Learning to Program Using D

Jesse Phillips

September 21, 2012

Contents

Chapter 1

Getting Started

1.1 Audience

The goal of this book is to provide the knowledge needed for one to become a good programmer and to show that D provides everything needed to go from a first time "Python" programmer to basic "C" programmer, so no programming experience is expected.

The book is intended to utilize most aspects of the D programming language and the standard library, Phobos. Most importantly it will use these to instill proper technique to both the reader and myself. However, this can make examples appear much longer and with greater complexity then needed to achieve the task, especially early on when the reader is unfamiliar with all things programming. I hope to be able to step into proper form quickly and cover the important points.

Examples are a huge part of this book. For those with background in another language these examples will be good to skim through as I'll be introducing constructs as they become useful and will not be providing entire sections to specific features, such as unit-tests and contracts. An index will of course be useful, once I decide to figure out how to do that.

History, comparisons, features, debates, and other language topics are the subject of this book. Teaching programming is the objective and that will require many of these, for example when discussing pointers it may be appropriate to point out traps that C has and D prevents.

1.2 Processing

A program is quite simply a list of instructions. These instructions may result in complex operations, but is nothing more than doing what is asked.

1. Go grocery shopping

The list above is quite short, but still provides an instruction for what should be done.

1. Enter Car
2. Drive to Fred Myers
3. Collect Milk
4. Collect Eggs
5. Collect Chicken Wings
6. Purchase Items
7. Drive Home

This list provides more detail on what is to be done. It names specific items that need to be purchased and the actions for getting to and from the store. Yet we can still go into even more detail on how to go grocery shopping.

1. Lift right leg
2. Move right leg in forward position
3. Place right leg down
4. Press off with left leg
5. Lift left leg
6. ...

This would take much longer to provide all the needed actions to make the trip to the store. Each one of these could easily be the construction of the previous.

1. Go grocery shopping
    (a) Enter Car
        i. Lift right leg
        ii. Move right leg in forward position
        iii. Place right leg down
        iv. Press of with left leg
        v. Lift left leg
        vi. ...
    (b) Drive to Fred Myers
    (c) Collect Milk

(d) Collect Eggs
(e) Collect Chicken Wings
(f) Purchase Items
(g) Drive Home

And these are exactly the kinds of things we want to build when making a program. A computer requires very specific instructions for everything it does; as we build our programs we want to make these instructions reusable.

1.3 Types in Programming

In the language of this book we have what is known as a *static type system* as opposed to a *dynamic type system*. A *type* is there to be sure you don't misrepresent something. A common toy used as a baby is one where you must place objects in their proper hole. Some holes are square, some rectangular, and other triangle. The objective is to select the right *type* of hole for the type of object, and you are prevented from mixing the types. The type system in a programming language has the same purpose as those holes in the toy.

Consider the representation the computer uses for everything. All of its information is held as an electrical current that is either there or it is not. In English we use a combination of twenty-six letters to construct ideas, other languages may have more or less. A computer only has the two and just like we can take words out of context and get a different meaning the same thing can happen to a computer.

It is generally easiest for humans to think of these patterns of ones and zeros, known as bits, as numbers especially since mathematical operations are primitives of the CPU. Consider a single character which is represented as 'a' in code. This as equivalent to the integer, *int*, representation of 97.

Listing 1-1: Type of 'a'

```
1 void main() {
2     static assert('a' == 97);
3 }
```

```
$ dmd example1.d
$ ./example1
```

In the example above I have introduced a language feature and the equality operator. *static assert* will be used throughout this book to demonstrate properties that are true. It takes what is known as an expression which evaluates to a boolean, keywordbool. A boolean being a type which hold either *true* or *false*. My expression is inside the parenthesis and uses equality, ==, which is different from assignment, =. The program will compile and result in no output when run. However the program fail to compile if you were to change either the letter used or the number.

Statements are a complete requested operation and usually are ended with a semicolon. In the code demonstrated *static assert* is the statement and **'a' == 97** is the expression. Expressions are comprised of different operators which result in need to evaluate the operations and would have no meaning if their result was not used.

The simplest operators to understand are those used in math. As you can see from the example below, basic math is quit familiar.

Listing 1-2: Integer Type

```
1 void main() {
2     static assert(2 + 2 == 4);
3     static assert(5 - 3 == 2);
4     static assert(21 * 2 == 42);
5     static assert(5 / 2 == 2);
6 }
```

```
$ dmd example2.d
$ ./example2
```

Take a closer look at the last statement with division. As mentioned earlier, the computer uses types to guide the interpretation of what the underlining bits mean. In this case we are working with the type *int*. Integers are only whole numbers and are unable to represent fractions of a whole. So when an operation is performed on them the result is truncated; it is not floored, brought to the lowest whole number, or rounded, brought to the nearest whole number. This can be important to remember when dealing with negative numbers as -3.6 is truncated to -3, floored to -4, and rounded to -4.

Instead the CPU has a different representation for what is known as a floating point, and has a number of types: *float*, *double*, *real*. Each of these provide a different amount of accuracy by increasing the bits used.

1.3.1 Floating Point

A float will consume 32 bits, a double 64, and a real is either 64 or 80 depending on the maximum representable by the CPU. But these details can be different or irrelevant depending on the language. For example Java only has double. Now let us see how we can use these in the language.

Listing 1-3: Floating Point Types

```
1 void main() {
2     static assert(5.0 / 2 == 2.5);
3     static assert(5.0 / 2 != 2);
4     static assert(5F / 2 == 2.5);
5 }
```

```
$ dmd example3.d
$ ./example3
```

In the new operator != is equivalent in meaning to ≠ in mathematics; the exclamation mark is read as "not." This shows when using a floating point number the result will not equal two as it does when using integers. The final assert show how to use a suffix to specify a float literal. The term *literal* refers to a hard coded element in the code. But to understand what I mean by that I must introduce variables.

There is one important note. This code is incorrect. At this time it is not important and a solution will be introduced later, but for now remember that checking equality of a floating point is usually not desired. When performing operations the result will be dictated by how accurate the number is represented. In more complicated operations instead of returning 2.5 the number could be 2.4999999 (where this is still less accurate than what might be returned) resulting in equality being false.

1.4 Precedence

The operator precedence is what defines the *order of evaluation* for an expression. What you know from math will be of great help to knowing basic ordering. Many operators exist which are not part of math.

Listing 1-4: Precedence

```
1 void main() {
2     static assert(3 + 2 * 3 == 9);
3     static assert((3 + 2) * 3 == 15);
4 }
```

```
$ dmd example4.d
$ ./example4
```

For now it is not important to cover all the operators and where they lay in terms of order of evaluation. Be aware that it exists and the parentheses can be used to to group evaluation if you are unsure how it will be evaluated.

1.5 Commenting

One of the key aspects to writing a program is communicating. The obvious communication is happening between the human and the computer. There is also communication with other programmers. However, we don't actually care about them, usually a programmer just needs to write some instructions down and will have little need to go over the source again once the program has been completed. The slightly hidden underpinning is that a program is never complete. The "other programmers" truly become the original author. Code

you write today, will be code you look at tomorrow. As you progress and desire to create new works, the old will always be guiding you.

