular programming language because of its easy-of-use but not as efficient as c and c++. - **perl**: Similar to python but with more complexity. In perl, there are thousands of ways to do the same task whereas in python there often is only one way. Having fewer choices can make the code easier to read because most developers will follow the same programming style. - **R**: Very useful and powerful for statistical analysis and plotting, but not design for larger and complex types of analysis.

In this book we will use c++ as the programming language of choice. This is clearly **not** because I think that c++ is the perfect programming language to start learning programming. Rather, my choice is made out of necessity that we developed *MrBayes* and *RevBayes* in c++. I hope that the extra complexity of c++ will not be a burden for this book.

Subsection 1.1.3: Setting up a code repository

Version control systems have a long history in software development. Early methods include *CVS* and *SVN* and today *git* is very popular. The general idea is that there is some general repository for your source code. On your computer, you have a *local copy* of the files in the repository. When you make any changes to these files, you should commit these changes to the global repository so that they become available to all your collaborators.

It is a very good habit to use repositories for all your projects. I even do this for projects that I haven't shared with anyone. The primary reasons are that (a) the code is securely stored in the cloud and cannot be deleted by accident, (b) I can always find the most recent version of the project files, and (c) I can trace back the history of my files, for example, if I need to revert back to an earlier version.

Today, a lot of open source software projects are hosted on GitHub. As an academic, you can get a free account which even includes private repositories. So you can create a repository for your code in this class without making it public. I recommend that you create such a repository now. You can name

it whatever you want, but it is recommendable to give it a name which you associate with this work. During one of our workshops, a student suggested we can have a special version of *RevBayes* call *SeBayes*. I'm not sure if the intended software would only work on *sebas*, but perhaps you get an idea of how to name your project.

If you don't want to use GitHub, there are other options such us Bitbucket and often local options at your university. There should really be no excuse to not using a code repository.

Subsection 1.1.4: Setting up your programming environment

Once you have created your repository for this project, it's time to fill it with the first files. A common layout is to have a directory called **src** within the project. The **src** directory will hold all your source code files. Let's conform with this standard and create the directory.

But before we go into writing your first program, we need to set-up your programming environment. Source code is *always* written in plain text files. That means, you can work on the source code using any text editor, for example - NotePad++ - Atom - Sublime - Textpad There are also more specialized programs for software development. These are usually called Integrative Development Environments (IDE). Examples of these are - XCode for Mac OS - Microsoft Visual Studio for Windows - Eclipse for all operating systems - Emacs If you have one of these programs installed, or want to install them, it might be worth to learn how to use it. The major advantage is that they include a compiler and make building your software much easier. They also include a *debugger* which comes in very handy when you are testing your code. Finally, most IDEs provide *syntax highlighting* and *code completion* and other nice features that can help you develop your software. Since there are several different IDEs available and each has their own ways of doing things, I will not go into their specific details here.

On Linux or Mac OS, make sure that you have the c++ compiler *gcc* installed. On Mac OS, you will get *gcc* if you install and run *XCode*, or install the *Command Line Tools*. Most Linux distributions have *gcc* automatically installed.

To compile *BabyBayes* in Windows is not so straightforward. You can use

Microsoft Visual Studio, which is a powerful IDE for Windows. Alternatively, you can compile *BabyBayes* using MinGW. You need to download the installer "mingw-get-setup.exe" and install it. Then open "MinGW Installation Manager" you just installed. Click the square before "mingw32-base" under "Basic Setup" and choose "Mark for Installation", then choose "Apply Changes" from the "Installation" menu. Additionally, add "C:\MinGW\bin" (path to your MinGW installation) to your system path in setting "Environment Variables".

Please take your time now and make sure you have all the software needed.

Section 1.2: Your first computer program

Subsection 1.2.1: The "Hello World!" program

Now we will write our very first computer program. As is tradition, we will write a very simple *Hello World* program. Almost every programming book starts with this exercise, so I'll do the same.

