COURSE 1

UNIT 1 : Introduction to Programming and this Course

Lesson 1.1 : Some thoughts about programming language tutorials and books.

The video for this lesson can be found here: http://youtu.be/IWPrfVQJGB4

First, I want to welcome you to "Computer Science for Everyone". I made this course in the hopes that those who are seeking to learn programming would be able to. I realize that this is a complex topic, and it may seem overwhelming at first.

Once you learn this skill, you will have many opportunities open to you that were not open before. You may be able to get a better job, or perhaps even to create your own product and start your own company. At the very least, you will have access to far more tools that will enable you to do more than you could do before. Therefore, I encourage you to stay with the course and complete each and every lesson.

Do not skim! Do not skip lessons!

When you go through the lessons in this course, I encourage you to go through each lesson word-for-word methodically. NEVER cut-paste the sample programs. Always write them out yourself. By skimming it is easy to miss important information, and I promise you that everything presented to you in these lessons is useful, and you should read it fully.

Also, do not skip a lesson. Always make sure you master the material in one lesson before you proceed to the next. I have written every lesson in order to prepare the next, and no lesson is designed to be read without having read *every* lesson that came before.

So many lessons may seem overwhelming, and if so I want to assure you that while there is a lot of material in this course, I go through it slowly. If you ever do get stuck, simply follow the discussion links at the bottom of each lesson. Unlike a book, this is an interactive course where you have direct access to myself and many others who are willing to help you to ensure that you can achieve your goal.

Do not worry if you have tried to learn programming before.

It may be that in the past you tried to learn programming and couldn't. It may also be that you have learned a programming language, perhaps from a book, but you are not able to actually sit down and "make" something.

There is a difference between knowing a programming language, such as what a book teaches, and knowing how to actually make something.

Most programming tutorials focus on how to do the most basic programming instructions like if, then, else, and while statements. All of the focus is on how a particular language does these things. Every programming language has this functionality, they all do it in their own unique way.

Very rarely do any of these tutorials explain beyond this. As a result, there are many people out there who have "learned programming" which effectively means that they can write any program so long as it consists of giving someone a prompt to type some text, doing some processing, and then finally displaying some text output to the screen.

This is what virtually every book you will buy at a book store will give you the ability to do. For this reason, there are plenty of people out there who understand how to write a program, and can probably effectively read someone else's source code - but they could never go out and actually build something.

What is the missing link?

Libraries. These are the TOOLS you need as a programmer to actually make things. In short, libraries provide you with functions that you can call rather easily in order to actually put your programming knowledge to work. For example, nothing in the core language of C gives you the ability to draw a circle. But a graphics library might very well have a function called: drawCircle().

This is how advanced applications and games are built. These libraries themselves are put together and packaged for programmers to use, and then the language serves as an interface between the programmer and the libraries.

We will be spending a great deal of time working with these types of libraries to build real, usable programs and games.

Lesson 1.2 : C, C++, Python, Ruby, Perl... Why are there so many languages? Which is best?

The video for this lesson can be found here: http://youtu.be/Fmdf_B3Sne0

This is bound to be a question foremost on a lot of people's minds from beginners on up. There is a lot of depth to this question, and I think this is a great place to continue to after Lesson 1.

As strange as it sounds, all programming languages, no matter how cryptic they appear, are designed to be understood only by humans, not computers. Even assembly language is written to be understood only by humans. There is only one language that your computer understands, the language of 1s and 0s.

The need for programming languages.

The magic of computing is that sequences of 1s and 0s flowing non stop inside of your computer make everything happen. Everything. However, no human can possibly understand or control this process. Even one simple programming instruction such as print "Hello World"; expands into more 1s and 0s than you could count in a lifetime.

The first fundamental principle of programming I want you to learn is this: Programming languages exist in order to make it possible to do a great many operations (think trillions) with very few instructions.

The second principle I want you to learn is related: Good programmers figure out ways to do complex tasks, and convert these into simple instructions.

For example, it takes a lot of code to figure out how to draw a circle on a screen, but once finished with that process you have a function called "draw circle" which you can use any time, anywhere, to draw circles. Thankfully, you will never have to worry about that.

If you want to design a game for example, you will NEVER have to struggle with learning how to draw circles, or create 3d objects, or create weapons, enemies, etc. All of this work has been done FOR YOU by all those who have come before since the dawn of computing.

Just about everything you can imagine is already out there. Everything from making windows appear on your screen, to dialog boxes, to volume controls, to libraries that play movies -- everything. All you have to do is learn how to obtain and use these and you will be able to produce just about anything.

Why are there so many languages?

In the end, new languages come to exist because people think they can do something better or in a more aesthetically pleasing manner than some existing language. A lot of this has to do with personal style. Some people prefer doing things one way, and other people prefer doing the same thing in a different way. For this reason, you are likely to find some languages suit you better than others.

Which language is best?

There is no "best" language. Every language is a tool designed to be useful in certain situations, and not as useful in other situations. You should always evaluate what you are trying to accomplish in deciding which language you want to use.

The more popular a language becomes, the more useful it becomes for two primary reasons:

1. Support. It is a lot easier to find help for more popular languages because there are more people using it. This means more tutorials, more reference guides, more help forums, etc.

2. Libraries. The more people that use a language, the more libraries are going to be built for it. The number and type of libraries available for a given language largely determine how useful the language as a whole is. No matter how useful or popular a language, without good libraries you can't build much with it.

In general, having a "vocabulary" of different languages is very helpful. Think of this as having a tool box with many tools. The more languages you know and the more libraries you know, the more you can do. There are many other considerations to this which we will be going over later on, but I wanted to provide a basic introduction here.

UNIT 2 : Binary, Learning to Count like a Computer

Lesson 2.1 : The Importance of understanding binary.

The video for this lesson can be found here: http://youtu.be/0qjEkh3P9RE

It may seem like binary is something you will never have to use in programming. The truth is, if all you planned to do was learn a language or make simple applications, this is probably true.

However, the ability to really make things requires that you understand binary for many reasons, some of which I want to explore here. The first major reason you should know binary is:

Working with data formats

It is important to understand that everything in your computer is encoded in binary. Everything that is encoded in binary (movies, music, etc) is done so according to extremely specific requirements. I want you to understand a bit about how this works.

In .bmp image files for example, you begin a file like this:

    <2 bytes> <4 bytes> ... and so on.

The first set of 2 bytes identify the format of the BMP file (Windows, OS/2, etc) and the set of 4 bytes immediately following specify the size of the file in bytes.

Why is it important to know binary in this case? You need to be able to state the size of the file - in binary.

Many format specifications you will encounter require knowledge of binary in order to write programs that can produce or read that type of data. Well designed data format specifications often use binary values in various ways. This is especially true any time within the format that some quantity has to be known. Almost all such quantities are represented in binary.

Flags

The next reason you should know binary involves understanding something called "flags". Flags are representations in binary of several true/false states of something. Lets say for example you are designing a game, and you need to keep track of the true/false state of eight weapons which may or may not be in your inventory.

You can do this with a single byte! Eight bits. Each position can represent a given weapon. 1 = yes you have it, 0 = no you do not. So for example:

0100 = (0 in the "plasma cannon" place, 1 in the "shotgun" place, 0 in the "handgun" place, and 0 in the "knife" place).

Adding a weapon to inventory, for example adding a "plasma cannon" would involve simply adding "eight" (or 1000) to the existing value.

You will run into flags often especially with data formats, and understanding how they work and how to turn on/off values will be important. You will run into plenty of cases where source code you read contains advanced operations on binary data, and without an understanding of how to count in binary you will be unable to properly understand this code. There are many other applications as well, but I want you to be familiar with a few so that as we get into advanced data formats later, you will be prepared.

Lesson 2.2 : Computers count differently than we do. Introduction to Binary.

The video for this lesson can be found here: http://youtu.be/-gKYHGvzVxU

A lot of comments are coming up about binary, hexadecimal, and in general how a computer counts. Now, this will not be the only lesson I do on this subject, but it is an important topic to cover correctly.

Go through this slowly, and please ask questions. That is why this is an interactive course.

Please remember this for all lessons:

This course is designed so that you can go as slow as you need to. Do not worry about falling behind, or taking too long to finish a lesson. Take as much time as you need to on each lesson. I and others here actively monitor all lessons for questions, and will continue to do so for the duration of the course. Some people may just be starting out, and that is fine. There is no need to rush to "catch up". Take your time.

How humans count

When we count, we count in "base ten." Effectively this means we start at 0, then 1, 2, 3, 4, 5, 6, 7, 8, 9 -- and then "ten". Why ten? Well, most likely because we have ten fingers. However, humans also count in base 60 - though you may not have been aware of it until now.

For example, is it 11:58 AM ? That is an example of "base 60." You start at 0, then you keep going until you reach 59. Then you go back to 0. Consider the similarities between:

... 17, 18, 19, 20, 21, 22 ...

and

... 4:57, 4:58, 4:59, 5:00, 5:01 ...

The general rule to remember is this: When one "column" is FULL, the next column over to the left increments by one and the column that becomes full becomes zero. For example:

18, 19, 20 <--- the "ones" column is now full, so it becomes zero. The column next to it (the "tens" column) increments by one to become 2.

How computers count

Remember that inside a computer everything is represented as 1s and 0s. A sequence of 1s and 0s is actually a number, the same as: 1,274 is a number. In base ten, a column is full once it reaches 9. In base 60 a column is full once it reaches 59. Well, in base 2 (binary, 1s and 0s), a column is full once it reaches ONE.

So, you start counting like this: 0, 1

Ok, what now?

Well, like we talked about - the column is now full. It is full because it contains the highest allowed number, in this case a 1. Therefore, the "full" column must now become zero, and the column over to the left must now increase. Since at present that column is empty, it must become a 1. So:

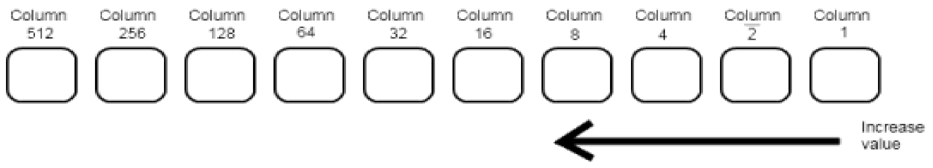
    00, 01, 10

ten? No. Two. Don't be confused. 10 all your life has meant "ten", but I want you to think of it as meaning something different: Two columns, and a value in each one.

Lets talk about the number 35 (thirty-five). It really means: 3 in the tens column, and five in the ones column. Each column in base-10 as you move over to the left becomes ten times the previous. So you go: ones, tens, hundreds, thousands, etc.

In base 2 (binary), each column doubles from the previous, so you go: 1, 2, 4, 8, 16, etc.

Here is a graphic that helps to illustrate this:



For example, the binary number: 0100 means this: You have a 0 in the ones place, a 0 in the twos place, and a 1 in the fours place, and a 0 in the eights place. Therefore, the number is "four".

So lets go back to counting in binary:

0, 1, 10 (because once a column is full, we go to the next column) then: 11 (three), 100 (four), 101 (five), 110 (six), 111 (seven).

Now, what do we do next? What would we do if the number was nine-hundred and ninety-nine? 999 ? Watch:

    999 + 1

```
    1000
```

Three columns are full, we go to the next one to the left. Now binary:

```
    111 + 1
```

1000 A thousand? No - eight. There is a one in the eights place, a 0 in the fours, a 0 in the twos and a 0 in the one's place.

It is very important that everyone masters this. Please feel free to ask any questions.

Lesson 2.3 : Patterns and Place Values of Binary

The video for this lesson can be found here: http://youtu.be/m1JtWKuTLR0

Earlier we went over the basics of binary and I explained how a base-two system is different from a base-ten system.

I realize some of this material may be difficult at first. Take your time, and ask questions. This is not a book but an interactive course and myself and others will be responding to any questions. Take your time. Go through this material slowly, do not skim it.

First, lets review the most important principles about binary. You might say that binary is how a computer "counts", but this is only a small piece of the story. Binary is the way a computer represents all numbers and data from simple counting to music files, to movies, etc.

Now, when we show binary numbers, we will typically write the numbers with spaces after each four digits. For example, instead of writing: 01100011 we would write: 0110 0011

Why is that? It simply makes it easier to read. Compare: 011111000001 to: 0111 1100 0001. Now we need to illustrate how to convert from binary to normal base-ten, and vice versa. Lets look at a table real quick:

```
    0000 : 0
    0001 : 1 (since there is a 1 in the ones place)
    0010 : 2 (since there is a 1 in the twos place)
    0011 : 3 (1 in two, 1 in one = 2+1 = 3)
    0100 : 4 (1 in four's place)
    0101 : 5 (1 in four, 1 in one = 4+1 = 5)
    0110 : 6 (1 in four, 1 in two = 4+2 = 6)
    0111 : 7 (1 in four, 1 in two, 1 in one = 4+2+1 = 7)
    1000 : 8 (1 in eight's place)
    1001 : 9 (1 in eight, 1 in one = 8+1 = 9)
```

Now what? We have used all our available digits from zero to nine. In base ten, you do not have any other digits to use. Here we can continue counting past ten by using letters. A can be ten, B can be eleven, and so on. You will see why soon. 1010 : A (1 in eight, 1 in two = 8+2 = 10)

```
    1011 : B (1 in eight, 1 in two, 1 in one = 8+2+1 = 11)
    1100 : C (1 in eight, 1 in four = 8+4 = 12)
    1101 : D (1 in eight, 1 in four, 1 in one = 8+4+1 = 13)
    1110 : E (1 in eight, 1 in four, 1 in two = 8+4+2 = 14)
    1111 : F (1 in eight, 1 in four, 1 in two, 1 in one = 8+4+2+1 = 15)
```

Examine only the column of this table containing the letters A through F. Now, if we were to stop here, what would be the next number? Lets go back to base ten for a moment, If we are at 9, what is the next number? The answer is "10" which means that the first column becomes 0, and the column next to it becomes 1.

So, if we count from 0 to F as above, what comes next? 10 -- except it doesn't mean ten. It doesn't mean two either. How much is it? Well, look at our above sequence - we went: 13, 14, 15 -- what comes next? sixteen! It is a curious fact that "10" (a one and a zero) means whatever base you are counting in. In base binary, 10 means two. In base ten, 10 means ten. In base sixteen, 10 means sixteen. And so on.

Therefore, in this new counting system with 0-9 and A-F, "10" means sixteen. This counting system called "base sixteen", or "hexadecimal" is extremely useful because you can represent ANY binary sequence using hexadecimal.

Lets keep counting so you can see that demonstrated:

```
    0000 1111 : F (1 in eight, 1 in four, 1 in two, 1 in one = 8+4+2+1 = 15)
    0001 0000 : 10 (not G, there is no such thing) (1 in sixteen's place)
```

Look at the binary of this. If we go 1, 2, 4, 8, 16 - then you will see clearly there is a 1 in the sixteen's place. Also, you will notice from the the above table that 0001 corresponds to 1, and 0000 corresponds to 0. It turns out that you can ALWAYS represent four binary digits with exactly one hexadecimal digit.

For example, 0110 1010 0011 - What is that in hexadecimal? Easy:

```
    0110 : six (6)
    1010 : ten (A)
    0011 : three (3)
```

Therefore, 0110 1010 0011 is: 6A3. It is that simple.

Now lets do it the other way around. How can you convert 5F1 from hexadecimal to binary? Well, what is five? 0101. What is F? 1111. What is one? 0001.

Therefore, 5F1 is: 0101 1111 0001

UNIT 3 : The Basics of Include Statements, Data, and RAM

Lesson 3.1 : Include Statements

The video for this lesson can be found here: http://youtu.be/VQ4enuvSe0w

I know many of you are anxious to begin writing your first program, and I am as eager to reach that point as you are. However, before we do, there are a number of important concepts I want to teach you. Be patient, and you will be programming in no time.

There is certain functionality that is shared by all languages. Some of this functionality is critical to understand even before you write your first line of real code.

Lets imagine you are trying to achieve some task inside a program you are writing, and you go to a forum to ask for help. Well, you are in luck because someone says "I wrote a function that does this already, here just include this code inside your program." This of course happens a lot.

There are really several ways you can do this. You could copy and paste the code right into your program. This can create issues because your program could become too long and difficult to understand. Just imagine how complicated it would be if you had to cut-and-paste lets say ten such files into your code. Also, imagine the headaches if you re-used this same code in other programs you are writing. What if you ever had to change something? You would have to change it in every file you cut and pasted the code into.

For this reason, virtually all languages have some form of an "include" statement. These include statements basically mean to cut-and-paste the contents of a file containing source code in that same programming language right into your program at the point you tell it to do the include.

In general it works like this:

```
include somefile.blah
```

As soon as you put that line in any of your programs, the whole contents of somefile.blah get placed right into your program, right where you typed that line.

This is important for many reasons. First, many libraries are contained in such files. Imagine a program that draws a circle, and lets say it relies on a "drawing" library that is five thousand lines of code long.

Which is easier, to write: include drawlibrary.blah into your program, or to cut and paste the whole contents of the file? You can see that there are many benefits to using "include" statements.

Remember that programmers are always looking for ways to make things easier, not harder. We like to avoid complications when possible.

Include statements were developed so that with a single line of code, you can put the whole contents of an entire file right into your program just as if you had typed the whole thing or copy-pasted it.

Addendum:

It is worth pointing out that the functionality I just described differs between programming languages. Some programming languages use the "Include" statements as a replacement for actually copy-pasting the entire contents of that file. Other languages use "Include" statements as a way to simply make functions found in the file available in the program you are writing.

The main thing that you need to understand however is that the purpose of using an "Include" statement in any language is to enable you to be able to use functions and commands that are available in the file you are including. For example, you may desire to write a program that draws a circle. To do so, you may need to "Include" a file that has a circle-drawing function. Once you "Include" the file, then you can draw the circle.

In this way, "Include" statements are closely related to the libraries we spoke about earlier. You will learn more about this as the course progresses.

Lesson 3.2 : How Programming Languages Work with Data

The video for this lesson can be found here: http://youtu.be/kDb_299qZMA

There are many types of data, ranging from simple (like numbers, letters, strings of text like "Hello", etc) to very complex data structures that could encode something like graphics or sound. All programming languages have built in mechanisms for understanding how to deal with the different types of data you will use.

Remember that all data, whether it was text, or numbers, or music is all going to be encoded in the same way. Binary. When you look inside your computer at the binary, you will not be able to tell the difference between one data type and another.

How can you know for example if: **0111 1110** is referring to a number, text, or part of something else? You can't! The same binary that means one thing if a number could mean something entirely different if part of a music file. That is why you must be specific in any program you write and state what type of data you are working with.

For example, if you are planning on having someone type text on their keyboard as part of your program, you need to tell the programming language that the type of data you expect to work with is text. If you are doing some addition on numbers, you need to tell the program that the type of data you expect to work with are numbers.

Each programming language has slightly different ways of doing this, however some things tend to be nearly universal. Concerning text, you usually will place the text inside either single quotes or double quotes. This tells the programming language that it is text.

For example, if I wrote "Hello Reddit" inside most programming languages, they will understand that data type as a string of text simply because I put it within quotes.

Many languages will understand numbers by just typing them out. Just simply typing 5 will be enough that the programming language knows you mean the number five.

Lesson 3.3 : Some Basics Concerning RAM

The video for this lesson can be found here: http://youtu.be/tZQAT1kkVoU

Unlike data stored on disk, ram (memory) exists only while your computer is turned on. As soon as you turn off your computer, everything that was in ram is gone. That is why if you were working on a document and forgot to save, you cannot get it back.

When you run a program on your computer, that program makes use of your ram to store and retrieve all sorts of data. For example, if you load a document in a word processor, the contents of that document can be loaded into your ram and then the program can manipulate the document as you edit it.

When you are satisfied, you tell the program to "save" your document, and this causes your program to take what was in RAM and store it onto your disk for permanent storage.

If you have four gigabytes of ram, that means that you have roughly four billion bytes, four billion sets of eight 1s and 0s available for any program that is running on your computer. Your operating system is responsible for ensuring that each program has enough to use, and for making sure that RAM in use by one program cannot be used by another until it is done.

Every one of those sequences of eight 1s and 0s has an address. The addresses start at 0 and work their way up to four billion. The exact way this is done is more complex, but for now - this is a simple description.

You as the programmer will need to store data at an address in ram, and then you need to be able to know where it is for later on. Lets say for example I have a string of text "Hello Reddit", and I put it in ram. If I want later to display that text, I have to first retrieve it from ram. That means I have to know where it was put, or what address it has.

It would be quite tedious if I had to remember some enormous number as an address in memory every time I needed to store something. This leads us to the next role a programming language has. Programming languages have functionality that keeps track of these addresses for us, and allows us to use plain-english names in place of these addresses, as well as for the contents of what we store.

Here is a sample of this in action. I tell my programming language to store the string of text "Hello Reddit" in memory somewhere. I have no way to know where. Then, I tell the programming language what I want to call that spot in memory. For example, I might call it: reddit_text

Later, I can simply type: print reddit_text and the programming language will do all the work of remembering where in memory it was stored, retrieving it, and actually printing the string of text "Hello Reddit".

Notice that the programming language is really keeping track of two things. First, it is keeping track of the contents of what I stored in ram. Secondly, it is keeping track of the address in ram so it can find it later. This second functionality will come in very handy as you will see.

Lesson 3.4 : Programs are Data Too!

The video for this lesson can be found here: http://youtu.be/BYPj1IBUeu4

We have already learned that data such as numbers, text, etc. is stored in ram memory at specific addresses. What you may not yet know is that when you run a program, it too gets loaded into memory the same way as if it was any other kind of data. In fact, as far as your computer is concerned, programs are data just like everything else.

So in addition to some sequence of binary like 0110 0111 being possibly a number or text like we talked about, it might also be part of a program.

Every single instruction that is ever processed by your computer is encoded the same way as everything else. You guessed it, Binary.

A program is fundamentally a sequence of many sets of 1s and 0s, each set being a unique instruction to tell your computer to do something. Some instructions might be small, like two bytes, and other instructions might be larger. Each instruction represents actual high/low voltage sequences which are transmitted directly to your CPU chip. Your CPU chip is designed to do many different things depending on exactly which sequence is received.

When a program is loaded into memory and executed, what happens is very simple. The first sequence of 1s and 0s, which is an actual command for the CPU, is sent to the CPU. The CPU then does what that instruction says to do.

This is known as "executing" an instruction. Then the next sequence is executed. Then the next. And so on. This is done extremely fast until every single instruction in the program has been executed. This process of executing one instruction after another is known as "program flow."

At the end of the entire program, after all of these instructions have been executed, we need one final instruction. Return control back to the operating system. This "return" instruction is special, and we will go into it in greater detail later.

Now, programs would be pretty boring if all they did was go through a set sequence until they were finished. It is often necessary in a program to specify different possibilities of how the program should flow. For example, maybe you want a program to do one thing if something is true and something else if it is false. We will describe this process soon.

UNIT 4 : About Program Flow, Functions, and Syntax

Lesson 4.1 : About Program Flow

The video for this lesson can be found here: http://youtu.be/fnp9wE_bSY4

We are getting closer to being able to write our first program. We are now going to start to learn about the structure that defines all programs. Now, it is true I could have you write a "first program" as many courses do, but all this would be is me giving you some code and telling you to type it verbatim and run it. I want it to be the case that when you write and run your first program, you really understand it. Patience, and you will be writing real programs in no time.

We talked about how a program is a sequence of instructions. Every program has an address in memory where it begins. The CPU looks at that address in memory and executes the instruction found there, then goes to the next instruction, and so on until the end of the program.

The way this works is simple: The CPU keeps track of the address in memory of the programming instructions to execute. Every time an instruction is executed, the CPU modifies its internal "address tracker" to point to the address of the next instruction.

Think of it like this: You have a list of tasks to complete today. You number each of these items on a piece of paper. Here is an example:

1. Fix breakfast.
2. Go to the bank.
3. Deposit check.
4. Pay electric bill.
5. Wash the car.
6. Drive home.

Each of these steps can be thought of as a programming instruction. Now imagine that each of these numbers one through six is the "address" of that particular instruction. For example, 3 is the "address" on the paper of the "Deposit check" instruction.

You point a pen at item one, you read it, and then you complete the task. After the task is complete, you mark it and point the pen to the next item on the list. You do this until everything is completed. The pen acts as a pointer to the instruction being executed so you can keep track of it.

At any given time during this process you need to be aware of what step you are on. In other words, you have to keep track of the address on the paper where the next instruction to execute is.

Inside your computer, your CPU has something called an "instruction pointer" which does exactly this. Think of it as being just like the pen in the example I gave. At any time during a program, the Instruction Pointer contains the address in ram of the next instruction to execute.

Once a programming instruction is executed, it does not get erased. It is still there in memory exactly where it was before. This means that your CPU could execute that same instruction again. For example, if you already drove to the bank, but later on found out that you had to go back, it would be possible for you to move the pen back to the instruction that says "Drive to the bank.". It is still written on the paper, and nothing stops you from executing that instruction again.

Very often in a program it is necessary to repeat an instruction. This is achieved by telling the CPU to "go back" to an address in memory of an instruction that has already executed.

For example, if you want to print "Hello Reddit" ten times, you would not need to write "Hello Reddit" ten times, you could simply tell your program to go back to that instruction and repeat it - ten times.

It used to be that programmers had to keep track of the addresses in memory of various parts of the program they might want to re-execute. Fortunately, modern programming languages remove all of that headache.

Lesson 4.2 : The Functions, Methods, Routines, etc.

The video for this lesson can be found here: http://youtu.be/VV2sgTT48tw

This lesson is a bit more intense than most. Take your time and work through this slowly, and ask questions if any part does not make sense to you. This is highly critical information to master.

You now know that your CPU keeps track of the address of programming instructions being executed using an "instruction pointer".

Everything we talked about in lesson 11 involved a single program in memory; a single list of tasks to do today. What would happen if you had two lists of tasks to do today instead of one?

First, notice that there is nothing preventing you from moving the pen at will between the two lists. You could do item #1 on the first list, followed by item #2, followed by item #3, then you might jump to the second list and complete all the items on that list, then jump back to where you left off on the first list.

It turns out that just like the above example, you can create as many programs as you want - each starting at its own unique address in ram. We call these smaller programs "functions" (though as you will see they can be called by other names as well). Each function has its own address in memory where it begins and has a list of programming instructions to execute.

In an earlier lesson, I explained that part of the job of a programming language is to keep track of memory addresses for data. I pointed out that you can give plain English names to any data you like, and the programming language does all the work of tracking its value and its memory address so you don't have to.

Well, we also said that programs and programming instructions are data just like everything else. Remember the "Programs are data too." lesson. Therefore, would it not make sense that you can keep track of addresses in memory of functions the same as you can any other data, by just stating plain English names? The answer is yes - you can. Every programming language makes this possible in fact.

I could choose to call one function by the name of:

```
business_to_do_today
```

and I could name another function:

```
personal_to_do_today
```

When I want a function to run, I can just call it by the name I gave it. The programming language takes care of all the details about where it is in memory, how to handle the instruction pointer, and everything else. As a programmer I do not have to worry about any of those details.

Every programming language does this differently. Some languages call these things functions, some call them routines, some methods, etc. The idea is the same. If it is a list of programming instructions meant to be executed and called by some plain English name, it is for all intents and purposes a "function" for the purposes of this lesson.

Lesson 4.3 : About Arguments and Return Values

The video for this lesson can be found here: http://youtu.be/TQE2QlNDO4Y

In an earlier lesson we learned the basics about functions. Primarily we learned that functions are programs that reside in memory just like data and that you can instruct your computer to jump to that point in memory, and execute a function. Now, I want to expand on that knowledge and explore what especially makes a function useful, and so critical to programmers.

Every program would be useless if it didn't have a way to display something to the screen. You could write a program for example that can convert a binary number to a hexadecimal number, but without a way to actually see the result you may as well have written nothing.

It should then be clear that one function that is packaged with just about every programming language, is some sort of ability to print text to the screen. Now, this varies from language to language.

Lets call this function "print" for the sake of this lesson. Imagine that "print" is a function that sits in memory, at some address, just like we talked about in the previous lesson. Now suppose I want to print some specific text like "Hello Reddit", how could I do it?

First, notice that it is not enough to simply call the "print" function. I have to have a way of specifying what it is I want to print. Whenever you give a function extra information that it needs in order to perform a task, that extra information is known as a "parameter" or an "argument".

You can give a function as many parameters as you like. For example, a "drawCircle" function might require all sorts of information. You probably need to specify the x and y coordinates on your screen where the circle will appear, the radius of the circle, the color of the circle, the thickness, maybe even whether or not it is a dashed line or a solid line.

Every programming language does this differently, but all programming languages give you the ability to send extra information to a function. This information is used by the function in order to complete the task you desired. Different incoming parameters will likely change the result of a function.

Whenever a function finishes executing, it passes control of the program back to the line of code where the function was called. However, it has an option of also sending back some information to tell about what it did, or whether or not it was successful, or even to return complex data.

This is called a "return value". In our drawCircle example, a return value might be something like this:

1. Successful
2. Not successful

This is a simple example, but lets say we have a function whose job is to take one string of text like "Hello Reddit" and turn it all uppercase so that it becomes: "HELLO REDDIT".

In this case, the return value would actually be the newly created string of text "HELLO REDDIT". In general, functions can return anything at all as a return value, and this can be used by whatever called the function. Functions are everywhere in programming. Even your main program is itself a function. Some programming languages in fact require you to expressly create a function for your main program.

Lesson 4.4 : About Syntax and Function Vocabulary

The video for this lesson can be found here: http://youtu.be/iPj9PomYQos

Every programming language has two aspects that you need to know about. The first is called syntax, and the second I am calling "function vocabulary" as part of this course.

Syntax refers to the specific way in which a function requires you to do all of the things we have talked about up until now. In most languages for example, you specify text by enclosing it within quotes. That is an example of syntax. Function vocabulary refers to knowing many different functions that are available. In much the same way as you build up a vocabulary of words in order to speak a language, you need to develop a vocabulary of functions in order to "know" a programming language.

When two languages allow you to create the same function called myFunction, and they both have you do so in a different way, that is an example of how the syntax of one language is different from the syntax of another.

Learning the syntax of a language is really all that is meant by learning a programming language. However, "knowing" a programming language in this way is quite useless. In addition to understanding the syntax of a language, you need to learn many of the functions that come packaged with the language.

Of course, even this is not really enough. We talked in one of the first lessons that even the grand sum of all the functionality that comes with a given language is not really enough to make any real application. For that, you need to also learn libraries, and the functions that go with them.

So lets sum this up a bit. You can't learn how to use the functions that come with a language properly until you understand the syntax of that language. Also, you can't get the most benefit from library functions without first understanding the functions that came packaged with the language.

Now, why is that? Because the functions that come packaged with a language are going to be the most basic functions you need. No library is going to re-create those functions, they are simply going to assume you already know how to use them. You will need to use them to do anything useful with more advanced functions like library functions.

Therefore, you learn any programming language by following these three steps:

1. Learn the syntax.
2. Learn the built in functions.
3. Obtain and learn the functions that come with libraries.

With #2 above, I am referring to those functions that typically come pre-packaged with a given programming language. These may also be in fact part of libraries, but only libraries supplied by the distributor of the compiler or interpreter for the programming language you are using.

With #3 above, I am referring to libraries built by developers who use the language. These libraries make it possible to do a great many operations that could not easily be done with just the functions that are included in #2. Some of these libraries are free (many in fact), and some cost a license fee.

Remember that what you can create is always limited by the types of functions available to you, and the more libraries you obtain and learn the more you can create in any language.

UNIT 5 : Your First Program, and beyond

Lesson 5.1 : Write Your First Program

The video for this lesson can be found here: http://youtu.be/T677kQzRLrQ

It is time to write your first program.

I am going to explain to you what the program is, and then I am going to give you everything you need to make it.

The goal is to create a program that will print the text "Hello Reddit!" to the screen.

The language we will be doing this in is called C. Here are the rules for C you need to know in order to make this program.

* We will be using a library that comes packaged with C. This library is called the "Standard Input/Output" library.

* To use the functions in this library, you have to include the file stdio.h at the top of your program. Remember I said that each programming language has a different way of doing this. In C, here is the syntax for doing that with any file:

```
#include <filename.blah>
```

Note that the greater than and less than sign are part of the instruction. They must also be present.

* I mentioned some programming languages require you to create a function in order to write a program. C is one such language. Therefore, you will have to create a function called main() for your program to work correctly. C has specific rules for this which are noted below.

For your main() function in C, you put:

```
int main(void) {
    .... any code goes here ...
}
```

The word "int" at the start simply means "integer". It specifies that the main() function will return some number as an indicator of whether or not it was successful. The "void" within the parentheses just means that you are not sending any arguments to the function. In other words, the main() function doesn't require any additional information to be sent to it in order to do its job. You will learn more about this later in the course.

* ALL code for the main() function must be between the opening "{" and the closing "}"
* The function in the "Standard Input/Output" library we are going to be using is called printf. This

function takes a single argument, the text you wish to print. C is one of the languages that encloses text within double-quotes.
* You call a function in C by simply putting the function name along with any arguments within parenthesis. At the end, you put a semi-colon ;

example_function("A text argument");

You may find during this course that I sometimes refer to the extra information you send to functions as parameters, and other times I refer to them as arguments. The correct terminology in C is "argument".

* At the end of the main() program in C, you should return a value. Typical is to simply return 0 for a successful program. You can do that with this command:

return 0;

Edit: Originally I had this saying return 1, which works fine - however it is true that for main() you return a 0 typically for success and a non-zero for failure. It is better to have return 0 for this example. 0 or 1 (or any number) will work fine, but to indicate a successful program, returning 0 from main() is best. Ironically, later on when you learn to write functions, you will discover that returning 1 from a function is usually better. We will get to that later. Just to be clear, when we write "return 0", this means that we are returning 0 from the main() function.

The number you return from a main() function identifies whether or not the program was successful.

You now have everything you need in order to write this first program in C. Try to do it yourself, and post it as a comment using the discussion link below if you like. First, go to Codepad and write your code, and then post the codepad link in the discussion below. Lets see how you do.

Lesson 5.2 : Review of Your First Program

The video for this lesson can be found here: http://youtu.be/Hqb7fOXgZd8

Congratulations on writing your first program in any language. Everyone did a great job on this first program, and for anyone who doesn't already know, here is the answer:

```
#include <stdio.h>

int main(void) {
   printf ("Hello Reddit!");
   return 0;
   }
```

I believe that from the lessons up until now, everyone should understand how and why this works. However, a few things should be addressed.

Here we had our first exposure to some of the syntax of a specific language, in this case C. We also learned our first simple function, printf().

First, in lesson 7 I explained that Include statements effectively copy and paste the contents of one source file so that you can use it in your program. For many languages, including C, this is exactly how it is done - however I want to go over a bit more of this.

The idea when using an Include statement in general is that you are saying "This file has something I want. I want to make the functions that are in this file available for use within my program." Every programming language makes it possible for you to separate code into multiple files, and then make these files available for programs as you desire.

When you say:

```
#include <stdio.h>
```

You are basically saying, "stdio.h has functions that I need to use." In this case, one of those functions is printf(). There are many others, and we will go over them later.

Now lets talk about the main() function. As I explained in previous lessons, some languages require you to define a "main" function, but I did not go into the details of why this makes sense.

When we talked about functions, we learned they had arguments (things you send to the function) as well as a return value (what the function "gives back" when it is finished running.)

Did you know that even programs you run operate in exactly this way? For example, with firefox you could run it by typing:

firefox.exe http://www.reddit.com

Well, in this case you are giving firefox an argument, and that argument is the URL you want to go to.

Next, programs tell the operating system an "exit status" which indicates whether the program was successful or had an error. When you return 0; you are telling the operating system "This program finished successfully." When you return any non-zero value, you are telling the operating system that there was a problem.

So from this explanation you should be able to understand that programs work in much the same way as any function works.

Remember in an earlier lesson we talked about the importance of specifying data types whenever you work with data, to tell whether or not you want to work with a number, or text, or something else.

When you define a function you have to specify what data type you will be using for the return value. For example, is it going to be returning a number, or something else?

The word "number" can mean several things. We will go over that, but for right now I want to introduce you to the most common number type. The integer.

Integers are all whole numbers (but there are limits to this, as we will learn), and in C you identify a data type as an integer by typing:

```
int
```

As you can see, "int" is short for integer. So, in our main program we are returning an int as a return value, a whole number. Therefore, we should specify this. We do so placing the keyword "int" in front of the function.

```
Int main(void) {
    printf ("Hello Reddit");
    return 0;
    }
```

Now we are saying "our main function returns an integer when it is done.

Lastly, by placing main(void) with "void" inside of the parenthesis, we are saying "We are not planning on sending any additional information to this program." Notice that there is a special keyword for "no parameters" in C. That keyword is "void".

Lesson 5.3 : How to Run Your First Program

The video for this lesson can be found here: http://youtu.be/bT9IMBFS3MU

Supplemental video on how to use CodeBlocks: http://youtu.be/dn7J5WuHqSg

Writing a program is all well and good, but is not of much use if you cannot actually run it to see the results. Part of the reason I chose C to begin with (we will be exploring other languages as well) is that there are so many great free C compilers available and other resources.

To complete this lesson, simply get the program you wrote in step 15 to run. If you need any help with this process, feel free to ask. Myself and others will help you get your first compiler installed and working.

On Linux

If you are on linux, then to compile and run a program all you have to do is save the program as a file, call it something like: firstprogram.c, then:

```
gcc firstprogram.c -o firstprogram
./firstprogram
```

If gcc is not installed, then you should be able to easily install it from your package manager. Just search for gcc, or for example in Ubuntu type:

```
sudo apt-get install build-essential
```

On Windows

You need to obtain a compiler. There are many great free compilers out there for windows including:

*** Recommended : http://www.codeblocks.org/downloads/**

Edit: get codeblocks-...mingw-setup.exe]

Having a problem which says "Invalid compiler" ? Try this

I ran across this on Google for those having issues:

Re: uses an invalid compiler. Skipping... « Reply #3 on: December 09, 2008, 05:38:31 am »

I encountered the same problem and solved it after a bit of tinkering... steps would be:

1. goto "Menu"->Settings ->"Compiler and Debugger" -> [It will open a new Tab]..... ->
2. In this tab, you have listings like...Compiler flags, Linker settings, Search subdirectories,...... next to that is ">" button, click on the ">" button 2-3 times, till you find "Toolchain executables" in same line.
3. In this window, set "compiler settings" to which ever directories your compiler is installed.
4. For varification, goto lower section of tab-menu and click on location button for gcc or g++, it shoulddirectly open a new browser window and gcc /g++ is selected.

Other good compilers for Windows are:

* http://www.microsoft.com/express/vc/Default.aspx
* http://www.delorie.com/djgpp/zip-picker.html

Edit: Here are some details on getting started with codeblocks

First, once you install it it may ask you for some options. All default options are fine.

When you get to the screen after installing it, there is a button that says "New Project." Click that, and choose "Console Application". Then it will take your through a "Wizard". Chose C as the language. Give it a title like First Program, give it a filename, and a directory to store it in.

I recommend you create a directory on your computer for your C programs.

On the next screen "Compiler configuration" leave everything as is. Then, when you are done with the wizard on your left side you will see a link under "Management" / "Projects" that says "Sources" Click that, and then you will see the file main.c Click on that.

You will see that Codeblocks by default already has a "Hello World" program pre-typed for you. There are slight differences to the one we wrote, but do not worry about that.

At the top there is a "play" button (a blue triangle icon) that when you mouse over says "Run". Click that button and it will ask you if you want to build the project, choose Yes. Then you will see the program run successfully. Congratulations, you can now build and run C programs.

Once you have a compiler installed and working, simply save the program you wrote as first.c and then use the compiler you installed to make first.c into first.exe, and run it. If you need help just ask.

On a Mac

* http://www.codeblocks.org/downloads/

If you get an error similar to "Nothing to be done", this is likely because a compiler is not installed. You can fix that by installing gcc which can be found here: http://www.tech-recipes.com/rx/726/mac-os-x-install-gcc-compiler/

OR:

To compile and run a program all you have to do is save the program as a file, call it something like: firstprogram.c, then do this from your terminal:

```
gcc firstprogram.c -o firstprogram
./firstprogram
```

Alternative Methods

Without downloading or installing any program, you can write and run your program here: http://www.codepad.org

UNIT 6 : Basic Data Types

Lesson 6.1 : The Basics of Signed and Unsigned Numbers

The video for this lesson can be found here: http://youtu.be/miwMEUfkqfY

This lesson is intended to demonstrate the basics behind "signed" and "unsigned" numbers. This lesson is not specific to any programming language, including C.

We have already discussed that computers represent all data as binary. We also discussed that it is impossible to distinguish between one data type and another just by looking at it, because any given sequence of binary could be part of absolutely anything such as a number, text, a movie, or even a program.

For this lesson, we are going to only discuss numbers. Lets examine the following binary number:

```
111
```

This is of course the number seven. If I asked you to store that inside of your computer, what is the minimum size you require? The answer is 3 bits. Each bit in your computer is effectively a 1 or a 0, and three of those bits will be enough to store any value from zero through seven.

Alright, but what happens if you need to store the number eight? Well, you cannot do it with three bits, you need at least four because eight is represented as 1000 which requires four bits, or four binary digits. If you needed to store a number 16 or greater, you need at least five bits. Here we learn two important principles:

1. The number of bits determines the maximum size of any number.
2. Adding just one extra bit to any binary sequence doubles its capacity.

For example, with three bits you can store a total of eight values (zero through seven, that is eight values including the zero). With four bits, you can store a total of sixteen values (zero through fifteen, including the zero). Each time you add a bit, you double the storage capacity.

I want to explore this a bit more so you can understand something about binary at a fundamental level. Lets look at a simple table of binary. We will start at zero, and add one to each new value, so you should be able to follow along.

```
0000
0001
0010
0011
0100
0101
0110
0111
```

Now, from eight onward:

```
1000
1001
1010
1011
1100
1101
1110
1111
```

And at fifteen we are done.

Did you notice that we simply repeated the first table of zero to seven a second time, only we had a 1 on the far left this time? This is because the 1 on the far left effectively means "add eight" since that is a 1 in the eight's place. If we started at 0 and counted to seven with the "add eight" present that is the same thing as counting from eight to fifteen.

We could also do something else here if we wanted to, we could choose to say that BOTH sequences are counting from zero to seven, except the far left digit is a "flag" indicating if we want the number to be positive or negative.

We could choose to say that instead of an "eights place", this fourth column from right to left means "positive or negative". We will say that a 0 here means positive, and a 1 means negative.

Whenever you encode a number in binary in such a way that it can be either a positive or a negative number, that is known as a "signed" number. This specific method I am introducing is known as "signed magnitude". It basically means that the furthest digit to the left is a flag indicating if this is a positive or a negative value. So, in our above example:

```
0011 = positive three 1011 = negative three.
```

Whenever you define a bit as a flag for stating if a number is positive or negative, that is called a "sign bit". In this case, a sign bit of 0 means "positive number" and a sign bit of 1 means "negative number".

Now here you should notice something important. When using four bits, if this were an unsigned number we could count all values from zero to fifteen. However, when we make the "eight's place" into a flag for positive or negative, we can only now count half as much as before, but we can do so in two different directions.

```
0000 = 0
0001 = +1
0010 = +2
0011 = +3
0100 = +4
0101 = +5
0110 = +6
0111 = +7
1000 = +8 OR 0 (negative zero is zero)
1001 = +9 OR -1
1010 = +10 OR -2
1011 = +11 OR -3
1100 = +12 OR -4
1101 = +13 OR -5
1110 = +14 OR -6
1111 = +15 OR -7
```

Remember, this is a system we are using just for the purpose of this lesson. This is neither the best nor the most efficient way to represent positive or negative numbers, and we will get to that later. Primarily I want you to get three things out of this lesson:

1. You can specify a number as being negative or positive with a "sign bit".
2. When you have a sign bit, you can only count half as far but you can do so in two directions, positive and negative.
3. The same exact binary can encode a signed number, or an unsigned number. For example, the binary sequence 1010 could mean either "ten" or "negative two".

Lesson 6.2 : The Basics of Numeric Overflow

Video for this lesson can be found here: http://youtu.be/Rq3vT2A8tGM

This lesson is not specific to any programming language, including C.

In the previous lesson we learned the basics for signed and unsigned numbers, and we also learned that the more bits are allowed for storage, the higher the number that can be represented.

Now, lets talk briefly about what happens when you have a set number of bits for storage, and you start counting. Lets use 3 bits for this example.

If I say that I have three bits available, I can now represent every value from zero through seven. That is eight total values. Lets start counting:

```
000
001
010
```

```
011
100
101
110
111
```

Now, what happens if I add one more? The answer is eight which is represented as 1000. But here we have a problem. We only have three bits for storing the number. That means it is impossible to store eight, we simply cannot do it.

With three bits we can represent every value from zero through seven, but there is absolutely no way we could ever represent eight. Therefore, if we add one more to this 111 and we are forced to stay within 3 bits, we will get this result:

```
110 = six, lets add one
111 = seven, lets add one
000 = Back to zero, lets add one
001 = one.
```

Why did this happen? Well first remember that the rules of binary state that if all the columns are full, and you add one, then they all become zero again. The other step is of course that we need to add one to the column on the far left, but here we have an issue. There is no column!

Whenever it happens that you use all the columns, and need to add one, you will always go back to zero and start over. This means that you can get some unexpected mathematical results. Lets go back to our example with 3 bits.

Lets add four and six. Both of these values can in fact fit in 3 bits, so it seems ok. However, look at the result:

```
  0100
+ 0110
---------
  1010
```

(don't worry too much about adding in binary. We will get to that later.)

Now, keep in mind that because we are limited to only 3 digits, the one in the eights place gets dropped. The final value we would get is: 010 - which is two!