Comments are valuable and should be used to exercise explanation rather than behavior. There are a couple types of documentation that will be written, documentation comments and code comments. Documentation comments are for explaining how to use and what it is used for. Code comments are intended to explain why something has been written.

There is a third type of comment, but is best handled by what is known as *self-documenting code*. Code should be written in a way that would make commenting on it redundant. This relies heavily on naming and structuring of the code, and as everything can be very subjective. This type of commenting does not replace the previous two already stated.

D provides a number of options for inserting comments into the code. The common comment style for C++, Java and many other languages.

```
/*
This Comment spans
multiple lines.
*/

// While this is only until the end of the line
```

The second example is a C style comment, though most C compilers will accept the first too.

```
/**
Documentation Comment
*/

/// Documentation Comment
```

Documentation comments are used to generate html pages that describe how to use code. These will be uncommon in this book, but are an important part of writing code.

```
/+
Nesting Comment
+/
```

Nesting comments are uncommon for languages, D provides them and also have a documentation comment form by adding another plus. The nesting is important due to the behavior of non-nesting comments.

```
/*
This is a Comment

/* I want to comment more */
```

```
but now I am not in a comment block

*/
```

Writing comments at the same time as writing code is difficult. It is recommended that an outline of the program be written in comments. Then fill in the code to fulfill each comment, leaving the comments. Commenting is something every programmer struggles with, it is hard to find the right information that is missing from the code. For this reason, my personal recommendation is to write comments while reading code too. If it takes some time to figure out what a piece of code is doing, write down what is found. Comments like "I don't get it" or "WTF?" are funny, yet the value for anyone is very limited.

1.5.1 Task Comments

Another use of comments in code is to mark a task which will need completed. Many IDEs and editors will recognise this hint keyord in a comment.

```
// TODO: Something to have done
```

However this is not a good way to keep a task list. One of the best uses is when writing the tasks within the function/file. As the tasks are implemented remove the TODO and leave it as a comment for the code. These can be good in small personal project or even minor things possibly related to code clean up. If it is between using this or not writing down the thought, use this.

Chapter 2

Variables, Functions, and Modules

As mentioned in Processing it is important to make reuse of the code you write. This is so important we'll get started on it right away with the three most common ways to do reuse.

2.1 Variables

With the basics of types we can start storing our values into variables.

Listing 2-1: Variable Introduction

```
1 void main() {
2     auto base = 4;
3     auto height = 3;
4
5     assert((base/2) * height == 6);
6 }
```

```
$ dmd example5.d
$ ./example5
```

This creates two variables *base* and *height* and initializes them. A language feature known as type inference through the use of *auto* allows the compiler to decide the type. The variables are used in a mathematical expression to calculate the area of a triangle, in this case it is six. This code could also be written by explicitly specifying the types of the variables.

Listing 2-2: Explicit Type Declaration

```
1 void main() {
2     int base = 4;
```

```
3     int height = 3;
4
5     assert((base/2) * height == 6);
6 }
```

```
$ dmd example6.d
$ ./example6
```

Do to the introduction of variables *static assert* is replaced by his runtime equivalent, *assert*. The static version means that the expression will be evaluated and checked during the compilation of the program. This is not possible with variables as the values are not know during the compilation. It is preferable to use *static assert* as errors will be found much sooner, but many times it will need to be checked during runtime.

Type inferences is extensively used throughout this book and in idiomatic D, so it is important to understand *literals* as they have types and will dictate the type of the variable during inference.

Listing 2-3: Integer Type

```
1 void main() {
2     auto base = 5;
3     auto height = 3;
4
5     assert((base/2) * height == 6);
6     assert((base/2) * height != 7.5);
7 }
```

```
$ dmd example7.d
$ ./example7
```

This example demonstrates the importance of knowing the types being stored. Rather than returning the correct answer we are faced with the same answer as before. This happens because the type of base is *int* and so the operation (base/2) performs what is commonly referred to as *integer division*. And the result is 2 instead of 2.5. Using the proper literals or specifying the required type would fix this issue. One of the goals for programming is to build the code to be reusable and one way to do this is to make a function and call to it.

2.2 Functions

A function is a container of executable code. Much like a book will refer different chapters for more details, a function is an encapsolation of operations that give a result.

Listing 2-4: Triangle Function

```
1 double triangleArea(double base, double height) {
2     return (base/2) * height;
3 }
4
5 void main() {
6     auto base = 5;
7     auto height = 3;
8
9     assert(triangleArea(base, height) == 7.5);
10 }
```

```
$ dmd example8.d
$ ./example8
```

In this case we have defined a function called triangleArea which accepts two values of type double and returns a value of type double. The *signature* of a function looks like this.

```
ReturnType functionName(Parameter list);
```

I have been avoiding a very important aspect to these examples. The *main* function. This is the entry point to the program. It follows the same guidelines as above, but the return type is *void* which indicates that nothing is returned. Its *parameter list* is also empty meaning nothing is to be passed to it.

2.2.1 Function Contracts

While building a program it is important to consider what the expected input will be. As the code will be written to handle the expected inputs it will be good practice to verify the parameters do not exceed those bounds.

As a piece of software is developed many parts of it are restructured, fixed, or improved in a number of ways. The two most common ways someone will test code that has been written is by compiling and running it. Once a feature has been added and tested using this method it becomes neglected as new features/behaviors are added. With this method of testing implementing a "Save as" feature could also affect "Save" causing it to perform the same operation. Since "Save as" is the new addition it gets the testing and approval of working.

While a large amount of effort goes into designing modularity to enable code reuse, this does have the drawback of changes making an impact in various areas. In some cases a feature could work because it relied on a bug that is later fixed, or purely out of a lack of full understanding on the programmers part. And this is why *regression testing* has become a key part of software development. A very popular form of regression tests is *unit testing*. In this case a "unit" is referring to an encapsulated operation of unspecified size where for a given input, an expected output should result. For now our units are functions.

Listing 2-5: Triangle Contracts

```
1 double triangleArea(double base, double height)
2 in {
3     assert(base > 0);
4     assert(height > 0);
5 } body {
6     return (base/2) * height;
7 }
8
9 unittest {
10    assert(triangleArea(5, 3) == 7.5);
11    assert(triangleArea(6, 10) == 30);
12 }
13
14 void main() {
15 }
```

```
$ dmd -unittest example9.d
$ ./example9
```

This book will continue to use these and other techniques to help demonstrate how a function operates and the manner in which to build tested code. The *in contract* is checking that the values passed into our function are positive numbers. The reason for this is because you really can't have a negative length so it would be very likely that the calling code has something wrong and needs correcting.

After the function there is a block named *unittest* which is not run with a normal compiler call. Please read your compilers documentation on how to enable Unit Tests. Once compiled with Unit Tests on, this code will be run prior to code in *main*. If one of the tests fail the program will terminate with an error massage. These blocks of code are able to have their own variables and usually consist of assertions that should always be true.

2.3 Modules

There is one more item to cover to create a usable program, instead of just a bunch of tests. Modules are similar to functions because they are another way to make reusable code. They are basically a collection of functions and variables that can be pulled into another module. So far the examples have all consisted of a single module. The module name is inferred by the file name, but it is recommended that one be specified in the programs you write, here is a modified version of the things we have learned.