Open a new text file and call it *main.cpp*. The extension *.cpp* is chosen by convention so that everyone sees this is a c++ source code file. Write the following few lines into the file and save it.

```
#include <iostream>

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

    std::cout << "Hello World!\n";

    return 0;
}
```

This is all you need to do for your very first program. I'll explain each line below but I know that you cannot wait to actually run this amazing program. So we need to compile the program before you can run it.

Subsection 1.2.2: Writing a *Makefile* to compile your code

If you are using an IDE, you might be able to click a button to compile and run the code. That would work. However, many programs, especially the ones that should run on a computer cluster or are distributed, are compiled using a *Makefile*. A *Makefile* is script that is run by the program called *make*. There are very many different ways to set up your *Makefile*. Here I'll just provide one way of how to do this, but you can find much more information on the web.

Create a new file called *Makefile*. This file should have **no extension** and should really be called only *Makefile*. As before, the file should be in your **src** directory. Now write the following content into the file.

```
CC           = g++
CPPFLAGS     = -O3

SRC       = main.cpp
OBJECTS   = main.o
PROGS     = BabyBayes

all: $(PROGS)

BabyBayes: $(OBJECTS)
    $(CC) $(CPPFLAGS) -o $@ $^

clean:
    rm -f *.o *~ $(PROGS)
```

The first two lines specify some variables used for compiling. The *g++* option means that we will use the *gcc* compiler and specifically it's c++ version. If you want to use a different compiler, then please change it to the name of your c++ compiler.

The second option, the *-O3*, specifies that the code will be compiled according to the third level optimization. This is the fastest and most efficient optimization. As a beginner, you should not worry about this and just keep it as is.

The next three lines specify your source code (main.cpp), the name of the

translated file (by convention, you replace the .cpp with .o for each file), and the name of your program (*BabyBayes*). When you will execute this file, it will by default start with the **all** section. In our **all** section, we define that we should perform the commands specified in the **$(PORGS)** section. Remember that we just defined *PROGS* to be *BabyBayes*, so this means that we will perform the commands in the **BabyBayes** section. This very cryptic line in the **BabyBayes** section performs the actual compilation. If you wanted to compile your *Hello World* program without a *Makefile*, you could have used

```
g++ -o MyProgram main.cpp
```

However, instead of writing this line every time you are compiling your software, it is easier to store the command in the *Makefile*. It is also easier to tell someone else to simply run *make* instead of providing a horribly long and complex compilation command. For example, imagine that a software like *RevBayes* has more than a thousand files, so the list of source files and inputs to *g++* becomes very, very large. Also, the order of the files is important, as you will learn later.

Subsection 1.2.3: Explaining the *Hello World* program

In our little *Hello World* program, we started with the line

```
#include <iostream>
```

So what does this **#include** do? The **#include** directive tells the compiler to load other files or libraries. By convention, we use for loading libraries and *<another_file.h>* for loading other source files. So in this case, we load the *iostream* library that belongs to the c++ standard. As you might have guessed, the *iostream* library defines the input/output functions.

The next text line in our program was

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

This line simple specifies that now the *main* function begins. First, it says that the return value of the function is of type **int**. Then, it says that the name of the function is **main**. Next, it says that there are two arguments, the first is called *argc* and is of type **int**. The second argument is called **argv**,

the values of the arguments, and is of type **int***[]. This is quite a lot for the beginning, and will be explained more later and hopefully make sense then. Don't worry too much about these details yet.

You might have noticed the curly brace { at the end of the line. By definition, a function, such as the *main* function, is followed by a command. However, you often want to perform more than one command within a function. The solution is to write a *block* of commands. A *block* is always surround by an opening and closing pair of curly braces { and }.

The next command it the only actual command of our program.

```
    std::cout << "Hello World!\n";
```

The *std::cout* says that we use the standard output, i.e., the terminal output, to print something. The prefix *std* specifies that *cout* lives in the namespace *std*. You will come across many functions from the *std* namespace over the next weeks.