Therefore, unexpectedly, we get the result that four and six gives us two! This is of course an error, and is caused by the fact that we are performing a mathematical operation whose result requires more bits of storage space than we have available to use.

Whenever this occurs, it is known as an "overflow".

Lesson 6.3 : The Basics of Fractional Numbers in Binary

The video for this lesson can be found here: http://youtu.be/Y4Q9PnjKhac

This lesson is designed to teach you about how fractional numbers are represented in binary in general. This lesson is not specific to any programming language, including "C".

In earlier lessons we learned how we can represent any whole number as a binary number. We even developed as system that allows us to do this for negative numbers. Therefore, as far as integers are concerned, you can represent any integer value as a binary number.

However, what happens when you have a number like 2.5 ? How can you store a number like this in binary?

First, lets look at how we do this in the base-ten (hereafter referred to as decimal) counting system that we are all used to. In decimal, how do you represent fractional parts? Well, you cut the number into two parts. The left hand side of the number is an integer, then you put a period, then you put the fractional part. Lets look at this in detail:

```
    172.31
```

Here we have the number one-hundred seventy-two, and we are defining a fractional part as thirty-one hundredths. If we looked at this according to what we have learned about place values, we see this:

There is a 1 in the hundreds place, a 7 in the tens place, and a 2 in the ones place. There is also a 3 in the tenths place, and a 1 in the hundredths place.

Note that we consider the place values to the right of the decimal point as fractional parts based on powers of ten, for example:

```
    .1 = 1/10
    .03 = 3/100
    .002 = 2/1000
```

So we go tenths, hundredths, thousandths, etc.

Now lets look at a totally different number, but in binary this time.

So in binary, we follow this same exact principle, but instead of using ten as the base, we are using two. Therefore instead of 1/10, 1/100, 1/1000 we will be doing: 1/2, 1/4, 1/8, and so on. Now lets look at this in action.

```
    0110.1000
```

Remember that to the left of the decimal point (called a radix point by the way) we have the value of six. If you do not understand why, please go back and re-read lessons 3 and 6. To the right of the decimal point, we have a 1 in the half's place. That means the final value is: 6.5

Suppose we had:

0011.1100

Now we have three to the left of the radix point, and to the right we have a 1 in the half's place, and a 1 in the fourth's place. That means one half plus one fourth which is three fourths. Therefore, the value here is: 3.75

Now what if we wanted to say: 8.1 - that is, eight and a tenth. This becomes trickier. Since in binary we are working with two as the base, there is no such thing as a tenth. We have to approximate by coming up with as close as possible of a value to 1/10th as possible.

0.1 = 1/2 (too high)
0.01 = 1/4 (too high)
0.001 = 1/8 (too high, but getting closer to a tenth)
0.0001 = 1/16 (too low)
0.00011 = 1/16 + 1/32 = 3/32 = 0.09375 -- very close to .1

Note that 3/30 would be exactly a tenth.

Note that the more digits we add, the closer we can get. For some numbers, we will be able to reach exactly the value we want. For others, we will be unable to. This is known as the level of precision.

Fortunately, programming languages take care of almost all the work associated with this, so it will not be something you have to worry about. However, understanding how this works will help you to understand upcoming lessons.

The last thing we need to address is a question you are sure to have: If we can only store 1s and 0s, how do we represent a radix point in binary? The answer is, you don't. Instead you specify the number of bits that will be to the right of the radix point, and the number of bits to the left.

If you wanted to store the value 6.5 in the computer, which is effectively: 0110.1000, all you need to do is store this: 01101000 and then instruct your program to treat the first four digits as being part of the whole number, and the last four digits to be part of the fractional number - to the right of the radix point. We will get into how this is done in a future lesson.

Lesson 6.4 : The Basics of Numeric Data Types in C

The video for this lesson can be found here: http://youtu.be/D8M6RifBoLk

Even though this lesson concerns the C programming language, this lesson, and all others, are applicable to all programming languages. Do not skip it (or any lesson) because you think it will not apply to some other language you are learning.

In previous lessons we have learned about many of the different ways you can represent a number as binary.

We have looked at integers, signed integers (that is, integers that can be positive or negative), and we have some understanding of fractional numbers and how they are stored in binary.

We also learned that to store a number in binary, we have to give it a set size and this size constrains us to certain maximum values. For example, if we allocate 4 bits of storage space, we are limited to numbers no larger than fifteen.

However, if we choose to store a signed number, then we are limited to values of:

-7 to +7

So when storing a number in your computer, there are the following things you have to decide:

1. How much space will the number occupy in memory?
2. What kind of number?

For example, an integer, or a signed integer, or a fractional number. All of these would be stored differently in binary as you have learned in previous lessons.

Every programming language has a way for you to specify the size in bits that a number will occupy, as well as what kind of number it is.

In C, there are specific keywords that are used to define these different kinds of numbers and their sizes. We will go over sizes a bit later, but for now lets just discuss the kinds of numbers.

* **int** : Just a normal integer, a whole number - which can be positive or negative.
* **signed int** : Same as the above.
* **unsigned int** : Same as an integer, but has no "sign bit", so only positive numbers are allowed.
* **float** : Will support fractional values, and also has a "sign bit" for positive and negative numbers.

Now, I should point out that what we have learned in the previous lesson is only part of the picture about how fractional values work, and we will get to more of this later.

You need to memorize these data types: int, signed int, unsigned int, float so that you will recognize and understand them when you encounter them later.

Even though we are talking about the C language, these names of number types are virtually universal across all languages.

Every compiler is set to allocate a certain number of bits for each data type, and there are a few extra keywords you can add that will increase or decrease the number of bits allocated for such a number. These values are not universal, and may differ between compilers.

By specifying the type of data (int, float, etc) and the size (long, short, etc) you are able to specify the different kinds of numbers you might want to store.

Now, if you see something like this in a C program:

```
unsigned short int total = 5;
```

It won't be a mystery to you anymore.

Lesson 6.5 : The char data type and the basics of ASCII

The video for this lesson can be found here: http://youtu.be/I-mFRR62Jgk

In the previous lesson we learned about numeric data types. The next data type we need to learn is called "char". The word "char" is short for "character" and refers typically to characters of text. Characters of text usually occupy one byte per character. One byte is almost always 8 bits, although there are some architectures for which this may not be true.

Lets imagine I say this:

```
printf("a");
```

We know that somehow "a" is being encoded in binary, but exactly how? The answer is that there is a table of binary values for every character on your keyboard. We are going to explore that in this lesson.

One such table that many C/C++ compilers use is the ASCII table. The ASCII table is simply a table of many different characters which can be represented in a single byte. All the letters on your keyboard (Assuming a US layout keyboard) are ASCII characters.

First, understand that every char data type occupies exactly one byte in memory. The data type "char" has a fixed size of one byte. In this lesson and throughout the course we will be assuming that one byte is eight bits, or eight binary digits. As I stated above, this is not 100% universal, and I encourage you to keep that in mind.

If we consider a byte as eight binary digits, you should be able to answer the question therefore how many possible values it can have and consequently how many possible ASCII character codes there are.

Note that each of these tables will show the hexadecimal values within parenthesis.

First lets look at A through Z.

```
0100 0001 (41) = 'A'
0100 0010 (42) = 'B'
0100 0011 (43) = 'C'
....
0101 1000 (58) = 'X'
0101 1001 (59) = 'Y'
0101 1010 (5A) = 'Z'
```

This is not hard to remember, and it will help you to memorize at least this structure. All you have to remember is this:

1. Each ASCII letter has 8 bits (1 byte),
2. All CAPITAL letters will begin with: 010.
(This means the first digit will be either a 4 or a 5 in hex, ex: 41 = 'A')
3. The last 5 bits start at 1 and work up to twenty-six. Note that the character 'Z' = 5A = 0101 1010

1 1010 = 26 (16 + 8 + 2)

Keep in mind that you are starting at: 0100 0001 (41) and just working your way up through the 26 capital letters.

Now, lets look at the lower-case letters:

```
0110 0001 (61) = 'a'
0110 0010 (62) = 'b'
0110 0011 (63) = 'c'
....
0111 1000 (78) = 'x'
0111 1001 (79) = 'y'
0111 1010 (7A) = 'z'
```

Notice that while all capital letters begin with 010, all lowercase letters begin with 011. You can also therefore see that this bit:

00**1**0 0000

is a FLAG for denoting uppercase or lowercase letters. This means that if you have an uppercase letter, you can make it lowercase by turning that flag on. If you have a lowercase letter, you can make it uppercase by turning that flag off.

For now, memorize this: capital letters begin with 010, lowercase letters begin with 011. The last five bits start at 1 for A/a and go through the 26 letters to Z/z.

Please remember that ASCII is only one way that you can encode characters. We will learn about others throughout this course. Also, remember that C is not required to use ASCII for encoding. The specifics concerning this are beyond the scope of this lesson, and will be discussed in greater detail later in the course.

Lesson 6.6 : The numbers on your keyboard as characters

The video for this lesson can be found here: http://youtu.be/7Ak8DN4NeE4

This is an important lesson for a number of reasons. First, it is an easy beginner misunderstanding to not realize that numbers on the keyboard are not treated as actual numbers by the computer.

It turns out that just as capital and lowercase letters are encoded in a special binary format, the same is true for numbers. First, let me show you the table, and the rules for this process:

As in the last table, the second column values are in hexadecimal.

```
0011 0000 = 30 = '0'
0011 0001 = 31 = '1'
0011 0010 = 32 = '2'
...
0011 0111 = 37 = '7'
```

```
0011 1000 = 38 = '8'
0011 1001 = 39 = '9'
```

Here you should already be able to see the structure of the number characters. All of them start with 0011 (3 in hex), and then you go from 0 to 9 in the last four bits.

Lets review this in the context of capital and lowercase letters:

```
Capital letters:

0100 0001 ('A') through 0101 1010 ('Z')

Lowercase letters:

0110 0001 ('a') through 0111 1010 ('z')

Numbers:

0011 0000 ('0') through 0011 1001 ('9')
```

This is just about all the ASCII you will ever have to know. The most important thing to understand in this lesson by far is this:

The character '4' is not at all the same thing as the number 4

And this goes for all characters.

However, as you can see from the above table - translating a character from ASCII to a real number is not very hard at all. If:

```
0011 1000
```

Is the character for the number '8', then how do we convert it to the ACTUAL number eight? We just make the first four digits all 0s. OR we can choose to just ignore the first four digits, and look only at the last four. In this way,

```
0011 1000
```

would become simply:

```
0000 1000
```

which is the actual number 8.

Please note that this lesson applies to ASCII. As I stated in the last lesson, ASCII is one of many ways to encode characters, and you should not assume that this is universal. The purpose of this lesson is to show you that even numbers have to be encoded as characters, and ASCII is one way this is done. We will explore this in greater detail later.

Lesson 6.7 : About maximum values of unsigned integers

The video for this lesson can be found here: http://youtu.be/abk_jxFbudI

So far, we have learned about a variety of data types. While the focus of these lessons has been the C programming language, I want to emphasize that these lessons are applicable in every programming language.

Fundamentally we learned that every number as well as every character of text is encoded in binary in a certain way and then stored in memory. We learned that you can specify exactly what type of format you wish to use with certain keywords such as "signed", or "float".

We learned in a previous lesson that the number of bits you have available for a number determines how big of a number you can have. For example, if you are limited to three bits, you could never have a number greater than seven.

Each data type has a set size in bits. This also means that each data type has a maximum number it can hold. If you are using a "sign bit", then there will be a maximum positive number and a maximum negative number it can hold.

Figuring this out is actually quite simple. There are three possible cases to consider:

"unsigned", "signed", or "float"

In this lesson, we will be looking only at "unsigned" data.

If a data type is unsigned, that means that it will only be positive numbers. This is easiest because figuring out the maximum size is simply two to the power of the number of bits - then subtract one.

For example, if we have 3 bits available, the maximum number we can hold is seven. That is because two to the power of three is eight. Eight minus one is seven.

Why do you subtract one? Because you always start counting at zero. If I count: 0, 1, 2, 3, 4, 5, 6, 7: I have counted eight total numbers including zero, but the maximum value is still seven - not eight.

So because you always start counting from zero, the maximum number of possibilities that can be contained in a set number of bits will always be exactly one greater than the highest possible unsigned integer value that can be contained in that same number of bits.

Now, lets see this in action:

If you have a numeric data type which is contained in two bytes (16 bits), then the maximum number of possible values is:

    2^16 = 65,536

Now, if we start at zero and count upward, the maximum value we could ever get to is exactly one less than this:

    65,535

Keep in mind, that in binary this value would be simply:

```
  1111 1111   1111 1111 (FF FF in hex)
```

This is another way of saying that with 65,535 we have used up all of the available 16 bits available for storage, and we simply cannot hold a larger number.

You may wonder why you need to know this, and the answer is very simple: Any time you ever have a mathematical calculation in your program that exceeds the maximum value of a given data type, your program will fail. In later lessons I will go into the specifics of what these limits actually are.

Remember that the maximum values as well as the size in bytes of each C data type may differ between C compilers. There are however certain requirements that all compilers must follow. We will explore that in greater detail later in the course.

Lesson 6.8 : Minimum and maximum values of signed integers

The video for this lesson can be found here: http://youtu.be/UHG3YCver1g

This lesson is a bit more intense than most, so please take your time and read through this carefully.

In Lesson 24 we learned how to calculate the maximum values that can be stored in a set number of bits. For example, we learned that if given 16 bits of storage space (two bytes) the highest integer number that can be stored is: 65,535.

Here I want to expand on this knowledge a bit. Lets consider that we still have two bytes to work with, 16 bits, but we want to store both positive and negative numbers. You should remember from previous lessons that this means we need to have a "sign bit".

The purpose of this lesson is to learn how to calculate the minimum and maximum values for "signed" integers.

You should remember from a previous lesson that using a sign bit cuts in half the total numbers that can be represented, but you are now able to count in two directions - both positive and negative.

Lets examine the situation when we have four bits to work with, and one of those bits is a "sign bit". Remember from the previous lesson that the total number of possibilities that can be stored in four bits is: 16 possibilities.

If we were to use one of those bits as a "sign bit", we now allow for two sets of eight possibilities. One set positive, and the other set negative.

If you need any review on this concept, please see lesson 18 and feel free to ask any questions you need to.

If you remember from that lesson, we had a curious situation. We had a positive zero, and a negative zero. This is not very efficient since we are wasting a value, since we only need to represent "zero" in one way. Lets look at that table again:

```
0000 = 0
0001 = +1
0010 = +2
0011 = +3
0100 = +4
0101 = +5
0110 = +6
0111 = +7
1000 = 0 <--- this is extra. We do not need it.
1001 = -1
1010 = -2
1011 = -3
1100 = -4
1101 = -5
1110 = -6
1111 = -7
```

If we choose to not use zero twice, then that frees up one extra value that we can use. We could set this extra value to anything we want. If we wanted to, we could set it to -8. Then our method would make it possible to show all numbers from -8 through +7. Also, this makes sense since for the extra value, the "sign bit" is already set to indicate this is a negative number.

Of course, this creates problems if you want to add or subtract based on this system, but we will get to that later. The exact mechanism by which this is done is still outside of the scope of this lesson.

Notice from the above table that exactly half of the total possibilities have the sign bit set to "positive", and exactly half of the total possibilities have the sign bit set to "negative".

Lets look at this in action. When we are considering two bytes, that gives us 65,536 possibilities. If we cut that in half, we get:

65,536 / 2 = 32,768

If we allowed for a "positive zero and a negative zero", that would mean we would have exactly 32,768 numbers where the "sign bit" was set to "positive", and exactly 32,768 numbers where the "sign bit" was set to "negative".

Since in both cases we would start counting at zero, that would mean our maximum positive value would be: 32,767 (just subtract one) and our minimum negative value would be: -32,767.

However, since we are NOT suggesting a "negative zero", we are able to have one extra negative number. Therefore, the final case is:

Any value from -32,768 to +32,767

Make sure you understand that before proceeding. Now we can develop a simple formula for this for all signed integers:

The minimum value will be:

negative ((2 to the power of the number of bits) divided by two)
or: 2^n / -2 where n = number of bits.

The maximum value

((2 to the power of the number of bits) divided by two) minus one
or: (2^n / 2) - 1 where n = number of bits.

Briefly, I want to touch on one more thing. I mentioned before that the method that I am showing you concerning "signed" and "unsigned" integers is actually different than how it is actually done.
One thing I want you to realize is that no matter how it was done, there are still always going to be a total of 32,768 values where the sign bit is set to positive, and 32,768 values where the sign bit is set to negative. This example uses two bytes of course, but the same applies no matter how many bytes of storage are used.

UNIT 7 : Variables and more

Lesson 7.1 : Introducing variables

The video for this lesson can be found here: http://youtu.be/sFp9f2QD_PM

We have learned previously that you can store all sorts of data in memory, including numbers, text, or pretty much anything you want. We have learned that to store anything you have to specify its size (using certain keywords like "short", etc.), and specifying its format (using keywords like int, char, etc.).

We have also learned that every programming language gives you the ability to give simple plain-English names to any data that you store in memory. Now we need to take this knowledge to the next level.

Whenever you create data and give it a simple name, that is usually called a "variable". For example I might tell my programming language that I want some data to be an integer, that I want to call that data "total", and that I wish to assign it some value like 5. I have now created a variable.

Lets suppose I want to do exactly this:

First, what kind of data type do I want? Well, it is a small number, and it is positive - so a "short unsigned int" makes perfect sense. Now, I have to give it a name. I will call it "total".

Now I have to give it some value. Here is how I do all of these steps:

```
short unsigned int total = 5;
```

Now, here are a few questions you need to be able to answer, along with their answers:

1. What is the variable's name? total

2. What is the data type for this variable? unsigned short int
3. Can negative numbers be stored in this variable? No

If you have been following all the lessons up until now well enough, you should be able to understand how this variable actually looks in binary, stored in memory. We know it is two bytes long, that is sixteen bits. We know that the binary for 5 is 0101. If we assume that this variable would take up two bytes, then it would look like this in memory:

```
0000 0000 0000 0101
```

Notice all the unused space. Because 2 bytes can hold up to 65,536 values, there is a lot of wasted space. I want to explain a few important facts concerning variables:

Since I have assigned this variable two-bytes, it will always contain 16 bits. These 16 bits will always be understood to be a number between 0 and 65,535. If I perform some mathematical operation that results in a number greater than 65,535 , then as we have seen in earlier lessons the result will be a wrong answer because no value that big can fit in 16 bits.

Always remember this: From the time you create a variable through to the end of a program, it will always be constrained to the size you gave it, and it will always be understood to have the data type and format that it had when it was first created.

Please be aware that "unsigned short int" is not required to always take up exactly two bytes. This as well as the size of data types in general may differ among C compilers. In this lesson, I used two bytes to illustrate the material presented.

Lesson 7.2 : The connection between function return values and variables

The video for this lesson can be found here: http://youtu.be/96VNEzoZg9c

This is a very important lesson, and is in fact one of the most important principles used in programming.

If you have observed the lessons up until now, you have noticed that when we created a function, the main() function, we gave it a data type just as if it was a variable. We wrote:

```
int main(void) {
```

We learned that any function can return a value. While I have not officially addressed it in a lesson until now, I want to point out that the printf() function returns an integer - which is the number of character that were printed.

Every function that returns a value always returns that value according to a specific data type. That means that the same data types available for creating variables such as short, int, char, etc. are also the same data types available for defining function return values.

Lets examine the following code:

```
printf("Hello Reddit!");
```

We know that printf() returns a value, the number of characters printed. In this case, 13. Remember that this is an int, because the function printf() has a return value of type int. Remember also that "int" is actually "signed int".

Lets imagine I create a variable of type signed int:

```
signed int total_characters = 0;
```

Now, I have created a variable called total_characters that can hold any value, so long as it is of the data type: "signed int". Notice that it should not be used to store any other data type -- only "signed int".

In addition to this variable, I also have the printf() function, which whenever ran will return a value that is also of the type "signed int". Any time I run the printf() function, it will return a "signed int" which will contain the number of characters that were printed.

What I want you to notice is this: The function printf() and the variable: total_characters are compatible. They share the same data type.

Whenever a variable is compatible with a function's return value, the variable can be assigned the return value of that function. What this means in simple terms is that because printf() returns a "signed int", and because the variable:

    total_characters

is defined as a "signed int", then this means that this variable can be assigned whatever value was returned by printf().

This is true for all variables and functions. If I create a variable like this:

```
unsigned short int total = 5;
```

Then this variable, "total", can now store the return value of any function whose return type is "unsigned short int". If I say:

```
char some_character = 'a';
```

Then this variable can store the return value of any function whose return type is "char".

This is only half the story however. This next part is equally important:

Whenever a function returns some value of a certain data type, that entire function can be used anywhere that this particular data type is expected just as if it were a variable.

In other words, since printf() returns an "int", I can use an entire printf() function anywhere that I can use an int. Anywhere at all. I can even perform mathematical operations like this:

```
int cool_trick = 0;
cool_trick = 5 + printf("Something");
```

Now the integer "cool_trick" will contain the value of 14. That is because printf() returns a 9 since it prints 9 characters. The 5 and 9 are added to get 14.

Notice also that "Something" still gets printed. This is because the printf() function still executes, and still does whatever it is designed to do. This is important for later, so keep it in mind.

I encourage you to experiment with this. Just take the first program you wrote, and modify it so that you use these concepts.

You can use printf() to print an integer like this:

```
printf("Some integer is: %d", variable_name_here);
```

The %d gets automatically replaced by whatever variable_name_here is. For example:

```
int total = 5;
printf ("The total is: %d", total);
```

Remember, ask questions if you need any help or if anything is unclear.

This lesson is for illustrative purposes. There are certain requirements as to what data types you can and should use as the return type for functions. For example, specifying an "unsigned short int" as a return type would generate C code that not all compilers would accept. These details are beyond the scope of this lesson, and will be discussed in greater detail later in the course.

Lesson 7.3 : Terminating strings of text and other data

The video for this lesson can be found here: http://youtu.be/1T_krhrLlKU

This lesson pertains to strings of text that are encoded as ASCII. There are other ways to encode text which are not covered in this lesson. However, the principles taught in this lesson are equally valid in such cases.

Earlier we learned about ASCII, and the different ways that characters are encoded inside of your computer. In this lesson we will look more closely at how text is stored in memory.

First, recall that every ASCII character is encoded in exactly 8 bits.

Recall that capital letters always begin with 010, with the final 5 bits counting from 1 to twenty-six corresponding to that letter of the alphabet. Lowercase letters do the same, but begin with 011. Finally, we learned that numbers begin with 0011 and the final four bits will give you the actual number.

You should have enough information then to understand that the text: "123" would be encoded thus:

```
0011 0001 : 0011 0010 : 0011 0011
```

I used the : character to separate bytes to make them easier to read.

Imagine now that I have some function that can print ASCII characters, and I point that function at this sequence of three bytes. It prints the first character and I see a "1" appear on my screen. Immediately after I see a "2" followed by a "3".

Think about this for a moment. When I put the sequence of three bytes into memory corresponding to the characters "123", it got placed a specific location in memory. Lets imagine what this might look like:

```
0011 0001 : 0011 0010 : 0011 0011
(our three bytes -- this is "123" encoded in ascii. 0011 0001 = "1", etc.)

0011 0001 : 0011 0010 : 0011 0011 :: 0101 1111 : 1001 0101
(our three bytes in memory -- the first three bytes are our "123")
```

You are probably asking what is this second set of two bytes after the :: in the above example. It is whatever just happens to be in ram following the three bytes we defined as "123". It could be left over data from some program that ran earlier. It could be absolutely anything. Always assume there is something in ram following any data you store.

Whenever you store some data in ram, there will be other data immediately before and immediately after it.

I presented the three bytes of "123" next to this mess of binary using a :: separator, so it was easy to understand. Lets see how it would look as individual bytes:

```
0011 0001 : 0011 0010 : 0011 0011 : 0101 1111 : 1001 0101
```

Can you tell that our "123" sequence is different from the two bytes that immediately follow it? No. That was the subject of an earlier lesson, you cannot distinguish data types just by looking at the binary. In fact, neither can your program. Neither can printf().

If I pointed a function like printf() at these five bytes, and told it to start printing - it would print our three characters "123" just fine.. but then what? It would keep going! Why? Because these extra bytes could be rendered as ASCII, regardless of what they were originally. What would happen then?

Have you ever seen a lot of strange characters get printed to your screen as a giant mess of weird letters? This happened because your computer started printing binary sequences it thought was ASCII, but which turned out to be who knows what.

Any sequence of eight bits can be rendered as some ASCII character, and this includes many especially strange characters that have nothing to with letters or numbers. Therefore, we must define some way that we can know where to stop printing characters.

I am presenting this lesson in the context of text strings, but this same principle applies any time you are processing data of a certain length. If you do not specify where to STOP, you may just keep on processing the data.

For example, just as it is possible to keep on printing sequences of 8 bits as though they were ASCII characters, it is also possible for a music-playing program to start trying to play what it thinks is music, which turns out to be something entirely different. The result of course would be some strange music.

So here we learn an important concept: You must always define a stopping point for any data. Always.

There are two ways you can do this:

1. Pre-define a set length. In our earlier example with the string of text "123", we could choose to define a set length of three bytes. Then we can tell a function to print only three bytes. It will stop knowing that it cannot go past that.
2. Define a character at the end of the text string that means "stop". Typically this is done using the binary sequence:

```
0000 0000
```

Effectively what this means is that we can have a string of text as long as we want, but we have to remember to put a special "all zeroes" byte at the end. Then we tell the function to stop when it reaches the "all zeroes" byte. The technical term for this kind of string of text is a "null terminated text string". We say "null terminated" because "null" is another name for "all zeroes".

So, how would our string of text "123" appear if we apply the concept of a null terminated text string? Like this:

```
0011 0001 : 0011 0010 : 0011 0011 : 0000 0000 :
```

Lesson 7.4 : More about printf() and introduction to placeholders

The video for this lesson can be found here: http://youtu.be/3CvPNW0hj1s

In an earlier lesson you learned how to do this:

```
printf("Hello Reddit!");
```

You learned that you send a string of text to the printf() function as a parameter, and then that function displays the text to the screen. We also learned that printf() returns an int value, which will be the number of characters printed.

Now we are going to learn that printf() is actually much more powerful than what we have learned up until now.

It is possible to use printf() to display not just set strings of text, but also to display other kinds of data, including integers.

In an earlier lesson I explained that you can do this:

```
int i = 5;
printf("The variable i is set to: %d", i);
```

And the result will be:

The variable i is set to: 5

Lets talk about how this works. First, notice that the %d never gets printed. This is because the printf() function knows that if you put %d it is intended to be a "place holder" for some other value.

Remember that we said before that matching data types is very important. Any time any function is going to operate on some data, it must know what kind of data it is, how big it is, and how it is formatted. Why? Because as we have learned in previous lessons the same binary can mean multiple things. A sequence of eight 1s and 0s might be an int or it might be a char, or anything at all.

printf() allows you to specify different place holders depending on the type of data of what you want to print. You must always match the correct data type to the correct place holder. We learned that %d means "integer", but lets look at some others:

%d or **%i** : signed integer
%u : unsigned integer
%c : single character
%s : A string of text like "Hello"

We will learn more later, but for now this is enough to proceed. With this information you should now be able to experiment with various data types we have already looked at and see how to use printf() to display different results.

UNIT 8 : Arrays and Pointers

Lesson 8.1 : Introducing arrays and pointers part one

The video for this lesson can be found here: http://youtu.be/9gq9uTe2XaA

This function assumes that all text strings are encoded as ASCII. Another assumption being made is that the "unsigned short int" data type is two bytes in size. This is not always the case, so you should be aware of that when reading this lesson.

In an earlier lesson we learned that we can use the printf() function to display text.

Lets briefly look at this text: "abc123"

Recall that it is encoded in memory like this:

```
0110 0001 : 0110 0010 : 0110 0011 : 0011 0001 : 0011 0010 : 0011 0011 : 0000 0000

   "a"    :    "b"    :    "c"    :    "1"    :    "2"    :    "3"    :
```

We store text in memory by creating a "train" of ASCII characters, then we end that train with a "null" character.

This entire "train" is stored in memory exactly as I showed above. Every character immediately follows the character before it. In computing, the word used for this is "string".

A "string" is one of the simplest forms of something called an "array". An array is a collection of data elements where each data element has the same data type. For example, in a string of text, you have a collection of data elements (characters) where each data element in this case has the data type char.

Arrays are incredibly useful in programming, and we will get into them more later on. Arrays are also often a source of misunderstanding for beginners, so I want to cover a few important points.

Remember from an earlier lesson that you never have to worry about the actual address in memory where a variable is stored, because this is done for you by the programming language. Also remember that you can give plain English names to variables.

Lets consider this code:

```
unsigned short int total = 5;
```

What is "total" ? It is both a way to refer to the address in memory where the value 5 is stored, and it is a way to refer to the value 5 itself.

Every variable has some address in memory. This address in memory is not the value of the variable. Theoretically, the variable "total" might exist at any of billions of possible addresses in memory - you have no idea which one. All you know is that indeed at some location in memory you will find this sequence:

some address in memory : 0000 0000   0000 0101 <--- This is our two-byte "unsigned short int total"

Now, for the sake of this lesson, lets give your computer a massive downgrade in RAM. Instead of you having gigabytes of RAM, you now only have 16 BYTES of ram. Lets examine how this would look.

On the left, I am going to put the address in RAM. On the right, I am going to put its contents - we are going to start with a blank slate of all zeroes to make this lesson easier.

Each address will be 4 bits in size (which gives us sixteen possible addresses in memory). At each address, there will be one BYTE of actual data stored - eight bits.

```
0000 : 0000 0000
```

```
0001 : 0000 0000
0010 : 0000 0000
0011 : 0000 0000
0100 : 0000 0000
0101 : 0000 0000
0110 : 0000 0000
0111 : 0000 0000
1000 : 0000 0000
1001 : 0000 0000
1010 : 0000 0000
1011 : 0000 0000
1100 : 0000 0000
1101 : 0000 0000
1110 : 0000 0000
1111 : 0000 0000
```

Now, lets imagine also for the sake of this lesson, that "unsigned short int" is only one byte in size, instead of two. Lets re-consider the following code:

```
unsigned short int total = 5;
```

Now, your programming language is going to choose somewhere in RAM to put this. This is as far as you are concerned entirely arbitrary. You have no idea where in RAM this value 5 is going to be placed.

Lets imagine that the variable "total" gets put in the memory address "eight" in our sixteen bytes of ram. Here is the new ram table with this modification:

```
...
0101 : 0000 0000
0110 : 0000 0000
0111 : 0000 0000
1000 : 0000 0101 <---- here is where we stored the variable "total"
1001 : 0000 0000
1010 : 0000 0000
1011 : 0000 0000
...
```

We can see therefore that the variable "total" actually refers to two different values. Five, and Eight. Eight refers to the location in ram where "total" is stored. Five refers to the numeric value stored at that location.

Lets go back to this statement:

What is "total"? It is both a way to refer to the address in memory where the value 5 is stored, and it is a way to refer to the value 5 itself.

This should make more sense to you now. We will talk more about this in the next lesson.

Lesson 8.2 : Introducing arrays and pointers part two

The video for this lesson can be found here: http://youtu.be/zYfdGxXx5y8

For the purpose of this lesson, assume that all text characters are encoded as ASCII.

In the previous lesson I showed you how to clearly visualize how variables are stored in memory. I also showed you that a variable really should be thought of in two different ways: the location of that variable in memory, and the actual value of the variable. Also, I showed you that these two values are not at all the same.

Now we are going to explore this further, and learn about how to use the memory addresses where variables are stored in a practical way. This will introduce you to the concept of a "pointer", which is a way to keep track of the address in memory of some data you are working with. We will talk about pointers more in future lessons.

Lets again consider the string of text "abc123". Lets review how it is stored in memory:

```
0110 0001 : 0110 0010 : 0110 0011 : 0011 0001 : 0011 0010 : 0011 0011 : 0000 0000

   "a"    :    "b"    :    "c"    :    "1"    :    "2"    :    "3"    :
```

Let's now store the string of text "abc123" into our 16-byte RAM from the previous lesson. Lets say that we will store it at position "eight" in RAM. Like this:

```
...
1000 : 0110 0001 <--- "a"
1001 : 0110 0010 <--- "b"
1010 : 0110 0011 <--- "c"
1011 : 0011 0001 <--- "1"
1100 : 0011 0010 <--- "2"
1101 : 0011 0011 <--- "3"
1110 : 0000 0000 <--- the null termination
...
```

I want you to observe the following fact: Every single character in our string of text has its own address in memory!

Even though our string as a whole starts at position 1000 (eight), each character in the string occupies a different location in memory. In fact, you could say that position 1000 (eight) only truly refers to the first character in the string, the "a" character.

Now I want you to do a mental experiment. On your own, follow these steps:

1. Start with the address 1000 in our 16 byte ram.
2. Say the character stored at that location.
3. Go to the very next address.
4. Repeat this process of saying characters until... the null termination is reached.

You just simulated exactly how the printf() function works!

Lesson 8.3 : Introducing the pointer data type

The video for this lesson can be found here: http://youtu.be/syll7VbleIg

For the purpose of this lesson, assume all text characters are encoded as ASCII.

Lets examine the following code:

```
char my_char = 'a';
```

Here I have stated that I am creating a new variable called my_char. I have stated to use the data type char for this variable. Now the variable my_char can hold any single-byte ASCII character I desire. In this case, I set the value to the character 'a'. This means that somewhere in memory, I have this:

```
address of my_char : 0110 0001 <--- "a"
```

We do not know what the address is, only that there is one. An address in memory is just a number, a sequence of 1s and 0s no more special than any other binary sequence.

Lets suppose that the variable "my_char" sits at position 1000 (eight) in our 16-byte RAM from the previous example:

```
...
0111 : 0000 0000
1000 : 0110 0001 <--- "a"; Here is my_char at position eight (1000)
1001 : 0000 0000
...
```

Notice that my_char has a value as well as an address. The value is 'a'. The address is 1000 (eight). Any time you create any variable, it will have a memory address.

Notice that the memory address of an ASCII character 'a' is not an ASCII character - it is a memory address. The data type of your variable is not the data type of the memory address where it is stored. A memory address does not care what kind of data is stored in it. I want to illustrate this using our 16-byte RAM example:

I am going to add some data before and after the 'a' to illustrate this.

```
0111 : 0000 0101 <--- This is the actual number 5
1000 : 0110 0001 <--- "a"; Here is my_char at position eight (1000)
1001 : 1100 0100 <--- This is actually the start of some unrelated data.
```

There are three totally different kinds of data above. However, the memory addresses still work the same way regardless of the data stored at that address. There is absolutely no direct relation between the memory address and the data stored at that address.

If I asked you, "Please show me the binary sequence that is stored at position 1001", you have all the information you need to give me those eight bits. The same is true if I asked you, "Please show me the binary sequence that is stored at position 0111", or any other location in our 16-byte RAM example.

Notice that while you can give me the eight bits of data stored at that location, you could not tell me whether it was to be a character, or a number, or something else - as we have learned in previous lessons.

Remember that we have learned that any binary sequence cannot be understood until we give it a "data type". A memory address is a binary sequence, just like any other. Therefore, memory addresses require a "data type" just as ints, or chars, or any other kind of data that might exist.

In computing, there is a term to describe a data type that is designed to hold memory addresses. This data type is called a "pointer". Any time you create a variable of the data type "pointer", you are creating a variable designed to hold a memory address.

Lesson 8.4 : How to create a pointer

The video for this lesson can be found here: http://youtu.be/Wlq-r4FTAUc

In the previous lesson we learned that it is possible to create variables that are designed to hold memory addresses. In this lesson we are going to explore how to create such a variable.

As we discussed, every memory address is the same regardless of what kind of data it contains. However, different data types occupy a different number of bytes in memory. For example, characters occupy one byte, short int might occupy two bytes, int might occupy four bytes (depending on the compiler), etc.

Lets look at the way an unsigned short int might be stored in our 16 byte ram example. In this case, we are going to assume that an unsigned short int takes up two bytes.

Lets imagine this code:

```
unsigned short int total = 50250;
```

So we have stated that that the variable total will contain a value of 50,250.

How would this look in binary?

```
1100 0100 : 0100 1010

2^15 + 2^14 + 2^10 + 2^6 + 2^3 + 2^1
32,768 + 16,384 + 1,024 + 64 + 8 + 2 = 50,250
```

Remember that this unsigned short integer takes up two bytes. Therefore, how would it be represented in memory? Let's store it at position eight in our 16-byte ram. Here is how it would look:

```
...
1000 : 1100 0100 <--- first half
1001 : 0100 1010 <--- second half
...
```

What I want you to notice is that obtaining the 8 bits at position 1000 is not enough to obtain the full value of this unsigned short int variable. It is however enough to start with. If we know that the unsigned short int starts at position 1000 then we know enough to get the value, we just have to remember to grab 16 bits instead of 8.

As you can see, you will get very different results if you expect there to be 8 bits as opposed to 16 bits. In our earlier example I said, "What is the value stored at the memory address 1000". You can see now that this is not enough. I need to really be more specific and say, "What is the sixteen bit value stored at memory address 1000.

Now for the next half of this lesson.

You do not create a pointer only to store a memory address, but in order to give you a way to see and work with the data at that address. For example, it would be utterly useless if I were to create a pointer to location 1000 in ram and have no method by which I could say, "What is at that location?".

When you create a pointer, you must specify how big the data is you are planning to use the pointer for. In other words, you must specify the type of data you are planning to use the pointer for.

If you are planning to use the pointer to look at ASCII characters in memory, then you need to specify, "I plan to use this pointer for type char". If you are planning to use the pointer to look at data of type unsigned short int, you must specify, "I plan to use this pointer for type unsigned short int".

When you create a pointer, you must specify the data type for what it will be pointing to.

Now, there is one more detail we have not covered. How do you tell a programming language like C that you want to create a pointer? In C, you simply put an asterisk in front of the variable name. That's it.

Lets look at this in practice. Note that in the below example, both lines are the same. C does not care about the space.

```c
int * my_pointer;
int *my_pointer;
```

There you see I have just created a pointer. Now, what data type do I expect it to point to? If you said int, you are of course correct. Why did I have to specify any data type? Because when later I want to say "What is at that location", C needs to know how much data from that location to give me.
In the next lesson we will explore pointers more, including seeing how to assign actual memory addresses to them.

Lesson 8.5 : Assigning a value to a pointer

The video for this lesson can be found here: http://youtu.be/HQQnWwgMC7E

As we discussed, a pointer is a variable that holds a memory address. You can think of "pointer" as the "data type" for memory addresses in general. When a pointer contains the memory address of a variable, it is said to "point" to the variable located at that address.

Lets suppose we want to create a pointer that will "point" to an unsigned short int. In other words, we want to create a variable (the pointer) of data type "memory address", and we want it to contain the memory address of a... unsigned short int variable.

To do this you would write:

```
unsigned short int *some_pointer;
```

This creates a pointer called "some_pointer", and we have stated that we will be using this pointer to hold a memory address for a variable of type unsigned short int. Keep in mind that a single pointer can only hold one memory address at a time.

Right now of course, this pointer effectively has no value. Because this pointer is designed to hold the memory address of an unsigned short int, then it should be obvious that we cannot use this pointer until we can give it a value. That value needs to be the memory address of an unsigned short int.

To make this possible, let's create a couple unsigned short int variables:

```
unsigned short int height = 5;
unsigned short int width = 10;

unsigned short int *my_pointer;
```

Notice I did not assign a value to the variable "my_pointer". What you should understand at this stage is that "my_pointer" is a variable designed to hold the memory address of any unsigned short int. We have not given it a value yet.

Remember that a pointer is useless if it does not contain a memory address.

Lets say that we want to give the pointer "my_pointer" the value of the memory address for the variable "width" in our above example. This means we need to have some way that we can write this line of code:

```
my_pointer = "address of" width;
```

This is not actual code, but it describes what we need to do. Now let's simplify this a bit. Rather than typing the words "address of", lets use a character on the keyboard to mean this same thing. Let's choose the & character.

In other words, lets just say that the "&" character means "address of" - just so we can write this line of code out simpler.

Now, if we wrote the same exact line of code having the & character in place of the words "address of", we might write this:

```
my_pointer = &width;
```

Believe it or not, that is exactly how it is done. In C, the & character literally means "address of". Any time you want to set a pointer to the address of some variable, you simply put this. Note that both of the below lines are exactly the same, white space doesn't matter:

```
pointer = &variable;
pointer = & variable;
```

This literally translates to:

pointer = the memory address of "variable";

Why do we want to do this? Because now we can look at that memory address, and obtain the value that is there. Now I can imagine you saying, "Sure, but we can do that already without pointers."

Yes, you can. But only for single variables of a given data type. This is not how real code works. In real code, you must be able to process large data structures, not simply an int or a char. A large data structure could be music, or graphics, or something else.

There is no data type built into C or any language for something so complex. Therefore, the way to process it is to set a pointer to the start of the data in memory, and then you process it by moving the pointer through the data as you perform actions on the data (like playing it to your speakers for example).

Lesson 8.6 : Getting the value stored at a memory address

The video for this lesson can be found here: http://youtu.be/iLY90p7Pit0

In the last two lessons we learned how to create a pointer, and how to assign a pointer the memory address of some variable.

Lets examine the following code:

```
int total = 5;
int *ptr = &total;
```

You should know from the previous lessons exactly what is happening here, but lets review it. First we are creating a variable called "total", that is of the type signed int, and assigning it the value 5. Secondly we are creating a pointer called "ptr" and giving it a value of... the memory address of "total".

Because the value of total is 5, if we were to run this command:

```
printf("The total is: %d", total);
```

We would see this output:

The total is: 5

Now, we said before that a pointer is useless if we do not have some way to read and use the data at the memory address the pointer refers to. So lets talk about how to do this.

Recall that C has an "address of" operator, the & character. Whenever we put the & character in front of a variable, we are saying "The address of". So if we write: &total this means "the memory address where total is stored".

Now, if I have created a pointer it makes sense that I will want to see what is actually stored at that address. In other words, I need a line of code that reads something like this:

```
printf("The total is: %d", <what is at the address of> ptr);
```

Recall that "ptr" is our pointer, and it points to total. The value of total is 5. This means that we desire the following output:

The total is: 5

So how do we achieve this goal? In the last lesson we learned that you can use the & character to mean "address of". It turns out you can use the * (asterisk) character to mean "what is at the address of". In other words, *ptr means "Whatever is stored at the memory address."

What memory address? The memory address that is stored inside of ptr. In this case, the memory address of the variable "total". Therefore, what do you think *ptr is equal to? If you said 5 - you are correct.

Therefore, to get the result we are looking for in our printf() statement, we would write this:

```
printf("The total is: %d", *ptr);
```

In this case *ptr means this:

ptr is a pointer to an integer. ptr has some value, a memory address (since it is a pointer). The memory address in ptr points to a variable, an integer called "total". The value of "total" is 5. Therefore, the value of *ptr is also 5.

The technical term for the * operator in this case is the "Dereference operator". The term "dereference" means to look not at the memory address, but at what is contained in that memory address.

Now, a very important clarification. In this code:

```
int total = 5;
int *ptr = &total;
```

The * character in this code does not mean "what is at the address of". There is a difference between when you create the pointer variable (which we are doing above) and when we use the pointer variable.

When you create the pointer variable, you use the character to indicate that you are creating a pointer. When you use the pointer later on, you can put a character in front of the pointer to indicate that you are not talking about the memory address itself, but what is stored there. This is an example of using the same operator, a * in this case, for two different purposes.

Lets consider this code:

```
int total = 5;
int *ptr = &total;
```

Only now that the ptr variable has been created can you use *ptr meaning: "The value stored at the memory address." Here is a summary of the different possible ways you can use the & and * operators when it comes to total and ptr in the above example.

1. ptr = This is the variable itself, designed to store a memory address. If it stores a memory address, then ptr refers to the memory address that is stored.
2. *ptr = This refers to the actual data stored at the memory address that ptr holds.
3. &total = This is the memory address of the variable total.
4. total = This is the variable itself, which is set to 5.

Earlier we pointed out that any function which returned a return value of a given data type could be used in place of that data type - anywhere in the program. For example, since printf() returns an int, you can use printf() in place of an int anywhere in a program that an int is expected.

The same concept holds true for pointers. If ptr is set to point at an integer (set to hold the memory address for an integer) for example, then *ptr can be used anywhere an integer is expected.

Why? Because *ptr actually is an integer in every sense of the word. It is the actual binary sequence - the actual integer - that was stored at that location. It has the correct size, data type, and everything.

*ptr is not just "like", or "equal to" the variable total, it is the variable total in every sense.

Lesson 8.7 : Use what you have learned

The video for this lesson can be found here: http://youtu.be/SPVNliAUNyQ

This is not a typical lesson. This is a challenge to you in order to give you the opportunity to apply what you have learned.

Create your own program that demonstrates as much as you can about the concepts you have learned up until now.

For example, use printf() to display text, integers, characters, memory addresses (use %p - see the comment thread on Lesson 35), and anything you want. Experiment with different ideas, and be creative. Also, use pointers.

Post your example programs in the comments on this thread. It will be interesting to see what everyone comes up with.

Be sure to put 4 spaces before each line for formatting so that it will look correct on Reddit. Alternatively, use http://www.codepad.org and put the URL for your code in a comment below.

Have fun!

UNIT 9 : Pointers Continued

Lesson 9.1 : Using pointers for directly manipulating data in memory

The video for this lesson can be found here: http://youtu.be/nFY9gZyB0E8

In an earlier lesson we saw that text is encoded as individual ASCII bytes and stored in memory like a train. We also learned that because each memory address only contains one byte of actual memory, that therefore each ASCII character had its own unique address in memory.