Listing 2-6: Unittesting Triangle Area

```
1 module triangle;
```

```
2
3 double triangleArea(double base, double height)
4 in {
5     assert(base >= 0);
6     assert(height >= 0);
7 } body {
8     return (base/2) * height;
9 }
10
11 unittest {
12     assert(triangleArea(5, 3) == 7.5);
13     assert(triangleArea(6, 10) == 30);
14 }
15
16 void main() {
17     auto base = 5;
18     auto height = 3;
19
20     assert(triangleArea(base, height) == 7.5);
21 }
```

```
$ dmd -unittest example_triangle.d
$ ./example_triangle
```

The module provides a name space for our triangle function. What this means is that the full name is actually triangle.triangleArea. But a fully qualified name is not needed; *triangleArea* is in the current name space. Modules allow code to be organized into common collections that are used by other collections. Now main() can move into its own module.

Listing 2-7: Triangle Main Module

```
1 module main;
2
3 import triangle;
4
5 void main() {
6     auto base = 5;
7     auto height = 3;
8
9     assert(triangleArea(base, height) == 7.5);
10 }
```

Listing 2-8: Triangle Module

```
1 module triangle;
2
```

```
3  /**
4   * Calculates the area of a triangle given
5   * the length of the base and height.
6   *
7   * The base and height must be positive numbers
8   * greater than zero.
9   *
10  * Parameters:
11  *      base = the length of the triangle base
12  *      height = the height of the triangle
13  */
14 double triangleArea(double base, double height)
15 in {
16      assert(base >= 0);
17      assert(height >= 0);
18 } body {
19      return (base/2) * height;
20 }
21
22 unittest {
23      assert(triangleArea(5, 3) == 7.5);
24      assert(triangleArea(6, 10) == 30);
25 }
```

```
$ dmd -unittest example_main.d example_triangle.d
$ ./example_main
```

By using *import* the compiler makes the symbols (function, variables, etc.) of the specified module available. There can be multiple files which import the same module allowing for greater reuse of code. Notice that each file is being passed to the compiler. In my case our triangle module is in example_triangle.d and the main module is in example_main.d. The output program will be named after the first file that is passed, it is customary, but not required, that the file contain the main method.

The *triangleArea* function has also been given a *documentation comment*. While very basic these comments are very important to describing what a function is for and how it is used. The depth needed for a good comment will vary depending on the depth of what is being commented. In this case the description of the parameters is probably unnecessary detail as it is just a different representation of the summary. For the DMD compiler the html documentation can be output into a directory called doc by passing the flag *-Dddoc* during compilation.

2.3.1 Packages

2.4 Common Errors

When you begin working with multiple files there a few things that may get missed and it is important to know how your compiler informs you of these mistakes.

Forgetting the main function:

Listing 2-9: Undefined Reference to Start

```
1 double triangleArea(double base, double height) {
2     return (base/2) * height;
3 }
```

```
$ dmd example13.d
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
bos2.a(dmain2_45f_1a5.o): In function `_D2rt6dmain24main
UiPPaZi7runMainMFZv':
src/rt/dmain2.d:(.text._D2rt6dmain24mainUiPPaZi7runMainM
FZv+0x18): undefined reference to `_Dmain'
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
bos2.a(thread_192_1b8.o): In function `_D4core6thread6Th
read6__ctorMFZC4core6thread6Thread':
src/core/thread.d:(.text._D4core6thread6Thread6__ctorMFZ
C4core6thread6Thread+0x26): undefined reference to
`_tlsend'
src/core/thread.d:(.text._D4core6thread6Thread6__ctorMFZ
C4core6thread6Thread+0x31): undefined reference to
`_tlsstart'
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
bos2.a(thread_1a2_6e4.o): In function `thread_attachThis
':
src/core/thread.d:(.text.thread_attachThis+0xf4):
undefined reference to `_tlsstart'
src/core/thread.d:(.text.thread_attachThis+0xff):
undefined reference to `_tlsend'
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
bos2.a(thread_180_1b8.o): In function `_D4core6thread6Th
read6__ctorMFPFZvmZC4core6thread6Thread':
src/core/thread.d:(.text._D4core6thread6Thread6__ctorMFP
FZvmZC4core6thread6Thread+0x2b): undefined reference to
`_tlsend'
src/core/thread.d:(.text._D4core6thread6Thread6__ctorMFP
FZvmZC4core6thread6Thread+0x36): undefined reference to
`_tlsstart'
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
```

```
bos2.a(thread_181_1b8.o): In function `_D4core6thread6Th
read6__ctorMFDFZvmZC4core6thread6Thread':
src/core/thread.d:(.text._D4core6thread6Thread6__ctorMFD
FZvmZC4core6thread6Thread+0x37): undefined reference to
`_tlsend'
src/core/thread.d:(.text._D4core6thread6Thread6__ctorMFD
FZvmZC4core6thread6Thread+0x42): undefined reference to
`_tlsstart'
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
bos2.a(deh2_441_525.o): In function `_D2rt4deh213__eh_fi
nddataFPvZPS2rt4deh29FuncTable':
src/rt/deh2.d:(.text._D2rt4deh213__eh_finddataFPvZPS2rt4
deh29FuncTable+0xa): undefined reference to `_deh_beg'
src/rt/deh2.d:(.text._D2rt4deh213__eh_finddataFPvZPS2rt4
deh29FuncTable+0x14): undefined reference to `_deh_beg'
src/rt/deh2.d:(.text._D2rt4deh213__eh_finddataFPvZPS2rt4
deh29FuncTable+0x1e): undefined reference to `_deh_end'
src/rt/deh2.d:(.text._D2rt4deh213__eh_finddataFPvZPS2rt4
deh29FuncTable+0x45): undefined reference to `_deh_end'
/usr/lib/gcc/x86_64-linux-gnu/4.7/../../../../lib/libpho
bos2.a(thread_17d_713.o): In function `thread_entryPoint
':
src/core/thread.d:(.text.thread_entryPoint+0xa1):
undefined reference to `_tlsend'
src/core/thread.d:(.text.thread_entryPoint+0xac):
undefined reference to `_tlsstart'
collect2: error: ld returned 1 exit status
--- errorlevel 1
```

The linker barfs a lot of information on this one, but you'll notice that one of the last things said is, "undefined reference to '_tlsstart.' And one of the first things is "undefined reference to '_Dmain.' These exact message my differ depending on the linker used, but the idea is that there is no starting location for your program (the main function).

Forgetting the import statement:

Listing 2-10: Undefined Identifier

```
1 void main() {
2     auto base = 5;
3     auto height = 3;
4
5     assert(triangleArea(base, height) == 7.5);
6 }
```

```
$ dmd example14.d
example14.d(6): Error: undefined identifier triangleArea
```

Providing incorrect types:

Listing 2-11: Cannot Implicitly Convert

```
1 void main() {
2     int foo = 64.42;
3 }
```

```
$ dmd example15.d
example15.d(3): Error: cannot implicitly convert
expression (64.42) of type double to int
```

Chapter 3

A Brief History

After some mostly unknown events that happened in the past, there was a brief period when a bunch of water filled sacks, which we will call human beings, had created what many believed to be the "way of the future." I guess they just didn't realize just how long "the future" was.