The right hand part of the command says that the text, which is called *string* in computer science jargon, is printed. Note that the direction of the << signals the direction; the text goes into the standard output *cout*.

Finally, we finish our program by return the number 0 and closing the command block with the closing curly brace }. By convention, if the program returns a 0, then the program quit successful. Otherwise, you can return different numbers to signal different errors during execution.

```
    return 0;
}
```

Subsection 1.2.4: Modifying the *Hello World* program

As a little exercise, and preparation for our *BabyBayes* software, let's change the *Hello World* text to print the information of your program. Change the commands to something like the following code.

```
    std::cout << "BabyBayes v1.0\n";
    std::cout << "Sebastian Höhna\n";
```

```
std::cout << "Ludwig-Maximilians-Universität München\n";
std::cout << "\n\n";
```

Now compile and run this new code. It still might look lame and boring, but getting this first step working is very crucial. If you succeeded, give yourself a little applause and a cup of coffee or tee; you've deserved it!

Subsection 1.2.5: The *main* function

The *main* function is the most important in every compiled software. The software *always* starts with the main function, and in programming languages you like **c**, **c++** and **Java** you always must have exactly one main function in your software.

This convention is very important for software development. When you execute a software, it will always start at the main function. How else would the software know where to start? Hence, when you execute a software, you are in fact only executing the main function of that software. What is happening inside the main function depends on the specific software and can be quite complex, most often including execution of other functions.

Subsection 1.2.6 :Functions take arguments and return values

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

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

}
```

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

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

but here is another try: *char* argv[]* is a vector of memory addresses, each of which indicates the starting address for a string. (Don't worry if this still doesn't make sense.)

What is contained in *int argc* and *char* argv[]*? *int argc* holds the number of strings (i.e., words) contained in *argv*. Create a new file called **main2.cpp** as a copy of your first **main.cpp**. Rewrite your *main* function to look like this:

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

   std::cout << "argc = " << argc << std::endl;
   for (int i=0; i<argc; i++) {
      std::cout << "argv[" << i << "] = " << argv[i] << std::endl;
   }

   return 0;
}
```

Now, compile the program using

```
g++ -o MyProgram main.cpp
```

and run it with

```
./MyProgram2
```

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

```
argc = 1
argv[0] = ./MyProgram2
```

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

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

```
./MyProgram2 David Swofford had a little lamb which was named, sensibly enough, PAUL
```

This line produces the following output on my computer:

```
argc = 13
```

```
argv[0] = ./MyProgram2
argv[1] = David
argv[2] = Swofford
argv[3] = had
argv[4] = a
argv[5] = little
argv[6] = lamb
argv[7] = which
argv[8] = was
argv[9] = named,
argv[10] = sensibly
argv[11] = enough,
argv[12] = PAUP
```

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

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

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

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

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

Section 1.3: Variables

Subsection 1.3.1 The basic variable types

Copy the **main.cpp** file and modify the *main* function so that it reads:

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

   int x = 3;
   std::cout << "x = " << x << std::endl;

   return 0;
}
```

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

```
x = 3
```

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

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

Most programmers, when declaring a variable, will also initialize it to some value. Normally, if I were declaring a variable like x, I would initialize its value to zero, *int* $x = 0$, so that I could rely on it being zero. Some compilers, when you declare a variable without initializing it (e.g., *int* x), initialize the value to zero. Other compilers, however, do not do this and the value of the variable will reflect the pattern of bits that happen to be at that memory address. Declaring and initializing the variable ensures that it has a value that you can rely on. Note that I added comments to the two lines, above; the words after the // are the comment and are not read by the compiler.

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

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

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

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

   return 0;
}
```

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

```
x = 3
y = 3.14
```

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

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

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

Subsection 1.3.2: Variables take up space

When you declare a variable, such as *int* $x = 0$, the operating system sets aside enough space on your computer's memory to represent the variable. Interestingly, c++ allows you to see how much space is set aside and even where the variable resides in memory. Now let's rewrite *main* to read:

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

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

   return 0;
}
```

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

```
x's value   = 0
x's address = 0x7ffeefbff57c
x's size    = 4
```

The first line of code, *int x = 0*, simply declares and initializes an integer variable called x. The next line of code should also make sense to you; it simply prints out the value of x, which because you initialized the variable at the same time that you declared it, is predictably zero. It's on the next line of code which prints out the memory address of x, *std::cout << "x's address =" << &x << std::endl*, where things get interesting. You can get the memory address of a variable by putting the ampersand symbol, '&,' in front of the variable's name. So, *&x* represents the memory address of the variable x. You can see that when I ran the program on my computer, the memory address is reported to be *0x7ffeefbff57c*.

The memory address deserves explanation. First of all, the memory on your computer is arrayed in bytes, each of which has an address. To understand this a bit more, imagine a city that consists only of a single street. Every house in the street gets a house number (i.e., an address) which are nice and tidy in an increasing order. You can then address the inhabitants of the city by their address (the house number). This is exactly the way how the memory in your computer is order and addressed by computer programs.

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

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

Variable	Number Bytes
int	4
unsigned int	4
float	4
double	8
char	1
bool	1

Subsection 1.3.3: Variables in computers have limits

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

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

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

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

Imagine we had only two digits to represent a base-10 number. What is the largest value that can be represented with two digits? The answer: 99 (or $10^2 - 1$). Similarly, the largest binary number that can be represented with 32 bits is *11111111111111111111111111111111*, or $2^{32} - 1$. For the *int* variable type, however, one of the 32 bits is used to indicate whether the number is positive or negative. Therefore, the largest value that can be stored using an *int* variable is $10^{31} - 1 = 2147483647$ whereas the smallest value is -2147483648. The actual story is a bit more complicated than I just made out, but you see the point: if we wanted to store a number larger than about 2.1 billion, we are going to run into problems. In fact, if we attempt to do so, we will get an overflow error from the computer. (The opposite problem — attempting to hold the value of a number that is too small — is called an 'underflow error.')

Note that you can get information on the limits of numerical representation from functions defined in the *limits* include file.

Similarly, we cannot represent the real numbers with complete accuracy. Try the following experiment; rewrite your project code so that it reads:

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

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

   float x = 0.1;
```

```
    double y = 0.1;

    std::cout << std::fixed << std::setprecision(50) << "x = " << x << std::endl;
    std::cout << std::fixed << std::setprecision(50) << "y = " << y << std::endl;

    return 0;
}
```

The output should look like this:

```
x = 0.10000000149011611938476562500000000000000000000000
y = 0.10000000000000000555111512312578270211815834045410
```

Neither number is exactly equal to 0.1, though both are quite close to that value. The reason is straight-forward: computers cannot represent every real number with complete precision, but rather approximates any particular number as well as it can. The representation of 0.1 is better when we use a *double* variable than when we use the *float* type. This makes sense because the internal representation of the real number for the *double* type uses twice as many bytes as the *float* type.

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

Subsection 1.3.4: You can use a variable to hold a memory address

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

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

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

```
#include <iostream>

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

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

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

   return 0;
}
```

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

```
x = 0
&x = 0x7ffeefbff56c
xPtr = 0x7ffeefbff56c
&xPtr = 0x7ffeefbff560
int size = 4
int* size = 8
```

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

Confusingly, different programmers will put the asterisk, which indicates that

the variable will hold a memory address, in different places. Compilers will accept the following as equivalent:

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

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

Two of the lines display the same memory address:

```
&x = 0x7ffeefbff56c
xPtr = 0x7ffeefbff56c
```

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

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

```
int** anotherDamnPointer = &xPtr;
```

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

Subsection 1.3.5: Dereferencing pointers

I mentioned that if you know the address of a variable, that you can manipulate it. You can do this by 'dereferencing' the pointer. Here's an example

using a re-written *main* function:

```
#include <iostream>

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

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

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

   return 0;
}
```

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

```
x = 0
x = 3
```

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