Lets review this by going back to our 16-byte RAM example, and store the simple string "abc123" at position eight (1000) in RAM.

```
...
1000 : 0110 0001 : 'a'
1001 : 0110 0010 : 'b'
1010 : 0110 0011 : 'c'
1011 : 0011 0001 : '1'
1100 : 0011 0010 : '2'
1101 : 0011 0011 : '3'
...
```

Did I make a mistake? I hope you noticed that I forgot to terminate the string with a null (all zeroes) byte.

Now, lets create a pointer called ptr which we will give the address of the first character in the string, the 'a'.

```
char *ptr = <address in memory of 'a'; 1000>;
```

This is of course not real syntax. For now, do not worry about how to actually do this, just understand that I have given the pointer ptr a value of 1000 which is the memory address of the 'a' character in our 16 byte ram.

Now we learned that the * character takes on a new meaning once the pointer has been created. Now we can use our pointer ptr in two ways in the source code:

    ptr = the address in memory of 'a', which is 1000.
    *ptr = 'a' itself, since it refers to "what is at the address 1000"

Notice that we have not created any char variable for the 'a' itself. The truth is, we do not have to. We are starting this example with our 16-byte ram in a specific "state" where the string exists already, so there is no need to create a character variable to hold something that is already in ram.

Up until now we have learned that you can use pointers to look at data in memory. For example, consider the following code:

```
int total = 5;
int *my_pointer = &total;

printf("The total is: %d", *my_pointer);
```

This code should make perfect sense to you. You should also know exactly what the above line of code will output:

```
The total is: 5
```

So here we have an example of using a pointer to "see" what is in memory. Now I am going to show you that you can use a pointer to "change" what is in memory also.

Let's go back to our 16-byte ram example. Here we have the pointer ptr which contains the address 1000 which corresponds to the 'a' character. The 'a' character in this case is the first of the string "abc123".

When we say *ptr, we are saying "The very data stored at the memory location 1000". If you change that data, you change the 'a' itself. Think about it. If we make a change to the data at position 1000, then it will no longer be an 'a'.

In fact, we could change it to anything we want. By using a pointer you can directly manipulate the data inside any memory address, and therefore you can change the data itself.

```
...
1000 : 0110 0001 : 'a' <----- ptr points here
1001 : 0110 0010 : 'b'
...
```

Since we know that ptr points to address 1000, we can change the contents at this address with this line of code:

```
*ptr = 'b';
```

What have we just done? We have written a line of C that reads like this:

    "Replace the binary sequence at position 1000 with the ASCII character 'b'"

After this line of code executes, here is the new state of memory:

```
...
1000 : 0110 0010 : 'b' <----- ptr still points here
1001 : 0110 0010 : 'b'
...
```

We have changed the 'a' to a 'b'.

Where did the 'a' go? It is gone. It is as if it never existed. Since the data itself has been changed in the memory location that 'a' used to reside at, the data that used to be 'a' is simply no more.

This means that if we create a variable and assign it some value, and then use a pointer to later change it, that original value is lost.

Consider this code:

```
int total = 5;
int *my_pointer = &total;

*my_pointer = 10;

printf("The total is: %d", total);
```

What do you think will be the output? Consider what is happening here. We are saying, "Lets create a pointer that contains the memory address of the total variable, and then lets use that pointer to replace whatever was at that memory address with a new value of ten."

This means that the variable total has been changed. The old value of 5 is gone forever, and it now has a new value of ten.
In this lesson you have learned that you can use pointers not only to look at memory directly, but also to change memory directly.

Lesson 9.2 : About changing the memory address stored in a pointer

The video for this lesson can be found here: http://youtu.be/-__3nIqIt4o

Remember that a pointer contains a value, a memory address. This is just a number, a binary sequence, no different than any other number. A pointer has no meaning except for the memory address it contains. If our pointer contains the memory address 1000, then it has no meaning except for the memory address 1000 and the data that resides at that memory address.

Let's look again at the 16-byte ram example from the previous lesson:

```
...
1000 : 0110 0001 : 'a' <--- ptr points here
1001 : 0110 0010 : 'b'
1010 : 0110 0011 : 'c'
1011 : 0011 0001 : '1'
```

```
1100 : 0011 0010 : '2'
1101 : 0011 0011 : '3'
...
```

Remember that since we are talking about a string of text, we are talking about data type char here which is always one byte in size. ASCII characters are always stored in a single byte of ram.

Notice that we have reverted back to the state of RAM from before we changed the 'a' to 'b'. We still have a pointer called ptr which contains the memory address 1000 and which therefore points to the 'a' character.
We know from the previous example that we can change the data at location 1000 by the following line of code:

```
*ptr = 'b';
```

What if we wanted to change the next character?

In general, if you want to look at or change any data in memory you only need to know the address of the data you want to change.

It turns out we already know the address of the next character in our string. It would be 1001 in ram, which is 1000 + 1. In other words, if we just add one to the address of 'a', we get the address of 'b'. If we add one to that address, we get the address of 'c', and so on.

If we want to change the 'a' in our ram, we simply set a pointer called ptr (for example) to 1000 and set *ptr to a new value. If we want to change the 'b' in our ram, we set ptr to point at 1001 (the address of 'b') and then we set *ptr to what we want. And so on.

We can see this in action with the following code:

```
        // To start with, ptr points to 1000
        // in memory which is where the 'a' resides.

*ptr = 'A';   // With this instruction we have changed
              // 'a' to 'A'

ptr = ptr + 1;  // by adding 1 to ptr, we are now pointing
                // to the address 1001, the 'b'

*ptr = 'B';   // Now we have changed 'b' (what was at 1001)
              // to 'B'

ptr = ptr + 1;  // By adding 1 to ptr, we are now pointing
                // to the address 1010 where 'c' is.

*ptr = 'C';   // Now we have changed 'c' to 'C' by
              // changing "what is at" that address.
```

What are we saying here? First of all the pointer ptr is pointing the memory address 1000, which is the 'a' in our 16-byte memory. By executing the instruction *ptr = 'A' we have changed the 'a' into an 'A', that is to say we have changed it from being lowercase to being uppercase.

Then, we added one to our pointer. Now instead of the pointer looking at position 1000 where the 'a' was, it is now looking at position 1001 where the 'b' is. Then we change the 'b' to 'B'. Finally we change the 'c' to 'C'.

Here is the state of our ram after these instructions have executed:

```
...
1000 : 0100 0001 : 'A'
1001 : 0100 0010 : 'B'
1010 : 0100 0011 : 'C' <--- ptr points here
1011 : 0011 0001 : '1'
1100 : 0011 0010 : '2'
1101 : 0011 0011 : '3'
...
```

Notice that ptr is pointing where we left it, at the address 1010.

We have changed the data that used to be "abc" and have turned it into "ABC". Also we have seen an important principle in action. It is often necessary when working with data to start at the beginning of the data, do some processing, and then continue through while each time incrementing a pointer so that it points to the next data we want to manipulate.

Also we have learned an important fact concerning pointers: You can add a value to a pointer and cause it to point to a different location in memory. In our example, we started at the address 1000 and then we added one so that we were pointing at the address 1001, then 1010, etc.

Whenever you change the memory address of a pointer, you are also changing what data the pointer "sees". In other words, if a pointer called ptr contains the memory address 1000, then *ptr will refer to the data at the address, for example an 'a'.

If we change the ptr so that it points to a different address, then *ptr takes on a new meaning.

Any time you change the memory address contained in a pointer, then you are changing the meaning of "what is at the address" of that pointer.

Lesson 9.3 : About pointers concerning multi-byte variables

The video for this lesson can be found here: http://youtu.be/MnHn4bZJvwU

Recall in the last lesson that we added one to our pointer in order to cause it to point to the next data element in memory.

Lets imagine a different case now. We are still going to use our 16 byte ram for this example, except instead of the string "abc123" we are going to use data of the type unsigned short int

Lets imagine the following code:

```
unsigned short int height = 10;
unsigned short int width = 14;

unsigned short int *ptr = &height;
```

Now in this example, we are creating two variables that each have a size of two bytes. This is because they each are of the data type unsigned short int, which is two bytes in size. (although this can differ between compilers).

Now, lets consider how they are stored in memory. Lets say that the first variable, height, is stored at memory address 1000 in our 16-byte ram.

```
...
1000 : 0000 0000 0000 1010 <--- height = 10; <--- ptr contains "1000"
...
```

Now keep in mind that because our variable is two bytes in size, it will take up two bytes of ram. To be truly accurate, our ram would therefore have to look like this:

```
...
1000 : 0000 0000 <--- first half of height; <--- ptr contains "1000"
1001 : 0000 1010 <--- second half of height.
...
```

Now lets go ahead and add the second variable width to our ram directly after height:

```
...
1000 : 0000 0000 <--- first half of height; <--- ptr contains "1000"
1001 : 0000 1010 <--- second half of height.
1010 : 0000 0000 <--- first half of width;
1011 : 0000 1110 <--- second half of width.
...
```

Do not think based on this example that variables are always placed one right after the other in ram when you create them.

Now we know that ptr is pointing to address 1000 which contains the start of the variable "height". So the next question to ask is what is the value *ptr is referring to?

Because ptr is pointing at address 1000, it might appear that *ptr would therefore be equal to: 0000 0000. After all, that is the data that is at the memory address 1000. This is not the case however.

Let's go back briefly to the lesson where we talked about how to create a pointer. We mentioned that it is important to specify the data type for what the pointer will be pointing to. In that lesson I explained that asking for the data at a memory address is not enough, you also have to specify how much data you are looking for.

In this case, I am not using ptr to point at one byte of data. I am using it to point at two bytes of data.

Therefore, *ptr will refer to: 0000 0000 0000 1010

The whole 16 bits that make up the variable height. Why? Because when we created the pointer ptr we specified that it will be used for pointing at variables of the data type unsigned short int.

What would happen if we set *ptr = 0; ?

Then C understands that because ptr was created to look at two-bytes, then *ptr=0 would set both bytes to zero. Let's expand on this a bit:

```
unsigned short int height = 10; // height is stored at the
                                // memory address 1000
unsigned short int *ptr = &height;

*ptr = 0;
```

The final result is:

```
...
1000 : 0000 0000 <--- ptr points here
1001 : 0000 0000
...
```

Think of *ptr = 0; as saying: "Store the unsigned short int value of zero (that means: 0000 0000 0000 0000) into the memory location at position 1000 in ram"

So you can see that *ptr will see and change two bytes of data which begin at whatever memory address is stored in ptr.

Now, consider if we want to change the value of "width" to ten. Based on the last lesson, we should be able to point our pointer to the memory address of width - which is two greater than the memory address of height. Would we then say ptr = ptr + 2; so we can point at the correct memory address?

No.

Because C understands we have created a pointer for type unsigned short int, it knows that if we increment our pointer by one, in fact if we do any mathematical operation on our pointer, that we are doing so on the understanding that each element we point to is an unsigned short int.

This means that C realizes that if we say ptr = ptr+1;, this means that we want to cause ptr to point at the next unsigned short int in memory, not the next byte in memory. In other words, this means that we want to cause ptr to point at the next two bytes in memory, and C assumes that those next two bytes are an unsigned short int.

In our last lesson because we were using the data type char, our pointer understood that we would be looking at data that was one byte in size. That is why adding one to the ptr in the last lesson resulted in the pointer address increasing by one byte.

In this example, because we are using the data type unsigned short int, our pointer understands that we will be looking at data that is two bytes in size. That is why adding one to the ptr in this lesson results in the pointer address increasing by two bytes.

This reasoning holds true for any data type.

So now, how do we change width to fourteen? Like this:

```
unsigned short int height = 10;  // Set height to ten.
unsigned short int width  = 5;    // set width to 5

unsigned short int *ptr     = &height;  // ptr contains the memory
                                         // address 1000 (eight)

ptr = ptr + 1;    // ptr now contains the memory address
                  // 1010 (ten)

*ptr = 14;  // Change the entire two-bytes at location
            // 1010 to fourteen: 0000 0000 0000 1110.
```

Keep in mind that this example is purely for the sake of this lesson. Our 16-byte ram is special because variables always get stored one after the other. In your real ram, this is not always the case. The above code should NOT be considered correct for this very reason. We will talk about how to actually do the above code correctly later.

The final state of ram after this code is:

```
...
1000 : 0000 0000 <--- first half of height;
1001 : 0000 1010 <--- second half of height.
1010 : 0000 0000 <--- first half of width; <--- ptr contains "1010"
1011 : 0000 1110 <--- second half of width.
...
```

Notice that width is now set to fourteen.

Lesson 9.4 : Pointers have memory addresses too!

The video for this lesson can be found here: http://youtu.be/0DgF3514nt8

Up until now we have learned a lot about pointers. In the previous examples I have used 16-byte ram to make it easy to understand how variables are stored in memory.

I have also showed how pointers can be used to point at variables inside memory.

Now we need to consider something else about pointers. Pointers have to exist somewhere inside of your memory also. That means whenever you create a pointer, it is given a place in memory to live just like any other variable.

Lets consider the following code:

```
unsigned short int height = 10;
unsigned short int *ptr = &height;
```

Only for the sake of this lesson, lets assume the following:

1. The data type unsigned short int is only one byte in size.
2. You only need one byte to store a memory address (eight bits).
3. Our pointer ptr will reside in only a single byte of memory.
4. the variable height will reside at position 1000 in memory.
5. The pointer ptr will reside at position 0100 in memory.

Now, lets imagine how our memory will look:

```
...
0011 :
0100 : 0000 1000 <--- ptr lives here. The value of ptr is 1000 (the memory address of height)
0101 :
0110 :
0111 :
1000 : 0000 1010 <--- height is here. The value of height is 1010 (ten)
1001 :
...
```

Notice that ptr is really no different than any variable. It has a memory address, and it has a value. It just so happens that the value it has is the memory address of another variable.

In an earlier lesson we saw that variables are plain English names that correspond to values stored at specific memory addresses. For example, if I type int height = 5, the programming language keeps track of the actual memory address where height resides.

When you create a pointer, you are giving the pointer a plain English name the exact same way as you do for a variable in general. A pointer has a memory address just like a variable does.

Your programming language keeps track of the address in memory of a pointer the same way as it does a variable. A pointer and a variable are much the same thing in this sense.

So lets consider the following code:

```
int height = 5;
int *my_pointer = &height;
```

Now, after these two lines of code, we have learned that:

1. If we write my_pointer we are referring to the value stored in my_pointer, which is the memory address of the variable height. In other words, my_pointer is equal to &height
2. If we write *my_pointer we are referring to "what is at" the memory address stored in my_pointer. In other words, *my_pointer refers to height.

What does &my_pointer refer to?

Well, if & means "address of", then &my_pointer would mean "The memory address where my_pointer itself is stored". Just the memory address of the pointer, not what is stored there. In our above 16-byte ram example, &my_pointer would refer to 0100 -- the address where my_pointer resides.
In summary: A pointer has to reside in memory somewhere, just like any variable. You can use the & "address

of" operator on a pointer in order to obtain the memory address where the pointer itself resides.

Lesson 9.5 : Why do I need to learn pointers?

The video for this lesson can be found here: http://youtu.be/hCFO3PYujuo

A lot of you have asked or have wondered why this material is worth knowing. This is a great question, and something I want to address.

It has been said in many books and by many students that pointers are one of the most difficult subjects in programming. For this reason I have taken the subject very slowly and I hope that everyone has been able to understand the material. Please let me know if that is not the case.

You may be wondering though why I am spending any time on it at all. Why do you need to see what is at some memory address of a variable that is already at that address? In other words, can't you just look at the variable?

First of all, I promise you that every lesson I have done throughout this course is something that will be useful to you as a programmer. Each lesson is also a pre-requisite for being able to do something greater.

Think of it like this. I had to teach you binary before I could teach you about how data is encoded, like ASCII for example. At the time, why anyone would ever need to know how to count in binary was a mystery to many of you.

It is the same with pointers. If you know pointers, you will be able to create programs, games, and applications that you simply would not be able make if you didn't know pointers. There is an enormous chasm of programmer skill between programmers who know pointers, and programmers who do not.

Now I want to show you just how fundamental pointers really are.

You are reading this text on a web browser, on a monitor. That monitor is re-drawing itself approximately 60-70 times per second. Each time it re-draws itself, a pointer, exactly like the pointers we have talked about, is scanning through your video memory, and replacing it with new data. This is in fact what makes your screen change. Every application that ever draws, writes, or otherwise makes any change to what is displayed on your screen uses pointers to do it.

If you have ever used a graphics program which had an "eye dropper" tool where you can select a color just by clicking on it, it determined the color you wanted using a pointer scanning your drawing in memory to see what color was stored at that memory location. In fact, the entire drawing exists as a data structure in memory. All the drawing, erasing, or anything else you do in an art program uses pointers to do it.

Program that play sound or music files work by having a pointer look at the start of the sound in memory, play a sound (which is itself a data structure requiring a pointer), and then the pointer goes through the data to the end of the song. The same thing is true with movies.

Every program you have ever used, game, or application, requires pointers to read and manipulate memory continually. Pointers are the only way you can see and understand any data that is greater than a few bytes in size. This is the fundamental point I want you to understand:

Pointers make it possible to read and manipulate data in memory which is larger and more complex than a data type (int, char, etc) can make possible. Which pretty much means... everything. Pointers are one of the most fundamental concepts in computing. Properly understood they empower you to do just about anything. Without them, you can do hardly anything.

In other words, the real question is not when do we use pointers. The real question is when do we not use pointers.
This course has barely started, and many great things are ahead.

Lesson 9.6 : Introducing the char* pointer

The video for this lesson can be found here: http://youtu.be/H2iVDvN4zek

As I mentioned before, pointers are powerful because they give you a way to read and write to data that is far more complex than the data types that C or any language gives you.

Now I am going to explain some of the mechanics of how this actually works. In other words, how do you read and manipulate a large data structure?

First I want to give you a small sneak peek at the future of this course. In C (or in any language really) the complexity of data follows this hierarchy:

1. single element of a given data type (char, int, etc)
2. text string (a type of simple array)
3. single dimensional arrays
4. multi-dimensional arrays
5. structures
6. And so on.

The more complex the data you can work with, the more and better things you can do. It is as simple as that.

In the very first lesson I commented about the difference between learning a language, and learning how to program. The purpose of this course is to teach you how to program. I am starting with C, and we will work into other languages as the course progresses.

Now we are going to advance our understanding past single data elements of a given data type, and work towards #2 on the list I showed you. To do that, I need to introduce a new concept to you.

Examine this code:

```
char my_character = 'a';
```

This makes sense because we are saying "Create a new variable called my_character and store the value 'a' there." This will be one byte in size.

What about this:

```
char my_text = "Hello Reddit!";
```

Think about what this is saying. It is saying store the entire string "Hello Reddit!" which is more than ten bytes into a single character -- which is one byte.

You cannot do that. So what data type makes it possible to create a string of text? The answer is - none. There is no 'string of text' data type.

This is very important. No variable will ever hold a string of text. There is simply no way to do this. Even a pointer cannot hold a string of text. A pointer can only hold a memory address.

Here is the key: a pointer cannot hold the string itself, but it can hold the memory address of.. the very first character of the string.

Consider this code:

```
char *my_pointer;
```

Here we have created a pointer called my_pointer which can be used to contain a memory address.

Before I continue, I need to teach you one more thing. Whenever you create a string of text in C such as with quotes, you are actually storing that string somewhere in memory. That means that a string of text, just like a variable, has some address in memory where it resides. To be clear, anything that is ever stored in ram has a memory address.

Now consider this code:

```
char *my_pointer;
my_pointer = "Hello Reddit!";

printf("The string is: %s  ", my_pointer);
```

Keep in mind that a pointer can only contain a memory address. Yet this works. This means that my_pointer must be assigned to a memory address. That means that "Hello Reddit!" must be a memory address.

This is exactly the case. When you write that line of code, you are effectively telling C to do two things:

1. Create the string of text "Hello Reddit!" and store in memory at some memory address.
2. Create a pointer called my_pointer and point it to the memory address where the string "Hello Reddit!" is stored.

Now you know how to cause a pointer to point to a string of text. Here is a sample program for you:

```
#include <stdio.h>

int main() {
   char *string;
   string = "Hello Reddit!";

   printf("The string is: %s  ", string);
   }
```

UNIT 10 : Introducing Strings and Constants

Lesson 10.1 : Introducing the Constant

The video for this lesson can be found here: http://youtu.be/hHEzjTk1oqg

Up until now we have only spoken about variables. We have learned that you can create a variable and then later you can change it. For example you can write:

```
int height = 5;
height = 2;
height = 10;
```

All of this is valid. There is nothing that stops you from storing a new value in a variable.

The reason we use the name "variable" is because variables can be changed. In other words, the data stored at the memory address of a variable can be read as well as written to.

This is not the case with a constant. A constant is data that is stored in ram just like a variable, but it cannot be changed. You can only read the data.

The first question you might have is, "When do you use a constant?" The truth is, you already have.

Consider this code:

```
char *string = "Hello Reddit!";
```

We know from the previous lesson that the text "Hello Reddit!" is stored in memory, and we can even set a pointer to it. However, when C created this string of text "Hello Reddit!", it created it as a constant. In fact, a string of text like this can be called a "string constant" or a "string literal". The meanings are the same.

If we create a pointer and point it at that text, we can read it. We should not however use a pointer to change it. This is because in the case of a constant, the data may exist in read-only RAM. I will show you the proper way to change data in a string of text later.

Just to review: A variable can be changed and is both readable and writable. A constant cannot be changed and is only readable.

Lesson 10.2 : Important review and clarification

The video for this lesson can be found here: http://youtu.be/a3Jkfu9JbIc

I have had a chance to go through and read hundreds of questions and replies from everyone. Before we proceed to the next lesson, there are some clarifications I need to make from previous lessons.

Some of these involve possible misunderstandings that may have come about from the way I explained a particular topic. Others involve questions and replies in the comment threads.

Include Statements

In this lesson I explained that you use include statements to "copy and paste" the contents of some programming source code into your existing program. With C and some languages, this is technically correct. It does not however address the real purpose.

It is extremely bad programming practice to do things this way. You do not put code in a file, and just cut-and-paste that code using an include statement to where you want it.

What you are really saying with an include statement is that you want to use the contents of the included file within your program. In other words, the include file contains code (functions especially) that will prove useful in your program.

Think of this line:

```
#include <stdio.h>
```

As meaning this:

I plan to use to use functions found in the stdio.h file in my program.

Basics of numeric overflow.

When we speak of numeric overflow in the context of this lesson, we are referring to any time a mathematical operation produces a result which cannot fit in the number of bits allocated. For example, if we have four bits and we perform a mathematical operation that results in the number twenty-five, that result will not fit in our four bits and thus we have a numeric overflow.

While there are similarities, this is not the same thing as a "stack overflow", which will be the subject of at least one future lesson.

Basics of fractional numbers in binary.

I have explained that the radix point is the term used for the equivalent of a "decimal point" in binary. I need to be extra clear on three points:

1. The radix point is not actually encoded in binary. Rather, it is understood by knowing how many bits correspond to the part of the number to the left of the radix point and how many bits correspond to the part of the number to the right of the radix point.
2. The number to the left of the radix point is known as the "integer part".
3. The number to the right of the radix point is known as the "fractional part".

The Basics of Numeric Data Types in C.

I have not covered all numeric data types in that lesson, or even up until now. We will be learning more as time goes on.

Also, as zahlman pointed out: The term 'long double' has nothing to do with combining a 'long' value and a 'double' value. It is actually often 10 bytes, although it may be stored in a 12-byte "slot" with the other two bytes being wasted.

Similarly, the sizes of a data type such as unsigned short int being two bytes long differs from compiler to compiler. This is true for all data types.

This does not matter if your goal is to write an executable program that you then deliver to systems running the same architecture and operating system as you. You will never need to worry about this in that case. That program will run on any computer that uses it if they have the same operating system and the same architecture (Example: 32 bit windows).

If however you are writing code that will be open-source, or you are otherwise going to make the source code available, you must be mindful that if you are using a data type of a specific size that may differ from compiler to compiler, you should let people know that as well as any other compiler unique optimizations and/or options you are using.

The "char" data type and the basics of ASCII.

Here and in general I showed you the following rules concerning how text and numbers are encoded in binary:

010 <-- All capital letters start with this. 011 <-- All lowercase letters start with this. 0011 <-- All numbers start with this.

Keep in mind that this is NOT reversible. This means that you cannot say that anything which starts with 010 is a capital letter for example. There are many ASCII characters that are not numbers or letters, and you should be mindful of this fact.

Introducing variables.

Here I showed how to create variables, for example:

```
int height = 10;
```

The question was raised, what if I didn't specify a value? What if I had just written:

```
int height;
```

This is legal, and will create the variable. However, anything at all could be its value. Why? Because when C creates the variable height it only decides where in memory it will reside. It does not set a value for it.

That means whatever binary sequence just happens to be at that memory address will be the value of the variable. This is not a good thing.

That is why as a programmer you should always initialize a variable to a value when you create it. Common practice is to assign a 0 to a variable until it is used if the value is not known at the time it is created.

About terminating strings of text and other data.

I am just going to quote zahlman here regarding the consequences of NOT terminating a string/other data properly:

The program might not print garbage; it might also crash, because as printf() cycles through memory looking for a byte that happens to have a zero value, it might encounter memory that doesn't belong to your program. The operating system generally doesn't like it very much when your program tries to work with memory that doesn't belong to it, and will put a stop to things.

Technically, the behaviour is undefined, which means anything is allowed to happen. A sobering thought. On Gamedev.net, it is a common joke to refer to accidental firing of nuclear missiles as a result. Your computer is almost certainly not actually capable of firing nuclear missiles, but it gets the point across.

Assigning a value to a pointer.

One thing I did not cover at the time I made the lesson, and therefore some people may have missed, is this: C is very forgiving when it comes to when you put spaces, and when you do not put spaces.

For example:

```
int     height = 1;
int height     = 1    ;
```

Are both valid, and will work fine. Although of course good practice is to format your code so it is easily readable. Also, notice that both have all necessary characters including the

```
int *ptr = &height;
int * ptr = & height;
int* ptr =     &height  ;
```

etc.

About pointers concerning multi-byte variables.

Here I showed you the following code:

```
int height = 5;
int width = 10;

int *some_pointer = &height;

*some_pointer = 8;
some_pointer = some_pointer + 1;

*some_pointer = 4;
```

Then I explained, as is logically apparent from reading the above code, that this has the effect of first changing height to 8, and then changing width to 4. This would be correct if height and width reside in memory one right after the other. This is one of the assumptions being made about our 16-byte ram used in these types of lessons.

Do not ever write code like this in a real program. Your compiler chooses where to put variables you create and you have no way to know if they will be stored in ram the way this example assumes they are. The code above from this lesson is for instructive purposes only, to show you how pointers work in general.

Some clarifications about printf

It has not been addressed directly in a lesson, even though you have likely seen it used. With printf you can give it multiple parameters if you want to print various types of data. For example, the following is perfectly valid:

```
int height = 10;
int width = 5;

printf("Height is %d and Width is %d  ", height, width);
```

This will have the output:

Height is 10 and Width is 5

Next, I have not specifically addressed the \n character (note I did not say characters).

There are special ASCII characters which have meaning other than what you see on your keyboard. These characters are useful for tabs, new lines, and more. The text \n actually turns into a single character which is ASCII and, for those who are curious, looks like this:

```
0000 1010
```

In other words, ten, or A in hexadecimal. This one character means "Go to the next line."

The final clarification I want to make concerning printf() is that many of you have used printf() with the %p option to print the addresses of pointers, which is great and helps to cement this understanding. However, be sure you remember the following:

```
printf("The address is: %p", some_pointer);
```

What will be printed is the address IN some_pointer, not the address OF some_pointer. Also, this is invalid:

```
int height = 10;
int *some_pointer = &height;

printf("The address is: %p", *some_pointer);
```

Do not do this. Remember that *some_pointer translates to: "Whatever is at the memory address stored in the pointer". It does not translate to: "The address of the pointer". It also does not translate to: "The address stored in the pointer".

Using *some_pointer in the above example results in: 10, the value stored in height. In other words, "what is at" the address stored in some_pointer is the value of height which is ten. Please ask questions if any of this is confusing to you.

Lesson 10.3 : More about Strings and Constants

The video for this lesson can be found here: http://youtu.be/-qujgzRSos4

In the last lesson I explained that in addition to variables, you can also create constants. I explained that with constants, you can read them but cannot change them. I know this is a mystery to many of you. Now we are going to cover this material in greater depth.

First of all, as we have discussed in previous lessons, whenever you create a string of text enclosed in quotes such as "Hello Reddit", this string of text must be stored somewhere in your memory. Lets go through the steps C (or any language) must go through in order to do this.

1. Find some location in memory to store the string of text.
2. At that location in memory, store an 'H'.
3. Exactly after the 'H', at the very next byte, store an 'e'
4. Continue through this process until all characters of the string have been stored.
5. Add a NULL byte (all zeroes) at the end.

Keep in mind that this means that if your string of text is five characters long, there will actually be six bytes of ram needed to store it.

Now consider this code:

```
char *string;
string = "Hello Reddit";
```

string is a pointer. Every pointer contains a memory address. string therefore contains the memory address to where this string begins. In other words, it points to the letter 'H'. It contains the memory address where 'H' is stored in memory.

What if we now want to change this string to something else? Could we write for example:

```
string = "A new string";
```

The truth is, we can. However, it does not change anything. What we are actually doing here is telling C to create a new string of text called "A new string" and to point the pointer string at this new string. Remember, string is a pointer. Pointers can only contain memory addresses. Not strings.

What happened to our old string of "Hello Reddit" ? It is still there, sitting in memory. However, we have no way to find it now because we changed where our pointer was looking. Consider this code:

```
string = "Hello Reddit";
string = "A new string";
string = "Hello Reddit";
```

With the third line of code, did we now set string back to what it was? No. It will be pointing to an entirely new memory address. We will now have multiple strings of "Hello Reddit" sitting in memory somewhere.

[Edit: It is true that many modern compilers will be smart and figure, "Why create a new string "Hello Reddit" ? Lets just use the one we have." However, the thing to keep in mind is that in general when you set a string pointer to a new quotes-enclosed string, you are not changing the text, you are creating a new string of text and pointing your string pointer at that new string of text.]

The next thing I need to address is that there are different "kinds" of memory. Now, this is a bit misleading because memory is memory, regardless of what is stored in it. However, your compiler as well as your operating system have rules about which memory you can use for various purposes. These kinds of memory exist as "ranges" within your available memory.

It works similarly to this:

```
<-------- Read Only -----------><--------- Read/Write --------->
```

This small diagram is purely for illustrative purposes. The real details of how this works are more complex, and you do not need to know them yet. What you need to know is that different ranges of memory can be used for different purposes.

Variables are placed into the "Read/Write" portion. Constants are placed into the "Read Only" portion. I hear you asking: "What? There is a read only range of memory?" And the answer is, yes.

Why is there a read only range of memory? Because as a programmer, some data you create must not change or it will cause your program to not work. Let me give you an example. Suppose in a program I need to draw circles, and I put the mathematical definition of PI as 3.14159. Do I ever want that to change? No, never.

When you as a programmer want to make sure that a constant doesn't change, then wouldn't it be nice if your computer worked with you on that goal? That is exactly the case. C stores constants in read only memory for your benefit. The idea is, if you define a constant, then you intend that it does not change.

Now, as many programmers reading this will attest, C's idea of your benefit and your idea of your benefit can differ. But it is the thought that counts.

At this stage you should understand:

1. What is a constant?
2. Why do constants exist?
3. Where and how are constants stored?

You are still probably wondering why did "Hello Reddit" get stored as a constant. The answer has to do with C syntax. If you write this code:

```
char *string = "Hello";
```

OR this code (they are both identical)

```
char *string;
string = "Hello";
```

Then C knows that you want "Hello" to be a constant. We will talk soon about other ways to define strings in such a way they are not a constant, but remember from here on out that using this above method to create a string, will make the string a constant.

You are probably wondering, "Shouldn't you have to use some sort of keyword to indicate that the string is a constant?"

Not in this case. It is implicitly understood by C that because we are creating a char* pointer, and pointing it to a string enclosed in quotes, that we intend for that string to be a constant.

You have used other examples of implicit keywords. For example:

```
int height=5;
```

Really means:

```
signed int height = 5;
```

Lesson 10.4 : A new way to visualize memory

The video for this lesson can be found here: http://youtu.be/v5YEjSXv8aQ

Up until now we have used 16-byte ram to illustrate concepts in pure binary. We cannot continue like this because eventually it becomes too complex. A large part of the skill that you need as a programmer is the ability to visualize concepts more and more abstractly.

So because of that, we are going to change how we look at our 16-byte ram. Lets imagine the text: "abc123" - with a null termination byte. Here is how it looks in our 16-byte ram at position: 1000

```
...
1000 : 0110 0001 : 'a'
1001 : 0110 0010 : 'b'
1010 : 0110 0011 : 'c'
1011 : 0011 0001 : '1'
1100 : 0011 0010 : '2'
1101 : 0011 0011 : '3'
1110 : 0000 0000 : null (also called \0)
...
```

Instead of visualizing ram like this, we will do so like this:

```
...
1000 : ['a']['b']['c']['1']['2']['3']['\0'] ...
...
```

Notice that the ... (ellipses) at the end of our "abc123" string indicates that other data might follow, but that we do not know or care what that data is.

We are still saying the exact same thing here as we did before. We are still looking at the exact same state of memory. The exact same bytes are storing the exact same values. We are just writing it out in a slightly more abstract way.

Each [ ] block represents a byte. We are simplifying what is contained in each byte.

You should be able to clearly look at any of these [ ] blocks and realize that there is a single byte, eight bits of data. From prior lessons you should also know what binary sequence is contained in each block.

You should understand that the memory address 1000 corresponds to the exact memory location that 'a' is stored. That 1001 corresponds to 'b' and so on.

Visualizing memory like this allows us to observe interesting details about our string. For example, you will notice that it is easy to see how many bytes the string represents. You can count out the number of characters and a \0 (null) character, and see exactly how many bytes are stored in memory.

Now if we want to study a more complex string, such as this:

```
1000 : ['H']['e']['l']['l']['o'][' ']['R']['e']['d']['d']['i']['t']['\0'] ...
```

It becomes easier to do, and to see each character living at its own byte, including the space character. You should still be able to understand the different binary sequences (roughly) in each byte, and understand that each character is stored in memory right after the previous character.

This method of visualizing the contents of ram will make the future lessons much easier to understand.

Lesson 10.5 : Introducing the character string as an array

The video for this lesson can be found here: http://youtu.be/7HAvRMUEURQ

In a previous lesson we learned how to make a string constant using a char pointer, and pointing it to a string of text within quotes. To be clear, we did not* learn how to store a string of text inside a pointer. That is impossible, and is a common beginner misunderstanding. Quick review:

```
char *string = "Hello Reddit";
```

We created a pointer of type char and we assigned it the memory address of the string "Hello Reddit";

In an earlier lesson, I introduced arrays. An array is a collection of data elements of the same data type that reside in memory one right after the other. This is very important as you will see. A string of text is the simplest example of an array.

With a string of text, you have a collection of data elements, in this case characters, each residing one after the other in memory. To create an array we basically need to follow these steps:

1. We choose a data type. Each element of the array must be the same data type.
2. We choose a size. In reality, this is optional, but for the purpose of this lesson it is worth having this as a step.
3. We store data into the array.

Remember that I said that a character string is an array. Lets look at our "abc123" from the previous example:

```
Figure (a)
1000 : ['a']['b']['c']['1']['2']['3']['\0'] ...
```

We have already seen how to create it as a constant. How do we create it in such a way we can modify it? The answer is, we tell C that we intend this to be an array of individual characters - not merely a pointer to a string constant.

Here is the code:

```
char string[7] = "abc123";
```

Here is what I am saying: Create a variable called string. Keep in mind that string is not really one single data element, but a chain of seven different bytes, each byte being an ASCII character. Notice I said seven. abc123 are six characters, but I stated seven to take into account the NULL byte at the end.

So here comes a question. What exactly is string? Is it a constant? Is it somehow encoded differently in memory to Figure (a) above? The answer for both questions is no.

It is not a constant first of all because we have specifically told C that we want an array of variables of type char. A variable can be modified, a constant cannot. By saying we want an array of variables, then C knows we plan on having the ability to modify them.

Is it encoded any differently? No, the same exact bytes are stored in exactly the same way. There is no difference.

Try this code:

```
char string[7] = "abc123";
printf("The string is: %s", string);
```

Now, notice I specified a size in bytes. It turns out that this is optional. If you do not know how many bytes you need for a string of text, you can put [] instead. For example:

```
char string[] = "abc123";
printf("The string is: %s", string);
```

Here you will get the same result.

Now, what is string itself? Behind the scenes, it is a pointer. However, you do not need to worry about this. As I stated in an earlier lesson, any time you are working with any type of data more complex than a single variable of a given data type, you are working with a pointer.

Programming languages, including C, give you some ability to work with pointers abstractly so you can work more efficiently. It is still important to understand the process that is going on behind the scenes, which is what these lessons are largely about.

Lesson 10.6 : Using pointers to manipulate character arrays

The video for this lesson can be found here: http://youtu.be/fl0QW7CPuec

In an earlier lesson we talked about setting a pointer so that it contains the memory address of a string constant. I pointed out that with a string constant you are able to read the characters of the string but you are not able to change them. Now we are going to look at a way to change a string character by character.

The concept we are going to look at is that of being able to start at the beginning of some data and change it by moving byte-by-byte through the data changing it as you go. This is a critical concept and we will be doing a great deal of this later.

First lets start with this code:

```
char string[] = "Hello Reddit";
char *my_pointer = string;

printf("The first character of the string is: %c", *my_pointer);
```

The output will be:

The first character of the string is: H

This should make sense to everyone at this point. *my_pointer refers to "what is at" the memory address stored in the pointer my_pointer. Because my_pointer is looking at the start of our array, it is therefore pointing to the 'H', the first character. This is what we should expect.

Notice that we do not need to put &string. This is because string, by being an array, is already effectively a pointer (though behind the scenes). Re-read the last lesson if that is unclear to you.

Because our string is part of an array of variables of type char, we can change it. Let's do so:

```
*my_pointer = 'h';
```

What we have done now is to change "what is at" the memory address which used to contain an 'H'. Now it contains an 'h'. This should be pretty simple to understand. Recall that we could not do this when we created the string using a char* pointer, because it was a constant.

Now, remember that because this string of text resides in memory with each character immediately following the character before it, adding one to our pointer will cause the pointer to point at the next character in the string. This is true for all C programs you will ever write.

This is perfectly valid:

```
char string[] = "Hello Reddit";
char *ptr = string;

*ptr = 'H';

ptr = ptr + 1;
*ptr = 'E';

ptr = ptr + 1;
*ptr = 'L';

ptr = ptr + 1;
*ptr = 'L';

ptr = ptr + 1;
*ptr = 'O';
```

This works fine because C will store your array of characters exactly the right way in memory, where each character will immediately follow the other character. This is one of the benefits of using an array in general with any data type. We do not have to worry about whether or not C will store this data properly in memory, the fact that we are specifying an array of characters guarantees it will be stored correctly.

Now notice that what we have done is very simple. We started at the first character of the array, we changed it, and then we continued through until we got to the end of the word "Hello". We have gone over this same concept in earlier lessons, but now for the first time we are actually able to do this in a real program.

If at the end of this, we run:

```
printf("The string is: %s  ", string);
```

We will get this output:

```
The string is: HELLO Reddit
```

Notice that it is perfectly ok that we "changed" the 'H' to an 'H'. When you assign a value to data at a location in memory, you are not necessarily changing it. You are simply stating "Let the value here become:

UNIT 11 : Conditional Flow Statements

Lesson 11.1 : Introducing Conditional Flow Statements

The video for this lesson can be found here: http://youtu.be/oyJb9UzCNDs

Welcome to Unit 11. In this unit we are going to be discussing conditional flow statements, which is a topic that applies not only to C, but to many other programming languages as well.

In fact, I would argue that you would be hard pressed to write any kind of useful program in any programming language without the use of conditional flow statements.

Ok, Let's begin.

Up until now we have only written code which follows this kind of a structure:

```
Statement 1;
Statement 2;
...
Statement N;
```

Basically, your code just executes a specific set of instructions one right after the other and then finishes. Now, you can do some interesting things as part of those instructions, such as printing text, or working with pointers, but still you must follow a specific path from start to finish. You have had no control over choosing which instructions to execute and when.

Now we are going to advance past this understanding. When you write a program, it is very often the case that you will not want to execute every single instruction. For example, you might be designing a shooting game, and you might write out some segment of programming code that is designed to execute only if your character gets shot. Such a program might be similar in structure to this:

```
... statements ...
... statements ...

If the character was shot, then do this:
    ... check the health ...
    ... display damage ...
    ... etc ...

... more statements ...
... more statements ...
```

In this example, you only want to check the health of the character and display damage if the character was shot. If the character was not shot, then you want to skip over those instructions. In other words, whether or not your program executes (or flows into) those instructions is conditional upon a certain test, in this case you are testing whether or not the character has been shot. This is why we call it a "conditional flow statement".

Before we proceed, let's clearly define a conditional flow statement: A conditional flow statement is a mechanism that allows you to branch your program flow into one or more alternate branches depending on the condition being tested. Conditional flow statements allow you the programmer to limit the execution of a segment of programming code, called a "block" of code, based on the condition being tested.

In our above example, the line saying "If the character was shot, then do this:" is an example of a conditional flow statement.

Next, we will get into the exact mechanism that allows you to do this.

Lesson 11.2 : The mechanisms for conditional flow statements

The video for this lesson can be found here: http://youtu.be/6ypJBg9SLi0

In the last lesson I introduced you to control flow statements, and I explained that this is a mechanism you can use in order to limit the execution of a segment or block of code based on a certain condition.

In this lesson I am going to explain how this is actually done, first in terms of how your computer does this behind the scenes, and then in the next lesson how to actually do this in C.

Just for a moment, suppose that your computer needs to compare two values to see if they are equal, let's suppose 5 and 3. How can this be done? How exactly is an electronic device like a computer able to perform comparisons?

It may surprise you to know that one of the ways your computer compares two things to see if they are equal is by using subtraction. If any two values are subtracted, it is very easy to know if they are equal. How? Simply by determining if the result of the subtraction is zero.

Any time two values are equal, the result must be zero. If the two values are not equal, then the result will not be zero. We do not need to know what the actual result was, we just need to know whether or not it was zero. If it was zero, the two values are equal, and if it is not zero, the two values are not equal.

Keep in mind that it does not matter how you subtract the two values. Whether you say 5-3 or 3-5, the result will still be "not zero" if the values are not equal, and the result will be "zero" if the values are equal.

So to recap, in order to test if two values are equal, you do the following:

1. You run a subtraction on the two values, and it does not matter the order in which they are subtracted.
2. You simply check to see if the result is zero or not. If the result is zero, the two values are equal.

Now of course you need to keep track of whether or not the result was zero for later on. Notice that you do not need to keep track of the actual result of the subtraction, you only need to keep track of whether or not that result was zero. Now think about this, how much space do we need in order to do this? How many bits?

The answer is that we need only a single bit. We can set that bit to "ON" if the result is zero, and we can set that bit to "OFF" if the result is not zero. From an earlier lesson, you should remember that a single bit being used to store a single true or false status is called a "flag".

So here is how this would work:

1. Compare two values (this means subtract them, but do so in a way that they are not actually changed). We need a machine code operation to do this, we will call it "COMPARE"
2. We could design our machine code in such a way that the if the result of the subtraction is zero, it automatically sets the "zero flag" to on and otherwise sets the "zero flag" to off.
3. Jump over the block of code associated with the conditional flow statement, but *only if* the zero flag is OFF. We can call this instruction "JUMP IF NOT ZERO"

So an if statement that looks like this:

```
... statements ...
... statements ...

IF 5 IS EQUAL TO 3, THEN DO:
    ;... block of code associated with conditional flow statement ...

... the program continues ...
```

Would break down into this, in machine code:

```
COMPARE 5 and 3
JUMP IF NOT ZERO TO: ... the program continues ...
```

and that's it! With only *two* machine code operations, we can do an if statement. Now you can see exactly how your computer is able to perform conditional flow statements behind the scenes.

Now, before I continue, I want to show you something.

What I just showed you is not hypothetical. There are many programming languages, and what I just showed you is a simplified version of "assembly language", which is a programming language that allows you to write code at such a low level, that every instruction you write gets directly translated to machine code, 1s and 0s. Rather than write out the actual 1s and 0s for each instruction, or "op code", assembly language allows you to use short mnemonics for these instructions. If I wanted to change my above example to assembly language, I would just do this:

```
CMP 5,3
JNZ memory_address_after_block_of_code
```

and that's it. Just two lines of machine code is all it takes in order to perform an if statement. Now in the next lesson, I will show you how this works with C.

Lesson 11.3 : Implementing an if statement in C

The video for this lesson can be found here: http://youtu.be/7AQkZduPJv0

You know from the previous lesson that evaluations of any test or comparison can only result in true or false, and that a flag on your CPU chip (the zero flag, ZF) is critical to this process.

Remember that there is a single-bit binary flag built onto your CPU-chip called the "zero flag". It is set to either 1 or 0. In fact, this single bit is used every time any program on your computer has any question or ever needs to test something. There is nothing else on your computer which has this purpose. Without this single bit, your entire computer would be rendered useless.

This explanation is a bit ironic, so bear with me: The zero flag is set to 1 if the result of the last operation was zero. In other words the zero flag exists in order to determine if the result of the last comparison was zero. True (1) means, "Yes, the result of the last comparison was zero". False (0) means, "The result of the last comparison was anything other than zero."

If this is unclear to you, consider the following. I want to check if 5 is equal to 3. If I run a subtraction (notice that it doesn't matter if I do 5-3 or 3-5), then I can check to see if I got a 0 as the result. If I did, then 5 is in fact equal to 3.