3.1 The Dinosaurs

Konrad Zuse in 1941 developed the first programmable computer. It was promptly destroyed in 1943 due to some disagreements between countries. Needles to say this was be for my time and programming was done on punched celluloid tape. Not the greatest medium for writing a program, but on the plus side the expected operations at that time were purely for number crunching (math problems). http://www.idsia.ch/ juergen/zuse.html

Another major player in defining and progressing what it means to be a computer was Allen Turing. Most notably was the definition he gave referred to as the "Turing Machine." While not practical design for a machine (who has infinite memory) it has become the basis for classifying programing languages as "Turing Complete" which in pop culture terms from the 1940's, "Anything you can do I can do also, I can do anything which you are able to do too."

3.1.1 Assembly

Assembly is a one to one correlation of instruction to machine operation. The detail required to write in assembly (ASM) is to the point that a load from memory must specify the exact register to hold it. Math operations aren't performed on numbers, they are requested to be run against multiple registers. Assembly is not really a specific language, it is representation of the hardware instructions; since hardware can contain a vastly different instruction set, assembly for one machine architecture could have a completely different representation for its instructions.

3.1.2 FORTRAN

Enter FORTRAN. John Backus was not fond of this and decided to represent higher level operations that would be translated into the instructions provided by the hardware. Higher level refers to being further from the details. At a low level you might see trees, water, and roads; the higher you are the less you would see until it eventually becomes Earth. This representation of the Earth would require something to translate this request for an Earth into the parts that actually make up an Earth. Thus the first compiler was written to translate FORTRAN to machine code.

And since some people enjoy having years, 1957.

http://en.wikipedia.org/wiki/Fortran

3.1.3 C

1972

C is yet another abstraction to the instructions provided by the hardware. Needles to say, it isn't going anywhere. This language built along side of the Unix operating system. It maintains a very close relationship to what is actually happening on the hardware. It has been an influencer to a great number of languages and the syntax has been claimed by many as well. D tries very hard to maintain that "if it looks like C then it either compiles with the same semantics or it fails to compile." This is not always true, but is very close. Where as C++ had to compile all C code without changing anything.

3.2 The Modern Times

While technology has been progressing very rapidly the design, principles, and techniques for programming languages haven't changed much. And in the computer world, the dinosaurs didn't all die out; which should not be a surprise as there was no catastrophic event which killed off all uses of these languages, requiring us to start from scratch.

What you'll see is different combinations of things already done one place or another. This is not to say there hasn't been major improvements, only to remind that improvements are incremental, its just that computers are really fast at incrementing now.

3.2.1 Is D a Duck?

There is an old test of no exactly know origin, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." And it is the principle behind what is known as *Duck Typing*. In this situation a type is only defined by what you can do with it and not what it has been declared to be. This is a behavior found in Python, however it is not the type system used in D. However *templates* in D do provide a means that resemble this behavior.

http://en.wikipedia.org/wiki/Duck_test

There is no right answer as both sides have there benefits and drawbacks. The choice to do *static typing* is one that provides more guarantees prior to ever running any code. And some from the dynamic side would argue that these guarantees are not enough benefit for the overhead they create.

Every variable in D has a specified type and that type can never be changed. It can be a burden to specify the type every time, so D has created a very good *type inference* as demonstrated earlier. There are limits just as there are clouds in the sky. Todo: Should not be here An interesting side effect of having an inferred return type was *Voldemort Types* discovered by Andrei Alexandrescu as these are types which can not be named.

3.3 The Rise and Fall of D

D has been the creation of *Walter Bright* and *Andrei Alexandrescu* to fulfill their masochistic desires. It has also been influenced and even developed by a small community of avid users that joined to make both Mr. Bright and Alexandrescu's life miserable. I could be wrong about this history though; it has been in development since 2001 with a version one release in 2007 and perhaps it was actually a joy to use and members not only desired for its success, but also didn't care as it was meeting some needs and doing so with a fun and engaging twist.

3.3.1 Walter Bright

Walter is a compiler writer with a background in mechanical engineering. He was once employed at Boeing and creator of the Empire video game. At a time when everyone was creating compilers that would translate C++ source code into C to finally be compiled as machine code, Walter wrote the first that compiled directly to machine code. Having enjoyed this great challenge and monster to upkeep, he decided that he could design and build a language that would be better and of course easier to implement.

While he was calling this new language the "Mars Programming Language" (his company being DigitalMars) when discussing his ideas with friends, he couldn't shake their referencing and calling it "D."

Much of Walter's original goals you could say have been revised since his initial go. And much change was mounting in 2006 when.

3.3.2 Andrei Alexandrescu

Alexandrescu came to the scene to help create a language which would address problems of the ~~future~~ today as well as the past.

Andrei came to us as a well known member of the C++ community. He came from the past, as the technology being developed in the US was slow to reach Romania. He did survive the time warp and was able to author a successful book, "Modern C++ Design."

He and Walter got to talking. Andrei wanted to take his knowledge and understanding, which he has graciously shared with the world, and packaging it into his own language. Walter advised that this was a very bad idea, and spoke of the deep lagoon which awaited. You see this could have been bad for business, Walter didn't want competition, or he really did know that many languages die during infancy because of the upfront work required to get a language up and running. Whatever the real reason, Andrei was convinced to join D and bring his offerings to the table.

And that is the story of the three wolfs and the ugly duckling.

3.3.3 The Community

As great as the D dictators have been, D would not have all the problems it does without the work, however subtle, from those members of the community past, present, and future.

I am afraid to provide this short list as many more have contributed to the progress of D and would be a shame to miss some of the influences that make a big impact but with little visibility in correlation. In alphabetical order:

- Bartosz Milewski
- Brad Roberts
- David Simcha
- Don Clugston
- Iain Buclaw
- Jonathan Davis
- Kris Bell
- Lars Kyllingstad
- Leonardo M
- Michael Parker
- Sean Kelly
- Steve Schveighoffer

3.3.4 Where D Has Gone

And now, for the sadist part of D's history, the story of its final demise.

Chapter 4

The User Interface at the Console

Now it is time to interact with the user using standard input and output, or stdio. This module is part of the standard library, Phobos.

4.1 Hello World!

One of the first programs generally written by a programmer after installing a compiler is the "Hello World" program. It is likely you used this program when following someone else's instructions for installing the D compiler. This simple program provides verification that the compiler is properly set up to access the standard library and provides the programmer with a visual cue that the program was correctly created.

Listing 4-1: Standard Output

```
1 module hello;
2
3 import std.stdio;
4
5 void main() {
6     writeln("Hello World!");
7 }
```

```
$ dmd example_hello.d
$ ./example_hello
Hello World!
```

Documentation can be found on http://www.dlang.org/phobos/

By importing stdio, which is part of the std package, a number of functions become available. We are taking advantage of code written by someone

else. The function *writeln* is written in a file std/stdio.d where the folder represents a package of modules. The standard library comes with every compiler implementation.

4.2 Listen to Me

To improve the computers communication skills it is important that it also learn to listen.

Listing 4-2: Standard Input

```
1 module username;
2
3 import std.stdio;
4
5 void main() {
6     writeln("Hello World!");
7
8     write("Please enter your name: ");
9     auto name = readln();
10
11    writefln("Hello %s and welcome to my world!", name);
12 }
```

```
$ dmd example_username.d
$ ./example_username
Hello World!
Please enter your name: Jesse
Hello Jesse
 and welcome to my world!
```

Haha, you thought I'd go over only one new function! Instead I have introduced, write, readln, and writefln respectively. The 'ln' stands for line, so the write method is the same as writeln, but does not place a new line on the end... The most interesting is writefln and writef. The 'f' stands for format and means the first argument will be a format string. This makes it easy to build a layout of what to output and insert variables into the needed location. In this case we are printing the *name* in the location of *%s*. There are other formatting options for spacing, alignment, and when printing numbers which you can reference at Formatting. Using %s will work for all types and recommended unless you need to change formatting. You can also produce a formatted string, and not have it output to stdout by using *std.string.format*. (Note that I am specifying a function *format* found in the package std, module string.)

One very noticeable aspect of the output is that it doesn't match what our format string specified... well actually it does. *readln* takes the users input and stores it in name, at the very end of enter the users name return is hit. This

new line character is captured by readln and thus printed where we ask it to be. The function std.string.strip can be used to remove this trailing character.

Being able to display data makes our previous work with *triangleArea* even more useful since we are able to display answers. Being able to ask for input from the user lets us get values from the user instead of hard-coding the question. But as you will see it isn't as straight forward as passing in the read values.

Listing 4-3: Triangle Calculator

```
1 import std.stdio;
2
3 import triangle;
4
5 void main() {
6     writeln("Area of Triangle Calculator");
7     writeln();
8
9     write("Triangle Height: ");
10    auto height = readln();
11
12    write("Triangle Base: ");
13    auto base = readln();
14
15    writeln("Area: ", triangleArea(base, height));
16 }
```

```
$ dmd example18.d example_triangle.d
example18.d(16): Error: function triangle.triangleArea
(double base, double height) is not callable using
argument types (string,string)
example18.d(16): Error: cannot implicitly convert
expression (base) of type string to double
example18.d(16): Error: cannot implicitly convert
expression (height) of type string to double
```

This code fails to compile because input from the user will be of type *string* and what we really want is to pass doubles to triangle function. To do this we use yet another library function. There are actually two functions which can be used for this purpose and it all depends on the goal you have when getting user input.

Listing 4-4: Triangle Calculator

```
1 import std.conv;
2 import std.exception;
3 import std.stdio;
4 import std.string;
5
```

```
 6 import triangle : triangleArea;
 7
 8 void main() {
 9     writeln("Area of Triangle Calculator");
10     writeln();
11
12     write("Triangle Height: ");
13     auto height = to!double(strip(readln()));
14     enforce(height > 0, "Height must be a positive");
15
16     write("Triangle Base: ");
17     auto base = to!double(strip(readln()));
18     enforce(base > 0, "Base must be a positive");
19
20     writeln("Area: ", triangleArea(base, height));
21 }
```

```
$ dmd example19.d example_triangle.d
$ ./example19
Area of Triangle Calculator

Triangle Height: 9
Triangle Base: 3
Area: 13.5
```

Now we have program which can be run to calculate the area of a triangle for most numbers. The import of triangle has another feature found in D, this is *selective imports*. Modules can contain many functions, using selective imports can prevent conflicts between modules. This book will make use of them so you can more easily see where used functions are coming from.

The use of *enforce* is somewhat redundant as *triangleArea* already has contracts. There is an important distinction for there usage. The program is requesting input from the user, this is an untrusted source. The use of assert is to identify locations that are bugs in your program, it is incorrect for the program to call *triangleArea* with numbers less than one. However, it is not a bug in the program if the user enters a negative number for any of the values. To ensure that this untrusted source does not cause the program to have a bug, the use of *enforce* allows an exception to be raised. There are many ways to deal with invalid inputs, and using exceptions tend to be the least user friendly but are great for personal projects.

An alternative would be to move these enforce statements *triangleArea*. The decisions to be had can come down to design choices and personal view. In this case it is considered wrong for *triangleArea* to be called with untrusted input.

4.3 A Word on double

The double only stores 64 bits, meaning there is only so many numbers that can be represented before it is all used up. The language can tell you the high and low of representable values for all integral types.

Listing 4-5: Limits of Double

```
1 void main() {
2     pragma(msg, "Minimum of double:");
3     pragma(msg, double.min);
4     pragma(msg, "Maximum of double:");
5     pragma(msg, double.max);
6 }
```

```
$ dmd example20.d
Minimum of double:
2.22507e-308
Maximum of double:
1.79769e+308
$ ./example20
```

So for the most part there isn't anything to worry about. But remember it may come up at some time. There is also another little thing that would make the program nicer to use, and that is waiting for the user to exit. We want someone to enter many requests instead of exiting after the first one. This requires a loop.

Did you see! This time running the program didn't do anything, instead we were given the values we requested while the compiler was running. *Pragma* is a command to the compiler and in this case it was asked to print out several messages (msg). This can only print on values known at the time of compile. It will get some more use in the feature.

4.4 Looping

A program starts at one point and continues until it runs out of instructions. During the life of the program it may require executing the same source code with slight modifications. While you may enjoy predicting how many times a user will want to calculate triangle areas when writing the program, usually it is better to get that information from the user.

Listing 4-6: Triangle Calculator

```
1 import std.conv : to;
2 import std.exception : enforce;
3 import std.stdio : write, writefln, readln;
4 import std.string : strip;
```

```
5
6 import triangle;
7
8 void main() {
9     string answer;
10
11     while(true) { // Loop with no exit condition
12         write("Enter the height or 'quit' when done: ");
13         answer = strip(readln());
14         if(answer == "quit")
15             break; // Break the loop
16
17         auto height = to!double(answer);
18         enforce(height > 0, "Height must be a positive");
19
20         write("Please enter the base: ");
21         answer = strip(readln());
22
23         auto base = to!double(answer);
24         enforce(base > 0, "Base must be a positive");
25         writefln("Area is %.3f",
26                 triangleArea(base, height));
27     }
28 }
```

```
$ dmd example21.d example_triangle.d
$ ./example21
Enter the height or 'quit' when done: 7
Please enter the base: 13
Area is 45.500
Enter the height or 'quit' when done: quit
```

The *while* loop takes an expression and continues to run while it evaluates to true. In this case I have provided *true*, which ironically always evaluates to true. This is because the necessary information to decide if the loop should continue or not is missing. Instead input from the user is used to make this decision by having them enter "quit". The *break* statement provides a means to stop execution of a loop, similarly a loop can be started from the beginning by using *continue*.

The introduction of a new function *std.string.strip* is due to a hidden character that represents the return/enter hit by the user. These characters come from a time when teletypewriters (TTY) were used. These devices required the transmission of a carriage return, which would bring the typewriter to a state for entry on the left side of the paper. The other operation was a line feed, which move the paper up for the next line of entry. Computers have continued to use these control codes, but it will depend on the operating system which

one is preferred. *nix machines have chosen the line feed, represented as *n* in a string. Apple choose the carriage return, represented as *r* in a string. Microsoft chose the combination of a carriage return and a line feed. Though remember that files can be transfered between operating systems and Apple has officially changed to using line feed, so it is best just to support all of them as the same.