In the context of comparing two values, Think of the zero flag as being an "equality" flag. 1 means the two values are equal (because subtracting them gives zero), and 0 means they are not equal.

Do not worry about using the zero-flag in your actual programs. You will never need to (unless you learn assembly language). I am presenting this to you mainly to show you some of the finer details of what goes on behind the scenes inside of your computer.

Imagine this code:

```c
int height = 5;
```

Now, I want to run a test based on the value of height being equal to 5. If it is equal to five, I want to printf() "The value is five!".

First, here is the C code to do this:

```
// Figure (a)

if (height == 5) {
   printf("The value is five! ");
}
```

This printf() statement will only execute if height is in fact equal to five. You will notice that there are two equal signs in that statement. Pronounce the two equal signs as: "is equal to". Why two equal signs? Because we have already defined what one equal sign means.

If I write: int height = 5; I am using one equal sign, and that means "Set the variable to some value." Therefore, since we have already defined that one equal sign means to set a value, we need a different operator to test if something is equal. Therefore, two equal signs (not one) are used to determine equality.

This is very important, so do not forget it. NEVER write code that looks like this when you want to test equality:

```
if (height = 5) { // <--- Very bad
   ...
}
```

Why? Because you are actually setting height to be equal to 5 as part of the statement. You are not testing whether height is equal to 5. Whenever you assign a value like this, the Zero Flag is set to 1, and thus the if statement will always be considered as being true. What is worse is that if you expect the if statement to be true, and you compile and run the program, it will appear to work perfectly.

I was not sure if I would show you what I am about to. This is actually a very simple process, and I think it is worth understanding. Here is how the machine code works:

In machine code (slightly translated), here is basically how the if statement in Figure (a) would look:

SUBTRACT 5 from whatever is in the variable height (however, do not change height)
(now a zero flag gets set to either 1 or 0)
IF Zero Flag (ZF) is set to 1 (meaning height is equal to 5), then: execute the printf statement

Believe it or not, that is all that happens. Just a few machine-code instructions are enough to do this, because the functionality to test equality is actually built right into your CPU itself. Notice that one of those machine-code instructions is itself an IF statement. That is how fundamental this is in computing.
So in this lesson you have learned that C or any language has the functionality to allow you to test equality between two different things, and that this functionality is so fundamental that it is actually physically built into your CPU.

Let me put this into concrete terms. Imagine the following if statement:

```
if (height == 5) {
```

The expression between the parenthesis will evaluate to something. Always. There will be some state of the "zero flag" once it evaluates. It will either evaluate to a 1 (true) or it will evaluate to a 0 (false).

Notice that there is an interesting correlation between logical truth and the zero flag. If the zero flag is set to 1, that is the same thing as saying true. If the zero flag is set to 0, that is the same thing as saying false. Remember the zero flag itself is merely the result of the last expression executed, such as a comparison statement. Recall that if two things are compared (using subtraction) then the result will be 0 if and only if they are equal. The zero flag is set to 1 in this case, meaning "The result was zero", and also meaning "true".

The above is a simple example because it involves only one expression, the height == 5, but you can have many such expressions in a conditional flow statement, such as an if statement. For example, I could write code that in English means: "If height is five, and width is three" which would look like this:

```
if (height == 5 && width == 3) {
```

Here I have introduced some new C syntax. The && when present in an if statement simply means "and". You do not put just one & character, but you have to put two. Two && reads "and".

Now, this is true but I want to explain it in a bit more depth. It really means "proceed to the next comparison only if the last one evaluated to true". In other words, the && itself implies a type of "if" statement.

It helps to think of the && as saying: "A new expression to be evaluated follows, but only evaluate it if the one just evaluated was true".

In other words, in the example if (height == 5 && width == 3) {, "is height equal to five" is evaluated. If the result is equal (meaning, zero flag is set), then and only then the next expression of "is width equal to 3" is evaluated. In this way every chain of the if statement is evaluated from left to right. The entire if statement will have the result of the last expression that was evaluated.

As this gets evaluated, it will follow these steps:

1. if (height == 5 && width == 3) {
2. if (1 && width == 3) {
3. if (1 && 1) {
4. if (1) {
5. Final result = 1 = true.

It might sound like I am picking at straws, or presenting useless information. This is absolutely not the case.

This is actually very important. Every extra expression in a conditional statement takes computing resources. Some such expressions will involve running functions, some of which can be enormously complex. Consider the following code:

```
// Figure (a)

int height = 5;
int width = 3;

if (height == 5 && width == 3 && printf("1234") == 4) {
    printf(" Test Finished");
}
```

Output:

```
1234
Test Finished
```

What you need to fundamentally understand here is that if you change any of the the three expressions in Figure (a) the if statement will fail. For example if you change height == 5 to height == 1, or you change width == 3 to width == 8, or you change the printf() from 4 to something else (or if you add extra text), if statement will fail. Notice that if you change the text in the printf() in the if statement, that printf() will still execute, but not the one that says "Test Finished".

Keep in mind that when an if statement fails, it will only fail once, on a given expression being evaluated. That means that any time an if statement fails, at least one expression has been evaluated. If it fails on the first expression, the evaluation still took place. If it fails on the third expression being evaluated, that still means the first, second, and third evaluation took place. However, the fourth expression (if there is one) will not execute.

If for example, we do this:

```
int height = 5;
int width = 3;

if (height == 5 && width == 1 && printf("1234") == 4) {
    printf(" Test Finished");
}
```

What happens? Neither printf() statement will execute. Why? Here is what happens:

First, height is evaluated to see if it is five. Since it is, the && means that we now evaluate the next expression. This will cause the next expression to execute, comparing the variable width and the value 1. Here the if statement fails. This means no further expressions will be evaluated. Once it has failed on any of its expressions, the entire if statement has failed and the final result will be "false". Because of this, the printf() statement inside the if() statement will not execute. This is extremely important, and you can use it to your advantage.

Whenever you have a series of comparisons to make, you should always execute them whenever possible in the order of least-computing-power-required to most-computing-power-required. In this way, if the if statement fails, it will fail with the least cost of computing resources.

I encourage anyone reading this lesson to experiment with the code in Figure (a) to see these concepts working for yourselves.

This knowledge will enable you to make programs that are faster, more efficient, and that make more logical sense.

Lesson 11.4 : More about if statements and logical operators

The video for this lesson can be found here: http://youtu.be/dumt3rvCz0k

Ok, so in the last lesson I showed you some of the basics about if statements and how to use the "logical AND" operator, which is two ampersands: &&. Now, in this lesson I am going to show you some of the more subtle details about the way if statements and certain logical operators work.

First of all, let's start by setting the integer variable height to contain five and let's start with a simple if statement.

```
#include <stdio.h>

int main(void) {

    int height = 5;

    if (height == 5) {
        printf("The value is 5");
    }

    return 0;
}
```

And we can see that the output is:

```
The value is: 5
```

Now what I have explained to you so far is that what is evaluated inside of the parenthesis of the if statement is going to result in either 0 or non-zero. If the result is 0, then the if statement will not execute. Any other value is considered true, and the if statement will execute.

Earlier I explained to you that the double equal sign basically means "is equal to". In this case, two values are compared and then if they are equal the result will be 1. Otherwise, the result will be 0. More specifically, the result will be either the integer 1 or the integer 0.

So in this case the entire expression "height == 5" is going to evaluate as the integer 1, because height is in fact equal to 5. Now if we had written "height == 3" instead, then this entire statement would have evaluated to 0, or false, because height is in fact not equal to 3.

Now let's suppose we create an integer variable called "holding_variable".

```
#include <stdio.h>

int main(void) {
    int height = 5;
    int holding_variable = 0;
    holding_variable = (height == 5);

    printf("The holding variable contains: %d", holding_variable);

    return 0;
}
```

Notice what we are setting "holding_variable" to. We are setting it to: (height == 5)

How can we do this? We can do this because (height == 5) will evaluate as an integer.

What do you think the output will be? The holding variable is going to be assigned an integer value of "height == 5". What will be the result? The result will be the integer 1, because height is in fact equal to 5. That will have the effect of setting "holding_variable" to be equal to the integer 1.

Ok, so let's continue.

Next, we are going to talk about the double ampersand which is the "Logical AND" operator. This operator works very similar to the equality operator in that it takes two values, acts on them, and the result is either a 1 or a 0. Like the equality operator, the result is an integer. The way the "Logical AND" operator works is simply this: If both operands are not zero, the result is 1. Otherwise, the result is 0.

For example, "4 && 1" is 1 (they are both non zero). "3 && 0" is 0 (One of the values is zero). "0 && 0" is of course 0.

Because the result of the logical AND operator is an integer, we can assign that result as a value to an integer. Let's do that:

```
holding_variable = ( 4 && 3 );
```

What is holding variable going to be set to now? Well, 4 is non zero, and 3 is non zero, so "4 && 3" should result in a 1. And as we run the program, you can see the result will be that holding_variable is set to 1.

Now let's go ahead and look at something a bit more complex.

```
#include <stdio.h>

int main(void) {
    int height = 5;

    if (height = 3) {
        printf("This executes.");
    }
}
```

Now what happens when we run this code? Why does it execute? The reason this printf executes is because "height = 3" will do two things. First of all, it will assign the variable "height" the value of 3. Second of all, the expression "height = 3" will itself evaluate to 3. The end result is to have written: if (3) {

```
#include <stdio.h>

int main(void) {
    int height = 5;

    if (height = 3) {          // Same as: if (3) {
        printf("This executes.");
    }

    return 0;
}
```

It is worth pointing out that if we had tried this instead:

```
if (height = 0) {   // Will not execute
    printf("This does not execute.");
}
```

that the printf() statement would not have executed. Why? Because "height = 0" will evaluate to 0, or false, and the if statement would not execute.

Now let's look at a much more complex example:

```
#include <stdio.h>

int main(void) {
    int height = 5;

    if (height = 3 && height == 5) {
        printf("The value of height is: %d", height);
    }
}
```

What do you think would happen here? If you try to predict the output of this code, you would find it difficult. It appears that we are doing this:

1. Set height equal to 3.
2. Check if height is equal to 5, which it won't be.
3. Evaluate the result, which would be "false".
4. The entire if statement is skipped over.

However, this is not the only possible way this code can be understood. Another way it can be understood is like this:

1. Set height equal to: (3 && height == 5)
2. Evaluate that entire result, which would be: 3 && 1, which is 1.
3. Now we are basically saying if (1) { and so the if statement executes, and height is set to 1!

Clearly this is not what we intended. And it brings up a very important point. Any time you are writing code and there is any possibility of ambiguity, you should use parenthesis to separate the expressions to ensure they are evaluated in the order you want. That way you are 100% sure every time that the code will execute as you intended. If we want to modify our above program to ensure that this evaluates correctly, we would write it like this:

```
#include <stdio.h>

int main(void) {
    int height = 5;

    if ( (height = 3) && (height == 5) ) {
        printf("The value of height is: %d", height);
    }
}
```

As you see, I have added parenthesis to remove any ambiguity. Now the code is far easier to read, and we do not have to wonder about the result.

Keep in mind that every expression we have looked at evaluates to something. For example, "height = 3" evaluates to 3, "height == 5" evaluates to 1 (if height is in fact equal to 5), and so on.

Alright, that concludes this lesson. I encourage you to practice with the concepts you have learned. Practice writing your own more complex if statements, using parenthesis to separate expressions. Practice demonstrating to yourself that == and && result in integers, and so on. As always, if you have any questions feel free to ask.

Lesson 11.5 : Introducing the concept of OR with Conditional Statements

The video for this lesson can be found here: http://youtu.be/s5GOtmQAj3Q

Every lesson in this course is designed to give you another tool that makes it possible for you to create more and more powerful programs. I imagine many of you are anxious to start writing more than just simple printf() statements, and we will get there.

In the last lesson we talked about using "AND" in a conditional flow statement, using the characters &&. This is useful when you want to evaluate multiple expressions and then execute code only when ALL of them evaluate to true.

However, there may be times that you want to execute code when ANY of them are true. For this, we have "OR".

In a conditional flow statement, the characters || (notice there are two of them, two pipe-bar characters) mean "or" in exactly the same way that && means "and".

Also, the || characters function in the same way as the && characters function. That is by evaluating only as much as they need to in order to confirm that the if statement should evaluate as true or false.

In English we might say "If height is equal to five OR width is equal to three". With OR we can write:

```
if (height == 5 || width == 3) {
```

Lets look at this in practice:

```
// Figure (a)

int height = 5;
int width = 0;

if (height == 5 || width == printf("123") ) {
    printf("If statement evaluated as true");
}
```

Output:

```
If statement evaluated as true.
```

Why didn't printf("123") execute? Because the if statement already met enough criteria to be evaluated as true.

Imagine in the middle of a clear day I say "If the sky is blue OR ..."

It doesn't matter what I say next. Why even read it? Since I used the word "OR", then nothing further needs to be considered. The entire statement is now true. This is even the case if the next thing I say is utterly absurd. Consider this:

If the sky is blue OR pink unicorns are flying around the sun : then do this :

Well, it will be done. Why? Because the sky is blue.

Whenever you use a conditional flow statement with OR, you must remember that the very first expression to evaluate as true causes the entire conditional flow statement to evaluate as true, and any other expressions will not be evaluated.

You can think of OR as saying "Evaluate the next expression only if the last one was FALSE". That is because if the last expression was TRUE, then the whole statement is already considered true. Notice this is the exact opposite of how && ("and") works. AND would say: "Evaluate the next expression only if the last one was TRUE".

Just as it is true with AND, you want to be strategic in your use of OR in a conditional flow statement. If it takes a great deal of computing power to determine one expression, and very little to determine another, you should always whenever possible order the expressions from least-computing-power to most-computing-power, just as with an AND statement.

Also, remember to be mindful of which expressions will be executed and which will be ignored entirely.

UNIT 12 : Loops and blocks of code

Lesson 12.1 : Introducing GOTO

The video for this lesson can be found here: http://youtu.be/Y-zfZUYX6ak

You have probably heard it said that using "goto" is considered one of the worst practices in programming. This is largely true. Why then am I spending an entire lesson teaching this? Well, several reasons:

1. Although it is poor practice in most languages, it is part of the core functionality built into your CPU chip. In other words, it is fundamental to computing as a whole, and any software written by any programming language uses it. You really should understand this concept.
2. It will help you to understand future lessons at a deeper level.
3. It will help you should you encounter this in some program someone else has written.

Now I want to add on a note to #3. You will never need to use a "go to" statement in C or most languages. There are almost always better ways to achieve the same purpose.

All of that said, let's begin.

Now that I have introduced conditional flow statements, I have shown you that it is possible to write a program that can choose to skip over instructions that should not be executed.

Consider this code:

```c
int height = 1;

if (height == 5) {
   printf("This gets skipped!");
}

// ... rest of program goes here ...
```

What is really happening here? At a machine code level, first a subtraction operation is performed using height and 5. If the result of that subtraction is zero (meaning that height IS equal to 5), then the printf() is executed. However, if the result is anything other than zero, what happens? It jumps over the code inside the if statement.

Now, I am sure you understand this at an abstract level, but how is this done exactly?

How is your CPU able to "jump over" instructions?

Recall from the lesson "Programs are data too" that a program is data that is stored in memory, just like any other data. We learned in earlier lessons about the instruction pointer built onto the CPU chip which contains the memory address of the next instruction to execute.

Each machine-code instruction of any program occupies a set number of bytes, and each instruction resides at a specific location in memory. One machine-code instruction might take up 2 bytes, and another might take up 4 bytes, etc.

To make this lesson even clearer, lets visualize this with real machine code. Do not worry about how this works or what it does. We are going to use our 16 byte ram and look at position 1000 (eight) where there just happens to be some machine code.

```
...
1000 : 1100 1101 <--- instruction pointer is pointing here
1001 : 0010 0001
1010 : 1100 1101 <--- start of next instruction
1011 : 0010 0000
...
```

Do not worry about how this works or what it does. The point is, this is real machine code in memory I have typed out for this lesson. These are not just random 1s and 0s, but actual machine code from a real program.

What you should notice here is that machine code looks just like anything else. These bytes could be characters, or numbers -- they just happen to be machine code. The instruction pointer on the CPU is pointing to position 1000 in that example, and knows therefore to execute that particular instruction.

Each instruction is located at its own address in memory. Each time your CPU is about to execute an instruction, the instruction pointer tells it where in memory that instruction is located. By changing the instruction pointer to point somewhere else instead, that instruction (the instruction located at the memory address you are now pointing at) will be executed instead of the instruction that would have been executed.

In other words, you can jump over code, or jump anywhere you want (forward or backwards) by simply changing the instruction pointer to a new memory address where a new instruction happens to be.

Imagine for example that we start at position 1000 (eight) in memory and start executing instructions one at a time until we get to position 1110 (fourteen). Lets suppose at position fourteen the instruction reads: "Change the instruction pointer so that it points back at 1000". What will happen? Well, our instruction will go BACK to position 1000 and execute all the instructions from 1000 back to 1110.

For this next example, I am making the assumption that every machine code instruction is exactly one byte long. This is not the case, so please keep in mind this is purely for illustrative purposes.

```
// ...
// 1000 : Instruction 1   <---------------------.
// 1001 : Instruction 2                         |
// 1010 : Instruction 3                    |
// 1011 : Instruction 4                    |
// 1100 : Instruction 5                    |
// 1101 : Instruction 6                    |
// 1110 : Set Instruction Pointer to 1000  ---'
```

Follow this in your mind. You will execute each instruction from 1 through 6, and then what? If this were the pen example in an earlier lesson, you are effectively "moving the pen backwards". Therefore, you will start all over from instruction 1.

Now to make this slightly more abstract, lets imagine that the memory address 1000 is given a name, such as label. In this case, we can effectively write the following:

```
label:
   ... the six instructions go here...

goto label;
```

The machine code instruction for this process is known as JUMP (JMP in assembly language).

Do not try this, even as an experiment. Why? Because if you look at the above example, this will go on forever. It will execute the six instructions, go back to instruction one, then execute the six instructions again, forever.

This has a name. Whenever it happens that the same instructions are executed over and over forever we call it an "infinite loop". We will talk more about loops and infinite loops in other lessons.

Why then use it at all? Because you can control it. Without it, conditional statements are impossible. However, when you are writing a program the real work involving "goto" statements is done behind the scenes. Also, instead of setting it to run forever, you can set it to execute a set of instructions a certain number of times - like 3 times. We will talk more about that in upcoming lessons.

Fundamentally what you have learned in this lesson is that there are mechanisms that make it possible to "jump around" in a program, and that this is done by changing where the instruction pointer is pointing. Whenever you set the instruction pointer to point somewhere other than where it was going to, that is known as a JUMP, or a "Go to" statement. We have also learned that this functionality is built right into your CPU chip. And finally, I have explained that you will never need to directly use this statement in most programming languages because that work is done for you behind the scenes.

Lesson 12.2 : About blocks of code

The video for this lesson can be found here: http://youtu.be/ohh-N9WBTmA

In the last lesson I explained that goto statements are done for you behind the scenes in most languages. Now I am going to explain how this is actually done.

First of all, as I have stated before, every single programming instruction itself has an address in memory. Consider this code:

```
// Figure (a)

int height = 5;

if (height == 5) {
    // ... some code ...
}
```

Every instruction in Figure (a) actually will have a memory address where that instruction lives. Therefore, to execute these instructions, your CPU must point the instruction pointer first at int height = 5; then at the if statement, and so on.

The way this actually works in machine code is a bit more complex, so keep in mind this is for illustrative purposes.

What you may be wondering at this point, considering we have seen this many times already throughout the lessons, is what on earth is the { and } character all about?

The answer is that these characters define memory addresses inside of the final machine code that will be your executable program. In other words, they are equivalent to the "goto" labels we saw in the previous lesson.

Consider this code:

```
if (height == 5)
{
    // ... some code ...
}
```

Do not worry that I changed the way the curly braces are formatted. Instead, try to understand this in a new way: The first { is an address in memory just before ... some code ... gets executed. The second } is an address in memory after ... some code ... gets executed.

In other words, we can define this same thing through the following example. This is not valid C code, but just something for illustrative purposes:

```
if (height == 5)
    start_of_code:
        // ... some code goes here ...
    end_of_code:
```

Notice I just replaced the braces with a label, similar to the last lesson. Now, lets consider the if statement itself in more clear terms:

```
compare height to 5
Check the zero flag.
```

If the zero flag is set to 1 (that means height is equal to 5) then *goto* start_of_code.

If the zero flag is set to 0 (that means height is not equal to 5) then *goto* end_of_code.

Now, lets take this one level deeper.

We do not need both of these "if" statements. We only need one. We just need one that says skip over.

Consider this same example again:

```
if (height == 5)
    start_of_code:
        ... some code goes here ...
    end_of_code:
```

Which translates to:

compare height to 5
Check the zero flag.

If the zero flag is set to 0 (that means height is not equal to 5) then *goto* end_of_code.
This means to skip over: ... some code goes here ...

I took out the instruction which went to start_of_code. Why? Because that is what would have happened anyways. In other words, the instruction pointer would have naturally gone to the next instruction, and so we do not need that line. We only need the line that says skip over.

Whenever you have a section of code that is defined within curly braces like { } we call that a block of code. A block of code is best understood as two labels one of which indicates the start of the block, and the other which indicates the end of the block of code, and the code itself contained inside.

Not all languages define blocks of code in this way. Some define blocks of code using simple indenting. Python is one such language. In Python, you would write an if statement like this:

Python Example:

```
if height == 5:
    ... this is a block of code that will execute ...
    ... only if height is equal to five ...

... the rest of the program goes here ...
```

Notice that we do not specify either a { or a }. We do not need to. Python is designed to understand blocks of code by simply indenting the block of code. This is a good idea since in C as well as most languages, even those using curly braces, this is usually how programmers typically choose to format blocks of code (by indenting the contents inside the curly braces).

Remember that all programming languages can only create machine code understandable by your CPU. No programming language can do something that another cannot. All are limited to the machine code instructions your CPU can execute. Once you learn how to program in C, or any language, you can take that understanding and learn any language you wish.

Throughout this course we will look at the various ways that these same operations and concepts are implemented in a variety of languages.

Lesson 12.3 : Introducing Loops

The video for this lesson can be found here: http://youtu.be/6lMW71vCJ-k

The last several lessons have explained how you can use conditional statements (like the if statement) to "skip over" code. We know that this works by trying to evaluate an expression as true or false. If it evaluates as true, then the code inside the block is executed. If it evaluates as false, then that block of code is skipped over.

For example:

```
if (height == 5) {
    // ... some code here ...

} // <--- end of block of code

// ... rest of program here ...
```

Intuitively, you would read this as "If the height is equal to five, THEN go into ... some code here ..., otherwise, go to ... rest of program here ...

However, this is not really the case. A conditional statement is really a conditional "jump over". That is to say, if the result of the if statement evaluates as false, then you jump over the block of code.

It is important to understand that all conditional statements have a built in goto statement that you cannot see. That goto statement will jump over the block of code when the conditional statement evaluates to false. If it evaluates as true, then there will be no jump and therefore the very next instructions will execute as if the conditional statement had not even been there.

For example:

```
int height = 5;

if (height == 5) {
    printf("Hello Reddit!  ");
}
```

If height is equal to 5, then the printf() instruction will execute as if there had not been any if statement altogether. The if statement really says:

"We *might* want to jump over this printf() statement. do so
if height is NOT equal to 5."

Now, lets continue.

In our last example we saw a simple example of a loop. The example we looked at was an infinite loop where the last instruction simply said to start over. Now lets look at a more concrete example.

The most basic type of loop is the "While" loop. The way it works is very simple: You have a block of code and a conditional statement. The last line in the block of code is a JUMP back to the conditional statement.

Lets see this in action.

```
Int height = 5;

while (height < 10) {
   printf("Hello Reddit!  ");
   height = height + 1;
}
```

The conditional statement here is "height < 10".

Now, lets look at how it will actually be understood by your computer. This is not valid C and is purely for illustrative purposes:

```
start_of_loop:
   compare height and 10
   if height is greater than or equal to 10: goto end_of_loop  <---yes! "greater than or equal to"

   printf("Hello Reddit! ");
   increase height by one.

   Goto start_of_loop

end_of_loop:
```

Did I make a mistake? The original statement said "height < 10", why therefore did I say "if height is greater than or equal to ten" in the above example? The answer is that we must think of this conditional statement as a conditional "jump over", not as a conditional "proceed".

The default behavior of your CPU is to proceed. We use a conditional statement only to change that default behavior. To have a machine code instruction that says "Go into the block of code if this is true" is just a waste of resources, since that is the default behavior anyways. What we really need is a statement that says "jump over this block of code if this is false".

Therefore, we say "greater than or equal to" in machine code because that is the condition on which we would jump.

The way you would intuitively read this code:

```
while (height < 10) {
    // ... More code here
}
```

is: "While the variable height is less than ten, then do what is inside the block of code." But the way your computer would read it is:

```
compare height to ten
goto end_of_block if height >= 10
... otherwise we will execute these instructions ...
end_of_block:
```

With a while loop, every time you reach the end of the block of code you return to the beginning where a simple question is asked: "Do we jump over the block of code this time?". If the answer is yes, then you jump over the block of code and the loop is over. If the answer is no, then the default behavior is to execute the instructions inside the block of code again.

Now lets look again at this example:

```
int height = 5;

while (height < 10) {
    printf("Hello Reddit!  ");
    height = height + 1;
}
```

So here is what happens:

Lets check to see if we jump over the block of code.
Do we? No, because height is not greater than or equal to ten.
Therefore we proceed. Now we execute our printf() function.
Then we add one to height.

Now what?

Since we have reached the } which defines the end of the block of code, we return to the start of the loop again. In other words, we repeat these same instructions, exactly as written, starting with the conditional statement.

Once we have done this five times, it will come to pass that when we read through these instructions, height will now be set to 10 (since each time we loop we are adding one to height). Therefore, the conditional statement will now evaluate like this: "Height is now no longer less than 10. Therefore, jump over the block of code." And thus ends our loop.

In this lesson I introduced you to the first and simplest kind of looping statement, the while loop. We learned that a while loop is effectively a goto statement built into a conditional statement. The conditional statement is evaluated every time the loop executes, including the first time. The loop will repeat itself until the conditional statement evaluates as false.

Lesson 12.4 : Introducing custom functions

The video for this lesson can be found here: http://youtu.be/r6NFEkABZlw

In general, a function is a block of code that you jump to from anywhere in your program. At the end of the function, you simply jump back to where you were.

On a machine code level, there is more that goes on here. For example, the assembly language instruction for calling a function is not JMP, it is CALL. There are certain differences between "JUMP" and "CALL" but we will not need to get into this as part of this lesson. For the sake of this lesson, the explanation I gave you above is enough. We will expand on this definition as we proceed.

Lets look at an example of a function in a real program:

```
#include <stdio.h>

int my_function(void);

int main(void) {

   printf("Calling our function...  ");

   my_function();
   // <--- function returns here when finished.

   return 0;
}

int my_function(void)
{   // <--- start_of_function
   printf("Inside the function!  ");

   return 1;   // <--- return to main program
}
```

Output:

```
Calling our function...
Inside the function!
```

Now lets talk about this. First of all, when we executed this line of code:

```
my_function();
```

This effectively means to jump to the line I marked as "start_of_function". We have defined this function as (void) which means that we are not sending it any parameters. If we wanted to, we could put some parameters in the parentheses and we will get to that in a later lesson.

One thing which may seem puzzling to you is that I have seemingly created the function twice. I have one line of code above main() which seems to create the same function as the code under main(). Why is that?

The top code with my_function tells C that we plan to have a function called my_function(). We are also telling C that we intend this function will return an integer, and that it will have no parameters.

If we did not do that, then when C reached the point in main() where we call our function, that function would logically not exist yet. That is because it is not really created until after main(). It is always good practice to define functions. By defining a function you let C, yourself, and anyone who reads your code know what functions you have made.

The above paragraph is only partially true. In truth, you can sometimes use a function that you haven't yet defined, depending on the function. We will discuss that more later. For the purpose of this lesson you should consider that any function should be defined before it is called.

You can call one function from another function. For example, we could have my_function look like this:

```
int my_function(void) {
    printf("Inside my_function  ");

    my_other_function();

    return 1;
}

int my_other_function() {
    printf("Inside my_other_function  ");

    return 1;
}
```

Keep in mind for the above example we should have defined both my_function and my_other_function like this:

```
int my_function(void);
int my_other_function(void);
```

Such definitions should always go at the top of your source-code as I illustrated in my sample program.

Very soon we will get into some powerful uses of functions, but this lesson is intended to only serve as an introduction to how to create and call your own custom built functions.

Lesson 12.5 : Introducing Boolean Logic

The video for this lesson can be found here: http://youtu.be/CV95MQ3hiLM

In earlier lessons on binary, I showed you the basics of how to "count like a computer". In this lesson, I am going to begin to show you how to "think" like one. Before we begin, however, I need to introduce a new concept as well as a new term.

"The Boolean"

Now, what exactly is a boolean? A boolean is simply a "thing" that can only be set to two values, for example "true" and "false", or "1" and "0". Because a bit can also only have two values, "1" and "0", we can think of a bit as a kind of boolean. This means that operations and concepts that apply to booleans, also apply to bits. It is this fact that makes understanding booleans useful.

There are two reasons you need to learn booleans:

1. Complex programming logic relies on booleans. When you need to determine something beyond a simple comparison, you can use booleans to do so.
2. The ability to manipulate data by manipulating individual bits requires that you understand booleans.

In fact, we will be looking at examples of this very soon.

The science of studying booleans is rooted in the study of logic. In fact, boolean operations can be thought of as logical operations.

Let me show you a simple example. Suppose I told you "Everything I say is false". Then you know that whatever I tell you, you simply need to flip its meaning. If there are only two possible values, "true", and "false", then you know to apply the following rule:

1. If I give you a "true", you change it to a "false".
2. If I give you a "false", you change it to a "true".

Now, if we think about this in terms of binary, we could say this: If I give you a 0, you change it to a 1. If I give you a 1, you change it to a 0.

This is the simplest kind of boolean operation, called "NOT". NOT just simply flips the value given to it. Here is how NOT works:

NOT 1 = 0
NOT 0 = 1

Notice that "NOT" only requires a single input, and gives a single output. This is what makes NOT the simplest of the boolean operators we will be looking at.

Now, let's suppose I told you two statements:

1. The earth is round.
2. The sky is up.

Now suppose I told you to evaluate whether I told you the complete truth, or not. For this to be the complete truth, *BOTH* statements must be true. If either
statement is false, then I have not given you the complete truth. We can think of this as a boolean operation requiring two inputs, and that both inputs must
be true for the output to be true. We can say it like this:

TRUE and TRUE = TRUE

TRUE and FALSE = FALSE (because both were not true)
FALSE and TRUE = FALSE
FALSE and FALSE = FALSE

Or, if we want to express "true" as "1", and "false" as "0", we can say:

1 AND 1 = 1
0 AND 1 = 0
1 AND 0 = 0
0 AND 0 = 0

The above is an example of a "truth table". A "truth table" is a table showing all possible inputs, and the final result, for a given boolean operation.

In this case, we are looking at the boolean operation "AND", which takes two inputs (unlike NOT) and returns a single output.

Now let's look at another. The "OR" boolean operation works like this: It takes two inputs, and if either input is TRUE then the final result is TRUE.

Here is the truth table for OR:

0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1

There are many other examples I can give you, and in fact there is a whole branch of mathematics devoted to this, which is called "Boolean Algebra". This is
a fundamental topic for electronics, and because computers are electronic devices, the value of understanding booleans carries over into programming.

So let's recap some of what we just learned:

1. A boolean is a "thing" that can be set to one of two values, such as "1" and "0" (synonymous with "true" and "false")
2. A boolean operator uses a specific set of rules in order to change one or more booleans to a new boolean

One major concept to understand is that when you are working with a Boolean, you can only ever get a Boolean back. In other words, any mixture of operations involving 1s and 0s will result in a single one or a single zero. The "NOT" operation is only a first example. There are more complicated operations as we will now see:

Keep in mind the following:

1. Each Boolean operator takes a set number of inputs. In the case of AND, and OR, that number is two. This means that AND as well as OR takes exactly two inputs. Both inputs are Booleans.
2. Every Boolean operator will have a truth table. The truth table will show all possible results for that operator. The number of rows for such a truth table will always be two to the power of the number of inputs required.

3. No matter how complex the operator, or mixture of operators, the final result of any Boolean operation is a single Boolean.

Now, let's try mixing two boolean operators together. Suppose I said "NOT OR". How would we construct that truth table?

Well, let's start with the truth table for OR:

```
0 OR 0 = 0
0 OR 1 = 1
1 OR 0 = 1
1 OR 1 = 1
```

Now, let's take the result of this, and send it into "NOT" :

```
0 OR 0 = 0; NOT 0 = 1
0 OR 1 = 1; NOT 1 = 0
1 OR 0 = 1; NOT 1 = 0
1 OR 1 = 1; NOT 1 = 0
```

Now we can construct the final truth table:

```
0 NOR 0 = 1
0 NOR 1 = 0
1 NOR 0 = 0
1 NOR 1 = 0
```

Notice I gave a better name to "NOT OR". I chose to call it NOR. Like "NOR" which means "NOT OR", there is another operation called "NAND" which means "NOT AND".

In the field of electronics these Boolean operations are formally known as logic gates. An "AND" logic gate really works like this:

You have two incoming wires and one outgoing wire. The outgoing wire will have an active current only if both incoming wires had an active current. Think of the two incoming wires as the Boolean inputs. Think of "active current" as a 1. If both "wires" (incoming Booleans) are active (1) then the result is a wire with active current (1). In other words, 1 AND 1 = 1.

Why am I teaching you this? Because C as well as many programming languages give you the ability to work with Booleans on this fundamental level, and as you will see it can be very useful. Besides using it in your own programs, you are likely to come across it being used in programs you might read. Remember that underneath it all, your computer is simply a very complex electronic device. Logic gates play a major role in how your computer works, and by understanding Booleans you will be able to perform powerful operations on data as well as understand your computer at a deeper level.

Lesson 12.6 : Introducing Bit Masking

The video for this lesson can be found here: http://youtu.be/QXaW5lj7Rs0

In this lesson I am going to show you why Boolean operations are so important. Earlier I have shown you that for ASCII characters, you know an uppercase or a lowercase letter based on a certain bit. Lets review that briefly:

```
0100 0001 : 'A'
0100 0010 : 'B'
...

0110 0001 : 'a'
0110 0010 : 'b'
...
```

The third bit defines if this is an uppercase or lowercase letter. How can we see that bit? The answer is by using Boolean operations. The technical term for what I am about to show you is called "bit masking".

The way you can see (or change) information about a single bit is by constructing a bit mask. Now I am going to illustrate this concept for you.

Imagine that I have the following byte, the letter 'A'.

```
0100 0001
```

I need to see the THIRD bit only. I do not care about the rest. I need to have some way of determining if the third bit is turned on, or the third bit is turned off. In other words, I need some unique operation that will result in one result if the third bit is turned on, and a different result if the third bit is turned off.

First of all, lets construct our bit mask:

```
0010 0000
```

Why is that the bit mask?

Think of it like having eight windows in a row. You are only interested in what is behind the third window. Therefore, you close all the others (set them to 0), and what you are left with is one open window.

Lets put our byte for 'A' and our bitmask together.

```
0100 0001 <-- 'A'
0010 0000 <-- Bitmask
```

Now lets use the AND Boolean operator on each bit. Remember, 1 AND 1 is 1. Anything else is 0. Watch how this works:

```
0100 0001 <-- 'A'
0010 0000 <-- Bitmask
---------
0000 0000 <--- After ANDing 'A' with the bitmask
```

What is the result? We get all zeroes. What if this had been a lowercase letter?

```
0110 0001 <-- 'a'
0010 0000 <-- Bitmask
---------
0010 0000 <--- After ANDing 'a' with the bitmask
```

Now here we can see the benefit of a Boolean operation. We now have a way to test a single bit in our byte to determine conclusively if it is uppercase, or lowercase. The process is very simple:

Given any character that we know is either uppercase or lowercase, by ANDing that character with 0x20 (0x20 means hexadecimal 20, binary 0010 0000), we know it is uppercase if the result is 0. We know it is lowercase if the result is 0x20. There are no other possible outcomes.

Why are there no other possible outcomes? Because our bitmask has only one bit turned on. When using AND, you can never get a result with more bits turned on than your bitmask.

Lets see this in action with a real function:

```
int is_lowercase(char test_character) {
    if (test_character & 0x20) {
        return 1;
    }

    return 0;
}
```

That is it. That is all you have to do in order to check if a letter is lowercase or uppercase. Now you can see why Booleans are important.

Notice that I used one & character. That is because one & character means "Boolean AND". That is NOT the same as the && characters which mean there will be another expression evaluated.

1. & means "apply the boolean AND operation"
2. && means "Another expression follows"

Let's walk through this function.

```
Int is_lowercase(char test_character) {
```

Here we are saying that this function will return an integer. We are giving it the name is_lowercase, and we are saying that it will accept a character as a single parameter.

From now on inside this function, test_character will refer to whatever character was sent to the function.

```
    If (test_character & 0x20) {
        return 1;
    }
```

This is a single expression: test_character & 0x20

(As stated above, this is NOT related to && in any way)

This just means we are taking whatever character was sent to the function, and doing the Boolean operation AND on each bit inside of the byte. What we will get back is a single byte. It is the exact same thing as this:

```
0110 0001 <-- 'a' (could be any character, this is test_character)
0010 0000 <-- Bitmask (this is 0x20)
---------
0010 0000 <--- After ANDing 'a' with the bitmask (this is the result)
```

This expression will result in one of two possibilities. It will be 0x20 if test_character turns out to be lower case. It will be 0 otherwise. If it is zero, then it will jump over this conditional statement and execute the very next instruction, which is return 0

If however it is not zero, which in this case it is not, then it will continue with the default behavior of executing whatever instructions are inside the block of code associated with our conditional statement. This means it will return 1.

Now, because our function returns 1 if it is lowercase, we can use is_lowercase() inside a conditional statement very easily. Consider this:

```
if (is_lowercase('a')) {
    printf("It is lowercase  ");
}
```

If the letter really is lower case, then is_lowercase() will return a 1. Therefore, the result of our if statement will be: if (1) {

[Edit: Quick note. The operations in this lesson, while heavily related to the Boolean operations of the previous lesson, are known technically as "bitwise" operations. We will discuss this more later.]

Here is a complete program that you can experiment with which illustrates this concept:

```
#include <stdio.h>

int is_lowercase(char);

int main(void) {

    char my_char = 'a';

    if (is_lowercase(my_char)) {
        printf("It is lower case!");
    }

    return 0;
}

int is_lowercase(char test_character) {
    if (test_character & 0x20) {
    return 1;
    }

    return 0;
}
```

Lesson 12.7 : Using bit masking to change data

The video for this lesson can be found here: http://youtu.be/ajBT1uOa9Ws

This is the logical continuation from the previous lesson. Just as you can use a bit mask to check if a certain bit (or bits) are set, you can also use a bit mask to set or unset the bits themselves.

For example, you can take a lowercase letter and make it a capital letter. However, for this lesson we will be doing something slightly different. We will be taking actual numbers (digits zero through nine) and turning them into characters on your keyboard.

Let's imagine this byte:

```
0000 0010 <--- Number 2 (Our "target")
```

This is the number two. We know from our earlier lessons that the character two would have the third and fourth bit turned on, and would look like this:

```
0011 0010 <--- Character '2'
```

There are a variety of ways to do this, but I want to show you how to do this using bitwise OR.

First, here is our bit mask:

```
0011 0000
```

OR will not turn off any bits, it can only turn bits on. Any bit that is turned on in the bit mask will be turned on in the result. Any bit that is already turned on in the target will be turned on in the result. Remember that 0 OR 0 = 0. Anything else = 1.

Now lets apply the bit mask using OR:

```
0000 0010 <--- Number 2 (Our "target")
0011 0000
---------
0011 0010 <--- after OR, we get the character '2'
```

In C, you specify a bitwise OR using the | character (once, not twice).

Do not confuse this with using || in a conditional statement.

This is the same reasoning we saw in the last lesson using & instead of &&.

Now lets see this in action.

```
char my_number = 2;

my_number = my_number | 0x30; // <--- 0x30 is: 0011 0000

printf("We can now print this number as a character: %c",my_number);
```

If we wanted to print a '2' without actually changing the contents of the byte, we could do this:

```
char my_number = 2;

printf("We can now print this number as a character without changing it: %c", (my_number | 0x30));
```

The above code works because printf() %c expects a character. (my_number | 0x30) is a character.

Remember that this only works for numbers 0 through 9.

What would have happened if we had started with a character 0 through 9 instead? Let's see:

```
0011 0010 <--- Character '2' (Our "target")
0011 0000
---------
0011 0010 <--- after OR, we get the character '2'
```

We will get '2' as a result. In other words, we do not have to first check if it was a number or a character, because we will always get the character version back. This is because OR will set bits to be on regardless of if they were already on or not.

Notice that you could not use OR in order to "see" a bit like we did with AND in the last example. Similarly, you cannot use AND to turn on a bit as we did in this example. Let's look at why this is:

```
0100 0001 <--- 'A'
0010 0000 <--- bitmask to turn the capital letter to lowercase
---------
0000 0000 <--- Result using AND. All we get is ZERO
```

Now, with an OR on the other hand, we cannot test individual bits like we did with AND. Consider this:

```
0100 0001 <--- 'A'
0010 0000 <--- bitmask to check if it is capital or lowercase
---------
0110 0001 <--- Result using OR.
```

Using OR, we get a lowercase 'a' regardless of if we started with a capital 'A' or a lowercase 'a'. Because we will always get the same result, we will never be able to know what we started with.

In other words, OR will set the bits no matter what they were. Any bit that is a 1 in the bit mask (or the target), will become a 1 in the final result.

with AND, any bit that is 0 in the bit mask (or the target) will become a 0 in the final result.

OR is best suited to unconditionally turning bits ON (setting them to 1)

AND is best suited to unconditionally turning bits OFF (setting them to 0), and also for testing which bits are on, because it will get different results depending on what bits were on and off to start with.

What if we want to alternate a bit from 1 to 0 or vice versa? For this, we use a bitwise operation called "XOR" which is short for "Exclusive OR". The OR we have been using is also called "Inclusive OR"

The idea of exclusive or is "It can be one, or the other, but not both". Exclusive OR is just like OR, except that 1 XOR 1 = 0. Everything else is the same.

Compare these two truth tables between OR and XOR:

```
0 OR 0 = 0    0 XOR 0 = 0
0 OR 1 = 1    0 XOR 1 = 1
1 OR 0 = 1    1 XOR 0 = 1
1 OR 1 = 1    1 XOR 1 = 0 <-- the only difference
```

This difference exists for a key reason. Observe what happens if we take our capital 'A' and apply our bit mask using XOR instead of OR.

```
0100 0001 <--- 'A'
0010 0000 <--- bitmask
---------
0110 0001 <--- 'a' is the result of XOR
```

Notice there is no difference between using XOR and using OR in the above example, but now lets take the output of this result (the 'a') and apply the same bitmask with XOR again:

```
0110 0001 <--- 'a'
0010 0000 <--- bitmask
---------
0100 0001 <--- 'A' is the result of XOR
```

We flipped it back to 'A'. In other words, XOR will toggle, or alternate any bits that are turned on in the bitmask. In this case, since the 3rd bit was turned on, XOR toggled the third bit on and off.

Notice that because we get a different result when we start with 'A' vs 'a', it is also possible to use XOR to test if a bit is turned on or off.

The only missing piece of the puzzle now is, how do you write XOR in C? The answer is by using the ^ character. Lets review:

1. & is AND
2. | is OR (inclusive OR)
3. ^ is XOR (exclusive OR)

The final notes I would add are this: When you are looking at source code that you will encounter, you will see these bitwise operations being used. This is true not just for C, but for any programming language. Being able to read that source code critically depends on your ability to understand why someone would use AND, vs OR, vs XOR.

In summary, remember this: By using bitwise operations you can test, set, unset, and toggle any bit in any data you wish.

Supplemental video on displaying data in memory as binary: http://youtu.be/MV-mFVX7VfY

Lesson 12.8 : Introducing data structures

The video for this lesson can be found here: http://youtu.be/O-qzfc6G6QI

Up until now we have only worked with simple data, starting at the basic data types and working our way into simple arrays such as text strings. In earlier lessons I have said that the only way to "see" or "work with" any data that is larger than a single variable of a basic data type (int, char, etc.) is by using pointers.

In this lesson we are going to explore what this actually means. What do I mean when I say "see" data? Well, real data comes in specially formatted packages which can only be understood by first understanding how it is formatted, and secondly by breaking it down into smaller pieces.

Here is a simple example:

    20091005 <-- Today's date in YYYYMMDD format (year, month, day)

This is a very basic data structure. Why is it a data structure? Because we are actually storing three different bits of information (data) together. It is a string of text, but the real meaning of the string of text is not "20091005", it is a date - October 5th, 2009. In other words, to be properly understood it must be broken into pieces, one unique piece for: month, day, and year.

First, lets create this string of text.

```
char date[] = "20091005";
```

Lets suppose we want the following printf() statement:

```
printf("The year is ___ and the month is ___ and the day is ___  ");
```

Notice that you cannot do this using the string we just created. It is too complex. It is a data structure. What we want is a way to break the data structure down into pieces, so that we can understand each piece properly.

We are using a date string as an example, but this same principle applies broadly in the field of computing. For example, graphics require data structures that contain different values for colors. Here is a simple example of such a data structure, which you have seen if you have worked with HTML:

```
color = FF22AA
```

This is a data structure which defines a color. For those not familiar with this, let me break it down. FF means how much RED. 22 means how much GREEN. and AA means how much BLUE. By mixing these values, you can get a wide spectrum of colors.