And finally when the answer is printed it is formated with *%.3f*. This says to print a floating point number with three digits after the decimal point. You can find more details at Formatting.

Sometimes having a more elaborate format for input can make things nicer. The function std.conv.parse will do a partial conversion of the input leaving the end. This is in contrast to std.conv.to which throws an *exception* as covered in **??**.

Listing 4-7: Triangle Calculator

```
1  import std.algorithm : munch;
2  import std.conv : parse;
3  import std.exception : enforce;
4  import std.stdio : write, writeln, writefln, readln;
5  import std.string : strip;
6
7  import triangle;
8
9  void main() {
10     string answer;
11
12     while(true) { // Loop with no exit condition
13         write("Enter height,base or quite when done: ");
14         answer = strip(readln());
15         if(answer == "quit")
16             break; // Break the loop
17
18         auto height = parse!double(answer);
19         if(height <= 0) {
20             writeln("Height must be a positive.");
21             continue;
22         }
23
24         answer.munch(","); // tasty
25
26         auto base = parse!double(answer);
27         if(base <= 0) {
28             writeln("Base must be a positive.");
29             continue;
30         }
31         writefln("Area is %.3f",
```

```
32                  triangleArea(base, height));
33      }
34  }
```

```
$ dmd example22.d example_triangle.d
$ ./example22
Enter height,base or quite when done: 52,72
Area is 1872.000
Enter height,base or quite when done: quit
```

Instead of using *enforce* the *height* and *base* have been checked using an *if* and the *while loop* is started from the beginning using *continue*. This will usually provide a better user experience and demonstrates why *triangleArea* does not except untrusted input, the program needs to handle invalid input as part of its usual flow rather than through exception handling.

At this point the fundamentals for programming have been introduced. The important pieces to take from this have been variables, types, functions, and looping. This book will continue to use these as more is being introduced.

4.4.1 Looping Without Loops

Recommended Later Reading

Loops are actually a very useful abstraction. It is an abstraction because most hardware does not have any notion of a loop. Instead it provides a means to jump execution to another location in memory. A language that really modeled this simplicity of moving through code was BASIC. Each line required a number and at any point a request could be made to *goto* any line and continue execution from there.

The *goto* statement has been around a while and has had criticism due to its ability to obscure program flow and dangers of skipping vitally important code to keep a sane state. Many languages have chosen not to include *goto* due to this reputation, while D has chosen to make them safer by restricting the "distance" of the jump.

In most cases *goto* is not needed as we have better constructs to handle most situations. However there are places that *goto* makes for a cleaner statement. Minor uses of *goto* will be used in this book, but for now this is an example of functional code, but very bad design to demonstrate the operation of this statement.

Listing 4-8: Looping with Goto

```
1 import std.stdio;
2 import std.string : strip;
3
4 void main() {
5     string name;
6     writeln("Welcome");
```

```
 7 start:
 8     goto ask;
 9
10 getName:
11     name = strip(readln());
12     goto checkExit;
13
14 print:
15     writefln("Now give me your money %s!", name);
16     goto start;
17
18 unused:
19     writeln("You'll never see me.");
20
21 ask:
22     write("What is your name? ");
23     goto getName;
24
25 checkExit:
26     if(name != "quit")
27         goto print;
28 }
```

```
$ dmd example23.d
$ ./example23
Welcome
What is your name? Jesse
Now give me your money Jesse!
What is your name? quit
```

D uses a labeling system. A line of code can be labeled by using a valid identifier name followed by a colon. The label is then used in the *goto* statement.

You will likely notice that following the program flow is more difficult and while similar to making a function call, the calls are embedded making the loop very hard to see at a glance.

Chapter 5

Arrays and Associative Arrays

Programming works by processing data; storing that data is important and when you have a lot of it, simple variables don't cut it. An array is a collection of data that is stored sequentially. Usually it is a contiguous block of memory, which is what is used in D. We will learn about Arrays at a low level in the chapter Pointers, Arrays and Structures.

5.1 Array

Let us build a simple word replacement game.

Listing 5-1: Mad Libs

```
1 import std.stdio : writefln, writef, readln;
2 import std.string : strip;
3
4 auto paragraph = "Programming works by processing %s; " ~
5    "storing the %s is %s and when %s has a lot of it, " ~
6    "simple %s don't cut it.";
7
8 auto requests = ["plural noun",
9                   "adjective",
10                  "name",
11                  "plural noun"];
12
13 void main() {
14     string[] words;
15
16     foreach(req; requests) {
17         writef("Please enter a %s: ", req);
```

```
18          words ~= strip(readln());
19      }
20
21      writefln(paragraph, words[0], words[0],
22              words[1], words[2], words[3]);
23  }
```

```
$ dmd example24.d
$ ./example24
Please enter a plural noun: fish
Please enter a adjective: slippery
Please enter a name: Harry Potter
Please enter a plural noun: daggers
Programming works by processing fish; storing the fish
is slippery and when Harry Potter has a lot of it,
simple daggers don't cut it.
```

This approach doesn't scale very well, but that is ok as I have some important things to say about it.

5.1.1 Selective Import

5.1.2 Global Variables

5.1.3 Array Litterals

There are two *global variables* which are declared outside of any function. The "story" is stored in *paragraph*, which you should note has *%s* throughout it. An array of string using an *array literal* follows *paragraph*. Each element is separated by a comma and the whole thing surrounded in square brackets. An array literal of integers would look like [1,2,3,4].

5.1.4 Foreach

The type for a string array is written *string[]*, used when declaring words. This will store the values provided by the user. The user input is concatenated to the end of words inside a *foreach* loop. The structure of a foreach loop is as follows.

```
foreach(Type Variable; Collection) { }
```

The Type can usually be inferred, the Variable is what will store each element, and the Collection is any type that can be iterated. For details on iterable types and using foreach check out Iterators.

5.1.5 Concatenation

The *concatenation operator*, tilde ( ), provides a means to combine two arrays and is commonly used with string types. By using = the value on the right is being appended to the end of array *words*. This operation may or may not reallocate and move the array to accommodate the needed space. Details on what this means is in the Pointers, Arrays and Structures chapter. You can also combine a number of other operators with assignment.

Listing 5-2: Assignment and Operation

```
1 void main() {
2     auto a = 10;
3     a /= 2;
4     assert(a == 5);
5     a += 1;
6     assert(a == 6);
7 }
```

```
$ dmd example25.d
$ ./example25
```

Another way to write the concatenation would have been like the following, however in this case it would always reallocate memory for the array.

```
words = words ~ readln().strip;
```

5.2 Associative Array

Let us take a look at another way to write the same program changing a few things.