However, a program like a web browser must be capable of taking FF22AA and converting it into three data elements, and then use those three elements to draw the proper color.

Lets go back to our printf() statement. We want to print the year, month, and day separately.

First of all, every data structure has a format. Some formats can be enormously complex, and could involve hundreds of pages of detail. Other formats, like this one, are simple.

In this case, we would define the format like this:

The first four characters are the year. The next two characters are the month. The final two characters are the day.

We could also word it like this:

```
year = 4 bytes
month = 2 bytes
day = 2 bytes
```

To parse any data structure, you must know its format. Once you know its format, the next step is to create a parsing algorithm.

A parsing algorithm is a "small program" designed to "understand" the data structure. In other words, to break it into easily managed data elements.

Lets create a pointer to our string:

```
char *my_pointer = string;
```

Why did I create a pointer? Remember, you have to create a pointer in order to see or work with anything larger than a single variable of the basic data types (int, char, etc). The pointer is like your eyes scanning words on a page to understand the meaning of a sentence.

What will our pointer do ? It will scan through this data structure string, and we will use the pointer to understand our data structure one byte at a time.

Since we know that the year will be four characters in size, lets create a simple string to hold it:

```
char year[5] = "YYYY";
```

Why 5 ? Because there will be FIVE elements in this array. The first four are the letters "YYYY". And the fifth will be the NUL character (all 0 byte) which terminates the string. Note that the proper term for this character of all 0 bytes is NUL with one L, not two. There is a reason for that which will be discussed later.

As you just saw, it takes 5 character bytes in order to properly contain the entire null terminated string "YYYY". Four bytes for the Ys, and one for the NUL at the end.

This is important, so remember it. The number in brackets for an array is the total number of elements you are creating. Whenever you intend for an array to hold a null terminated string, always remember to allow room for the final termination character. So if we plan to create a null terminated string with 8 characters, we need an array with 9 elements.

Notice that for the year array I set this to YYYY temporarily and we will replace those Ys with the actual numbers later. It is always good to initialize any variable, array, etc to something so that you do not get unpredictable results.

Now, lets do the same thing for month, and day:

```
char month[3] = "MM";
char day[3] = "DD";
```

Notice again I put enough room for a \0 terminating character. Just to see how this works, lets see this in action before we parse our date string:

```
printf("The Year is: %s and the Month is: %s and the Day is: %s  ",year, month, day);
```

Output:

```
The Year is: YYYY and the Month is: MM and the Day is: DD
```

These arrays: year, month, day are known as "data containers" and are designed to hold the individual elements of our parsed date string. The logic here is simple:

1. We have a string of some data format which really contains 3 different bits of information.
2. We plan to "understand" those pieces.
3. Therefore, we need to create containers to hold them so that when we "pull them out" of the main data structure we have somewhere to put our newly understood data.

Now, lets begin. First of all, we know that the first four characters are the year. We also know our pointer is pointing at the first such character. Therefore:

```
year[0] = *my_pointer;        // first digit; same thing as *(my_pointer + 0)
year[1] = *(my_pointer + 1);    // second digit of year
year[2] = *(my_pointer + 2);    // third digit
year[3] = *(my_pointer + 3);    // fourth digit
```

We do not need to write year[4] = '' because it has already been done. How was it done? When we wrote the string "YYYY" C automatically put a NUL at the end. Think of this process as simply replacing the four Ys with the 2009 in the date string. Make sure you understand the process of how we used the pointer to assign values to the individual characters in the array.

Notice that rather than actually move the pointer, we have kept it pointing to the first character in our date string. We are using an "offset" which we add to the pointer in order to obtain the value for bytes that are further away in memory.

saying *(my_pointer + 3) is just like saying "Whatever is at the memory address in (my_pointer + 3). So if my_pointer was the memory address eight, then (my_pointer + 3) would be the memory address eleven.

Now, lets do the same thing for month:

```
month[0] = *(my_pointer + 4);
month[1] = *(my_pointer + 5);
```

Finally, day:

```
day[0] = *(my_pointer + 6);
day[1] = *(my_pointer + 7);
```

Notice that each array starts with ZERO in brackets. That is to say, we do not start with day[1], but with day[0]. Always remember this. Every array always starts at 0. So lets review a couple important facts concerning arrays:

1. When you define the array, the number in brackets is how many elements of the array you are creating.
2. When you use the array, the number in brackets is the "offset" from the first element of the array. [0] would mean no offset (thus the first element). [2] would mean an offset of +2 from the FIRST element, thus [2] is actually the third element. [8] would be the 9th element. Remember, we start at 0 and count from there.

And we are done. Now I have shown you the first example of how you can use a pointer to truly "see" data that is more complex than a simple text string.

Now, lets finish our printf() statement:

```
printf("The Year is: %s and the Month is: %s and the Day is: %s  ",year, month, day);
```

Here is the completed program which illustrates this lesson:

```
#include <stdio.h>

int main() {

    char date[]      = "20091005";

    char year[5]    = "YYYY";
    char month[3] = "MM";
    char day[3]     = "DD";

    char *my_pointer = date;

    year[0] = *(my_pointer);
    year[1] = *(my_pointer + 1);
    year[2] = *(my_pointer + 2);
    year[3] = *(my_pointer + 3);

    month[0] = *(my_pointer + 4);
    month[1] = *(my_pointer + 5);

    day[0] = *(my_pointer + 6);
    day[1] = *(my_pointer + 7);

    printf("The Year is: %s and the Month is: %s and the Day is: %s  ", year, month, day);

    return 0;
}
```

UNIT 13 : Basics of Algorithm Design

Lesson 13.1 : The Basics of Algorithm Design Part One

The video for this lesson can be found here: http://youtu.be/DscD95i1FZY

In the previous lesson I showed how you can take a string of text formatted as a date in YYYYMMDD format, and convert it into three valid null terminated strings, one for year, month, and day.

This was a simple example of something called an algorithm. An algorithm is a sequence of steps you take inside of a program to perform some complicated task. Usually algorithms involve loops as they need to repeat instructions numerous times.

Algorithm design refers in part to taking some process you want to accomplish and turning it into a working system of loops which get the job done. In this lesson I am not just going to simply explain what an algorithm is. I am going to show you the thought processes behind it.

Why do you need to do this? Because algorithms can often expand into more lines of code than you could write in a reasonable amount of time. For example, try to write an algorithm that uniquely draws the correct color for every pixel on your screen with one line of programming code per pixel.

Properly being able to write and read complex algorithms is a critical skill that any serious programmer needs. At this stage in the course, I am going to show you how to construct and de-construct some simple algorithms using our previous example as the base point.

Keep in mind from the previous lesson that we had something that looked like this:

```
// Figure (a)

year[0] = *my_pointer;        // same as *(my_pointer + 0)
year[1] = *(my_pointer + 1);
year[2] = *(my_pointer + 2);
year[3] = *(my_pointer + 3);
```

What you should notice about this is that there is "a number" that changes according to a set pattern. In this case, it is simply adding one each time. Whenever you see code that repeats and uses an incrementing number or numbers then you should realize that you can use a loop to achieve the same goal. Here is how we would do this.

First ask yourself how many times will the loop need to execute? Four times. Why? Because there are four lines of code in Figure (a). Let's therefore construct the loop skeleton:

```
int i = 0;
while (i < 4) {
    ... code goes here ...
    i = i + 1;
}
```

Why is this the loop skeleton? Because it will execute whatever is inside it four times. Each time this loop executes, the variable i (which starts at 0) will increase by one.

The next step is to take a candidate line of code from our working code, and simply cut-and-paste it into our loop. A candidate line of code is a line of code that appears well suited for the loop. Our first line of code would be a poor candidate in its present form. Let me show you:

```
year[0] = *my_pointer;
```

That is our first line of code. What is wrong with it? It only takes into account one changing value (the 0 in brackets) but it ignores the other changing value (the number after my_pointer). A much better candidate will take into account both changing values.

I have picked one such candidate and pasted it into our loop:

```
while (i < 4) {
    ... code goes here ...
    year[2] = *(my_pointer + 2); <--- I just cut and pasted this.
    i = i + 1;
}
```

Notice I have not changed anything yet. I just pasted it as is.

Now, the final step is to change what we pasted so that it will work properly within the loop. In other words, we want to make sure that our newly constructed loop is identical to the code we started with.

```
while (i < 4) {
    year[i] = *(my_pointer + i); <--- I just changed the number to i
    i = i + 1;
}
```

Why did I change the number to i? Because i is a variable that will change exactly the same way our number did in Figure (a).

Now, lets expand this loop to see what it will do. We do this by stating the starting condition, and then writing out the code that will execute as one line for each time the loop will execute. Note that i = i + 1 is part of the code that will execute. I put all the code that will execute each time on its own line for better readability.

```
    i = 0;
    year[i] = *(my_pointer + i);     i = i +1;
    year[i] = *(my_pointer + i);     i = i +1;
    year[i] = *(my_pointer + i);     i = i +1;
    year[i] = *(my_pointer + i);     i = i +1;
```

Here it will stop, since the next time the conditional statement is evaluated the variable i will no longer be less than four (it will be exactly four) therefore, it will jump over our code at that point.

Notice that I put the incrementing statements at the end of each line so that it is easier to read. These are "reminders" so I can see exactly how the loop is changing each time.

If you mentally follow that in your mind, you should see the result is this:

```
    year[0] = *(my_pointer + 0);
    year[1] = *(my_pointer + 1);
    year[2] = *(my_pointer + 2);
    year[3] = *(my_pointer + 3);
```

If you do not see how this is, the next example will help to clarify this.

Now, lets do the same thing with month, and day.

Recall that month looks like this:

```
month[0] = *(my_pointer + 4);
month[1] = *(my_pointer + 5);
```

Now lets turn it into a loop. Notice there are two different incrementing values this time. You have a 0 and a 1, and you also have a 4 and a 5. To turn this into a loop, we need two variables. (Actually we don't, but we will get to that.)

Let's reset the variable i back to 0, and lets create a second variable called j.

Also, I am going to introduce you to a new shortcut. You can use i++ instead of i = i + 1;

In general, any variable you need to increment by one, you can say: variable++.

```
i = 0;        // We do not need to say int i = 0 because we already did that.
int j = 4;

while (i < 2 && j < 6) {
   month[i] = *(my_pointer + j);

   i++;
   j++;
}
```

Let's expand it:

```
i = 0; j = 4;
month[i] = *(my_pointer + j);     i = i+1;     j = j+1;
month[i] = *(my_pointer + j);     i = i+1;     j = j+1;
```

What will it expand into? To find out I am just going to plug in the values for i and j that I defined for just the first line of code:

```
month[0] = *(my_pointer + 4); i = i+1; j = j+1;
```

I put in 0 for i, and 4 for j. This is what I defined them to be. Why did I define them to be 0 and 4? Because in our main code, month starts at 0 on the left and 4 on the right.

Now, I will see that i increases by one, and so does j. Therefore, I will do the same thing for the second line of code.

```
month[1] = *(my_pointer + 5); i = i+1; j = j+1;
```

Now lets put both lines together, and take out the "reminders" at the end:

```
month[0] = *(my_pointer + 4);
month[1] = *(my_pointer + 5);
```

As you can see, it expands to exactly what we started with.

Finally, we can do the same thing for day. Recall that day looks like this:

```
day[0] = *(my_pointer + 6);
day[1] = *(my_pointer + 7);
```

Again, we need two variables. One to go from 0 to 1, and one to go from 6 to 7. Here is how we do this:

```
i = 0;
j = 6;

while (i < 2 && j < 8) {
    day[i] = *(my_pointer + j);

    i++;
    j++;
}
```

You should be able to see on your own that it expands to:

```
day[0] = *(my_pointer + 6);
day[1] = *(my_pointer + 7);
```

Let's look at the entire thing now, entirely finished. We will start from here in the next lesson.

```
// First Year
int i = 0;
while (i < 4) {
    year[i] = *(my_pointer + i);

    i++;
}

// Now Month
i = 0;
```

```
int j = 4;

while (i < 2 && j < 6) {
    month[i] = *(my_pointer + j);

    i++;
    j++;
}

// Now Day
i = 0;
j = 6;

while (i < 2 && j < 8) {
    day[i] = *(my_pointer + j);

    i++;
    j++;
}
```

Lesson 13.2 : The Basics of Algorithm Design Part Three

The video for this lesson can be found here: http://youtu.be/5jyEiJ3o-ko

Supplemental video on printing binary data: http://youtu.be/XbL69Z_-keY

In our last lesson we ended up with a series of while loops which all had three things in common:

1. They all had some initial state, we set a variable or variables to a value to start with.
2. They all had a conditional statement to know when the loop was done.
3. They all incremented the variables at the end.

It turns out that this three step process is used so much that programming languages have created a sort of "short hand" while loop called a for loop.

Here I will show you the while loop for year in our previous lesson, and then I will show you how to write this same code using a for loop.

While loop:

```

// Figure (a)

while (i < 4) {
    year[i] = *(my_pointer + i);

    i++;
}
```

For loop:

```
for (i = 0; i < 4; i++) {
    year[i] = *(my_pointer + i);
}
```

We have combined the starting conditions, the conditional statement, and the final incrementing statements into a single expression where semi-colons separate the three. Lets look at this in parts:

for (**i = 0;** i < 10; i++) {

This is our starting condition. We are setting the variable i to 0. This is the same thing as the code above our while loop. This part in bold executes before the loop begins. This is very important to remember. It is not part of the loop, it only creates a starting condition which the loop needs in order to work properly.

for (i = 0; **i < 10;** i++) {

This is our conditional statement. This is exactly the same as what goes in the parentheses of the while loop.

for (i = 0; i < 10; **i++**) {

This is the final statement that will execute at the end of the loop. It is identical to putting the incrementing statement at the end of our while loop.

Now, lets put this together to transform all of our loops in the previous example to a while loop just as we did in Figure (a).

```
for (i = 0, j = 4; i < 2 && j < 6; i++, j++) {
    month[i] = *(my_pointer + j);
}

for (i = 0, j = 6; i < 2 && j < 8; i++, j++) {
    day[i] = *(my_pointer + j);
}
```

Notice that we used commas to separate statements inside our loop expressions, NOT semicolons.

For example, we wrote: i = 0, j = 6 and we did not write: i = 0; j = 6.

Now here is our final code again, but this time using for loops instead of while loops.

Notice that we initialized our variables before any of the loops began. This is good practice as we are defining ahead of time which variables we intend to use for our loops. This also lets a programmer reading the source code understand that none of these loops will require more than two variables.

```
// First we create and initialize the variables we will
// need for the loop.
```

```
int i = 0;
int j = 0;

// First Year
for (i = 0; i < 4; i++) {
    year[i] = *(my_pointer + i);
}

// Now Month
for (i = 0, j = 4; i < 2 && j < 6; i++, j++) {
    month[i] = *(my_pointer + j);
}

// Now Day
for (i = 0, j = 6; i < 2 && j < 8; i++, j++) {
    day[i] = *(my_pointer + j);
}
```

Lesson 13.3 : The Basics of Algorithm Design Part Four

The video for this lesson can be found here: http://youtu.be/IXBZ9zHkBv4

In the last several lessons I have shown you various ways to do the same task, which is to take a string of text formatted as: YYYYMMDD and convert it into three separate strings: one for year, month, and day.

These lessons are designed not only to show you how to create algorithms, but also how to read them. For that reason, this lesson is going to work backwards, starting at the final algorithm (similar to what you might actually encounter) and then showing you how to decipher it.

Please take your time through this and all lessons. I am proceeding through this subject slowly in order to avoid confusing anyone. This is complex material, and should be read carefully. Do not be afraid to ask questions. If you have a question, odds are someone else has the same question and can benefit from you asking it. Since this is not a book but an interactive course, you should take advantage of that to learn as much as possible.

Here is the algorithm we are going to work with. This is exactly the same as what we did in Lesson 59, 60, and 61.

```
// Figure (a) : Algorithm to convert YYYYMMDD to 3 strings.

for (i = 0; i < 4; i++) {
    if (i < 2) {
        day[i]     = date[i+6];
        month[i] = date[i+4];
    }

    year[i] = date[i];
}
```

This is the final algorithm. This is the type of thing you are likely to see when you read source code. It may look scary, but do not worry. After this lesson you will be able to read algorithms such as these (and write them) quite easily. That is, so long as you take this slowly and ask questions when you need to.

Notice I did away with the pointer indexing. In reality I didn't, it is just that arrays do this behind the scenes for you. I am using array indexing which is cleaner and easier to understand.

First let me explain why I took such an easy to read example in Lesson 59 and transformed it into what you see here. Remember that the code in Lesson 59 is not really the same as the code here even though it does the same task. They both achieve the same goal, but the code in FIgure (a) does so much cleaner, faster, and more efficiently.

In general you will find that algorithms made to be fast and efficient will also tend to be difficult to understand at a glance. This is because we are writing instructions for a machine, not a person.

Now let's begin the lesson.

First of all, never be intimidated by code. Think of it as a box that contains something, and that you have to peel away the layers in order to understand it. Not even the most seasoned programmers will be able to look at a complex algorithm at a glance and understand how it works.

You will find that no matter how intimidating it appears at first, it is actually easy to understand if you take it in parts, and slowly convert it backwards. Take your time. Patience is a major part of being a programmer.

If you just stare at any lesson in this course and try to see it all at once, it will appear to be nothing more than a jumble of numbers, letters, and cryptic terms. When you see things like that, it can be intimidating.

Now, lets take this algorithm apart:

Notice there is a for loop which says that the contents inside it are going to execute.. how many times? four. So the first step to expanding this algorithm will be to take the contents of the for loop and examine each iteration, each time keeping in mind that the variable i will increase by one.

Whenever you talk about a loop, you describe each time the loop runs as an 'iteration'. Whenever you want to decipher an algorithm, you must start by understanding the first iteration.

Here is the first iteration of our loop, with i=0 as our starting state.

```
i = 0;

if (i < 2) { // <-- is i less than 2?
            // Yes, it is the first iteration.

    day[0]    = date[0+6];
    month[0] = date[0+4];
}

year[0] = date[0];
```

Notice all I really did here was to remove the for loop start and end, so I can just observe the contents inside it.

Do we really need an if statement here? No. We just need to remember that the contents of the if statement execute when i is less than two. In other words, the first two times. Since this is the first iteration, will the contents of the if statement execute? Of course.

So let's re-write our expansion of the first iteration:

```
day[i]    = date[i+6];
month[i] = date[i+4];
year[i]   = date[i];
```

I took out the code for the if statement start and end. Why? Because i is less than 2. Therefore, whatever was inside the if statement will execute.

Now, lets fill in values for i.

```
day[0]    = date[0+6];
month[0] = date[0+4];
year[0]   = date[0];
```

Now, 0+6 is just six, so let's fix that:

```
day[0]     = date[6];
month[0] = date[4];
year[0]   = date[0];
```

Now we have readable code. This is the first iteration of our loop. Now we just have to figure out what this code means.

Observe date in memory:

```
Y Y Y Y M M D D
0 1 2 3 4 5 6 7
```

I labeled each character so this will make more sense.

1. Set the first character of the string day to be character #6 of date (Remember, we start at 0)
2. Set the first character of the string month to be character #4 of date
3. Set the first character of year to be the first character of date.

So what we have are really three simultaneous processes going on. The first digit of YEAR is set. The first digit of MONTH is set. The first digit of DAY is set. We know the first digit of YEAR is the first digit of the string in general. We know the first digit of MONTH is digit #4 (when starting at 0). We know the first digit of DAY is digit #6 (when starting at 0).

Now, look again at our original loop:

```
// Figure (a)

for (i = 0; i < 4; i++) {
   if (i < 2) {
      day[i]     = date[i+6];
      month[i] = date[i+4];
   }

   year[i] = date[i];
}
```

Now lets look again at just the first iteration:

```
// Figure (b) : First iteration

day[0]     =date[6];
month[0] = date[4];
year[0]    = date[0];
```

Make sure you can understand how the first iteration transforms from the loop to the simplified version in Figure (b). We will explore more of this in upcoming lessons.
Why do you need to know how to understand an algorithm? Because you will encounter them, and you will need to write them. Every application and game uses them in one form or another, and they are not difficult if you take the material slowly.

Lesson 13.4 : The Basics of Algorithm Design Part Four

The video for this lesson can be found here: http://youtu.be/FLn6pCk6OIs

In this concluding lesson of our four-part series on Algorithm Design, we are going to take the for loop in the previous example and break it down into the actual four iterations that will take place.
First, here is the original algorithm:

```
// Figure (a)

for (i = 0; i < 4; i++) {
   if (i < 2) {
      day[i]     = date[i+6];
      month[i] = date[i+4];
   }
```

```
    year[i] = date[i];
}
```

Now, here is the first iteration of that algorithm, which we learned from the previous lesson:

```
// Figure (b) : First iteration

day[0]     = date[6];
month[0] = date[4];
year[0]    = date[0];
```

And here we are. Looking at the for loop itself, and having written out the first iteration, now we have to determine the second iteration. This is not that difficult.

The first question to consider when advancing from one iteration to the next is this: What variables have changed, and how? In this case, the answer is easy. There is only one variable i, and it has increased by one.

The second question to consider is this: Are there any lines of code from the first iteration that will not execute this time? Also, are there any lines of code that did not execute in the first iteration that will execute this time?

In this case, the answer is no. The only conditional statement inside our loop asks if the variable i is less than two. Since i is still less than two (the variable i will be set to 1 after the first iteration), then we know the same general lines of code will execute as the first iteration.

Therefore, here is the second iteration:

```
Figure (c) : Second iteration

day[1]     = date[7];
month[1] = date[5];
year[1]    = date[1];
```

Do you see that all we really did was to increase the numbers by one? Recall how the string date looks in memory:

```
Y Y Y Y M M D D
0 1 2 3 4 5 6 7
```

So the second iteration reads like this:

After setting the first character for year, month, day (the first iteration), now we:

1. Set character #1 of day to character #7 of date.
2. Set character #1 of month to character #5 of date.

3. Set character #1 of year to character #1 of date.

Remember that when you are talking about an array index or a pointer offset, that 0 really means the first one. You always start counting with zero. If you want the third item, that is item #2. The first item is item #0.

So now the second iteration should make sense. You should understand at this point that two iterations is all it takes to set both day and month. After this second iteration, we have actually already processed the entire string YYYYMMDD with only TWO characters not processed. They are the last two characters of YYYY.

Now, we need simply two more iterations to process those characters. To do the third iteration we must ask our questions: How is the variable changing? What lines of code will be present on this iteration?

The answers are: The variable i increases by one. In this case, the code inside the conditional statement will not execute, so we can ignore it. Here is the third iteration:

```
year[2]  = date[2];
```

Now, you should easily see the fourth iteration.

```
year[3]  = date[3];
```

Now we are done. We have officially processed all of the string YYYYMMDD. Now lets put it all together:

The fully expanded for loop:

```
// First iteration (Processes 3 characters of YYYYMMDD)

day[0]    = date[6];
month[0] = date[4];
year[0]    = date[0];


// Second iteration (Processes 3 more characters)

day[1]    = date[7];
month[1] = date[5];
year[1]    = date[1];


// Final two iterations process remaining two characters

year[2]    = date[2];
year[3]    = date[3];
```

Notice that our for loop expands to only eight actual statements, one for each character of our YYYYMMDD string. We can complete those eight statements with only four iterations of a for loop, and one variable.

Keep in mind that the purpose of these four lessons was to introduce you to algorithms in general. There are many ways to accomplish similar tasks, and I do not want anyone to think that you must use an algorithm like this for simple string functions.

There are a wide variety of pre-built functions which will do this for you easily, and they will be the subject of future lessons.

UNIT 14 : Multi-Dimensional Arrays

Lesson 14.1 : Introducing Multi-Dimensional Arrays

The video for this lesson can be found here: http://youtu.be/XUKsRn60KSA

In this lesson I'm going to introduce you to multi-dimensional arrays. We're going to start by looking at a two dimension array, and then we will look at 3 and higher dimensional arrays in later lessons.

So first of all, what do I mean when I say a "multi-dimensional array" ?

Let's go ahead and look at the kind of array that we have already seen.

```
#include <stdio.h>

int main(void) {
   char string[] = "hello";

   return 0;
}
```

So this is a one dimension array. That is because it is an array of individual data elements, in this case characters. We can think of a one dimensional array, for the sake of this lesson, as being a "word", or a simple string of text.

A two dimensional array is simply an array of one dimensional arrays. Now if we think of a one dimensional array as being a word, then we can think of a two dimensional array as simply being an array of words.

How would that look in memory?

let's first of all create three different words.

```
char word1 = "One";
char word2 = "Two";
char word3 = "Three";
```

Here we have simply created three different variables called word1, word2, and word3. This is not a two dimensional array. It is just simply three individual one dimensional arrays, or three individual strings of text.

If we were to imagine that these three strings of text were stored in memory one after the other it would look like this.

```
['O']['n']['e']['\0']['T']['w']['o']['\0']['T']['h']['r']['e']['e']['\0']
```

word1 would be the word "One" followed by the null termination character. word2 would be the word "Two" followed by the null termination character, and word3 would be the word "Three" followed by the null termination character.

For the sake of this lesson, in order to explain to you how two dimensional arrays work, I need to be able to store these strings of text one after the other in memory. How can I do that?

The answer is that I can just simply create a new array of characters and I will call it an_array_of_words and I will set it equal to: "One\0Two\0Three".

```
char an_array_of_words[] = "One\0Two\0Three";
```

Keep in mind that any time you have a string of text enclosed in double quotes, there is always going to be a null termination character at the end. By creating that line of code, I am storing each string of text one immediately after the other in memory. In effect, I am creating a kind of two dimensional array.

What happens if I try to print this string of text: an_array_of_words by sending it to the printf function?

Output:
One

It is going to print just the word "One". Why? Because when it starts printing it is going to look at the "O", then the "n", and then the "e", and then it is going to reach the null termination character which means to stop printing, and it is not going to go beyond that.

So it is very easy to print the first word of my one dimensional array. Now, we have worked with pointers before, let's go ahead and create a pointer and set that pointer to the memory address of the first element of an_array_of_words:

```
char an_array_of_words[] = "One\0Two\0Three";
char *my_pointer = an_array_of_words;

printf("%s \n", an_array_of_words);
```

So in this case we are pointing to the capital "O" in the word "One". So if we try to send the pointer to the printf() function and we run the program, we get exactly the same result. We are still printing the first word.

Output:
One

How can we print the second word? The answer is that we just simply have to move the pointer. Right now the pointer is pointing to offset 0, which is the start of the array as well as the start of the word "One". If we want to print the word "Two", then we need to move the pointer by adding to the offset.

Offset 0 is the "O" in "One". Offset 1 is the "n" in "One". Offset 2 is the "e" in One. Offset 3 is the null termination character which ends the first word. Therefore, offset 4 is the capital "T" which is the first character of the word "Two".

So if we tell the printf() function to use my_pointer offset 4, we are adding 4 to the memory address, then it is going to start printing with the "T" in "Two", and it will stop when it reaches the next null termination character.

```
printf("%s \n", (my_pointer + 4) );
```

If we run the program, you will see that it prints the word "Two".

Output:
Two

What I want you to understand is that whenever you have a two dimensional array, what you really have is a one dimensional array where each element (in this case each word) is simply stored one after the other in memory.

The way that you can get to each individual array element is by using an offset. If we wanted to print the final word, "Three", then we would change our offset. Instead of using offset 4, we would use offset 8.

If we use offset 8 and we run the program, we will print the final word "Three".

```
printf("%s \n", (my_pointer + 8) );
```

Output:
Three

Now, what if we wanted to print a specific character? Let's say we want to print the "r" in "Three".

First of all, we need to change our printf() function to use %c instead of %s. This means instead of the printf() function expecting a memory address to a string of text, it will expect the contents of a memory address, an actual character.

By simply placing an asterisk in front of the pointer and offset, we are no longer saying the memory address but we are specifying the actual character stored at that memory address.

```
printf("%c \n", *(my_pointer + 8) );
```

Output:
T

If we run the program with offset 8, then we will print "T". That is simply because the character located at the memory address "my_pointer + 8" is the "T" in "Three".

Now if we want to print the letter "r", we simply need to add to that offset. There are two ways we can do that. First of all, we could say "my_pointer + 10" which would work just fine, and that would get us to the specific character that we want to print.

```
printf("%c \n", *(my_pointer + 10) );
```

Or, you can use two offsets. You can use one offset (the +8) in order to get to the first element of the word "Three", and then you can use another offset which is how many additional characters after that. So you can say 8 + 2 and get the exact same result, as shown here:

```
printf("%c \n", *(my_pointer + 8 + 2) );
```

Now the reason I would do it that way is because it is easier to understand intuitively that I am going to a specific word, and then I am saying that I want a specific character within that word. It is a two part process. First of all, I am moving to the word and then I am adding to that in order to reach the specific character.

Now what if I wanted to print the "o" in "Two" ? Well, remember that the word "Two" begins at offset 4. The "o" would be the element index 2 in the word "Two". The "T" is at index 0, the "w" at index 1, and the "o" at index 2.

So this:

```
printf("%c \n", *(my_pointer + 4 + 2) );
```

will print the "o" in the word "Two".

Now hopefully here you see why we start array elements at index zero. If we were looking at offset 0, we would say "4 + 0" and that is the first character. That way when you are performing mathematical operations in order to move offsets around, you can always be looking at the first character by not adding anything to the offset.

Now let's take a look at another example.

Suppose we have the exact same string of text, and I want you to use the printf() function to print a specific character. Now, if I were to write this:

```
char an_array_of_words[] = "One\0Two\0Three";
printf("%c \n", an_array_of_words[0]);
```

then I am sending specifically the character "O" from the word "One" to the printf() function. If I run this, I would simply print the "O" from the word "One".

I want you to see that writing this:

```
an_array_of_words[0]
```

is the same thing as writing this:

```
*(an_array_of_words + 0)
```

In both cases, we are getting the exact same character. We are just using a different syntax in order to do so. In the first example, we are using the array indexing syntax, where we are saying "get array element index 0 of the array called an_array_of_words, which would be the first element of that array which is the "O" in "One".

In the second example on the other hand, we are saying "Locate the memory address of that array, and get what is at that memory address." Because we have an offset of 0, we are just saying to get what is at that memory address, which is the "O" in "One".

So if we run the printf() function with either of these two examples, we will get the exact same result. We are saying the exact same thing, we are just using a different syntax.

Now if we are looking for a specific element of the array, for example the "w" in the word "Two", then of course like I showed you, we can simply add to that offset. If we add 0 we will get the "T" and if we add 1 we get the "w". Therefore, you can use this syntax in order to get the "w" in two.

```
printf("%c \n", *(an_array_of_words + 4 + 1) );

// or we can write

printf("%c \n", an_array_of_words[4 + 1] );
```

This is the same thing as if we had written + 5

It is the same thing. If we run this program you will see that it prints the "w". What I want you to notice is the very clear relationship between pointers and arrays. You can write the exact same code using the array index syntax using the brackets as you can using a pointer and and using the offset.

They are just two different ways of writing the exact same code. So any time you see something like this:

```
array[5]
```

You should be able to now think of it as being this:

```
*(array + 5)
```

(Editor's note: This is assuming we are talking about character arrays similar to the examples in this lesson. In later lessons, this will be expanded on for other kinds of arrays.)

In both cases, you are reaching the same element in memory. Now in the next lesson we will look at this in greater detail, but I want to make sure you understand from this lesson are the following points:

1. A two dimensional array is simply an array of one dimensional arrays.
2. You can use either the array bracket syntax or the pointer offset syntax in order to reach the same element in memory.
3. Lastly, to reach a specific element, you can use multiple offsets. Your first offset will reach a specific starting element, and then an additional offset or offsets added to that allow you to reach the specific element that you want.

Lesson 14.2 : Introducing Multi-Dimensional Arrays (part two)

The video for this lesson can be found here: http://youtu.be/XpyfG-sgJzk

Take your time through this lesson. Make sure you are here only if you have already mastered all previous lessons.

In the last lesson I gave you a general overview of multi-dimensional arrays and how they are implemented in memory. Each programming language implements arrays somewhat differently, and the topic of this lesson is how to actually implement a two dimensional array in C.

Before we begin, let me show you what a multidimensional array will look like:

To create a 2d array of these six words (meaning, a 2d array with 6 elements, each one a word with up to six characters), I have to specify this. Here is how I do so :

```
char first_array[6][6];
```

We will go over the details of how this works during this lesson, and we will come back to this at the end.

In the last lesson I showed you that you can start at the "first letter" of a given word, and that by knowing that first letter you can add some offset in order to reach any letter in that word.

Let me illustrate this again:

"RedditProgrammingClasses"

Here are three unique strings in memory. Each of them starts at a different memory address. The first word starts at position 0. The second word starts at position 6, and so on.

There is a problem with this approach. It works as long as you know exactly where in memory each word starts. However, this is not uniform. We might get a result similar to this:

1. The first word starts at position 0
2. The second word starts at position 28
3. The third word starts at position 40
4. The fourth word starts at position 44

There is nothing uniform about this. Because of that, we are forced to uniquely keep track of the starting offset number for every element in our "array". A true array requires that all offsets are uniform. Let's see what this means.

```
"OneTwoSix"
```

Here we have three strings of text that are of uniform length. How many numbers do we have to keep track of to know the starting location of each one? Only one number. Watch:

1. The first word starts at position 0.
2. The second word starts at position 3.
3. The third word starts at position 6.

Where would the fourth word start at if we kept our words of uniform length? It would start at position 9. And so on.

In other words, rather than keep track of the unique starting location for each word, we can simply know that we multiply the number 3 by which word it is (first, second, etc) and we get the correct offset.

1. The first word is at position 0 which is: 3*0
2. The second word is at position 1 which is: 3*1
3. The third word is at position 2 which is: 3*2
4. The Nth word is at position N which is: 3*N

So you can see that mathematically it is very convenient to keep each word in our array of uniform length. We call such "words" in general, elements. So to rephrase, it is important that all array elements have uniform length.

This way we can find the start of any member element by simply multiplying the length of each element times the "element number". For example, if each element is 10 bytes in size, and we want the third element, it would start on byte #20. The first element would start at byte #0 and span until byte #9. The second element would start at byte #10 and span until byte #19. The third element would span from byte #20 until byte #29.

Now, what happens when words are NOT of uniform length? Well, we have to make them uniform in order to make this system work. This requires that we "waste" some space. Here is how it works.

```
"RedditProgrammingClasses" will become:
```

```
Redditxxxxx
Programming
Classesxxxx
```

Which in memory will really be: RedditxxxxxProgrammingClassesxxxx

I added x's at the end of each word so that now it is proper length. Notice that this has the effect of lining up the words in a grid. This means that we can now quickly reach the start of any word in the array, and that each future word will be reached just as easily.

So here you can see the first issue concerning arrays in general. Every member element of the array must be the same length.

Now, it is very important that you understand how to calculate the total size of an array. It is not hard, and you only need to know two things:

1. The size of any element of the array (it should be uniform).
2. The total number of elements.

By just multiplying these together, you get the total size of an array. Let's look at this in action with various array sizes.

First, a simple string of text - one dimensional array:

```
char string[] = "Hello";
```

Total number of elements? 6 (including the termination character). Total size of each element? 1. Total size of array = 1*6 = 6. Easy.

Now let's create a two dimensional array filled with "Hello" (or equally sized words). Let's suppose it has 3 elements.

How many elements? 3. The total size of each element? 6 (We are using "Hello" sized words). Total size of our 2d array? 3*6 = 18. In other words, we have 18 characters in which to contain exactly three words which each contain exactly 6 characters.

What about a 3d array with 2 elements? Well, what is the size of each element? 18. Why? Because that is the size of a 2d array. A 3d array will be an array of 2d arrays. So 18 * 2 = 36.

Therefore, a 3d array of this type will have exactly 36 characters to contain exactly two 2d arrays which each contain exactly 3 words that each contain 6 characters.

As I just explained, any Nth dimensional array will be an array of [N-1] elements. A 5th dimensional array is an array of 4d elements. And so on.

Now, what if this was a 3 dimensional array? Here is how this would work:

First, imagine our string of text of three words (our 2d array) as ONE string of text. Remember, inside of your memory even a 6th dimensional text array is only ONE string of text.

```
RedditxxxxxProgrammingClassesxxxx
```

That is a 2 dimensional array. A three dimensional array would be an array of these. For example:

```
RedditxxxxxProgrammingClassesxxxx
RedditxxxxxProgrammingClassesxxxx
RedditxxxxxProgrammingClassesxxxx
```

I left the words the same so it is easier to understand. Here we have three elements of a three dimensional array. Why is it a 3 dimensional array? Because each element is a two dimensional array.

Now you should be able to see that if we lined them in memory as a single string: we will have one offset to reach the next 3-dimensional array member of our array, a different offset to reach the next 2-dimensional array member, and a different offset to reach the exact character we want.

For example, if we want to reach the letter "a" in "Classes" in the 3rd "row" of our three dimensional array, we would do this:

```
string[2][2][2]
```

Remember. We start counting array indexes at 0. 0 means first.

The first [2] means "third element" of our 3d array. In other words, the third row. The second [2] means "Third word" which is "Classes". The third [2] means "Third letter" which is "a". But all of this translates to:

```
string[ ( 2*size_of_3d_element ) + ( 2*size_of_2d_element )  + 2]
```

How big is a 3d element? How many characters are in the sample 3 rows I gave above? Well, each word has 11 characters (including x's). 11*3 = 33. That is the size of each 3d element in our array.

In other words, each element of our 3d array has exactly 33 characters in which to contain exactly 3 words which each contain 11 characters. Remember, each word must be of uniform length.

So you should realize something by now. To create any array you must know the size of each element of that array. If I want to store six words in an array:

```
One
Two
Three
Four
Five
Six
```

I must know the maximum size I will need. This will mean the size of the biggest element, in this case, "Three" which is 6 characters (including a termination character).

Now, to recap: To create a 2d array of these six words (meaning, a 2d array with 6 elements, each one a word with up to six characters), I have to specify this. Here is how I do so :

```
char first_array[6][6];
```

This creates a 6x6 array. Six elements. Each element six characters (max). What is the total size of the array? 36.

In the next lesson we will explore this further, including how to assign values to array elements and more.

Lesson 14.3 : Introducing Multi-Dimensional Arrays (part three)

The video for this lesson can be found here: http://youtu.be/jcygT7Rzp6o

In the last lesson I explained the basic structure of arrays and how they are implemented in memory. In this lesson I am going to show you how to actually create and initialize them.

Lets suppose that we want an array that will contain ten words. Each word will be a maximum of 9 characters (including the string termination character, called NUL).

Here is how we would do this:

```
char first_2d_array[10][9];
```

Now, this allocates 90 bytes of storage space for whatever we want to put in those 90 bytes. We are saying that we intend to store ten unique elements, each element containing up to 9 characters.

Now, how can we assign values to these? You might think you can do this:

```
first_2d_array[0] = "One";
```

But you cannot. C understands "One" to mean a string of text, a constant, that resides somewhere in memory. This is not what you want. You want to actually store the individual characters of "One" into first_2d_array[0] one letter at a time.

There are effectively two ways to do this. The hard way, and the easy way.

First, the hard way. You can individually store each character one at a time like this:

```
first_2d_array[0][0] = 'O';
first_2d_array[0][1] = 'n';
first_2d_array[0][2] = 'e';
first_2d_array[0][3] = ' ';
```

Thankfully, the developers of C understood how tiresome a process that would be. Therefore, they created a whole range of string functions which make doing things like this quite simple.

There is a function called strcpy() which is built into C, just like printf() is built into C. You use strcpy to copy strings. str is short for string. cpy is short for copy.

The syntax for strcpy() is fairly simple. It takes two parameters. The first parameter is where you are putting the string. The second parameter is the string itself.

So, instead of that tedious process to copy the characters "One" one at a time, I can simply write this:

strcpy(first_2d_array[0], "One");

And that is all I have to do. Can you do it without using the built in strcpy function? Sure. But this is much easier. If you really wanted to, you could do this yourself with a simple algorithm:

```
char *tempstring = "One";
int i = 0;

for (i = 0; i < 4; i++) {
    first_2d_array[0][i] = *(tempstring + i);
}
```

Just a quick review. Keep in mind we are creating a char pointer to a string constant "One". We are not storing the string "One" inside a pointer. Also, keep in mind that our small algorithm is taking into account the NUL termination character. This is because it starts at 0, and ends at 3. That is four total characters. O, n, e, and \0.

So it is not the case that you must use strcpy() to copy a string into an array. However, this is there for your convenience, so it makes sense to use it.

The first parameter is where you want to put the string. The second parameter is the string itself.

Now, lets use strcpy() to initialize each array element of our 2d array. Recall that a 2d array will be an array of 1d arrays. In this case, 1d arrays are simply text strings.

Because part of this course is about showing you the thought processes that go into programming in general, I think it may serve helpful to show you the actual process I would go about writing this out - even for this lesson.

First, I would just copy-paste the strcpy() lines that I need. I need ten of them since I am going to be setting this up for ten different strings.

```
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[0], "One");
```

Now, since that copy-paste operation is fairly fast (I do it without even thinking about it), the next step is to just go through and change the elements.

```
strcpy(first_2d_array[0], "One");
strcpy(first_2d_array[1], "Two");
strcpy(first_2d_array[2], "Three");
strcpy(first_2d_array[3], "Four");
strcpy(first_2d_array[4], "Five");
strcpy(first_2d_array[5], "Six");
strcpy(first_2d_array[6], "Seven");
strcpy(first_2d_array[7], "Eight");
strcpy(first_2d_array[8], "Nine");
strcpy(first_2d_array[9], "Ten");
```

I would actually be able to do this even faster using my editor of choice.

Now remember that each of these strcpy() operations are going to be taking into account the NUL termination character. Why? Because we are giving it double quoted strings as a 2nd parameter. A double quoted string has a NUL termination character automatically at the end.

So now, how can we display that these strings were properly created? Well, we could use ten different printf() statements, but why not just have a for loop execute ten times?

```
int i=0;
for (; i < 10; i++) {
    printf("String #%d is %s  ", i, first_2d_array[i]);
}
```

Here is the final program so you can experiment with it:

```
#include <stdio.h>
#include <string.h>

int main(void) {

    char first_2d_array[10][9];

    strcpy(first_2d_array[0], "One");
    strcpy(first_2d_array[1], "Two");
    strcpy(first_2d_array[2], "Three");
    strcpy(first_2d_array[3], "Four");
    strcpy(first_2d_array[4], "Five");
    strcpy(first_2d_array[5], "Six");
    strcpy(first_2d_array[6], "Seven");
    strcpy(first_2d_array[7], "Eight");
    strcpy(first_2d_array[8], "Nine");
    strcpy(first_2d_array[9], "Ten");

    int i=0;
    for (; i < 10; i++) {
       printf("String # %d is %s  ", i, first_2d_array[i]);
    }

    return 0;
}
```

Notice with the for loop I did not put anything in the initial state. I just put a single semicolon. This is because I already established the initial state above the for loop.

One more note. Just as we have to include stdio.h for printf() and related functions, we need to include string.h for string functions like strcpy().

Lesson 14.4 : More on Multi-Dimensional Arrays

NOTE: There is no transcript for this lesson. The video for this lesson can be found here: http://youtu.be/oSOfcFvj5vw

Lesson 14.5 : More on Multi-Dimensional Arrays and Introducing strcpy()

NOTE: There is no transcript for this lesson. The video for this lesson can be found here: http://youtu.be/MVkLoOoG5fA

UNIT 15 : Review of Pointers

Lesson 15.1 : Review of Pointers (part one)

The video for this lesson can be found here: http://youtu.be/UOQJQaNI4aQ

Upcoming lessons are going to rely heavily on you having mastered pointers. Before we begin these advanced lessons, I think it is imperative that we spend some time reviewing pointers and especially covering some of the finer details we have not yet had a chance to cover.

Let's begin.

A pointer is a variable that can hold a memory address. More specifically, a pointer is something that can only hold a memory address. If a pointer has any value at all, that value is a memory address. The only thing you can assign to a pointer, is a memory address.

Now, how do you create a pointer? You specify the data type of the pointer you want, followed by an asterisk * character, followed by the name of the variable. Why do you need to specify a data type? Because besides holding a memory address, a pointer needs to know how big something is that it will be looking at, as well as what format the data will be in. This is because pointers are used to "look at" the contents of memory at a given memory address.

Here is how to create a pointer:

```
char *my_pointer;
```

You can optionally assign the pointer a value.. a what? A memory address.

To assign a pointer a value at the same time as you create it, you can use the equal sign. Now, the equal sign is a bit misleading sometimes. Consider this example:

```
char *string = "Hello Reddit";
```

It sounds like I am saying: Make string a pointer equal to the string "Hello Reddit". This is false. Where is the confusion in this situation? It is in our understanding of an equal sign. Now, let's change that understanding in a way that will greatly help you to understand this course.

A single equal sign is an assignment operator. That's all. When you see an equal sign, you are seeing an "assignment operation". In other words, some thing is being set to some value. With this in mind, let's re-think this statement:

```
char *string = "Hello Reddit";
```

Instead of seeing this as: Set *string equal to the string "Hello Reddit", see it like this instead: Perform an assignment operation involving *string (a pointer) and "Hello Reddit" (a string).

string is a pointer. It requires a memory address. It cannot be set "equal" to "Hello Reddit" because that is not a memory address, it is a string. However, while "Hello Reddit" is not a memory address, it has a memory address.

Therefore, we are saying that we are performing some assignment operation involving something that requires a memory address, and something that has a memory address.

```
char *string = "Hello Reddit";
char *string (assignment operation involving) "Hello Reddit";
```

So if you think of the equal sign from now on as an operation that is intended to assign a value to something on the left, based on something on the right, a lot of syntax will start to make more sense - especially pointers.

One note of caution: The above method works great if you are looking at valid C syntax involving pointers that you cannot understand, and trying to make sense of it. It does not work in reverse. You cannot set a pointer equal to just any thing (a variable for example) and expect that C will simply understand you mean "the address of the thing on the right". That is why you have the & "address of" operator.

We will cover this more in the next lessons.

Lesson 15.2 : Review of Pointers (part two)

The video for this lesson can be found here: http://youtu.be/mtnJX1Q8tLI

I am taking this review process slowly, to make sure that each concept is entirely mastered before we proceed. The next topic I want to review about pointers involves the differences between using the * character with a pointer.

The * character means exactly two things concerning a pointer. It means either "Make this variable a pointer", or it means: "What is at the address of the pointer". There is no other possibility.

When you first create a pointer, you use the * character and it means "I am making a pointer." That is all it means.

After a pointer is created, even if it has not yet been assigned, the * character takes on new meaning. It now means for the rest of your program: "what is at the address of".

```
char *my_pointer = ...  // <--- Here and *only* here, the
                         // * character means "I am creating
                        // a pointer.
...
... // <--- For the rest of the program, the
... // * character when used with this pointer will
... // mean "what is at the address contained in the pointer"
```

So that covers the * character. At this stage it should be very clear to you that any time you ever see the * character used with a pointer, other than its creation, it means "what is at the address of" the memory address in the pointer.

Now, there is a term for this process. It is called dereferencing. This is the act of "seeing" what is at the memory address contained in a pointer. For example:

```
char my_character = 'a';
char *my_pointer = &my_character;
printf("The character is %c ", *my_pointer);
```

In the third line, we are seeing the * character being used, and it is not part of the creation of the pointer, therefore the * character means we are asking for "what is at the memory address" of the pointer. Which is of course, the character 'a'.

Now, lastly lets review the & character in the context of pointers. It simply means "address of" and it will give you the memory address that anything resides at. You can use the & "address of" operator with variables, pointers, arrays, array elements, and more.

It might help to think of the & operator as a function that returns a memory address.

Lesson 15.3 : Review of Pointers (part three)

The video for this lesson can be found here: http://youtu.be/lSvxsR7Ihdg

I had originally planned a different lesson here, but it is clear to me that at least a few details regarding pointers need to be addressed. The upcoming lessons will rely on pointers heavily, so it is critical that you understand them thoroughly. Please do not be afraid to ask questions if anything is unclear to you.

The topic of this lesson is, "What can you assign to a pointer, and how?". Some things in C act like memory addresses, and some do not. It is important to understand this. There are only three things we have to consider in this lesson, so this is not difficult:

1. Single variables of a given data type. (ex: int, char, etc)
2. Double quoted strings. (ex: "Hello")
3. Single quoted characters. (ex: 'a')

First, variables. All variables. If it is a variable, then it is not treated as a memory address by C. Here are some examples:

```
int i = 5;
char my_char = 'a';
unsigned short int height = 10;
```

All of these single variables of a given data type will never be seen by C to be a memory address. Why? Because they are all small enough that they can be processed "as is". What I mean by this is, you can never directly assign one of these to a pointer. The following is very wrong.

Bad:

```
char some_character = 'z';
char *my_pointer = some_character;  // <- Bad.
                                     // C does *not* see
                                    // "some_character"
                                    // as a memory address.
```

Good:

```
char some_character = 'z';
char *my_pointer = &some_character;  // <- Good. The &
                                      // operator gets
                                     // the memory address.
```

The only way to get the memory address for variables such as these, is to use the & "address of" operator.

Next: Double quoted strings of text.

Any time you ever see a double quoted string of text, C understands it by using a memory address. Keep in mind it is unnecessary to use an & character with a double quoted string of text.

&"Hello" <--- invalid syntax in this case, and unnecessary.

Any double quoted string can be thought of as already having the & character in front of it. Why? Because a string of text is too big to fit in any of the standard data types, and therefore it must be understood using a pointer. In other words, it must be understood as being a sequence of bytes that start at a given memory address. To work with a string of text, you must have the memory address where it starts.

Therefore, a double quoted string is treated like a memory address so far as pointer assignment operations are concerned.

Example:

```
char *string = "Hello";
// "Hello" is treated like a memory address.
```

A memory address to what? To where "Hello" is stored in memory.

Now, single quoted characters. (Example: 'a')

A single quoted character is simply an ASCII value. For example, 'a' is a value of: 0110 0001. It is no different to say 'a' or to say 0x61 (the hex value for 0110 0001). It is the same thing. Whenever you use a value such as a number or a character, it is seen by C as simply a numeric value, no different than any other.

The following two lines are perfectly identical:

```
char my_char = 'a';
char my_char = 0x61;
```

0x61 is 'a'. It is just a number.

You cannot assign a pointer the value of a character, and you cannot put a & in front of a character to get its memory address. Why? Because it never needs a memory address, it is simply a number.

```
char *some_pointer = &'b';
// Invalid syntax. 'b' is just a value,
// it has no memory address.
```

Therefore, if you want a pointer to contain the memory address of a character, you must first store that character as a value into a variable of data type char, like so:

```
char my_character = 'b';
char *my_pointer = &my_character;
```

So lets review so far:

1. Single variables of a given data type. If you want to assign their memory address to a pointer, you must use the & character.
2. Double quoted strings. You cannot use a & character and you do not need to, because C understands these as having a memory address. Therefore, you can assign a double quoted string directly to a pointer. This means to set the pointer equal to the memory address where the string begins.
3. Single quoted characters. As far as C is concerned, these are no different than numbers. You would not say the memory address of the number 5 for example, similarly you would not say the memory address of a single character, such as 'a'.

Note that #3 above does not apply to variables of type char. A variable of type char is covered by #1 above. It is just a variable, and the only way you can get the memory address of such a variable is with the & character.

```
char my_char = 'a';
// my_char is a variable stored in memory.
// If you want the memory address, use: &my_char

char my_char = 'a';
// 'a' is a character. It has no memory
// address. It is not stored in memory.
// It is simply a number, the number 0x61 (61 hex)
```

Lesson 15.4 : Review of Pointers (part four)

The video for this lesson can be found here: http://youtu.be/zxPRcZ-xfBo

Now, everything should be clear about pointers concerning everything we have covered so far with one exception: arrays. Here we are going to finish this review by studying arrays as they relate to pointers.

Let's consider this code:

```
char my_string[] = "Hello Reddit";
char *my_pointer;
```

Ok, we now have a pointer called my_pointer. We want it to "point to" our array. What does this mean? It could actually mean several things.

Most likely, it means we want to create a pointer that will contain the memory address where our array begins, and it would be used to read one character of the array at a time.

This is the first example we will look at, as it is the simplest. In this case, from the above code, there are two ways to do this.

First, I can say this:

```
my_pointer = my_string;
```

Why? because my_string is already understood by C to be a memory address. Any array name is seen as a memory address. More specifically, it is seen as the memory address of the exact start of the array.

Another way we can achieve the exact same goal, is like this:

```
my_pointer = &my_string[0];
```

This is saying, "Assign my_pointer the memory address of the first character in the array my_string".

It makes sense right? If we wanted to set my_pointer to be equal to the second character, we would write: my_pointer = &my_string[1]; Consider that my_string[0] is the first element of the array. Therefore, &my_string[0] should be the memory address of the first element of the array. So this much should make sense to you.

These two examples are the same. We can set a pointer to contain the memory address of the first element of an array which means exactly the same thing as setting the pointer to contain the memory address to the start of the array itself. Why? Because the start of the array is the first element.

So both of these lines of code achieve the same goal:

```
my_pointer = my_string;
my_pointer = &my_string[0];
```

Ok, so you may be figuring there is a third possibility here:

```
my_pointer = &my_string;
```

This is the exception to the rule. This does not mean the same thing as the other two. Why? Well, that requires a bit of explaining. That explanation is the topic of the next lesson.

Lesson 15.5 : Review of Pointers (part five)

The video for this lesson can be found here: http://youtu.be/5XcPIyOH-Ko

Recall from earlier lessons that I stated you have to give a pointer a data type so that C knows what you will be pointing to. For example, you have to say char * if you want it to be a pointer to data type char.

Why was that? Because if I told you "Give me what is at memory address 1000" you would not know how many bits to give me. In other words, the data type of a pointer is useful in part because it tells how much we plan to see and work with at once. If we have a pointer of type char, then we expect to see and work with data in increments of one byte at a time.

Recall our code from the last example:

```
char my_string[] = "Hello Reddit";

char *my_pointer = &my_string;
// This is different, and *incorrect*. We will see why now.
```

When you say &my_string, you are saying something rather interesting. You are saying that you want the memory address of the "whole array", not just the memory address of the first element.

How is that possible? The whole array doesn't have a memory address. That is exactly right. However, an integer doesn't really have "a memory address of the whole thing" either, since it contains lets say four bytes.

What happens when you tell C to create a pointer to an integer? Well, it creates a pointer that expects to see a whole integer every time you use it.

In other words, we are effectively telling C "I want to see the entire array with one pointer, not just one character at a time." All this really means is that when we plan to "read" from a memory address, we will not be asking for eight bits - but however large the array itself is.

In our example, "Hello Reddit" (and a NUL character) is exactly 13 bytes. Therefore, by saying &my_string we are telling C "I want a pointer which is pointing to the start of the array my_string" - so far so good. But then you are also saying: "When I use this pointer, I plan to use it to see 13 bytes at once."

This is the key difference. C is a unique and powerful language in large part because of the way you can use slightly different syntaxes to mean different processes and operations. This can also serve to make C more confusing than other languages.

Keep in mind that setting the pointer equal to &my_string is incorrect. Why? Because we did not define our pointer as a pointer to a data type of multiple array elements, but as a pointer to a single character. Therefore, if you try to do this you will almost certainly get a compiler warning saying that you are using a pointer of an "invalid type" in your assignment operation. Now you know why.

Whenever you use just the name of an array, C understands that as the memory address where the array begins. Of course, the memory address where the array begins is also the memory address where its first element is located.

However, whenever you use the name of the array with an & "address of" operator, you are saying "the address of the whole array". What this really means is that you are changing the data type of the pointer itself. Instead of saying "I have a pointer which will point to a single byte of data type char", you are saying instead: "I have a pointer which will point to N bytes, each byte being of data type char".

Now, let's look at other examples using this same syntax so that it will make sense to you:

```
int height = 5;
int *my_pointer = &height;
```

Assume int is 4 bytes. What are we saying here? We are saying that we want a pointer that will point to the single memory address where "height" begins, but that it will expect to see four bytes at a time.

Now this:

```
char my_string[] = "Hello";
char *my_pointer = &my_string;
```

"I want a pointer called my_pointer that will contain the memory address where the array my_string begins in memory. However, I want to use this pointer to see the whole array at once."

Now, to wrap this up, lets put into plain English the three ways to use an array with a pointer that we discussed:

1. my_pointer = my_string Means: Assign the memory address of my_string into my_pointer, and my_pointer will expect to see one element of the array at a time. (In this case, one character at a time)
2. my_pointer = &my_string[0] Means: Assign the memory address of the first element of my_string into my_pointer, and my_pointer will expect to see one element of the array at a time. (In this case, one character at a time) #1 and #2 are the same thing.
3. my_pointer = &my_string Means: Assign the memory address of my_string into my_pointer, and my_pointer will expect to see the entire array each time it is used. This is incorrect usage. Use #1 or #2.

Therefore, do not use &array to get the memory address of an array. Simply using array or &array[0] is sufficient.

At this stage, everything we have covered so far involving pointers should make sense. If anything is unclear, please present your questions. All future lessons will rely on you having mastered this material.

UNIT 16 : Pointer Offsets and Array Indexing

Lesson 16.1 : Using Pointers with Offsets

The video for this lesson can be found here: http://youtu.be/3y3BqAU7CqM

The subject of pointers is confusing to many beginners, and yet it is a critical concept to master. Part of the difficulty with the subject is the difficulty in visualizing pointers in a way that makes them easy to understand. Throughout these next lessons I will be expanding on ways to better visualize pointers, and to better understand their uses and purpose.

This lesson is going to focus on the methods of using an "offset" with a pointer. This is a critical concept to master. Some of this material has already been covered, so this lesson may feel like a review to some extent.

First of all, what is an offset? An offset is a way of identifying a location given a starting point. Nothing more.

If I told you "Start at your house, then go 100 miles north", that is an example of an offset. 100 miles is a distance away from your house, and that defines an exact location of something. Remember that your computer's memory is linear, thus any offset will be simply plus some number, or minus some number.

With pointers, you use an offset to locate an exact thing in memory that you want to see or work with. If I give you this string for example:

```
char *string = "Hello Reddit";
```

How can we find the 'R' in Reddit? Well, we do not know the actual memory address that 'R' is found at, but we do know the actual memory address that the 'H' is found at. Therefore, we have all the information we need in order to find 'R'. If we simply add 6 to the memory address, we get the memory address of 'R'. If we are at 'R', and we want to find 'H', we subtract six.

This is a critical concept to master. Any element of any data structure can be located by knowing the location in memory of any other element of the same data structure, provided you have a way to determine the distance between the two. This goes for arrays, data structures, anything at all.

Now let's expand this statement: All elements of any data structure can be located by an offset (plus or minus) from any other element.

Suppose I didn't know where 'H' was, but I knew where R was in memory. Then it is possible to find 'H'. If I know where the 'o' is, I can find every letter from the 'H' of "Hello" to the 't' of "Reddit".

So now I have explained how offsets are useful, and what they are, but how do you use them? You add or subtract some quantity from the memory address of the source element. The source element is the element that you are going to measure the offset from.

Think of the source element as being "your house" in the example I gave. You will either add or subtract some value to that "source element" and in so doing you will be able to reach any element in the data structure.

What do you get if you add a number to a memory address?

Another memory address. It is as simple as that. A pointer is always going to have a memory address, so if you add to it, or subtract from it, you are still going to get a different memory address as a result. The word "different" is important here. Unless you are adding or subtracting a 0, you will get a different memory address, and therefore different data.

Memory address plus 100 means: "Give me the memory address 100 bytes forward from where I am."
Memory address minus 50 means: "Give me the memory address 50 bytes backward from where I am."

[Note: Above example assumes a char pointer, otherwise it will not be "50 bytes", but the concept still applies.]

You can also set "bookmarks" inside of memory, and come back to it later. You might create a pointer to "mark" where the string "Hello Reddit" begins in a complex data structure, and set a different pointer to "mark" where the string "Hello Reddit" ends.

By doing this you could have two or more pointers working on data simultaneously, by each starting at different locations within the data.

There are many reasons you may want to do that, not the least of which is sorting algorithms. That will be the subject of future lessons.

To wrap this up: A pointer can be added to, or subtracted from. Doing so just results in a new memory address that is a set number of bytes away from where you started. If you add to a pointer, you go forward into memory. If you subtract from a pointer, you go backwards.

Knowing how to properly use offsets is an important skill and will empower you to be able to achieve results that otherwise would not be possible.

Lesson 16.2 : Introducing Array Indexing as Pointer Offsets (Part One)

The video for this lesson can be found here: http://youtu.be/7s9sZ_lYttw

This is the first lesson on a series designed to increase your understanding of arrays and pointers, and also to see how they work together.

Before we begin, there is one major difference between pointers and arrays that I need to address. Pointers can be represented (and are) in machine code instructions. Indeed, pointer functionality is built right into your CPU.

This is not the case with Arrays. Arrays are therefore a construct of programming languages such as C, but are not directly implemented as machine code instructions on your CPU the way pointers to memory addresses are. In fact, we use arrays simply as a way to make working with pointers easier.

Let's examine a simple array of text characters:

```
char my_string[] = "Hello Reddit";
```

At this point, you should fully understand that we are creating an array called my_string and storing this text at the memory address of my_string. Each character in "Hello Reddit" is stored one at a time at its own unique address in memory, starting with the 'H', then the 'e', and so on. Each character resides in memory immediately after the character preceding it. All of the characters are stored in memory one immediately after the other, each character having a memory address that is exactly one greater than the memory address before it.

C has unique syntax for dealing with arrays. For example, we have to use the brackets [] after our array name. When we want an array index, we have to put it inside of those brackets. These are all constructs of the C programming language, and most languages have similar constructs for dealing with arrays.

In this lesson we are going to implement a two-dimensional array of text strings using only pointers, not arrays. This will help solidify the understanding that array indexing is really just using a pointer with an offset.

Here is the goal:

I intend to create an array of four strings of text, each string a maximum of six characters long. I will then create a printf() statement that will print each of these four strings just as if they had been arrays.

Here are the four strings of text:

```
[0] : "One"
[1] : "Two"
[2] : "Three"
[3] : "Four"
```

Why did I choose a maximum size of six characters? The longest word is "Three", which is five characters. I also need to account for a NUL termination character, which is why the need for six characters.

Whenever you perform an action that is designed to "give you space to work in", this process is known as allocation. In this case, I am allocating 24 bytes of memory to hold what will become my 4x6 array. This is because we have four elements that will each have six characters.

When I write this line of code:

```c
char my_string[4][6];
```

I am allocating 4x6 = 24 bytes to use for the array my_string. In this lesson, I am going to use a different method but I still need to allocate 24 bytes.
There are various ways I can allocate 24 bytes of storage. However, for the purpose of this lesson, lets do so like this:

```c
char storage[] = "12345678901234567890123";
```

Why did I choose the characters I did? It makes it easier to count. In this way you can start at 1, go to 9 and then the next 0 is "ten". Then you can go to "twenty", and then you can see I stop at 3. Therefore, 23 characters. The last character is of course invisible, the NUL string termination character. That makes 24 total characters. This has no bearing outside of this lesson, but I thought I should clarify why I chose the characters

I did to avoid any confusion.

Ah, but wait a minute. I said we would do this without arrays, yet I created an array called storage. Starting the lesson like this will make it easier to understand, but we will do this without any arrays before we are done.

So what you should know at this point is that I have created a string of text, consisting of 23 visible characters and an invisible NUL termination character. That gives me 24 bytes I can read or manipulate.

Now I have achieved step one, I have obtained (or allocated) the total number of bytes I need for my task.

Lesson 16.3 : Introducing Array Indexing as Pointer Offsets (Part Two)

The video for this lesson can be found here: http://youtu.be/IewsNZv6Mso

Recall in our previous lesson we have set out to create a two dimensional array using only pointers. Our array is to consist of four words each having a maximum of six total bytes in length.

The first step in our task is to allocate the storage we will need. In this case, we need 24 bytes. Even though it violates the spirit of the lesson, we are temporarily using the following method to allocate our 24 bytes.

```
char storage[] = "12345678901234567890123";
```

Now we are ready for part two of our lesson.

Once we have allocated the memory we will need, the next step is to actually start putting in the data. Lets recall the words:

```
[0] : "One"
[1] : "Two"
[2] : "Three"
[3] : "Four"
```

Well, it only makes sense to start with the first word, "One".

Before we examine how to put this word into our string of text, lets decide where to put it. Where would the first element of an array normally go? At the very start of the array.

So, lets put the word "One" (including the NUL termination character) into our array at the very first byte using a pointer. First, lets create the pointer.

char *ptr = &storage[0];

Our pointer now contains the memory address of the first byte, which is where we want to put the 'O' for one. Let's store the word "One" like this:

```
// Figure (a)

*(ptr + 0) = 'O';    // <-- At the memory address where storage begins, put an 'O'
*(ptr + 1) = 'n';    // <-- At the very next byte, put an 'n'. And so on.
*(ptr + 2) = 'e';
*(ptr + 3) = ' ';
```

Remember that '\0' is a single character which is: 0000 0000. Also, remember that ptr throughout this lesson only contains the memory address to the start of storage. We are NOT changing the value of ptr. Rather, we are using offsets to locate a new memory address by starting with the memory address in ptr, and then adding some number.

Notice the similarities to the above code, and the way we would do the same thing with an array:

```
// Figure (b)

storage[0] = 'O';
storage[1] = 'n';
storage[2] = 'e';
storage[3] = ' ';
```

The code in Figure (a) and the code in Figure (b) are identical.

Now we are done with the first word. Let's put in the second word. Where does it go? We would not begin our second word right after the first word. Why? Because as you learned in earlier lessons arrays must meet the criteria that all elements are the same length. What is the length we chose in this case? six. Meaning, the second word must start at byte #6. In other words:

```
// Figure (c)

Bytes  0,  1,  2,  3,  4,  5 : "One"
Bytes  6,  7,  8,  9, 10, 11 : "Two"
Bytes 12, 13, 14, 15, 16, 17 : "Three"
Bytes 18, 19, 20, 21, 22, 23 : "Four"
```

Because we are starting at 0 and counting to 23, that is 24 total bytes.

Even if each word doesn't fill up the six bytes allocated to it, those six bytes are still reserved just for that word. So where does the second word begin? Byte #6.

Before we put it in, I should make a comment. Keep in mind that we have started out with all 24 of these bytes initialized to a character that we know. When we are done we will look at how the final string will look.

Now, we know the second word will start at byte #6, so lets put it in:

```
*(ptr + 6) = 'T';
```

```
*(ptr + 7) = 'w';
*(ptr + 8) = 'o';
*(ptr + 9) = ' ';
```

Done. Notice that saying storage[6] = 'T' achieves the same thing as the first line of the above code.

The third word will begin at byte #12. Notice that this is 6*2. Just as we talked about, you can find the start of any element in an array by multiplying that element number (in this case, element 2 since it is the third word and you start counting at 0) times the size of each element (which is 6). 6*2 = 12.

The first word starts at position 0 because 6*0 is 0. The second word at 6 because 6*1 is 6. The third word at 12 because 6*2 is 12. And so on. Remember, we start counting array elements at zero. The first is 0, second is 1, and so on. If this is confusing to you, examine Figure (c) and notice what byte # each array element starts at. Notice the third element starts at byte 12.

```
*(ptr + 12) = 'T';
*(ptr + 13) = 'h';
*(ptr + 14) = 'r';
*(ptr + 15) = 'e';
*(ptr + 16) = 'e';
*(ptr + 17) = ' ';
```

Now the fourth word. 6*3 is 18, so that is where the fourth word will begin.

```
*(ptr + 18) = 'F';
*(ptr + 19) = 'o';
*(ptr + 20) = 'u';
*(ptr + 21) = 'r';
*(ptr + 22) = ' ';
```

Notice that "Four" follows immediately after "Three" in our array, and that is not the case with the other elements. This is because we chose the size of our array based on the size of "Three". There is no wasted space between where "Three" ends and "Four" begins.

Now we have stored all the words. Here is what our string now looks like:

```
"One$__Two$__Three$Four$_"
```

I needed some way to represent the invisible character NUL, so I chose a $. The underscores represent data that has not been set. In other words, wasted space. The dollar signs and underscores are just for this lesson, not part of C itself.

Remember that we started the word "Three" at position 12. Why? because "Three" is word number 2 (0, 1, 2). If we wanted the 'r' in three, we would say: 12+2 which is 14. Look above at the code where we stored "Three" into memory and you will see that character 14 is in fact 'r'. You should mentally experiment with other concepts concerning arrays and use the above examples as a guide. For example, how would you find the 2nd letter of the word "Four" ?

In the next lesson we will look at how to use the strings we have stored in memory as if they were arrays.

Lesson 16.4 : Introducing Array Indexing as Pointer Offsets (Part Three)

The text in this lesson is almost identical to Lesson 74.

We are going to make one change from the last lesson. Instead of storage being defined as a single variable array, for this lesson, we are going to imagine that it was created like this:

```
char storage[4][6];
```

What you are going to read is a modified version of the last lesson which will contrast the differences between a single dimensional array, and a two dimensional array.

Now lets begin right after we create a pointer to the start of the array.

Our pointer now contains the memory address of the first byte of our two dimensional 4x6 array. The first byte of our array is where we want to put the 'O' for one. Let's store the word "One" like this:

```
// Figure (a)

*(ptr + (0 + 0)) = 'O';
*(ptr + (0 + 1)) = 'n'; // <-- same as storage[0][1] = 'n';
*(ptr + (0 + 2)) = 'e';
*(ptr + (0 + 3)) = ' ';
```

Notice the similarities to the above code, and the way we would do the same thing with a two-dimensional array:

```
// Figure (b)

storage[0][0] = 'O';
storage[0][1] = 'n';
storage[0][2] = 'e';
storage[0][3] = ' ';
```

So our two dimensional array storage has four words. storage[0] is of course the word "One". Therefore, storage[0][0] is the first character of "One" which is 'O'.

The code in Figure (a) and the code in Figure (b) are identical.

Now we are done with the first word. Let's put in the second word.

We would not begin our second word right after the first word in memory. Why? Because as you learned in earlier lessons arrays must meet the criteria that all elements are the same length. What is the length we chose in this case? six. Meaning, the second word must start at byte #6. In other words:

```
// Figure (c)

storage[0] :  0,  1,  2,  3,  4,  5 : "One"
storage[1] :  6,  7,  8,  9, 10, 11 : "Two"
storage[2] : 12, 13, 14, 15, 16, 17 : "Three"
storage[3] : 18, 19, 20, 21, 22, 23 : "Four"
```

Because we are starting at 0 and counting to 23, that is 24 total bytes.

Even if each word doesn't fill up the six bytes allocated to it, those six bytes are still reserved just for that word. So where does storage[1] (the second word) begin? Byte #6.

Now, we know the second word will start at byte #6, so lets put it in:

```
*(ptr + (6 + 0)) = 'T';  // <-- Same as storage[1][0] = 'T'
*(ptr + (6 + 1)) = 'w';
*(ptr + (6 + 2)) = 'o';  // <-- Same as storage[1][2] = 'o';
*(ptr + (6 + 3)) = ' ';
```

Each letter of this word is identifiable using an offset from where the word begins. Since the word "Two" begins at byte #6, then we simply add a number to 6 to get the correct letter position.

The third word will begin at byte #12. Notice that this is 6*2. Just as we talked about, you can find the start of any element in an array by multiplying that element number (in this case, element 2 since it is the third word and you start counting at 0) times the size of each element. 6*2 = 12.

Now let's store "Three" starting at byte #12:

```
*(ptr + (12 + 0)) = 'T';
*(ptr + (12 + 1)) = 'h'; // <-- same as storage[2][1] = 'h';
*(ptr + (12 + 2)) = 'r';
*(ptr + (12 + 3)) = 'e'; // <-- same as storage[2][3] = 'e';
*(ptr + (12 + 4)) = 'e';
*(ptr + (12 + 5)) = ' ';
```

Now the fourth word. 6*3 is 18, so that is where the fourth word will begin.

However, this time let's make a change. Instead of saying that each letter of "Four" is understood by adding 18 to some number, let's just represent 18 as being six times three. It means exactly the same thing.

```
*(ptr + ((6*3) + 0)) = 'F'; // <-- Same as storage[3][0] = 'F';
*(ptr + ((6*3) + 1)) = 'o';
*(ptr + ((6*3) + 2)) = 'u';
*(ptr + ((6*3) + 3)) = 'r'; // <-- Same as storage[3][3] = 'r';
*(ptr + ((6*3) + 4)) = ' ';
```

Why did we do it this way? Because now you can clearly see the relation between offsets and array indexing. It is as follows:

```
array[x][y] means *(ptr + (SIZE * x) + y)
```

In this case, SIZE was 6 because each element is 6 bytes in size.

Notice that "Four" follows immediately after "Three" in our array, and that is not the case with the other elements. This is because we chose the size of our array based on the size of "Three". There is no wasted space between where "Three" ends and "Four" begins.

Now we have stored all the words. Here is what our string now looks like:

```
"One$__Two$__Three$Four$_"
```

Remember that we started the word "Three" at position 12. Why? because "Three" is word number 2 (count: 0, 1, 2). If we wanted the 'r' in three, we would say: 12+2 which is 14. We can also do this by saying: (6*2) + 2.

Now some closing notes:

The purpose of this lesson is to help you visualize pointers, offsets, and array indexing better. Notice how in the last lesson you understood each pointer offset as simply a number being added to the start of the array.

In this lesson, I showed you that array indexes are more properly understood by multiplying the size of an element times the element number, and then add another offset to this in order to get the actual element.

Observe how this progresses:

```
storage[0][0]          is the same as
    *(ptr + (6*0) + 0)    is the same as
        *(ptr + 0)        is the same as
            *ptr

storage[1][3]          is the same as
    *(ptr + (6*1) + 3)    is the same as
        *(ptr + 6 + 3)    is the same as
            *(ptr + 9)
```

OR

```
array[x][y]    is   *(ptr + (size * x) + y)
```

In the end, you get just a number. You can say that the 'r' in three is found by just saying byte #14, or by saying 12 + 2 (since "Three" starts at byte #12). You can also get this same result by saying: (6 * 2) + 2. The end result is the same.

One thing I hope you have learned from this lesson is that any dimensional array is in truth, just a one dimensional array. Earlier lessons concerning arrays and pointers should now make more sense to you.

Lesson 16.5 : Array Indexing as Pointer Offsets (Part Four)

In this lesson, we are simply going to start with the string that we created in our earlier lessons. There is no need to go through and recreate it, so here it is:

```
"One$__Two$__Three$Four$_"
```

Now, recall from the previous lesson that:

```
storage[0] :  0,  1,  2,  3,  4,  5 : "One"
storage[1] :  6.  7.  8.  9. 10, 11 : "Two"
storage[2] : 12, 13, 14, 15, 16, 17 : "Three"
storage[3] : 18, 19, 20, 21, 22, 23 : "Four"
```

Now we have all the information we need to finish. The last step in our task is to use printf() to actually display these strings as if they were arrays.

Normally we would do this:

```
printf("Here is a string %s", string_goes_here);
```

But what exactly goes there? A pointer to a string. In other words, you send the memory address of the first character you want to print, and printf() will continue printing until it encounters a NUL termination.

Let's now see this in action:

```
printf("The 1st string is: %s  ", (ptr + 0));
printf("The 2nd string is: %s  ", (ptr + 6));
printf("The 3rd string is: %s  ", (ptr + 12));
printf("The 4th string is: %s  ", (ptr + 18));
```

Notice that ptr + 0 is the same thing as ptr. Here you see that I am just giving printf() the correct memory address to the start of the string I want to print.

In our next lesson we will do away with the storage array altogether.
Now, here is a complete program showing this whole process:

```
#include <stdio.h>

int main() {

    char storage[]   = "12345678901234567890123";

    char *ptr = &storage[0];

    *(ptr + (6*0) + 0) = 'O';
    *(ptr + (6*0) + 1) = 'n';
    *(ptr + (6*0) + 2) = 'e';
    *(ptr + (6*0) + 3) = ' ';

    *(ptr + (6*1) + 0) = 'T';
    *(ptr + (6*1) + 1) = 'w';
    *(ptr + (6*1) + 2) = 'o';
    *(ptr + (6*1) + 3) = ' ';

    *(ptr + (6*2) + 0) = 'T';
    *(ptr + (6*2) + 1) = 'h';
    *(ptr + (6*2) + 2) = 'r';
    *(ptr + (6*2) + 3) = 'e';
    *(ptr + (6*2) + 4) = 'e';
    *(ptr + (6*2) + 5) = ' ';

    *(ptr + (6*3) + 0) = 'F';
    *(ptr + (6*3) + 1) = 'o';
    *(ptr + (6*3) + 2) = 'u';
    *(ptr + (6*3) + 3) = 'r';
    *(ptr + (6*3) + 4) = ' ';

    printf("The 1st string is: %s  ", (ptr + (6*0) + 0) );
    printf("The 2nd string is: %s  ", (ptr + (6*1) + 0) );
    printf("The 3rd string is: %s  ", (ptr + (6*2) + 0) );
    printf("The 4th string is: %s  ", (ptr + (6*3) + 0) );

    return 0;
}
```

UNIT 17 : Memory Allocation and Data Structures

Lesson 17.1 : Introducing memory allocation using malloc()

In the last series of lessons I used an array in order to allocate space for something I needed. This is, as you can imagine, a poor way to do things.

There are a variety of problems with that approach. One of the biggest problems is that sometimes you do not know just how much space you have to allocate. Let's suppose you are writing an application and you need to allocate space to hold the document someone is working on.

Whenever we refer to the process of allocating memory while a program is running, we speak of this as "dynamic memory allocation".

You should see that this is a rather fundamental capability that is needed for any programming language. Some do this behind the scenes, but all of them in one form or another must give you a way to allocate enough memory for some task you want to achieve.

In C, we do this using a function called malloc(). This is short for "memory allocation".

malloc() will grab however many bytes we tell it to. So for example:

```
malloc(24);  // <--- Reserves 24 bytes for us to do what we need to do.
```

We still do not have all that we need. Knowing that there are 24 bytes of memory available for our use is good, but how do we actually use it? The first thing we need to know is, where are these 24 bytes?

Somewhere in memory there are 24 bytes that we can use, but where? Well, if we are talking about needing something to contain a memory address, what are we talking about? A pointer.

So you use malloc() with a pointer. It should make sense. I need to point some pointer at the 24 bytes in order to be able to use them. Doing so is very simple:

```
char *my_pointer;
```

There we go. Now I have a pointer. Now where do I point it? I point it at the 24 bytes malloc() will set up, like this:

```
char *my_pointer = malloc(24);
```

That is all there is to it. Now I have allocated 24 bytes of storage, and my_pointer can now be used to read and write to those 24 bytes of space.

I could put data into these 24 bytes in a variety of ways. One way is by just writing directly to the pointer offset I want. For example:

```
*(my_pointer + 0) = 'O';
*(my_pointer + 1) = 'n';
*(my_pointer + 2) = 'e';
*(my_pointer + 3) = '\0';
```

Are you starting to see the connection?

These 24 bytes are just like any other. I have told C to reserve 24 bytes of memory for me to work with, and I can do with those 24 bytes whatever I want.

It turns out that the example program in Lesson 76 will work just fine if you make two simple modifications:

DELETE THIS LINE:

```
char storage[] = "12345678901234567890123";
```

Then, change this:

```
char *ptr = &storage[0];
```

to:

```
char *ptr = malloc(24);
```

One more note, you need to add the following include file:

```
#include <stdlib.h>
```

If you do that, you will see the program in Lesson 76 works fine. You should therefore understand now that malloc() is just a way to allocate memory to work with.

The last thing to know is that when you are done with the memory allocated, you should free it so that it is available for other purposes. This is done with the free() function, like this:

```
free(ptr);
```

Remember that ptr is the pointer which points to our allocated memory.

Here is our final "array simulation" program, with no real arrays used:

```
#include <stdio.h>
#include <stdlib.h>

int main() {

    // We need 24 bytes to hold a 4x6 array
    char *ptr = malloc(24);

    // array[0] is the word "One"
    *(ptr + (6*0) + 0) = 'O';
    *(ptr + (6*0) + 1) = 'n';
    *(ptr + (6*0) + 2) = 'e';
    *(ptr + (6*0) + 3) = '\0';

    // array[1] is the word "Two"
    *(ptr + (6*1) + 0) = 'T';
    *(ptr + (6*1) + 1) = 'w';
    *(ptr + (6*1) + 2) = 'o';
    *(ptr + (6*1) + 3) = '\0';

    // array[2] is the word "Three"
    *(ptr + (6*2) + 0) = 'T';
    *(ptr + (6*2) + 1) = 'h';
    *(ptr + (6*2) + 2) = 'r';
    *(ptr + (6*2) + 3) = 'e';
    *(ptr + (6*2) + 4) = 'e';
    *(ptr + (6*2) + 5) = '\0';

    // array[3] is the word "Four"
    *(ptr + (6*3) + 0) = 'F';
    *(ptr + (6*3) + 1) = 'o';
    *(ptr + (6*3) + 2) = 'u';
    *(ptr + (6*3) + 3) = 'r';
    *(ptr + (6*3) + 4) = '\0';

    // Print the four words
    printf("The 1st string is: %s  ", (ptr + (6*0) + 0) );
    printf("The 2nd string is: %s  ", (ptr + (6*1) + 0) );
    printf("The 3rd string is: %s  ", (ptr + (6*2) + 0) );
    printf("The 4th string is: %s  ", (ptr + (6*3) + 0) );

    // Free up our allocated memory, since we are done with it.
    free(ptr);

    return 0;
}
```

Remember that malloc() doesn't actually set the bytes it allocates to 0 or anything, so you must do this yourself. It just picks some chunk of memory with whatever is already in it, and gives it to you. This memory could be something left over from an earlier program that ran. We will talk more about this later.

Also, keep in mind I didn't have to do this one character at a time. I did that in order to make this lesson clearer.

Lesson 17.2 : Continued Introduction to Data Structures in C

Having just finished arrays, and multi-dimensional arrays, it is now time to learn about true data structures. This is a critical and fundamental topic to master for any serious programmer.

We have covered a lot of material in the course up to this point, but I have good news for you. This is truly the last major pre-requisite you need to master before we can start reading and writing "real programs". Rest assured, the course will not end at that point. That is when it will truly begin.

This lesson is meant to answer two questions before we delve into this complex topic: What exactly is a data structure, and why are they important?

A data structure is a collection of data elements. In this sense, it is similar to an array. Indeed an array is itself a simple form of a data structure. There are two key differences however:

1. An array requires that all elements be of the same data type. A true data structure has no such requirement. For example, you can have integers mixed with strings of text mixed with music, graphics, and anything else you can imagine.
2. An array requires that all elements be of the same length. A true data structure has no such requirement. You can have some elements 10 bytes long, some elements 3 bytes long, and some elements a thousand bytes long.

Now, why do you need to learn data structures?

For starters, every single file format that exists is itself a data structure. If you ever want to read a .doc or write to a .pdf; if you want to display a .gif or do anything at all with any file format that you can imagine, then you must understand and be able to create and work with data structures.

Here are some examples:

In a word processing application, there is likely to be a complex data structure for the text you type, since it needs to keep track of not only the text itself but also the font, color, background color, whether or not it is bold, and more.

In a game, you would likely use a data structure for each weapon. You need to keep track of the weapon's name, how much ammunition, maybe the strength of a shot and various other details. Similarly, you would likely have a data structure to keep track of each enemy. Information contained in the data structure may include the location of the enemy on the map, the type of enemy, the speed it can travel, direction it faces, etc.

There is not a single major application or game I can think of which does not use data structures heavily. You simply must learn this.

Also, data structures are a pre-requisite for something called "Object Oriented Programming" (OOP), which will itself be the topic of future lessons. The concept of an "object" itself originated with data structures.

We have a lot of material to cover, so let's begin.

Lesson 17.3 : The need to describe a data structure

Before we can learn to work with data structures, we must first learn how to create them. Data structures are created differently than anything you have seen up until this point. There are a lot of details to this, and so I plan to go through this material slowly and thoroughly.

The first thing you must know about a data structure is that you cannot use it until you describe it.

Up until now we have not had a requirement to describe how we are using the data we create. For example, I can allocate 24 bytes of memory then put a 'O' at position 0, followed by 'ne' to make the word "One" for example. I can write to position #16, or #21. As soon as I have the memory to work with, I can put anything I want anywhere I want.

With a data structure however, I have to describe exactly how I plan to use the bytes of memory I allocate before I can actually use them. If I plan to have a string of text from bytes 0 through 10, I must state this. If I plan to have an integer starting at byte #18, I must state this also. Every detail concerning the data structure must be described fully before I can do anything.

Why is that? Because you cannot index a data structure like an array. This is because elements of a data structure are not necessarily the same data type, and/or the same length. Therefore, how can C possibly know where one element ends and another begins? It can't, which is why you have to describe a data structure before you can use it.

Here is a very simple data structure, with data already in it:

```
Figure (a)

"Reddit$Programming$Classes$"
```

Notice that '$' is really the NUL character, and the '$' is just for the purpose of this lesson, not part of C itself. The actual string doesn't have $ characters, but has the NUL termination character where the $ characters would go.

Why is this not an array? Because each element is not the same length. Notice I did not put any "filler" text to make the word "Reddit" and the word "Classes" as long as the word "Programming".

How would we define this data structure? For each element, we need to state three things:

1. Where does each element begin?
2. How long is each element?
3. What data type is each element?

Notice that with arrays all 3 of these questions are answered already. In the case of a data structure however, we must answer these questions before we can do anything.

Now, with the example in Figure (a), how do we answer that?

First of all, we know that this data structure consists of three words (strings of text). Each word has a varying length. Therefore, each word starts at a unique byte position that cannot be determined unless it is stated.

The word "Reddit" starts at... 0. That is easy enough. What about "Programming", where does it start? It starts at byte #7 (Don't forget the NUL character at the end of each string). And finally, "Classes". Where does it begin? byte #19.

Hopefully at this stage it is not too confusing. We have to tell C where each element starts. In other words we need to state:

1. The first word starts at byte #0 and is 7 bytes in length, of type char.
2. The second word starts at byte #7 and is 12 bytes in length, also of type char.
3. The third word starts at byte #19 and is 8 characters in length.

Why do we need to state this? Because each element is of a different length. In an array with each element being the same length you can simply multiply that length times the element number to get its starting position. That method will not work here simply because we are using different lengths.

So, the first thing we have established is that you must describe a data structure before you can use it. This description must include all the elements of the data structure, their data type, and their length.

In the next lesson I am going to show you how to do this.

Lesson 17.4 : Introducing the struct keyword

In the last lesson I explained that you must describe a data structure before you can use it. In C, you do this by using the struct keyword. The syntax for struct basically works like this:

```
struct <give it a name> {
   // ... each element is described here ...
   // ...
};  // <-- notice you end it with a ; (semicolon)
```

From the above description you can see that you must give each structure a name. We have seen the same thing when you create variables or arrays, however it is different in this case.

Unlike variables or arrays, the name we give to a struct is not related to the data itself. Remember, this is only a description, not the actual data. This is not even setting up a place to put the actual data. It is only describing it.

Therefore, the name we give the structure is only the name of the description of the data, not the data itself.

Why do you have to give the description a name? Suppose you want to describe multiple kinds of data structures, how would you be able to differentiate between them? You give each data structure description a name simply because it makes it easy for you to specify which data structure description you are using.

Now, let's talk about what goes inside the curly braces. We are still using this data structure from our previous lesson:

```
"Reddit$Programming$Classes"
```

So what we really need here are three character arrays. One for "Reddit", one for "Programming", and one for "Classes".

I will call the description: first_description

```
struct first_description {
    char first_word[7];
    char second_word[12];
    char third_word[8];
};
```

Notice I gave each element a name. I can name them anything I want, just like variables. Notice I also had to define the type of data (char arrays in this case) and the length in bytes.

These are not variables in the typical sense. I am not actually creating anything here, I am only describing what I intend to create. I have not created a variable called "first_name" which is 10 bytes long, rather I have told C that at some point I intend to create this variable as part of my data structure.

Everything involving the struct keyword is only a description.

That is basically all there is to it. I have now described the data structure. Before we move on to the next lesson, I want to give you one more example of describing a data structure:

```
struct student_records {
    int student_record_number;
    char first_name[30];
    char last_name[30];
    char birth_date[10];

};
```

Here I have described a data structure which will be used to keep track of student records for a school. Notice that I can mix different data types with no problem.

Remember that these are only descriptions, not the actual data structures. You are only telling C "At some point I plan to use a chunk of memory in this way". In the next lessons you will see how to actually use the data structure you have described.

Lesson 17.5 : Allocating memory for a data structure

The first step in being able to create a data structure is describing it and giving that description a name. However, we still cannot do anything. Just as we saw in previous lessons, before you can actually do any task you must give yourself some memory to work with.

So it is in the case of a data structure. Recall our struct definition looks like this:

```
struct first_description {
    char first_word[7];
    char second_word[12];
    char third_word[8];
};
```

How much memory do we need? Well, 7+12+8 is.. 27. Therefore, we need to allocate 27 bytes in order to use this data structure. Does this mean that every time you need to allocate space for a data structure that you must manually add up the length of each element? Thankfully, no.

C has a built in function called sizeof() which will tell you the size in bytes of virtually anything. We can get the total size in bytes that we need for any data structure using the sizeof() function.

We know that we are going to be allocating 27 bytes of memory for our data structure. That should tell you that we will be using the malloc() function. The malloc() function returns a pointer to the memory that is allocated.

How did we use malloc() last time? Like this:

```
char *some_pointer = malloc(100);
```

100 is the number of bytes, and we have to use it with a pointer. What we are saying here is simply this:

"Allocate 100 bytes of memory and store the memory address in some_pointer"

With a structure, we cannot do things quite so easily. What kind of pointer do we use? We cannot use a char * pointer because the data is a mixture of characters, integers, and who knows what else.

Remember that the whole reason you specify a data type for a pointer is so that the pointer knows how big and of what format the data will be. We therefore need to create a pointer that knows that we are using a data structure, and more specifically one that knows exactly how the data structure will work.

Why? Because we will be using this pointer to look at chunks of memory that we have allocated. How else will we see what is inside our data structure? Our pointer will not know when we are looking at an integer, or a string of text, or anything else unless we somehow include that in the pointer's definition.

That may sound like a difficult thing to do, but fortunately it is built right into the C language. We can actually tell C to create:

A pointer to the data structure itself.

Let me expand on that a bit. We are not going to say "Create a pointer to data type char" or "Create a pointer to an array". We are going to say:

"Create a pointer specifically to a data structure which has three elements, the first element having 7 bytes, the next element having 12 bytes, and the last element having 8 bytes."

In other words, we are going to have a pointer that is perfectly fitted to our data structure. This will give us the ability to easily see and work with all of the elements in the memory we will allocate. C will automatically know when we are looking at an integer, or a character string, or anything at all.

Watch this syntax carefully:

```
struct first_description *our_pointer = malloc(27);
```

First notice the struct keyword. Then you see the name of the description we created earlier. This tells C exactly what kind of data structure we plan to work with. Next you see that we put a * character. This tells C that we are creating a pointer. Then the pointer name - whatever we want. Finally, we are pointing our pointer to 27 bytes that we just allocated for this data structure.