Listing 5-3: Mad Libs

```
1 import std.stdio : writefln, writef, readln;
2 import std.string : strip;
3
4 auto paragraph = "Programming works by processing %s; " ~
5     "storing the %s is %s and when %s has a lot of it, " ~
6     "simple %s don't cut it.";
7
8 string[string] requests;
9
10 void main() {
11     requests = ["plural noun1":"",
12                 "adjective":"",
13                 "name":"",
14                 "plural noun2":""];
```

```
15      foreach(speech, ref req; requests) {
16          writef("Please enter a %s: ", speech);
17          req = strip(readln());
18      }
19
20      writefln(paragraph, requests["plural noun1"],
21              requests["plural noun1"],
22              requests["adjective"],
23              requests["name"], requests["plural noun2"]);
24  }
```

```
$ dmd example26.d
$ ./example26
Please enter a name: Billy Blaze
Please enter a plural noun1: tazers
Please enter a plural noun2: pogosticks
Please enter a adjective: adventurous
Programming works by processing tazers; storing the
tazers is adventurous and when Billy Blaze has a lot of
it, simple pogosticks don't cut it.
```

For this we replace the need for words by using an *associative array* (also known as hashtables, dictionaries, key-value pairs). One trade-off with this approach is that numbers had to be added so that the keys were all unique. It would be possible to strip them off before asking the question. A literal for an associative array is surrounded by brackets like an array, but also provides a key in the form *Key:Value*. In this case *string* is used for the keys and values and the value is being set to an empty string. Use of any other type for key and value is possible, but it must be consistent throughout the whole array.

I have also added to the *foreach* with an extra variable and the use of *ref*. Since an associative array has the key and value for each item you can request them by providing two variables, the key is stored in *speech* and value in *req* which is determined by the order they appear. The value is requested by references, this allows changing its value in the loop and the change being reflected in the associative array.

The last change is instead of indexing with a number that is done with arrays, indexing uses a string which is equal to the key desired.

The thing to know about associative arrays is that they will use a magnitude more memory then the items they hold and iterating over the elements does not have a defined order.

5.3 Common Errors

A foreach loop was used to modify data which was stored in an associative array. This was possible because we asked for a reference to the data by preceding the

variable name with *ref*. Had this been left off the change would not be visible from the container being processed.

Listing 5-4: Foreach by Value

```
1 void main() {
2     auto arr = [1,2,3,4];
3     foreach(val; arr) {
4         val = val * 2;
5     }
6     assert(arr == [1,2,3,4]);
7 }
```

```
$ dmd example27.d
$ ./example27
```

Using ref:

Listing 5-5: Foreach by Reference

```
1 void main() {
2     auto arr = [1,2,3,4];
3     foreach(ref val; arr) {
4         val = val * 2;
5     }
6     assert(arr == [2,4,6,8]);
7 }
```

```
$ dmd example28.d
$ ./example28
```

Preventing modification:

Listing 5-6: Foreach over Constant

```
1 void main() {
2     auto arr = [1,2,3,4];
3     foreach(const int val; arr) {
4         val = val * 2;
5     }
6 }
```

```
$ dmd example29.d
example29.d(5): Error: variable example29.main.val
cannot modify const
```

Due to an implementation bug for my compiler the type was needed to get this to print the correct compiler error.

Chapter 6

The Compilation Model

It is time to remove a little bit of magic from the process of creating a program out of source code. This was touched on as part of some Common Errors, notably in 2.4. The compiler is actually part of an ecosystem and many times the lines become blurred and hard to distinguish. These are the stages going from the higher level component to the details of that component.

1. Build system
   (a) Maintain program/library dependencies
   (b) Execute commands resulting in a usable program/library
2. Compiler
   (a) Lexical Analysis
   (b) Parsing
   (c) Language translation
   (d) Optimizations
3. Linker
   (a) Combine machine code from multiple locations
   (b) Optimizations

Each stage is feeding into the next and interacting with one may not require explicit interaction for the next. Keep these stages in mind when requesting help, it is not necessary to know which stage has failed only that enough information should be provided so that someone else can help identify it.

6.1 Asking for Help

Provide your kind problem solver with as much relevant information as possible.

- A small segment of code which demonstrates the problem.
- Command used to compile.
- The error message.
- Any steps you may have already taken.
- Additional information that my be relevant.

Step one directs your attention to what your expectations are of working code and can lead to self answers. The code should be as small as possible as this eliminates unneeded details for willing to guide you to an answer. And once it is explained you'll have a nice concise code example of what not to do. However, if the error message is one you do not understand, it may be best to skip this in order to learn how the code should be reduced.

The command which is causing issues can be very important. This speaks to the compiler being used and other missing flags that could be needed to help resolve the problem.

An error message is vital to solving any problem as it gives direction. To say "the program below doesn't compile." limits the ability to address the issue you're having. Someone could easily take the time to place your code in a file and compile and run. There is little more than "works for me" to report back. And this statement is actually very specific to where the problem lies, though many will assume you could mean it doesn't link. Saying it doesn't compile indicates that the source code is being reported as invalid, something has failed during lexing or parsing.

It shows good faith to have attempted and search for a solution to the problem. The problem is unlikely to be unique, though it is understandable that someone new will have trouble locating a solution. Learning how to read and search for problems is very valuable so make an attempt even when it is small.

Finally, if you think there is something that could have influenced your problem state what it is. What operating system is used? Did it compile before upgrading to a new version of the compiler?

Be considerate to those taking their time to help. You won't have to include these for every single question; as you learn more it will be easier to identify where relevant information is. And people will guide you with further questions when needed.

6.2 Linker

6.3 Compiler

6.4 Build System

Chapter 7

Pointers, Arrays and Structures

Now that a basis for programming has been presented to "get things done" it is time to learn about things closer to the machine. This chapter will be similar to learning C, but remember that there are things the C compiler will allow that D will require to be explicitly stated.

Let us look at the behavior of value types and how to work with pointers.

Listing 7-1: Using Pointers

```
1 void main() {
2     int k = 6;
3
4     triple(k); // Pass the value 6
5     assert(k == 6); // Is not tripled
6
7     triple(&k); // Pass an address to k
8     assert(k == 18); // Is tripled
9 }
10
11 void triple(int k) {
12     k = k + k + k;
13 }
14
15 void triple(int* k) {
16     *k = *k + *k + *k;
17 }
```

```
$ dmd example30.d
$ ./example30
```

Here are two functions one which takes an integer and one which takes a pointer to an integer. There are three important things to remember when dealing with pointers. One is that the type for a pointer is denoted with a asterisk, *SomeType**.

A pointer is a location in memory. Consider the duodecimal system. The library will organize its books and provide a set of letters and numbers to identify the location any given book can be found in the library. Similarly the memory in the computer has an associated number for each location.

Variables have a location in memory, when a variable is used the value at that location is then retrieved and used in the operation. Pointers just store that address into a variable, and yes they also have a memory location which can be stored in a variable.

The second thing to remember is the address is taken by the address of operator *&*. And finally you are able to get the value at the address by using the dereference operator *. Here are some rather useless examples of using these operations.

Listing 7-2: Using Pointers

```
1 void main() {
2     int num = 7;
3     int* pnum = &num; // Address of num assigned to pnum
4     int num2 = *&num; // Store num's value in num2
5     *pnum = 5; // Assign 5 to the location pnum refers
6     assert(num == 5);
7     assert(num2 == 7);
8 }
```

```
$ dmd example31.d
$ ./example31
```

Note that line four is equivalent to int num2 = num;.

7.1 Binary

Remember a pointer is only a variable which is storing an address for a location in memory. A memory address is just a number, though usually a fairly large number. For this reason it is simple to observe the value of a pointer. The value however is not represented using the decimal system, instead hexadecimal is used.