That is all there is to it. There is however an issue. The issue is, how do we know we need 27 bytes? In this case we just counted them, but this is risky and in some cases not practical. Let's see how to do this same exact definition (equally valid) using the sizeof() function:

```
struct first_description *our_pointer = malloc( sizeof(*our_pointer));
```

Notice I put: sizeof(*pointer_name). Notice I do this in the same line that I created the pointer. C can determine the size we need by just looking at the data structure description we made earlier, and we can plug our pointer right into the sizeof() function. sizeof(*our_pointer) is the same exact thing as 27.

These two lines are identical in function:

```
struct first_description *our_pointer = malloc( sizeof(*our_pointer));
struct first_description *our_pointer = malloc( 27 );
```

Both are saying that we are allocating 27 bytes. One lets C do the math for us, and the other we are doing the math ourselves.

The purpose of this lesson was to learn how to actually allocate enough memory to work with a data structure definition you have made. In our case, we have described a data structure and we gave our description the name: "first_description". Then I showed that you can allocate such a data structure using malloc() and a pointer just by typing this line of code:

```
struct description_name *pointer_name = malloc(size);
```

and size should be: sizeof(*pointer_name)

Lesson 17.6 : Using a data structure

Now you know how to describe and allocate memory for a data structure. Let's review the code we have already written:

```
struct first_description {
    char first_word[7];
    char second_word[12];
    char third_word[8];
};

struct first_description *our_pointer = malloc( sizeof(*our_pointer));
```

Now we have a chunk of 27 bytes somewhere in memory that C understands we plan to use for our data structure. What is in those 27 bytes is anyone's guess, as we have not yet set them to anything. Remember that malloc() does not actually initialize memory, it only gives it to us so that we can use it.

Now we need the ability to store data into the elements of our structure. This is not difficult at all. We treat our data elements exactly as you would expect. For example, to store a string of text into a char array, we would use the strcpy() function that you learned earlier.

Let's go ahead and set up our data structure.

Before I show you the actual syntax, let me show you a more intuitive syntax:

```
strcpy(first_word, "Reddit");
strcpy(second_word, "Programming");
strcpy(third_word, "Classes");
```

Here you understand exactly what we are trying to do. However, this is not going to work because these variables (first_word, second_word, third_word) do not actually exist as unique variables. They are part of a data structure we created.

The question is, how can we tell C that we are talking about elements inside our data structure? Here is how you do it:

```
data_structure.element_name
```

This is the syntax which makes it possible to use elements in a data structure. Remember that "data_structure" is not at all the same as "description name". Rather, it is the actual data structure that was created.

We created our data structure using a pointer. We gave the pointer a name of: our_pointer. Now, observe these two key facts:

1. our_pointer is the memory address where the structure begins. All pointers contain a memory address.
2. *our_pointer is "what is at" that memory address. In other words... *our_pointer is the actual data

structure.

Do not get these mixed up. our_pointer is the memory address. *our_pointer is the actual data structure. Now, let's go back to our syntax.

We need to say:

```
strcpy( (*our_pointer).first_word, "Reddit");
strcpy( (*our_pointer).second_word, "Programming");
strcpy( (*our_pointer).third_word, "Classes");
```

This is entirely valid syntax.

(*our_pointer) is the data structure itself. We put a period and then the element name. Now, we could printf() any of these like so:

```
printf("One string is %s  ", (*our_pointer).first_word);
```

Output:

```
Reddit
```

You can probably guess from this lesson that structures must be used a lot, and that this syntax (*structure_pointer).element_name must therefore also be used a lot. It turns out that the developers of C decided to make a short-hand method for this syntax, which makes programming a lot easier and more intuitive.

Instead of having to write this:

```
(*our_pointer).first_word
```

You can write this instead:

```
our_pointer->first_word
```

They are identical. The second is just a short-cut method of doing this.

Now, I have created a program illustrating all that you just learned, including being able to see that the memory of a structure is exactly as I said it would be.

This program is the next lesson.

Lesson 17.7 : Sample program illustrating data structures

First you will see the program itself, then you will see the same program with additional notes explaining what is going on.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {

    struct first_description {
        char first_word[7];
        char second_word[12];
        char third_word[8];
    };

    struct first_description *our_pointer = malloc(sizeof(*our_pointer) );

    char *charptr = (char*) our_pointer;

    strcpy(our_pointer->first_word, "Reddit");
    strcpy(our_pointer->second_word, "Programming");
    strcpy(our_pointer->third_word, "Classes");

    printf("The first word is: %s  ", our_pointer->first_word);
    printf("The second word is: %s  ", our_pointer->second_word);
    printf("The third word is: %s  ", our_pointer->third_word);

    printf(" ");

    printf("Our data structure looks like this in memory: ");

    int i=0;
    for (; i < 27; i++) {
            if ( *(charptr + i) == 0) {
                *(charptr + i) = '$';
            }

            printf("%c", *(charptr + i));
    }

    printf(" ");

    free(our_pointer);

    return 0;
}
```

Now I will explain the program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

These include files give us printf(), malloc(), and strcpy().

```
int main(void) {

    struct first_description {
        char first_word[7];
        char second_word[12];
        char third_word[8];
    };
```

Above: Here is our structure description. We are not actually creating any data structure here, just telling C what we intend to create. No data is being initialized. This is a description and nothing more.

```
    struct first_description *our_pointer = malloc(sizeof(*our_pointer) );
```

We are allocating 27 bytes of memory using this malloc() statement. Then we are creating a special pointer called our_pointer which C understands points to this kind of data structure. After this line of code, our data structure is ready to be used.

```
    char *charptr = (char*) our_pointer;
```

I plan to scan our data structure to display the final memory contents at the end of this program. To do that, I am creating a new pointer called charptr which I am stating is going to be a char * pointer. I am setting this pointer to look at the memory address where our structure begins.

```
    strcpy(our_pointer->first_word, "Reddit");
    strcpy(our_pointer->second_word, "Programming");
    strcpy(our_pointer->third_word, "Classes");
```

Here I am simply assigning the strings into the character arrays that are part of our data structure.

```
    printf("The first word is: %s  ", our_pointer->first_word);
    printf("The second word is: %s  ", our_pointer->second_word);
    printf("The third word is: %s  ", our_pointer->third_word);
```

I am displaying the three words, each element of our data structure.

```
    printf(" ");

    printf("Our data structure looks like this in memory: ");

    int i=0;
    for (; i < 27; i++) {
        if ( *(charptr + i) == 0) {
            *(charptr + i) = '$';
        }

        printf("%c", *(charptr + i));
    }
```

Now I have a for loop which will go through all 27 bytes and display the character represented. If it is a NUL character, I am having it display a $ instead by actually changing that character in memory to a $.

```
    printf(" ");
```

Now I need to free the memory I allocated using malloc()

```
    free(our_pointer);

    return 0;
}
```

Output:

```
The first word is: Reddit
The second word is: Programming
The third word is: Classes

Our data structure looks like this in memory: Reddit$Programming$Classes$
```

Lesson 17.8 : You can make your own data type using struct

We have learned about different data types: int, char, etc. We have learned that a data type effectively has two characteristics:

1. length
2. format

Format of course meaning "how it is understood". You will notice that structures are very similar in this respect, in that all data structures have a length and a format. The format being the description of the data structure, and the length being what sizeof() returns.

It turns out you can create your own data type using a structure definition you made. Once you make this data type, you can create "variables" using it just as you can with int, char, or any other data type.

To do this, you must use the typedef keyword, and slightly change the way you set up your definition earlier. Instead of this:

```
struct first_description {
    char first_word[7];
    char second_word[12];
    char third_word[8];
};
```

You do this:

```
typedef struct first_description {
    char first_word[7];
    char second_word[12];
    char third_word[8];
} reddit_type;
```

I just created a new data type called "reddit_type". I can call it anything I want. In this case, any time I want to use this kind of data structure I can do so using the word "reddit_type" and I can do so just as if it was char, or int, or anything else.

Instead of writing this:

```
struct first_description *our_pointer = malloc( sizeof(*our_pointer));
```

I can write this:

```
reddit_type *our_pointer = malloc( sizeof(*our_pointer) );
```

What I am literally saying here is: "Make a pointer of data type reddit_type and allocate 27 bytes for it."

nothing else changes. If you make just those changes to example 83 you will see the program works identical.

This is extremely important because you can create functions that have a return type of reddit_type, or that take parameters that are reddit_type.

You will run into this a lot in C. You will be reading source code and see a function definition that looks like this:

```
    circle some_function(
```

This means that some_function returns.. a circle? Yep, the programmer used typedef to create a data type called circle.

Congratulations! You have just finished this course. There are no more lessons. You should return to the homepage and proceed to the next course.

COURSE 2

Unit 1 : Preparing to write Tic-Tac-Toe

Lesson 1.1 : Tic-Tac-Toe with A.I.

This lesson could also be titled: "The Real Course Begins".

We have finally reached the point in this course where you can start applying your knowledge to making "real" programs. We must start with simple programs and then we will work our way up. Of course, we are still going to be learning about new concepts along the way. Only now you will be learning about such concepts in the context of practical application.

Rather than show you how to write a simple 2-player tic-tac-toe game, I figured it would be better to show you how to write a tic-tac-toe engine that could play against and beat a human player given the opportunity; even better, one that can tell you as soon as a position is winnable, or lost, or drawn.

Also, you will be doing a large part of the work in writing this program, just like you did when you wrote your first C program back in Lesson 15 (2 weeks ago by the way). For the most part I will be simply describing what you need to do, and making sure you have all the tools you need to do it. I know that tic-tac-toe is probably not the most exciting project, but we have to start somewhere. We will get into bigger and better things soon enough.

The first thing I need to tell you is that to create a tic-tac-toe engine that is capable of calculating N moves ahead on a tic-tac-toe board, we will need to create a model of a tic-tac-toe board. To do this, we need to create a data structure that contains a 3x3 grid for putting X's and O's.

Why not an array? Well, if you created an array then you would be able to keep track of one tic-tac-toe position. You could create an array of such arrays.. but this gets unnecessarily complicated. Also, a data structure is capable of holding additional information. For example, you could have an int variable which tracks how many moves have been played. You could have another variable which tracks whether or not the position is winnable, and so on. Your data structure could contain additional "meta data" which is useful to the program.

Another major advantage to using a data structure is that once we have our data structure definition, we can create many data structures. In other words, we can have a unique tic-tac-toe board data structure for every position our artificial-intelligence engine evaluates.

Before we begin, let me describe some other practical applications of using multiple data structures in programming:

Suppose you are writing a chess program that can calculate moves on a chess board. You need a way to evaluate each position, and therefore you could create a data structure of a chess position. You can create many copies of this data structure that way you can evaluate many possibilities at once.

Here is another example. Suppose you are writing an artificial intelligence for an enemy in a computer game. You may desire a way to calculate the consequences of a specific action the character may take. You could create a unique data structure based on the expected result of a particular action and compare it to similar data structures created based on the expected result of a different action.

If you want to create the ability to undo some operation in a graphics program, one way to do this is to save the previous state of the drawing as a data structure. You could have numerous such structures each one containing a different drawing, and then a routine that can switch to one based on the undo command. Similarly, you could save "states" of a drawing that can be worked on uniquely, such as in layering.

If you are writing a web browser, you can have a data structure for a "tab", and then each time a tab is opened you can just create multiple copies of that data structure. Remember that each tab would be able to have its own unique data.

As you can see, there are countless applications for why you might need multiple data structures. This is also one reason why the typedef method I showed you in the previous lesson is so widely used. If I need to create a tic-tac-toe board, I can do so like this:

```
tictactoe_board *new_board = malloc(sizeof(*new_board));
```

in this case, tictactoe_board is some typedef to a data structure of what a tic-tac-toe board consists of. I could create an array of such boards to hold a new position after a move.

Lesson 1.2 : The need to initialize data

Don't let the title fool you. We are still working on our Tic-Tac-Toe project, and will be for the next lessons. However, each lesson as we do so will be introducing new topics in programming and software development. It makes better sense to title the lesson by the topic we will focus on.

The purpose of this project is to write a tic-tac-toe game which can be played with one or two players, and which is capable of "understanding" the position. We are not looking for speed or efficiency in this project, just a working program that can help to illustrate concepts we have learned already as well as to introduce new concepts.

As I said in the last lesson, we will need to construct a data structure in order to hold an instance of a tic-tac-toe board. Think of this as a digital representation or "model" of the tic-tac-toe board itself. This is our "data" component.

The data component is where all information related to our tic-tac-toe board will be stored. The rest of the program will use this data component in order to do all of the processing which will make our program work.

This processing is achieved through a range of functions designed for different tasks related to reading from and writing to the data. Some categories of such functions include:

1. Data Initialization
2. Data Manipulation Operations
3. Evaluation and Analysis
4. Rendering and Display

These are the four categories we will start with. Keep in mind that these topics are unlikely to be taught in any tutorial/book whose purpose is to teach a programming language in general. These have nothing to do with a particular programming language, but programming as a whole.

We are going to start with the "Data Initialization" category.

Initializing data simply means setting it to something before you begin working with it. More specifically, it means setting it to what you know will work and will be expected by the program.

Any real program requires this. For a web application, this starting state may be an HTML file with pre-built tables and spaces (often called place holders, or data containers) for data to go in. For a game, this might be a blank scenery with no mountains/enemies/other objects. For a graphics program, a blank drawing, and so on.

Now, you must always initialize data to a known working starting point. This is rarely if ever going to be simply "blank space". You never leave data uninitialized.

There are many reasons why you must create a base state. The most important reason is that every function you write should expect a certain kind of data. If the data is not exactly the way a function expects, then that function may not work correctly.

Therefore, every function should have a well defined expectation of exactly what will be the state of the data at the time this function will run. Further, it must have a mechanism to react if the state of the data is not as expected. In other words, it must be able to recognize and react when it is not given what was expected. Do not forget this.

By knowing exactly what kind of data a function expects, it is much easier to create tests for functions as well as to develop for one function without worrying about others. A common mistake found with novice programmers is that the entire program is best described as a "pass the baton" relay race where no function has any clearly defined expectations for what it will receive from the last function. The programmer just assumes that everything will work as expected.

This leads to many problems. Imagine you have ten functions where function #1 does something, sends what it does to #2, which sends it to #3 and so on. By the time it reaches function #10 (or worse yet, the end user), something may have gone horribly wrong. Finding what went wrong and troubleshooting it will prove to be a nightmare as any function along the way could have been the culprit. Worse still is that fixing such a problem is almost guaranteed to create new problems.

Therefore, in this course I will be showing you how to avoid such problems by showing you the proper technique for developing functions in a program.

Lesson 1.3 : Introducing the constructor function

In the last lesson I went into detail about the need for any program to initialize a data structure to a known initial working state.

A function whose job is to allocate memory for data structures, initialize data, and overall "prepare" for the program to run is called a "constructor function". The constructor function is always ran first, before anything else. It will "pave the road" for future functions. In our example we will want a constructor function which can initialize a "tic tac toe board" to some starting point.

First, we need to decide on an initial state. I propose that we have our tic tac toe board consist of three rows of three underscore characters. This will be easy to work with, and when you see it displayed on the screen it will look like this:

```
___
___
___
```

So, how can we represent that as a data structure?

It looks to me that the easiest way is to create a structure similar to this:

```
typedef struct tictactoe_board_description {
    char square[3][3];
} tictactoe_board;
```

Here I created an array of characters called square which will be a 3x3 grid. We will decide that each square can have either an underscore (initial or default state), an 'X', or an 'O'.

Now, how can I initialize all squares to underscores? Let's write a simple constructor function:

```
int init_board(tictactoe_board *board) {

    int i = 0;
    int j = 0; // used for for loop

    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            board->square[i][j] = '_';  // Set each square to _
        }
    }

    return 1;
}
```

The word init in our function name is short for initialize. This function is different than any we have done up until now. Let's evaluate it a bit. First of all, it takes a rather strange parameter: a tic tac toe board.

Notice how much the typedef keyword helps us. I can create a function that receives a tic tac toe board as an argument. However, this is only part of the story. Notice that my function looks like this:

```
(tictactoe_board *board)
```

Why the *? Because I am not sending a tic tac toe board to the function. I am sending a pointer to the tic tac toe board. All I am sending the function is the memory address along with the understanding that this memory address is the start of a range of memory (the size of our structure) that can be used as the data structure described earlier as a tic tac toe board.

Keep in mind we have only created a description of that data structure, we have not created any instances of it yet. However, the description alone is enough for our function to know how to handle any memory range which we define as a tic tac toe board, provided of course that the memory range is the correct size in bytes.

Most likely, the memory address will contain who knows what left over garbage from earlier programs. The purpose of the constructor function is to transform that memory from junk data into our tic tac toe board. This may still be a bit confusing, so let me explain it one more time:

A pointer to a data structure means "Here is some memory that can be used for this purpose." The struct description tells C how we intend to use that memory. Now let me show you a visualization of this:

```
1000 : 0110 1111 : 0011 1110 : 0001 0000 : 0000 0000 : 0100 0011 :0011 1111 <--- a range of memory with 1s/0s left over.
```

Imagine we just told C that we plan to use this memory at position 1000 (eight) for our data structure of a 3x3 grid of characters.

All C gives us is the address "1000". That address does NOT "have our data structure". It is just a sequence of 1s and 0s left over in memory, which could be ANYTHING. However, C now understands how we intend to use that memory. It understands that the first byte now means: "square[0][0]". It understands that the third byte now means: square[0][2]. And so on. A data structure is only a set of rules for how to understand data in a range of memory, in this case six bytes.

So when our constructor function receives the pointer to some memory like this, it doesn't care what is in that memory. It only cares that C has established the correct rules for understanding it. Those rules are of course our data structure description. If that is still unclear, please tell me.

Now, back to the function.

We also gave our parameter a name, "board". This is the name that our function will use to identify this pointer inside the function. This has nothing to do with what if anything it may have been called in other functions, including main().

Every time you create a parameter for a function, you give it a name that is used just by that function. I chose the name "board", but I could have chosen anything else. It does not matter at all what it may be called outside of the function, or even if it is also called "board".

When you create a function, you choose a name for every parameter you give it, and that name will be the name used inside that function. In this case, the function expects a parameter called "board" which is a pointer of the data type "tic-tac-toe board".

Now from that line onward to the end of the function:

1. board refers to the memory address.
2. *board refers to what is at that memory address, the actual data structure.

Therefore, inside the function if I write: (*board).square then I am referring to the member element "square" in our data structure that will be sent to this function. Remember from before that there is a short hand way of writing this: board->square.

Therefore, by writing:

```
board->square[0][0] = '_';
```

I would be setting the first row and first column square to an underscore. By doing this for all 3 rows, for all 3 columns, I set all squares to an underscore. That is why I used the variables i and j to each go through the 3 rows and 3 columns, setting each one to an underscore.

Finally, the function returns an int. Why do I not return the tic tac toe board? I could (that will be the subject of later lessons), but I do not have to. Why I do not have to is the subject of the next lesson.

Lesson 1.4 : Introducing Pass-by-Reference and Pass-by-Value

In the last example I showed you a function which received a pointer to a data structure, and returned an int. I imagine a question that could be on someone's mind is, how do I "get back" the data structure from the function?

Well, it turns out that I already have it back. When you send a pointer to a function, that function can read and write straight to the actual memory itself. It doesn't have to send anything back. Consider this code:

```
int height = 5;
int *ptr = &height;

*ptr = 2;
```

What is height now? Height is set to 2. It is no longer set to 5. Why? Because by changing the actual data stored at the memory address, I changed height itself.

Therefore, the function in the last lesson which sets the underscores to the memory address of the pointer it received has changed those underscores for any other function which will look at that same memory address. As soon as the constructor is done with that operation, every other function now has an initialized tic tac toe board without even having to talk to the constructor function.

There are two ways you can send something to a function. The first is called Pass by Reference.

Consider the following program:

```
#include <stdio.h>
```

```
int main(void) {

    int height = 5;
    printf("Height is: %d  ", height);

    change_height(&height); // Pass by reference

    printf("Height is now: %d  ", height);

    return 0;
}

int change_height(int *ptr) {
    *ptr = 2;

    return 1;
}
```

Output:

```
Height is: 5
Height is now: 2
```

Notice therefore that the main() function (or any function) can simply send the memory address of any variable, array, structure, etc. to a function. The function can then change the data in place at that memory address. This is because by sending the actual memory address where the data is located, any changes to that data become universal to anything else looking at that same memory address.

At the same time, I do not have to send a memory address to a function. For example, I can have a function like this:

```
int some_function(int height, int width) {
    return height*width;
}
```

In this case, even if I change height or width inside the function, it will not change it anywhere else. Why? Because I am not sending the actual integers to the function, I am sending a copy of them. C will actually create a copy of these variables when they are sent to the function. These copies will of course reside at different memory addresses than the originals.

Now this is easy to remember:

1. If you send the memory address (aka a pointer) to a function, then anything that function does on the data is universal.
2. If you send the name of a variable to a function, not a pointer, then a copy of that variable will be created and sent to the function. Any changes done on such variables will be visible only within the function.

We refer to #1 as "Pass by Reference". This means you "pass" an argument to a function by sending it the memory address.

We refer to #2 as "Pass by Value". This means you "pass" an argument to a function by creating a copy of it.

Here is a program illustrating pass by value:

```
#include <stdio.h>

int main(void) {

    int height = 5;
    printf("Height is: %d  ", height);

    wont_change_height(height);  // Pass by value

    printf("Back in main() it is: %d  ", height);

    return 0;
}

int wont_change_height(int some_integer) {
    some_integer = 2;
    printf("Inside the function height is now: %d  ", some_integer);

    return 1;
}
```

Notice that inside a function you can call a variable whatever you want. I can call it height before I send it, then call it some_integer when it is inside the function. This is useful because it is not necessary to remember what a variable name was before it was sent to a function. Also, it is ok if two functions have parameter names that are the same. We will talk more about this later.

It is worth pointing out that in C, there is no true "pass by reference". Instead, you pass a pointer "by value". For the purpose of understanding this lesson, think of passing a pointer as a form of "pass by reference". However, remember that the truth is you are not actually passing anything by reference, you are just passing a pointer by value.

Lesson 1.5 : Introducing the Stack

In the last lesson we covered the different methods you can use to send variables to a function. However, we have not yet covered what it really means to "send" something to a function to begin with. That is the purpose of this lesson.

Right now, this process is black magic. You write: some_function(height) and somehow the variable height "gets sent". What does this mean? How is a variable or a pointer "sent" to a function? How can two functions communicate to each other?

To explain this, I have to introduce you to something called the Stack.

The stack is a special range of memory in your computer which is used to store and retrieve data in a unique way.

Recall from earlier lessons that a CPU chip has a built in register called the "Instruction Pointer". The "Instruction Pointer" contains the memory address of the next instruction to execute. It turns out that the Instruction Pointer is not the only register on your CPU chip that stores memory addresses.

There is an additional register on your CPU chip known as the "Stack Pointer", or SP for short. Keep in mind that the Instruction Pointer and the Stack Pointer are different.

[Note: However, to be entirely technically accurate, there are some architectures which do not have a dedicated "Stack Pointer" register. These architectures use other registers to act as a Stack Pointer. This however does not affect the lesson.]

Different ranges of memory have different purposes. The "Stack Pointer" register contains memory addresses in much the same way as the "Instruction Pointer" register does, just that it contains memory addresses located in a different range of memory. Each range of memory has a different name. One range of memory is called the stack. The Stack Pointer looks at memory addresses within the stack.

To properly understand the stack, it is important to understand that functions have no way to communicate to each other directly. For example, my main() function has no way to communicate to the init_board() function.

Why is that? When you CALL a function, it is little more than a glorified "Go to" statement. You are saying to "go to" the point where the function begins. There is no way to somehow send data that can hitch a ride on a goto statement. There is no way for a function once called to "look back" at what an earlier function was doing.

This is where the stack comes in. Before a function CALLs another function, it can put data onto the stack in memory. Notice I did not say "into", I said "onto". We will get to that.

Then inside the function the stack can be read in order to see what data was sent to the function. The stack is a type of "middle man" for communicating between functions. All functions can read from and write to the stack. Therefore, main() can put something on to the stack, and then init_board() can read it off of the stack. Similarly, other functions can communicate in this way.

There is more to the stack than just this, but that will be the topic of future lessons.

In summary, here is what you should know from this lesson: Functions cannot talk to other functions directly. They must use the stack in order to communicate.

How this works is the topic of the next lesson.

Lesson 1.6 : Introducing PUSH and POP

In the last lesson you learned that there is a range of memory known as the stack. You also learned that this range of memory serves as a data storage and retrieval bin that functions use in order to communicate to each other.

In this lesson I am going to explain some of the basics concerning the mechanics of how the stack works.

The first thing to understand about the stack is that there are two machine code instructions built into your CPU chip which are used to write data to the stack, and to retrieve data from the stack. These two functions are:

PUSH and POP

PUSH means: "store data" to the stack.

POP means: "retrieve data" from the stack.

Now that you know what each of these instructions is designed to do, let's talk about how they do it.

We have already learned that any programming language provides a mechanism to store data in memory at a specific memory address. Similarly, there obviously exists a mechanism to read the data from memory based on knowing that memory address.

The key point to this process is that in order to store or retrieve data you must specify the memory address where you are either putting the data, or reading it from.

On a machine level, the process to specify a memory address takes time and CPU resources. Machine code instructions that require a memory address are going to be slower and more resource intensive than machine code instructions that do not require a memory address.

When you call a function in C or any language, the speed at which you can get in and out of that function is of tremendous importance. This is why PUSH and POP are so important. You see, PUSH and POP do not require any memory address. This is what makes the stack unique.

The stack is different from other ranges of memory because every time you store something onto the stack using the PUSH instruction, you store that item (variable, pointer, etc.) on top of the rest of the items already on the stack. Whenever you use the POP instruction to retrieve something from the stack, you only retrieve the item which is on top of the stack. Therefore, neither PUSH nor POP require you to specify a memory address.

Here is an example of a stack. This stack is half full.

```
0000 0000   <-- empty
0000 0000   <-- empty
0000 0000   <-- empty
1010 0101
0010 1001
1011 0101
```

If I use PUSH to store something onto the stack. It will always go to the first "unused" spot located at the top of the stack, on top of the data already there. So if I were to store 1111 1111 to the above stack, the result after will be this:

```
0000 0000   <-- empty
0000 0000   <-- empty
1111 1111   <-- I just stored this.
1010 0101
0010 1001
1011 0101
```

So each time you use the PUSH instruction the data will go "onto" the stack. Meaning, it will go into the first unused memory address; the top of the stack.

Now, let's talk about POP.

Just as PUSH stores data at the top of the stack, POP retrieves data from the top of the stack. So if I have a stack that looks like this:

```
0000 0000   <-- empty
0000 0000   <-- empty
1111 1111
1010 0101
0010 1001
1011 0101
```

If I use POP, the value I will get from my POP is: 1111 1111 (the last item that was PUSH'ed)

If I use POP again, the value I will get is: 1010 0101 (the last item before that)

If I use POP again and again, the last item I will get from the stack will be the first item that was put in it. For this reason, the stack is referred to as a LIFO data structure. LIFO means "Last in, First out". It is called this because the last item stored onto the stack will be the first item retrieved from it.

Each time you use PUSH, the stack gets bigger. Each time you use POP, the stack gets smaller. PUSH will always put data on top of the stack. POP will always take the top-most data off of the stack, retrieving the data.

The only question that should remain then is this: How does PUSH and POP know what memory address is the top of the stack? Well, since we are talking about needing to track a memory address, what do you imagine we need? A pointer!

This is the purpose of the Stack Pointer. The Stack Pointer contains the memory address of the top of the stack.

Lesson 1.7 : How a function call works

Go through this lesson slowly. It is more intense than most. Take your time. Let me know if any part of this is unclear.

Remember, this is only an introduction to a rather complex subject. Do not worry if some of it doesn't make sense, we will go over it in greater detail later in the course. Try to get as much as you can from this lesson. Later in the course it will be made more clear.

In the last two lessons I have explained that the purpose of the stack is in part a way for functions to receive parameters. In this lesson I want to explain more about the mechanics of this process.

One thing you should already realize is that a function call has to know where to return when it is done. This may sound simple, but it ceases to be so simple when you consider that any function can call another function from anywhere in your program.

How therefore does a function know where to return? Well, the answer is that when you first call a function, the Instruction Pointer (the register that stores which instruction to execute next) is placed onto the stack. This Instruction Pointer can be used to know where we were in the program flow prior to the function call.

I am going to write out a small part of a C program using "line numbers", and I want you to imagine that each line number corresponds to some memory address where the machine code is located.

Figure (a)

```
   void main(void) {
1      int height = 5;
2      int width = 2;
3
4      calculate_area(height, width);
5
   }

   int calculate_area(int height, int width) {
7
8      return height*width;
9
   }
```

Do not be concerned with the numbers I chose. I just needed a unique way to label each line that is important to this lesson.

When we start executing the main() function, we go through lines 1, 2, and 3. At this point, C understands that we want to CALL a function. What happens at this point?

Keep in mind that we need to achieve 3 things:

1. We need to save the location of the memory address we will return to when the function is done. This means we need to save the Instruction Pointer.
2. We need to store the variables height, and width onto the stack so the function can see and use them.
3. We need to "goto" the function itself.

There is more that happens as part of this process, but this is all you should be concerned about at this stage.

Now at line #3, we will basically execute machine code instructions similar to this:

```
push width       ; Store width onto the stack using PUSH
push height      ; Store height onto the stack using PUSH

push eip         ; Store the Instruction Pointer (called eip) onto the stack. This is done as part of "call" below.
call calculate_area    ; Call the actual function
```

This is assembly language, but it translates directly to machine code.

[Edit: One note about the above code, in reality the CALL statement itself automatically pushes the instruction pointer onto the stack. I illustrated that here so you could see the process more clearly, but no one writing actual assembly code would write "push eip" as an instruction. With function parameters however, you would have to push them manually.]

Now you should understand that our stack will now look like this with respect to the function parameters:

```
... top of stack where new elements can go ...
height
width
```

Notice that height is at the top of the stack even though width was pushed first. Also, keep in mind the instruction pointer has also been pushed to the stack as part of the "call" instruction.

Now you can see how exactly parameters are sent to a function. Whether a pointer, or a variable, etc. the calling function, such as main(), places the parameters onto the stack. Then the function reads them off of the stack in reverse order to how they were stored.

Now you should be able to clearly see why you must specify how many parameters a function will take as well as their data types. A function needs to know how many items to read from the stack and how big each item is.

Were this to be done incorrectly, a function could take off "too much" or "too little" from the stack and thus absolutely break everything in the program. This is why C forces you to precisely define function parameters.

Now, once we get to line 9 in Figure (a), the function has finished executing. At around this stage it needs to have a return value.

How return values work is the subject of the next lesson.

Lesson 1.8 : How function return values work

I split this into two lessons to make it easier to grasp.

Remember as I said in the last lesson, this is just a general introduction to how this process works. Do not worry if you do not fully understand it at the end of this lesson, just try to get as much as you can. Later in the course it will be made more clear.

I think it is important to preface this lesson by saying this: This lesson focuses on how a function truly returns a value at the machine code level. C and other languages are not always so "pure". There are many complexities to how a programming language such as C makes it possible to "return" data that is larger than what is allowed at the machine code levels.

These "returns" however are not true returns, but involve creative memory copying and pointer usage by the compiler. In the end, these operations lead to vastly slower and intensive processes than the type of return illustrated in this lesson.

We will get into the specifics of those processes later in the course.

A function typically returns a value using a CPU register named EAX.

EAX is just another register, like the "Instruction Pointer" or the "Stack Pointer".

Registers are simply places on the CPU that data can be stored, sort of like an on-chip memory. Each register is only capable of holding a small amount of information, limited to the chip architecture. For example, a 32 bit chip can hold a 32 bit data element in a register (typically).

No register is very large. However, what is stored inside a register can be acted on by the CPU faster than anything stored in memory. This is because when the data is in a CPU register, that data is literally on the CPU itself.

You will learn more about microprocessor architecture later in the course. However, for now you should know this. The return value of a function is typically stored in the EAX register.

This means that before a function returns control to whatever called it, it must simply make sure that the value of EAX will be what that function intended to be the return value. If for example a function needed to return 5 as a return value, the function stores the value 5 into the EAX register.

This is what will typically happen at the end of a function call:

1. The function will finish doing what it was designed to do.
2. The function will store a return value into EAX
3. The function will return control back to whatever called it.
4. The caller will then read EAX and understand that as the return value.

Now it should be very clear to you why a function returns one value. The EAX register is only designed to hold one value. Also, it should be clear to you why a function cannot return a value larger than X-bits (depending on your chip architecture).

You can however return large data structures (arrays, structs, etc) by using a pointer. That is because a pointer, a memory address, will fit into the EAX register just fine.

Because I showed you the assembly language/machine code instructions for push and pop, I might as well show you the assembly language for storing a value into EAX. It is not very complex.

```
mov eax,0     ; This will "return 0". It is short for "Move 0 into EAX"
```

All of this work that I have described is done for you behind the scenes by C.

The goal of these last lessons is for you to understand how in general parameters are sent to functions (the stack), how the function knows where to return (IP register gets put on the stack), and how in general return values work (the eax register).

Lesson 1.9 : Introducing Casts

In this lesson I am going to introduce you to a concept called type casting. This refers to the process of treating some data of one data type as though it were another data type.

To understand this process, consider this: What does an array of 10 characters look like in memory? Well, if each character is one byte long, then an array of ten characters would be 10 bytes in size.

Now that covers how big it is. What does it look like? The answer: Whatever happens to be in those ten bytes. Any possible sequence of 1s and 0s is fine. It doesn't matter. An array of ten characters has absolutely no requirement concerning how the 1s and 0s inside those 10 bytes look.

This is important to understand. Any memory range that is 10 bytes long can be understood as being 10 characters regardless of what is actually contained in the memory. Similarly, any 4 bytes of memory can be considered to be an integer regardless of what is actually contained in the memory.

A data type is only a description of how to understand a range of memory. The same 9 bytes of memory that can be a tic tac toe board in our earlier lesson could be nine ASCII characters. The same 90 bytes of memory that can be ten tic-tac-toe boards could just as easily be 90 bytes of some sound file.

Below I am demonstrating how an unsigned short int (assume 2 bytes) and an array of characters 2 bytes long sees the same data:

```
  Figure (a)

   0100 0001    :    0100 0010

  unsigned short int = 16,706
  ____________/ _____________
/
 ______/          ______/
char[0] = 'A'      char[1] = 'B'
```

It is the same memory: 0100000101000010

This same sequence of 1s and 0s can mean 'AB' or it can mean 16,706 and it could also mean two squares of a tic-tac-toe board. Anything is possible. There are no rules for how a sequence of 1s and 0s are to be interpreted.

If I create an array of two characters in C, all that happens is C chooses a place in memory for those two characters to live. nothing changes in memory. Whatever was there before, will still be there.

If you look at that statement another way: Whatever was there before, will become understood as being two characters.

If I create an unsigned short int in C, it works the same way. Nothing is actually changed in memory. C just chooses a location in memory for the unsigned short int to live. Whatever happened to be at that address remains at that address. However, whatever was at that address is now understood to be an integer.

With this in mind, why then could I not transform the two characters 'A', and 'B' into some integer number? Similarly, why can I not take some integer value and convert it to several characters?

The answer is: You can.

Whenever you tell C or any programming language to treat a value of one data type as though it was a value of a different data type, this is known as type casting. It simply means that you are wanting to take a sequence of 1s and 0s that can be interpreted one way, and interpret it a different way instead.

You could do this as many times as you want. You can even have the same data in memory being used in your programs in multiple ways simultaneously. You could have a printf() statement which says AB: 16706 using the same sequence of memory for both the character interpretation, and the integer interpretation.

[Note: The way this actually works is slightly different, but we will get to that soon enough.]

There are many powerful uses of this which we will go over in future lessons. You saw one example of this in an earlier lesson when I created a char * pointer by casting it from a data structure pointer in order to go byte-by-byte through my data structure to show that it looked like this in memory:

```
Reddit$Programming$Classes$
```

In the following lessons there are two kinds of casts we will look at: value casts, and pointer casts.

A value cast refers to when we cast an actual sequence of 1s and 0s, a value, something stored in some variable. An example of this is taking an integer value and converting it into ASCII characters such as in Figure (a).

A pointer cast refers to when we take a pointer of one data type and we tell it to continue to point where it is pointing, but to treat what it is pointing to like something else.

Think of a data type in general as a pair of colored glasses. If you put on red tinted glasses, everything you look at is red. However, if you take off the red glasses and put on green tinted glasses, you are still looking at the same data, but now it has turned green.

Casting can be thought of as switching glasses from one color tint to another. You will still be looking at exactly the same data, but you will be seeing it as something entirely different.

To wrap up this lesson: Any sequence of bytes can be understood as anything you want, even if you have already told C to treat it as something else. The process of understanding the same data but as a different data type is known as type casting.

That is all for tonight. I will do more tomorrow. I didn't have very much time to get to questions but tomorrow I expect to catch up.

Lesson 1.10 : A new way to understand memory and data types

In the last lesson I explained that using type casting it is possible to cause data which was created as one data type to be considered as though it were another data type.

In this lesson I want to change slightly how you look at variables, arrays, structures, and pointers.

Whenever you create something of any data type, all that happens is the size of that item in bytes is allocated by C, and that memory is now available for your use.

Let's imagine that we have a chunk of memory that looks like this:

```
Figure (a) : Before

...
0110 0011
1111 0111
1001 1100
0011 0000
0001 0000
...
```

Imagine that this represents the total memory that is free for use for the program we are writing. If I create a variable, like this:

```
char my_char;
```

C is going to find some spot of that available memory, and give it to my_char. It will not change what is at that memory address. The same range of memory will look exactly as it did before. It is just that one byte of that memory has been reserved for my_char.

If I write this:

```
int height;
```

Similarly, four bytes (typically) have been reserved for height. These four bytes will still contain whatever was in them. Now we have used up all 5 bytes. How will the range of memory look after the char my_char and int height instructions? Exactly as it did before!

```
Figure (b) : After

...
0110 0011
1111 0111
1001 1100
0011 0000
```

```
0001 0000
...
```

The act of creating a variable is only the act of reserving some chunk of memory for a data type, pointer, array, structure, etc. Anything you create without initializing it will have whatever value was already at those bytes.

Now suppose you want to create an array of ten characters. In order to do this, all you need are ten bytes. Having the ten bytes is the same thing as having your array.

In other words, here I am creating my array of ten characters:

```
char *my_characters = malloc(10);
```

I just created an array of 10 characters. Why? Because 10 bytes is ten characters, if I choose to look at those 10 bytes in that way.

If I want to create an array of five integers, I can do this:

```
int *my_integers = malloc(5 * sizeof(int) );
```

Most likely 20, but at any rate I now have my array of five integers. Why? Because 20 bytes is five integers if I choose to see it that way.

What you need to understand from this lesson is that having the correct sized chunk of memory for your data type is the same thing as having the data type itself. The data type just describes how you intend to understand the memory you have allocated for it.

Suppose I want a 3x3x3 array of characters. I can simply allocate 27 bytes, and I have my array. 27 bytes is an array of 3x3x3 characters if I choose to see it as such. Therefore, this command:

```
char *my_array = malloc(27);
```

This gives me a 3x3x3 array. What if I want an array that is 3x9 ? How about 9x3 ? Same thing. The above line of code will generate any possible data type that is designed to be 27 bytes in size.

After this lesson, you should never wonder "I need an array of 5 structures, how do I do that?" Answer - you simply create a pointer of your structure type and you point it to a memory address that has (5 * size_of_structure) bytes.

UNIT 2 : Casts, Pointers, and Arrays

Lesson 2.1 : Using Casts with Pointers (Part One)

In the last lesson I showed you that any range of memory can be used for any purpose that can fit inside that memory. In this next series of lessons I want to illustrate this by actually re-using the same ten bytes of memory in this way. I believe that doing this will give you a greater understanding for how casts, pointers, and data types in general work.

In this lesson I am going to show you how to write a program which allocates 10 bytes of space and then uses that ten bytes of space as:

1. ten characters
2. two integers (which leaves unused space)
3. A 2x5 array of text
4. A data structure having two strings of text, one 4 chars and one 6 chars (including NUL)

First, let's allocate our memory:

```
char *main_pointer = malloc(10);
```

There. Now main_pointer is a pointer which is looking at the first byte of a ten-byte memory space that we have allocated.

First, lets set up ten characters, and print the text using printf():

I am going to use a mixture of methods here. They are all doing exactly what we want.

```
*(main_pointer + 0) = 'A';
*(main_pointer + 1) = 'B';

main_pointer[2] = 'C';
main_pointer[3] = 'D';

strcpy( (main_pointer + 4), "EFGHI");
```

Our final string will look like this: "ABCDEFGHI" (with a NUL) at the end.

Notice that using array indexing or pointer indexing makes no difference. Notice that when I use my pointer as an array, I do not put a dereference symbol (meaning a * character) in front of it. An array is the pointer itself.

Let's print it:

```
printf("Our ten bytes of memory contain the string: %s  ",main_pointer);
```

Output:

```
Our ten bytes of memory contain the string: ABCDEFGHI
```

Now, I am not going to create a new ten bytes to work with. Rather, I am going to change the way C understands the ten bytes we already have. In the compiler I am using, an integer is 4 bytes.

I can only fit two 4-byte integers in a 10-byte space. So lets consider how this will work:

```
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 <-- ten bytes
```

Before we can choose to use these bytes for integers, we have to decide where they will go. I could really do this any way I want. They do not have to be directly connected. For example, I could have my two integers occupy:

```
B0 B1 B2 B3    and    B6 B7 B8 B9
```

There is only one rule I must remember. I cannot mix the order of the bytes. For example, I could not do: B2 B4 B1 B6, nor could I do: B1 B2 B3 B7.

Now, how do I get C to read my ten bytes as two integers? First of all, we need to decide how they will be stored in the ten bytes. I propose that we use the example above where we use bytes 0 1 2 3 and 6 7 8 9.

What I really need to know is the starting point of each integer. In this case byte #0 and byte #6. I know that there is room to store my integers correctly.

Now, let's set the integers to some value, and use printf() to display them. Before we do that however, let's see what they already are. How can we do that?

First, consider this code and the result. Remember that main_pointer is the pointer we already created and is pointing to the ten bytes in memory which presently have: "ABCDEFGHI".