Computers run off of the simple notion of "power here" and "no power here." This binary system makes it natural to store numbers in powers of two. Similarly we have ten fingers making it natural to "store" numbers of ten (decimal). Let us take the simple task of counting to ten: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. What comes after nine? Zero right? We have run out of digits to represent our number, so we begin to reuse what we have and place a one before the zero to indicate we have filled the first column up one time.

Now instead count to three: 0, 1, 10, 11. Did you forget already, a computer can only say yes power and no power. If it is going to represent a number greater than one it will need to add an additional circuit to say the previous one has filled once. But again, it can't indicate more than a single overflow resulting in the number four: 100. This is not one-hundred, eleven, or ten. Easily this could have all been represented with dots and dashes: . - -. – -.. but now how will I indicate the end of a sentence? Actually it harder to read so the standard 0 and 1 symbols will work fine. Besides you may have thought I was just saying "atnmd" and just been very confused.

Just as decimal increments for powers of ten (10, 100, 1000), binary increases its representative storage for powers of two.

```
 0: 0
 2: 10
 4: 100
 8: 1000
16: 10000
```

In decimal sixteen is simply two digits, while binary has just been growing at an alarming rate. This is where hexadecimal comes in to save the day.

7.2 Hexadecimal

For a 32-bit computer addresses are stored in...32-bits and 64-bits for a...64-bit machine. This is a lot of number so representing it in binary is out of the question.

01111111001100010101011001100100000111111100000

Oh, umm. How did that get there. But why not use decimal you ask? After all this address could easily be said to be at:

139849879523296

Simply because the number is of no use. What is important is being able to communicate with the computer and for it to communicate back in the most efficient manner. Hexadecimal is a number in base 16, meaning a slot will overflow for the sixteenth value. This address is represented as:

7F3156640FE0

Roman does not provide digits ten or higher, a hexadecimal uses the letters A through F to correspond to 10 and 15 respectively. Thus you won't reach 10 until your sixteenth number. This is being brought up because our memory addresses will be presented in this form and it is important to understand this numbering system when working with computers in general. Let us take a look at a program shows us addresses.

Listing 7-3: Hexadecimal Address

```
1 import std.stdio;
2 import std.conv;
3
```

```
4 void main() {
5     int num = 7;
6     int* pnum = &num;
7     int num2 = *&num;
8
9     pragma(msg, "Pointer size: " ~ to!string(pnum.sizeof));
10
11    writeln("&num : ", &num);
12    writeln(" pnum: ", pnum);
13    writeln("&num2: ", &num2);
14
15    assert(&num == pnum);
16    assert(&num != &num2);
17 }
```

```
$ dmd example32.d
Pointer size: 8
$ ./example32
&num : 7FFFF87601B8
 pnum: 7FFFF87601B8
&num2: 7FFFF87601C8
```

7.2.1 Addresses in More Detail

If you work making any attempt to check your understanding of bytes, bits, and address size, then you may have noticed that for a 64-bit system the address is only 48 bits. This is due to the architecture created by AMD for 64-bit systems. Here is a breakdown of the address used earlier.

```
7  F  3  1  5  6  6  4  0  F  E  0
1  2  3  4  5  6  7  8  9  10 11 12
```

Each digit in hexadecimal corresponds to for bits as F in binary, 1111, is for bits. The address given was twelve digits, twelve times four is:

Listing 7-4: Programming Math

```
1 import std.conv;
2 import std.math;
3
4 void main() {
5     pragma(msg, 12 * 4);
6     pragma(msg, "Addressable Bytes: " ~ to!string(pow(2, 12 * 4)));
7 }
```

```
$ dmd example33.d
48
Addressable Bytes: 0
$ ./example33
```

7.3 Arrays and Pointer Arithmetic

Arrays are just contiguous blocks of memory where each element is of equal size. We have seen how to use indexing to obtain an element from these arrays, now a pointer into the array will provide this access.

Listing 7-5: Arrays as Pointers

```
1 void main() {
2     auto arr = [3,4,12,23];
3     assert(sum(arr.ptr, arr.length) == 42);
4 }
5
6 int sum(in int* values, in size_t length) pure
7 in {
8     assert(values !is null);
9 } body {
10     int total;
11     foreach(i; 0..length) {
12         total += values[i];
13     }
14     return total;
15 }
```

```
$ dmd example34.d
$ ./example34
```

A function which takes an pointer and a length is to force the world of pointers into the function. We require the length because pointers do not come with how many elements exist in them. The function is also defined with *in* and *pure* as the arguments are not modified and we will not be affecting external state.

In the main function we pass our array as a pointer by using the *ptr* property. This is done instead of taking the address because an array in D is a struct with a pointer and length. Though using &arr will still provide the same information as arr.ptr.

The *sum* function accesses each element in the pointer and does so just like you would an array. However the foreach loop is changed to iterate over index values instead of the values inside the pointer.

Listing 7-6: Arrays as Pointers

```
1 void main() {
2     auto arr = [3,4,12,23];
3     assert(sum(arr.ptr, arr.length) == 42);
4 }
5
6 int sum(in int* values, in size_t length) pure
```

```
7 in {
8     assert(values !is null);
9 } body {
10    int total;
11    foreach(i; 0..length) {
12        total += *(values + i); // Use arithmetic
13    }
14    return total;
15 }
```

```
$ dmd example35.d
$ ./example35
```

This is a very minor modification to the previous example. Instead of indexing into the pointer the index is added to the pointer and the new location is dereferenced.

Pointer arithmetic is applying basic math operations, addition or subtraction, to an address.

7.4 Common Errors

D's garbage collector will not collect items which are referenced by a pointer, however when working with value types the address will be on the stack and if the stack frame returns your pointer will no longer be valid.

Listing 7-7: Returning Local Pointer

```
1 void main() {
2     auto p = getValue();
3 }
4
5 int* getValue() {
6     int num = 7;
7     return &num;
8 }
```

```
$ dmd example36.d
example36.d(7): Error: escaping reference to local num
```

The *stack frame* is named as it stacks a frame for the function within memory. When a function iscalled the parameters are placed on this stack along with a reservation for the result and any new variables created during the execution. The real nice thing about the stack is when the function completes there is no need for cleanup TODO: finish:wq :wq

Chapter 8

Iterators

Programming is all about processing data. At a very high level this means looking at and looking for specific kinds of data which can be culminated, modified, and pushed out as either more data or as useful information. Unlike data, information has meaning which usually comes from associating multiple types of data.

Consider: 15.5, 18, 21, 75.5. Can you tell me anything about this data? This data represents specific ages in a persons life. In fact they're just some numbers the United States has decided are "defining" times in someones life. At fifteen and one-half one can get a permit to drive, at eighteen one is consider an adult and can by cigarettes, at the magical age of twenty-one one can purchase alcohol, and seventy-five and one-half the government requires funds be removed from your retirement accounts. It is surprising how little meaning there is in pure data.

Iterators are collections of data that provide a means to move/observe/progress/iterate through this data. In D the *foreach* loop is one of the main mechiacnisms for performing operations on these iterables. There are several options for creating an iterator; the most common is known an a *range*.

8.1 Java Iterable

Chapter 9

Formatting