```
printf("The integer at byte #0 is set to: %d  ", (int)*main_pointer);
printf("The integer at byte #6 is set to: %d  ", (int) *(main_pointer + 6));
```

Output:

```
The integer at byte #0 is set to: 65
The integer at byte #6 is set to: 71
```

Notice that 65 is 'A' (decimal, not hex), and 71 is 'G'. So we can see here that by putting (int) in front of the *our_pointer, we are telling printf() to treat that character as if it were an integer.

However, that only treats one byte as an integer. We want to treat four bytes as an integer. How can we do that?

You see, there is a problem with our pointer. Our pointer is designed to look at memory in char-sized chunks. We can keep using our ten bytes of memory, but we really should consider a different kind of pointer: one that can read memory in int-sized chunks.

Let's create it:

```
int *int_pointer;
```

I haven't given it a memory address yet. What memory address do we want to give it? We want to give it the memory address of our ten bytes that we have already allocated. What is that memory address? It is: main_pointer. Remember we already have a pointer that contains the right memory address. Therefore, we need to set it to that:

```
int *int_pointer = main_pointer;
```

One small problem. an int * pointer expects to look at memory in int-sized chunks. A char * pointer expects to look at memory in char-sized chunks. We cannot assign the memory address of a char * pointer that easily without our compiler complaining.

Our compiler is concerned that we do not really know what we are doing, that we are not aware we are trying to take the memory address inside a char * pointer and put that memory address into a int * pointer. How can we tell our compiler that we intend to do this? The same way we told printf() that we intended to treat a character as an int. Observe:

```
int *int_pointer = (int *) main_pointer;
```

By simply putting (int *) we have stated that we are assigning the new pointer int_pointer the same memory address as main_pointer, but that we want to treat our new pointer as an (int *) instead of a (char *).

What have we just done? We have created a second pointer which points to our ten byte memory space. Our first pointer is already pointing to this same exact spot in memory. How are the two pointers different? They are different in mainly two ways:

1. How they understand dereferencing
2. How they understand pointer arithmetic

Remember that dereferencing means that you put a * character in front of a pointer to get "what is at" the memory address contained in the pointer. As you saw in our printf() example, a char * pointer understands dereferencing as "one byte".

Therefore, if I say *main_pointer with or without a type cast it will still mean "one byte". The same is true for any offset I add to that pointer. Therefore:

```
*main_pointer        <-- one byte
main_pointer[6]      <-- one byte
*(main_pointer + 4)  <-- one byte
```

No matter what location in memory I dereference with this pointer, I am only going to get one byte.

The second way the two pointers are different concerns pointer arithmetic. If I say: main_pointer + 1: It means to change the memory address by only one byte. Why one byte? Because that is how big a char is. However with our new pointer if I were to say the same thing: int_pointer + 1 The memory address will be four bytes different than it was. If I added 2 then our int_pointer would point 8 bytes away (4 * 2).

What you should understand from the above text is that in order to have a true four-byte integer out of our 10-byte data, I need to create an integer pointer. I create an integer pointer by type casting the (int *) data type and using the same memory address as was already in the other pointer.

Now consider this:

```
int *int_pointer = (int *) main_pointer;    // <-- create the pointer int_pointer using the same memory address
```

Now let's set values using this pointer:

```
*(int_pointer + 0) = 53200;
*(int_pointer + 1) = 32000;
```

Remember of course that int_pointer + 0 is the same as int_pointer. However, writing it this way makes it clearer. Notice I chose two numbers that are too big to fit inside a byte. Let's see if this works:

```
printf("The first integer is set to: %d  ", *int_pointer);
printf("The second integer is set to: %d  ", *(int_pointer + 1));
```

Output:

```
The first integer is set to: 53200
The second integer is set to: 32000
```

This proves we are using more than just one byte of our allocated memory. Notice that we could not do this with a char * pointer, and that is why we have to use an int * pointer.

Notice we did not have to use any type cast in our printf() statement. Because int_pointer is already set to point at the data type int, then any time we use *int_pointer it will be dereferenced as a true 4-byte integer.

What bytes am I using in the printf() statement? Let's consider this. I have 10 total bytes. I pointed int_pointer at the first byte. I stored the integer value "53,200" into those first four bytes. Then at a position four bytes away from where I started, I stored the value "32,000" into those four bytes. Therefore:

```
<  B0 B1 B2 B3  >   < B4 B5 B6 B7 >      <--- Here are my two integers in the above printf()
```

Now keep something in mind, I said we would use bytes #0 and bytes #6. In the above example I actually used byte #0 and byte #4. This is because when our int_pointer is used without an offset, it is pointing to byte #0. When we add a +1 as an offset, that will cause it to point at byte #4.

If adding to an int pointer can only be done in increments of four, how can I position the pointer at byte #7 ?

That is the subject of the next lesson.

Lesson 2.2 : Using Casts with Pointers (Part Two)

In this lesson I am doing something highly unorthodox for the purpose of better explaining and illustrating pointers and casts. I therefore feel it is necessary to preface this lesson. This is purely for illustrative purposes. I want you to see that bytes are just bytes, and pointers are just pointers. The purpose of this lesson is to help you to understand the function of pointers and casts better.

This is poor programming practice to actually do what is shown in this lesson in any real program. Just remember as you read this lesson, this is how pointers and casts work, but not the best or most efficient way to actually use them.

In the last lesson, I showed you how we could use 10 bytes of memory as characters, then as integers. The one difference in the last lesson from what we had planned is that we created two integers that were right next to each other, as opposed to creating one integer at byte 0, and another integer at byte #6 (which is actually a rather dumb thing to do, but I am showing this to you so that you understand pointers better). In this lesson, we are going to continue but this time with the original plan.

So, to be clear, what I want is this:

```
< B0 B1 B2 B3  > < B6 B7 B8 B9 >   <--- two 4-byte integers
```

B4 and B5 will be "wasted space".

As you recall in the last lesson, in order to be able to use

```
char *main_pointer = malloc(10);
```

Then we created an integer pointer, like this:

```
int *int_pointer = (int *) main_pointer;
```

This created a pointer of type int * (pointer to integer. That is what the * means). It contains the memory address of the start of our 10 bytes. This means that we have bytes:

```
int *int_pointer2 = (int *) (main_pointer + 6);
```

Notice what I did here. (main_pointer + 6) is a memory address. That memory address corresponds to byte #6. We are type casting it using (int *). Our type cast allows us to assign the new value (after the type cast) to a new pointer we created called int_pointer2.

Keep in mind at this stage we have three total pointers. main_pointer is our primary pointer which is pointing at byte #0 of a ten byte memory space. int_pointer is an integer pointer which points at the same address as main_pointer, except instead of expecting to see char, it expects to see int. Finally, int_pointer2 is our third pointer, just like int_pointer except it points at byte #6.

It is perfectly ok to have many pointers all looking at the same range of memory. it is perfectly ok if multiple pointers contain the same memory address, or act on the same data. You must however remember that if two pointers contain the same memory address, and the value at that memory address changes, that change will be visible by all pointers which point to that memory address.

Alright, so now lets consider the result of this code:

```
int *int_pointer1 = ( int *) (main_pointer + 0);
int *int_pointer2 = ( int *) (main_pointer + 6);

*(int_pointer1 + 0) = 53200;
*(int_pointer2 + 0) = 32000;
```

Now, if I were to printf() both integers, I would see the values "53200" and "32000" just as we would expect. Moreover, if I were to look at each byte one at a time, I would see that the first integer is occupying bytes

Lesson 2.3 : Introducing Arrays of Pointers (Part One)

This lesson may appear a bit scary at first, but it is only because I use the word pointer about a thousand times. Take it slowly, and read through the whole lesson. This is often a difficult topic, and I have done my best to explain it thoroughly. Please let me know if any of this is unclear.

In the last lesson I showed you a situation where we had two related pointers to which we gave the names: int_pointer1 and int_pointer2. It should be apparent that if you have a situation that you are giving such names to variables or pointers in general, you should consider an array instead.

Why? Let's consider I have 5 such pointers:

```
int *int_pointer1 = ...
int *int_pointer2 = ...
int *int_pointer3 = ...
int *int_pointer4 = ...
int *int_pointer5 = ...
```

Now, how would I be able to create a for loop, or any kind of loop, that could use each one? I could never do something like this:

```
for (i = 1; i <= 5; i++) {
    ... int_pointer i    (ex: int_pointer1, int_pointer2, etc.)
}
```

There is simply no way to do this. C will not be able to take a partial variable name like int_pointer and figure out how to add an extra number at the end as part of a loop. On the other hand, if I have an array like this:

```
for (i = 1; i <= 5; i++) {
    ... int_pointer[i] ...    (ex: int_pointer[1], int_pointer[2],etc.)
}
```

Now this is doable. I can easily put a number inside of brackets as an array element. So to be clear, my goal in this lesson is to figure out a way that I can use int_pointer1 and int_pointer2 as if they were elements of an array.

First, realize that they are both pointers. Therefore, I need to create an array of pointers. Without evaluating the exact syntax just yet, let's imagine how this would work. Here is a description of the array of pointers I plan to create:

int_pointer[1] = Element [1] will be a pointer to some integer in memory.
int_pointer[2] = Element [2] will be a pointer to a different integer in memory.

That is our goal. So, let's begin.

First of all, think of this process the same as you would any other array. If we want to create an array of three pointers, then we need somewhere in memory to put them. We have been spending a lot of time using the malloc() function, so I want to do the same here.

Remember from previous lessons that having the memory is the same thing as having the data type that will fit in that chunk of memory. For example, having 3 bytes of memory is the same thing as having an array of three characters.

From this, you should be able to figure out that we need to malloc() a portion of memory large enough to fit three pointers to type int. Why three and not two? Because it will be more instructive for this lesson. How do we know how large a space to allocate?

Well, on a 32 bit system, it is likely that each pointer is 32 bits (4 bytes) in size. However, this is not true for all systems. Therefore: We cannot use malloc() with the number 4 just because we think (or know) that the size of bytes we need is 4. That may be true on our system, but not others. In other words, we cannot ever trust that we know the size of any data type - with one exception: char.

Let me say that again, as this is very important: Whenever you allocate or write any code which depends on a size of a given data type, you must use sizeof() to get the correct value. You should never assume ints are 4 bytes, or pointers are 4 bytes, etc. Always use sizeof(). There is one exception. A char data type is always going to be one byte in size.

Now let's continue.

An array of pointers will look something like this in memory:

```
['Pointer #1']['Pointer #2']['Pointer #3']...
```

For this lesson, assume each "block" of the above represents 4 bytes of memory. In other words, we are assuming that a pointer takes up 4 bytes. This means that if we were to give memory addresses to each of these pointers, it would be something like this:

```
B0 : ['Pointer #1']
B4 : ['Pointer #2']
B8 : ['Pointer #3']
```

Notice that each pointer starts 4 bytes later than the last one started. This is the very definition of an array. An array is a repeating collection of the same data type stored sequentially in memory one element immediately after the other.

How do you work with any array? You use a pointer. Therefore, if I am working with an array of pointers, I will need a pointer.

But a pointer to what? What will our pointer be pointing to? Will it be pointing to integers? Characters?... No, it will be pointing to.. pointers!

Why? Because each pointer is to be stored in memory sequentially at B0, B4, B8, etc. That is the very definition of an array of pointers, which is what we want to create. We therefore need some pointer which will work like this:

```
pointer[0] = "point to B0";
pointer[1] = "point to B4";
pointer[2] = "point to B8";
```

Do not worry about syntax right now. The above is a description of what we want, not actual syntax. Just understand that we need a pointer which can point to B0, then B4, then B8. However, remember that array indexing is a shortcut for pointer offsets. Therefore, the above 3 lines could also be written like this.

```
*(pointer + 0) = "point to B0"
*(pointer + 1) = "point to B4"
*(pointer + 2) = "point to B8"
```

Why does adding one to our pointer get us to B4? Because the idea of pointer arithmetic is that you always increment by the size of the data type you are pointing to. Now, what kind of data type will we be pointing to? A pointer!

We are assuming in this lesson that a pointer is 4 bytes in size. So the first thing we need to consider is that we need to create a pointer of the data type pointer.

How do we create a pointer in general? Like this:

```
data_type *pointer ...
```

What is our data type if we want an array of int pointers? Our data type is (int *) which means "a pointer to an integer".

Therefore, what we want will be similar to this:

```
(int *) *pointer ...
```

What does this mean? It means "create a pointer called *pointer". Of the data type "pointer to int".

We are creating something called *pointer which will contain a memory address. The memory address it will contain will be the memory address where a pointer resides.

The syntax I showed you above is nearly correct. Let's take out the parentheses from (int *) for the data type, and see what we get:

```
(int *)  *pointer_name = ...
```

Becomes:

```
int *   *pointer_name = ...
```

This is correct. We are saying to create a pointer called "pointer_name" which will point to the data type: int * which means "pointer to an integer".

So, we know that pointer_name is a pointer. We know therefore that it will contain memory addresses. What kind of memory addresses will it point to? A memory address that has a pointer.

Let's go back to our earlier example:

```
B0 : ['Pointer to integer #1']
B4 : ['Pointer to integer #2']
B8 : ['Pointer to integer #3']
```

Let's consider our new pointer called pointer_name we just created. Can it point to B0? Yes. Why? Because B0 is a memory address which stores an integer pointer. It can point to B4 or B8 for the same reason. It can point to any memory address which contains a pointer to an integer.

Now, whenever we have done this before, we have used malloc() based on the size of the final array we want. This is no different, so lets create the actual array now:

```
int *    *ptr_array = malloc(3*sizeof(int *));
```

int * is our data type. The second * indicates we are creating something that is a pointer to our data type (which is int *). Finally, ptr_array is the name we are giving our pointer.

Finally, how big of a space do we get in memory to use it? Assuming that sizeof(int *) returns 4 bytes, we would get 12 bytes. Remember that this sizeof() can be thought of as "size of a pointer", since all pointers will be the same size.

Now, how do we use it? Well, lets evaluate some facts.

ptr_array is a pointer. It points to B0 of the 12 bytes we just allocated. So therefore:

```
ptr_array = Byte #0 (The actual memory address)
*ptr_array = *B0 (What is *at* Byte #0)
```

So what is *ptr_array ? It will be the actual pointer that resides at Byte #0.

How we can use the pointer at Byte #0 is the subject of the next lesson.

Lesson 2.4 : Introducing Arrays of Pointers (Part Two)

Before I continue the last lesson, I want to spend some time talking about how to think about pointers in general.

It is possible in C to have, for example: "a pointer to a pointer to a pointer to an integer." If you assume that I or any skilled programmer can follow that in our minds, you are wrong.

When you look at a string of text like this:

```
"Hello Reddit"
```

Do you try to understand the individual ASCII of each letter? No, of course not. Could you imagine how hard it would be to learn programming if you had to? It is enough to know that each letter has an individual ASCII value, and that you can find it by zeroing in on one such letter.

Pointers are the same way. Any pointer at all no matter how complex, whether it is a pointer to a single character or a pointer to... 10 levels deep of pointers to pointers, it is still just a pointer.

A pointer is just a variable of the data type memory address.

It is an exercise in futility to try and understand every detail about a chain of complex pointers. Further, if you have to, then you are doing something wrong in your program. A good program should never require that you worry about all the fine details about every variable, pointer, and data element that is in the program.

That is simply impossible for anyone to do.

I am going to present "pointers to pointers" as data types below. First I am going to do it wrong. Then I am going to do it right.

The wrong way to understand pointer data types:

```
int * = a pointer to an integer.
int ** = a pointer to a pointer to an integer.
int *** = a pointer to a pointer to a pointer to an integer.
```

Confused yet? Good! It means you are human.

Now, the right way to look at pointer data types:

```
int * = a pointer to an integer.
```

So far so good... What about int ** ?

Do not think of it as int **. Think of it as "two star int". A two star int is a pointer to a one star int.

What about int *** ? Well, it is a three star int. Therefore, it points to a two star int data type. What is a two star int data type? It doesn't matter. It is some valid data type to which it can point. The exact specifics of that data type are understandable if you look at it closely. That is all that you need.

The idea that there is a "two star int" data type is massively easier to understand than that there is some "(int **)" data type.

Don't worry if you are a bit confused. It will all be crystal clear in a minute.

Now, let's demonstrate this. Do you understand what this is:

```
int ******some_pointer = ?
```

If you start by saying "Well it is a pointer to a poi..." you are doing it wrong. Count the * characters. There are six. This is therefore a six star int. Subtract one. That is the data type we are pointing to. In other words, this points to a pointer of the data type "five star int". Any six star int is a pointer to a five star int.

What is a "five star int"? It doesn't matter. It is enough to see this and realize that it is some level of pointers. More than that is utterly unnecessary. Just realize that this is some data type. That our pointer can therefore point to variables of this data type, and that is all you need to know.

Yes, it really is ok to say: "This is a pointer to a five star int." When you mentally evaluate code like this, that is what you are really doing. The meaning of five star int is something that becomes apparent once you think about it. It is not something you need to know when you first look at the code.

With this in mind, let's write out a very simple program to demonstrate multiple levels of pointers:

```
int height = 5;

int *one_star_int = &height;

int **two_star_int = &one_star_int;

int ***three_star_int = &two_star_int;

int ****four_star_int = &three_star_int;

printf("The actual value of height is: %d  ", ****four_star_int);
```

Was that hard? Notice something interesting. When we de-reference the final pointer, the "four star int", we used four stars to do it. If we were de-referencing a "three star int" to get the final value, we would have used three stars.

Here are printf() for each level of int pointer we did:

```
printf("The actual value of height is: %d  ", ****four_star_int);
printf("The actual value of height is: %d  ", ***three_star_int);
printf("The actual value of height is: %d  ", **two_star_int);
printf("The actual value of height is: %d  ", *one_star_int);
```

How do we know that each of these are going to give us the original value ? Because the number of * characters corresponds to how many levels deep the pointer is. It is as simple as that.

It would be a great idea if you took the material in this lesson and experimented with it so that you can better understand multiple levels of pointers.

Lesson 2.5 : A quick review on Casting

I realize that casting is a confusing topic at first, and I have created this quick review to help anyone who may be struggling with the concept. Do not worry if even after reading this the entire concept is not clear to you. Everything will be covered in greater detail later.

In earlier lessons, here is how we set up ten bytes to work with:

```
char *main_pointer = malloc(10);
```

Now main_pointer is a char pointer that is pointing to Byte 0 of our 10 byte working space. Now, let's create an integer pointer, and cause it also to point at Byte 0.

```
int *int_pointer = (int *) main_pointer;
```

This may be unclear to you. Why are we writing (int *) here?

In this case, (int *) is a cast. Our cast works by simply saying that we intend to use main_pointer as though it were an int * pointer, for the purpose of this assignment.

You can see that main_pointer is a pointer to a char while int_pointer is a pointer to an int. The two data types are not the same and are therefore not compatible with each other. We need a cast in order to make this assignment work.

In general, you use casts when something requires one data type and you have the correct data, but it is of the wrong data type.

When I type:

```
(int *) main_pointer
```

I am attempting to produce data of type (int *) by using main_pointer as input.

The result of this casting operation can then be assigned to our int_pointer we created.

When I write this code:

```
int *int_pointer = (int *) main_pointer;
```

What happens is something similar to this:

1. Transform main_pointer to an (int *) data type.
2. Take the result of this transformation, and assign it to int_pointer.
3. This transformation does not actually change main_pointer.

Remember that main_pointer is not actually changed. The cast operation takes place without changing anything, it just creates something new which can be used to assign a value to int_pointer.

Casting is used when we need to take something of one data type and use it as another.

When it comes to pointers, remember that all pointers differ only by the data type they point to. The memory address where an int begins is the same kind of memory address where a char begins.

Therefore, if you have a pointer which points to the right memory address, but the wrong data type, it is very easy to change that. All you have to do is put the cast of the correct data type in front of the pointer, and you instantly have a new pointer of the correct data type.

Here are some examples:

I have a pointer called my_int_pointer which was designed to point to integers. It points to the memory address we will call B4 in some allocated memory. I need to look at B4, but as a character rather than as an integer. All I need to type is this:

```
char *my_char_pointer = (char *) my_int_pointer;
```

Now, suppose I have the exact opposite. I have a pointer called my_char_pointer which points to the correct address where an integer is stored in memory. I need to see that integer, therefore I can create a new pointer called my_int_pointer like this:

```
int *my_int_pointer = (int *) my_char_pointer;
```

And that is all there is to it. Just remember that you use a cast in order to take something of one data type and change it so it can be used as a different data type altogether. Also remember, the thing being casted is not actually changed.

We will cover casts in greater detail later in the course.

If you are unsure whether you understand this material well enough to proceed, consider this code:

```
int *int_pointer = (int *) some_other_pointer;
```

The purpose is to create an integer pointer called int_pointer using a pointer to a different data type. The (int *) is used to specify that we want to convert some_other_pointer to an integer pointer. Remember, some_other_pointer is not actually changed.

As long as you understand the above paragraph and you understand the purpose of the above code, then you can safely proceed to the next lesson. We will cover more details later.

Lesson 2.6 : Arrays of Pointers Continued (Part One)

This is a complex topic, so I am splitting it into two lessons to make it easier to grasp.

In this lesson we are going to continue the project we started earlier, when I showed you how to use 10 bytes of memory simultaneously for various purposes.

Recall that our ten bytes look like this:

```
B0 B1 B2 B3 B4 B5 B6 B7 B8 B9
```

We allocate our ten bytes like this:

```
char *main_pointer = malloc(10);
```

That gives us ten bytes to work with, and our pointer main_pointer is now pointing to the first byte (Byte #0, or B0) of our ten bytes of working space.

Now, we need to create two integer pointers. We will point one of them to B0 and the other to B6.

Remember, this lesson is just a demonstration and is only for illustrative purposes. I just want you to see that this can be done, and how it can be done.

Now, pointing an integer pointer to the start of our ten byte array is easy. We already have a pointer that points there called main_pointer. Since it already points to the right address, the only thing we need to do is create an int pointer using the main_pointer with a cast.

```
int *int_pointer1 = (int *) main_pointer;
```

Based on the last lesson, you should understand what this does. Now, we have one integer pointer called int_pointer1 which is pointing to B0 in our ten byte memory space.

Now let's create a second pointer, which will point at B6:

```
int *int_pointer2 = (int *) (main_pointer + 6);
```

Remember that main_pointer + 6 is just a different memory address. We are doing the same thing as before, only now int_pointer2 will point to B6 instead of B0 in our ten bytes.

So far so good.

Now there is just one problem. These pointer names are not good. It is poor practice to name two related items as int_pointer1 and int_pointer2. Why not just use an array of int pointers? In this lesson I am going to show you how.

First of all, one way to create an array is simply to allocate the memory that the array will require. In this case, we will allocate array space for two integer pointers.

How much space do we need? Well, let's consider how two integer pointers will look like in memory:

```
Horizontal View:
    [Integer Pointer #1][Integer Pointer #2]...

Vertical View:
    B0 : [Integer Pointer #1]
    B4 : [Integer Pointer #2]
```

If we assume that any pointer is 4 bytes in size, then we need 8 bytes of space to store two such pointers. Because we need to allocate 8 bytes, we need the malloc() command. It will work like this:

```
malloc(8);
```

Except, not quite. If you recall from earlier lessons, you should always use sizeof() for any data type so that you are absolutely sure that your program will work for as many computers and compilers as possible. In this case, it happens to be that 8 bytes is correct. That is not guaranteed to be the case all the time. Therefore, we write this instead:

```
malloc(2 * sizeof( int* ) )
```

This will give us the same result. This just means we are allocating 8 bytes of storage space for our array of two pointers. Any time we use malloc() we need a pointer to point at the space we allocated. Well, what kind of pointer do we need?

To answer that, we need to answer this:

What will our pointer point to ? It will point to one star ints. Why? Because we are creating an array that will consist of (int *) pointers.

What do you call a pointer which points to one star ints ? A two star int. Similarly, what do you call a pointer that points to four star ints ? A five star int. Any "higher level" pointer is simply one that points to one level lower.

Here is how we create our two_star_int which will point at our array of int * (one star int) pointers.

```
int **two_star_pointer = malloc(2 * sizeof( int * ) );
```

If you are confused, let me explain a bit more. We are creating an array of int * pointers. In other words, we are creating an array of one star int pointers. Whenever you create any array, you need a pointer that can "point to" elements of that array. In this case, each element of the array will be a one star int. We need something that can point to a one star int. That means, we need a two star int.

If all you have gotten out of this lesson is that we need a two star int to point at our array of one star ints, then you are fine.

Now consider the following:

1. two_star_int is the actual memory address of Byte #0. This will be the start of our eight byte working space.
2. *two_star_int is "what is at" that memory address. What is at that memory address? A one star int.

Lesson 2.7 : Arrays of Pointers Continued (Part Two)

In the last lesson I explained that because we are creating an array of one star int pointers, we need a two star int pointer to point to elements of that array.

Put in more technical terms, because we intend to have an array of (int *) elements, we need to use an (int **) pointer to point at each of those elements.

Why is that? Because the thing that points to a one star int is a two star int.

Now, we created the array already. We did so with this command:

```
int **two_star_pointer = malloc(2 * sizeof( int * ) );
```

Now the only question that remains is, how can we use the array?

Well, with any array we basically need to be able to do the following:

1. We need a way to "set" the value of each element of the array
2. We need a way to "see" the value of each element of the array.

Let's create a couple of simple integers to work with:

```
int height = 5;
int width = 10;
```

Alright, now let's again examine our array in memory:

```
B0 : [Integer Pointer #1]
B4 : [Integer Pointer #2]
```

Now we know that two_star_pointer already contains the memory address for B0. Therefore, logic would tell us that (two star pointer + 1) would refer to the memory address at B4. Why does it add 4? Because our data type is four bytes in size. Pointer arithmetic makes this possible.

Let's re-word the above paragraph:

two_star_pointer is the memory address where a one star pointer lives.

(two_star_pointer + 1) is the memory address where a different one star pointer lives.

Alright, so we just need a way to take these two pointers and assign them a value. What two pointers? The two pointers at B0 and B4. It doesn't matter that we haven't set them yet. They are still there, waiting to be set. That is the nature of allocating any memory.

Remember that if I say: char *some_char_pointer = malloc(10), I have already created my array of ten characters. Each character will be set to whatever just happens to be in the memory I allocated.

It is the same thing here. We already have our two one star int pointers. They just happen to be set to whatever the malloc(8) gave them. Now it is time to change that, by setting them to the proper values we want them to have.

If two_star_pointer is the memory address of a working one star int pointer, then: *two_star_pointer is the one star int pointer itself.

So how can we set that one star int pointer to have some value? Like this:

```
*two_star_pointer = &height;
```

Think about it. *two_star_pointer refers to what is actually at that memory address, which is a live working one star int pointer. Therefore, by saying *two_star_pointer we are no longer talking about our two star pointer. We are talking about what is at that memory address, which happens to be a one star int pointer.

Let me say this again. As soon as you put a * character in front of ANY pointer, you are no longer talking about that pointer. You are now talking about whatever it points to. In this case, by putting a single * character in front of two_star_pointer, we are no longer talking about two_star_pointer. We are talking about what it points to. What does it point to? A one star int.

Therefore, typing this:

```
*two_star_pointer = &height;
```

We are saying "Take the thing that two_star_pointer actually points to, and set that thing to &height. What is that thing? That thing is a one star int pointer. Therefore, we are setting the one star int pointer at B0 to &height;

Look at this in action:

```
int **two_star_pointer = malloc(2 * sizeof( int * ) );

int height = 5;

*two_star_pointer = &height;

printf("The value of height is: %d ", **two_star_pointer);
```

Why did I use two stars in the printf? Because one star would have referred to a one star int. Two stars would refer to a "zero star int", in other words, the integer itself. Whenever you put two stars in front of a two star pointer, you are dereferencing it two layers deep, thus arriving at the original integer value.

So let's review this a bit. Imagine I have something like this:

```
int ***three_star_int = ...
```

Now, consider this:

1. If I write: three_star_int then I am referring to the actual three star int. I am referring to the actual memory address, the pointer itself.
2. If I write: *three_star_int then I am no longer referring to the three star int. I am referring to a two star int. Why? Because that is the thing a three star int points to.
3. If I write: **three_star_int then I am no longer referring to the three star int. I am referring to a one star int.
4. If I write ***three_star_int then I am refering to the thing a one star int points to, In other words: the actual int itself that is being pointed to at the end.

Now let me show you a simple way to understand this. Any time you have a pointer of certain levels deep, and you see it being dereferenced by a certain number of * characters, just subtract the stars from the type of pointer, to get the kind of pointer now referred to.

For example:

```
**three_star_int ...
```

Ok, three minus two (there are two stars) is one. Therefore, **three_star_int is referring to a one star int.

```
****six_star_int ...
```

Six minus four is two. This is therefore referring to a two star int.

```
*two_star_int ...
```

Two minus one is one. This is therefore referring to a one star int. In other words, just a general pointer to an integer.

So with that in mind, we can see why this works:

```
*two_star_pointer = &height;
```

Now, remember that our two_star_pointer points to an array with two elements. How can we set the other element? Like this:

```
*(two_star_pointer + 1) = &width;
```

Ok, now how can we use printf() to display the actual values of height and width? Like this:

```
printf("The values are %d and %d ", **two_star_int, **(two_star_int+ 1) );
```

It should make sense. Why didn't we use one star? Because one star means that we are putting the one star pointer into printf for %d. That would not be correct, we need to put the actual int itself, which would be our two star int dereferenced twice.

Here is a sample program that helps make this clear:

```
int height = 5;
int width = 10;

int **two_star_int = malloc(2 * sizeof(int *) );

*(two_star_int + 0) = &height;
*(two_star_int + 1) = &width;

printf("The values are: %d and %d  ", **two_star_int, **(two_star_int + 1) );
```

If any of this is still unclear, please let me know.

The above code and how it works should make perfect sense to you at this point.

Lesson 2.8 : Arrays of Pointers Continued (Part Three)

In the last lesson I showed you how to create an array of one star pointers. I showed you that you do so by creating a two star pointer that can point to these array elements. The only thing I didn't show you is how to do this using array indexing, rather than pointer offsets.

It is not that different.

Here is the sample program we looked at in the last lesson:

```
int height = 5;
int width = 10;

int **two_star_int = malloc(2 * sizeof(int *) );

*(two_star_int + 0) = &height;
*(two_star_int + 1) = &width;

printf("The values are: %d and %d  ", **two_star_int, **(two_star_int + 1) );
```

Here is this same program, using array indexing:

```
int height = 5;
int width = 10;

int **two_star_int = malloc(2 * sizeof(int *) );

two_star_int[0] = &height;
two_star_int[1] = &width;

printf("The values are: %d and %d  ", *two_star_int[0],*two_star_int[1] );
```

It is exactly the same thing. Remember that array indexing is just another way of using pointer offsets.

```
two_star_int[0] is the same thing as: *(two_star_int + 0)
two_star_int[1] is the same thing as: *(two_star_int + 1)
```

Similarly:

```
*two_star_int[0] is the same thing as: **(two_star_int + 0)
*two_star_int[1] is the same thing as: **(two_star_int + 1)
```

Now that we have this out of the way, we can finally resume our lesson on the ten bytes.

Remember that our goal through all of these lessons is simply to create an integer pointer at B0, and another one at B6. We want our ten bytes of allocated memory to look like this:

```
< B0 B1 B2 B3 > B4 B5 < B6 B7 B8 B9 >
```

The < > shows where we want our integers to be.

B4 and B5 will be "unused space". Now, let's wrap this up with everything we just learned in the last several lessons.

First, lets allocate ten bytes of memory using a char * pointer:

```
char *main_pointer = malloc(10);
```

Now, let's create an array of two integer pointers:

```
int **int_pointer_array = malloc(2 * sizeof( int * ) );
```

The above creates our array of two integer pointers. One will be at byte #0, and the other at byte #4 of this eight-byte working space. Remember that this is not the same as our ten byte working space we also created. We are creating a different eight byte working space to hold two four-byte one star pointers.

We have a ten-byte working space and we have an eight byte working space.

Now, let's set the first integer pointer in our eight byte working space to point at Byte #0 in our ten byte working space.

```
int_pointer_array[0] = (int *) main_pointer;
```

Remember that the above line is the same thing as:

```
*(int_pointer_array + 0) = (int * ) main_pointer;
```

This is no different than when we wrote: int_pointer1 = (int *) main_pointer; earlier in the lessons. It is just that now we are doing this using an array element.

Consider that we wrote this: *(int_pointer_array + 0). This is the same thing as: *int_pointer_array. Why? Because adding 0 can be ignored altogether.

By putting a * in front of int_pointer_array we are no longer talking about int_pointer_array. Now we are referring to the one star int that it points to. Any time you put a * character in front of any pointer you are no longer referring to the pointer, but what it points to.

With this code:

```
*(int_pointer_array + 0) = (int *) main_pointer;
```

OR

```
int_pointer_array[0] = (int *) main_pointer;
```

we are setting the first one star int in our array to the memory address that main_pointer is pointing to. That means we are now pointing to byte #0 of our ten-byte working space.

Now, let's set the second integer pointer in the array similarly:

```
int_pointer_array[1] = (int *) (main_pointer + 6);
```

Remember, this is the same thing as:

```
*(int_pointer_array + 1) = (int *) (main_pointer + 6);
```

There you have it. Now let's assign some value to these two integers:

```
*int_pointer_array[0] = 5;
*int_pointer_array[1] = 15;
```

Why a * in front? Because we want to peel away all the layers of our two star int pointer. The array indexing automatically peels away one layer. A * character in front peels away the second layer. Why does array indexing peel off one layer? Because using array indexing is the same thing as using a * in front of the pointer with an offset.

Remember this:

1. int_pointer_array[0] is the same thing as: *(int_pointer_array + 0).
2. *int_pointer_array[0] is the same thing as: **(int_pointer_array + 0).

Array indexing removes one layer. A * character removes another layer. Therefore, a * combined with array indexing will remove two layers in exactly the same way that ** would remove two layers.

Now finally, we just need a printf() to display the values:

```
printf("The values are %d and %d", *int_pointer_array[0],*int_pointer_array[1]);
```

In the next lesson I will show you the final working program of everything you just learned. You will be able to see in action how we were able to use and re-use the same ten bytes of memory for different purposes.

Lesson 2.9 : Sample Program demonstrating pointers, casts, and arrays of pointeers

Here is the entire program with comments. Remember, this is just a demonstration and is for illustrative purposes only.

If this looks difficult, don't worry too much. You are not expected to memorize any of this yet, just to be able to read the code and understand how it works. If this is too difficult, see Lesson 104 and then come back to this lesson.

To make this even easier to read, I have placed the output of printf() statements INSIDE the code.

Read through this slowly. Take your time, line by line. This is also a lesson, not just a sample program. Read through the comments, code, and output carefully. Ask questions if any part of this is unclear to you.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    // For looping purposes
    int i=0;

    // Allocate a ten-byte working space
    char *main_pointer = malloc(10);

    // Set the first two bytes of this working space to 'AB' using the pointer offset method.
    *(main_pointer + 0) = 'A';
    *(main_pointer + 1) = 'B';

    // Set the next two bytes to: 'CD' using array indexing.
    main_pointer[2] = 'C';
    main_pointer[3] = 'D';

    // Set the rest of the string using the strcpy() function.
    strcpy( (main_pointer + 4), "EFGHI");

    // At this stage, our entire string is set to: ABCDEFGHI<NUL>
    printf("First we use our ten bytes as a string like this: %s  ",main_pointer);

// Output: First we use our ten bytes as a string like this: ABCDEFGHI

    // Let's go through all ten bytes and display the hex value of each character
    printf("Our ten bytes of memory look like this: (41 is A, 42 is B, etc.) :  ");
    for (i = 0; i < 10; i++) {
        printf("%02x ", (unsigned char) *(main_pointer+i));
    }

// Output: Our ten bytes of memory look like this: (41 is A, 42 is B, etc.) :

// Output: 41 42 43 44 45 46 47 48 49 00

    printf("  ");

    // Now let's create an array of two integer pointers
    int **int_pointer_array = malloc(2 * sizeof( int * ) );


    // Set the first of these integer pointers to point at byte #0 of our ten-byte working space
    // and set the second to point at byte #6 of our ten-byte working space.
    int_pointer_array[0] = (int *) main_pointer;
    int_pointer_array[1] = (int *) (main_pointer + 6);

    printf("Now we will use B0->B3 as an integer, and B6->B9 as another integer... ");

// Output: Now we will use B0->B3 as an integer, and B6->B9 as another integer...

// (Note: remember this is B0->B3 of our ten byte working space.)

    // Give these two pointers a value.
    *int_pointer_array[0] = 5;
    *int_pointer_array[1] = 15;

    // Using printf() we prove that the values we set are accurate, and we can see how they are represented
    // as occupying 4 bytes of memory, the way a true int is expected to

    printf("The first integer is: %d (hex: %08x)  ",*int_pointer_array[0], (unsigned int) *int_pointer_array[0]);
    printf("The second integer is: %d (hex: %08x)  ",*int_pointer_array[1], (unsigned int) *int_pointer_array[1]);

// Output: The first integer is: 5 (hex: 00000005)

// Output: The second integer is: 15 (hex: 0000000f)

    printf(" ");
    printf("Our entire ten byte memory space now looks like this:  ");

    // Again we go through all 10 bytes and display their new contents.
    // It is easy to see that the first four bytes and the last four bytes are
    // the integers we created.

    for (i = 0; i < 10; i++) {
        printf("%02x ", (unsigned char) *(main_pointer+i));
    }

    printf(" ");

// Output: Our entire ten byte memory space now looks like this:

// Output: 05 00 00 00 45 46 0f 00 00 00

// (Note: Notice that the integers are 05 00 00 00, rather than 00 00 00 05. We will get to that later.)

    // Finally we demonstrate that bytes #4 and #5 are unaffected, and that our integer values remain set.
    printf(" Bytes #4 and #5 are set to: %c and %c  ", *(main_pointer + 4), *(main_pointer + 5));
    printf(" ");
    printf("Our two integers are set to: %d and %d  ",*int_pointer_array[0], *int_pointer_array[1]);

// Output: Notice that Bytes #4 and #5 are unaffected and remain set to: E and F

// Output: Still, our two integers are set to: 5 and 15 and occupy this same 10 byte space

    free(main_pointer);
    free(int_pointer_array);
    return 0;
  }
```

Output:

```
    First we use our ten bytes as a string like this: ABCDEFGHI
    Our ten bytes of memory look like this: (41 is A, 42 is B, etc.) :
    41 42 43 44 45 46 47 48 49 00

    Now we will use B0->B3 as an integer, and B6->B9 as another integer...
    The first integer is: 5 (hex: 00000005)
    The second integer is: 15 (hex: 0000000f)

    Our entire ten byte memory space now looks like this:
    05 00 00 00 45 46 0f 00 00 00
```

Notice that Bytes #4 and #5 are unaffected and remain set to: E and F

Still, our two integers are set to: 5 and 15 and occupy this same 10 byte space

It may be beneficial for you to write this code into your editor so you can see "color highlighting". Alternatively, you may want to write it at www.codepad.org.

Remember that this is only a demonstration. We are doing some rather unusual and unorthodox things here. The entire purpose of this is simply to show you how these concepts can be used to directly manipulate memory in interesting ways.

I highly recommend that you type out this program, line by line, into your own editor. Not copy and paste, but actually type it out. This will greatly help you to understand the material. Do this even if you get a different result. Remember that this is designed to work where an integer is 4 bytes in size.

Lesson 2.10 : Sample Program Revisited

Here is the same sample program you just looked at, except I have removed all the printf() statements, as well as all unnecessary code. This way you can look at just the "core" process of setting the 10 bytes, and setting the two integers at B0 and B6.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void){

// Allocate a ten-byte working space
char *main_pointer = malloc(10);

// Set the entire string to "ABCDEFGHI<NUL>"
strcpy(main_pointer, "ABCDEFGHI");

// At this stage, our ten bytes look like this: ABCDEFGHI<NUL>

    // Now let's create an array of two integer pointers
    // In other words, create a "two star int" that will point at "one star ints"

    int **int_pointer_array = malloc(2 * sizeof( int * ) );

    // Set the first of these integer pointers to point at byte #0 of our ten-byte working space
    // and set the second to point at byte #6 of our ten-byte working space.

    int_pointer_array[0] = (int *) (main_pointer + 0);
    int_pointer_array[1] = (int *) (main_pointer + 6);

    // Give these two pointers a value.
    *int_pointer_array[0] = 5;
    *int_pointer_array[1] = 15;

    // At this stage, our ten bytes look like this <B0: First integer = 5 > E F <B6: Second integer = 15 >

    free(main_pointer);
    free(int_pointer_array);

    return 0;
  }
```

It would be beneficial for you to type this program out into your own editor, and add printf() statements in various places to demonstrate how this works.

UNIT 3 : More on Multi-Dimensional Arrays and Functions

Lesson 3.1 : On the address-of operator and pointers

Up until now, you have seen pointers and the '&' "address of" operator as two distinct elements of the C language.

You have understood that a pointer means "something that contains the memory address of a variable (or pointer)". Whereas, you have understood the '&' character to mean: "The address of operator"

Now I want to help you to take this understanding to the next level.

Consider this code:

```
int height = 5;
int *int_pointer = &height;
```

Here is what you may not realize: &height is a pointer!

Why is that? Because &height is itself the memory address where height is stored, just like a pointer. Further, &height is of exactly the correct data type that an int * pointer expects.. just like a pointer. It meets all the requirements that a pointer needs to meet.

Therefore, it is a pointer.

From now on when you see operations involving the & operator, understand that the & operator is itself making a pointer. This same concept applies to multiple level deep pointers.

Consider the following:

```
int ***three_star_int = &two_star_int;
```

Now, three_star_int is a pointer. What does it point to? It points to a two_star_int.

What is &two_star_int ? It is also a pointer. What does it point to? It points to a two_star_int. Consider that the '&' character can be considered as "pointer to" just as easily as it can be considered as "address of". The same terms mean the same thing. Containing the address of something is the same thing as pointing to it.

In fact, &two_star_int is of exactly the same data type as a pointer that is created like this: int ***three_star_int. That is why you can assign &two_star_int as the value of three_star_int.

Consider this code:

```
int height = 5;
char my_char = 'a';

int *int_pointer = &my_char;
```

We get a compiler warning. What does it say?

```
Warning: initialization from incompatible pointer type
```

What does this mean? It means that C is understanding &my_char to be a pointer of one data type, and that we are assigning it incorrectly to a pointer of a different data type. The term: "from incompatible pointer type" means that C understands &my_char as a "pointer type".

Now consider the following:

```
int height = 5;
int *my_pointer = &height;
```

'&height' is a pointer of type int. my_pointer is also a pointer of type int. Therefore, the assignment is valid.

If you are still not convinced, consider this:

```
char my_char = 'a';

char *char_pointer = &my_char;
int *int_pointer = char_pointer;  <-- this line creates the warning message
```

We will get the exact same warning message: Warning: initialization from incompatible pointer type

So let's recap this short lesson: When C sees the '&' operator, it understands that you are creating a pointer. What you create using the '&' operator can be assigned to pointers of that data type because this itself is a pointer to that data type.

Lesson 3.2 : Understanding Multi-Dimensional Arrays Better (Part One)

As you recall from earlier lessons, we are in the middle of a series of lessons designed to show you creative ways to use and re-use a ten-byte working space we created with a simple malloc() operation.

We stated that we were going to use our ten bytes in the following four interesting ways:

1. As ten characters
2. As two integers
3. As a 2x5 text array
4. As a data structure

We are now half way through this series. The next goal is to show you how to use these 10 bytes as a 2x5 text array.

Let's discuss the structure behind a multi-dimensional array.

If we start with the basics, you should remember that any array is defined as a set of items of the same data type stored sequentially in memory.

The simplest kind of array is a one dimensional array. Here is an example of a one dimensional array:

```
char simple_array[] = "abcdef";
```

In this example, simplearray[0] means 'a'. simplearray[2] means 'c'. This is easy to understand because the number inside the brackets always perfectly corresponds to the element of the array that you are referring to. [2] means item #2 (when starting at 0).

If it is a one dimensional array, I can give you any possible array index (such as: [21]) and you will have all the information you need to know in order to find the exact item in memory that I am referring to. Consider the following examples:

```
Figure (a)

array[23], array[10], array[34] ...
```

Each of these examples are easy to understand. You just consider the number that is in brackets and you know exactly what item of the array I am referring to. For example, array[10] would refer to element #10 (when starting from 0). (Meaning, Element #0, Element #1, Element #2... All the way to Element #10.)

Consider how much this changes when I switch to a two-dimensional array such as in the following examples:

```
array[3][2], array[10][12], array[9][2]
```

These three examples are not nearly as simple as those found in Figure (a).

Remember from earlier lessons that memory is linear inside a computer. Therefore any multi-dimensional array is actually just a one dimensional array in disguise. It may then seem that you can determine just by looking at the above examples where exactly I am referring to within a two dimensional array.

Let's start with this example: array[3][2]. Where am I referring to here?

You might try some creative math to answer this question. Are you trying to add 3 and 2 together? How about multiply them together?

You can try all the creative math you want, but you will not be able to answer the question. It is impossible to know where I am referring to inside the array from just this information.

Remember this: Before you can calculate any array index of a multi-dimensional array, you must know the dimensions of the array when it was created.

So if I am considering the element at: array[3][2] in some two dimensional array, I must know how that array was created. If it was created as a 6x6 array then this will have a totally different meaning than if it was created as a 10x5 array, or anything else.

Notice that for a one dimensional array this is not important. If it is a one dimensional array, then [20] always means element #20 no matter what. As soon as we advance to anything higher than one dimension however, this rule applies.

Therefore to answer the question posed earlier, two pieces of information must be known: First, the statement that created the array. Second, the array index we are considering.

Here is an example of a two dimensional array being created. The numbers within the brackets define the maximum dimensions of the array. These numbers are necessary in order to calculate any index of this array.

```
char array_2d[10][5];    // <--- This creates a two dimensional 10x5 array
```

Now that we know how the array was created, we can convert any index of this 10x5 array to an offset. Now and only now it is possible to tell you exactly what array_2d[3][2] would be referring to.

From this lesson, you should gain the following key information: It is impossible to convert any multi dimensional array to a pointer offset without having the starting dimensions of the array. This is because different starting dimensions will cause the same array index to refer to a different location in memory.

In the next lesson, we will look at this further.

Lesson 3.3 : Understanding Multi-Dimensional Arrays Better (Part Two)

This lesson is a bit more intense than most. Go through this material slowly.

In the last lesson I explained that in order to understand an index of a multi-dimensional array, you need to have not only the index you are considering, but you also must know the dimensions of the array at the time it was created.

Understanding how multi-dimensional arrays work is critical no matter what language you are programming in. The problem many beginners have is that it is natural to try and understand an array as a grid. For example, a 5x10 array is thought of as having 5 rows with 10 columns (or the other way around).

There are two major problems with this approach. First, memory in your computer is not arranged as a grid, it is arranged as a straight line. Therefore any true visualization should be as close as possible to how your computer would understand a multi-dimensional array.

The second problem is that this method breaks down once you get past three dimensions. How could you mentally visualize a four or five dimensional array with this type of method? You cannot.

It is important to remember that all multi-dimensional arrays are simply one-dimensional arrays in disguise. The process of converting any multi-dimensional array to a pointer offset is the same process for converting that multi-dimensional array to a one-dimensional array.

In order to do this effectively, you must be able to visualize the array you are trying to convert. In this lesson I am going to show you some methods for doing this, as well as the mathematics behind the process.

First we are going to start with a two dimensional array:

```
char 2d_array[10][10];
```

Here we are stating that we have an array of ten elements, and each element itself has ten elements. We can immediately see that the total size of this array is 100, because ten times ten is one hundred.

I started with this array because it is two-dimensional, and is therefore easier to visualize. In this case, you could think of this array as a 10x10 grid with no problem.

I am going to draw out some of this grid in order to make this lesson clearer:

```
     0  1  2  3  4  5  6  7  8  9

0    0  1  2  3  4  5  6  7  8  9
1   10 11 12 13 14 15 16 17 18 19
2   20 21 22 23 24 25 26 27 28 29
.   ...
7   70 71 72 73 74 75 76 77 78 79
8   80 81 82 83 84 85 86 87 88 89
9   90 91 92 93 94 95 96 97 98 99
```

Note that you can identify any element of this array by simply lining up the grid. For example, I can see from this array that [0][0] would be the very first element, which is called '0'. I can also see that [2][5] would be 25 (twenty-five). It is easy to just line up the grid and find the exact "pointer offset" for any element of our two dimensional array.

This method falls apart though if we consider this array to be three dimensional. Certainly I cannot draw a 3D representation of such an array in this text box. However, I can do something better.

First, I want you to notice something interesting about our two dimensional array. You have been using it to count all your life. This is just a representation of our base ten numbering system made into an array.

Notice therefore that I chose [10][10] because it gives us a "ones" place, and a "tens" place. If you look up at the array chart, you will see that each row is a new "ten", and each column is a new "one".

How else could you visualize this array? You could visualize it by simply understanding that each time you add a "ten", you jump forward by ten ones. Each time you add a "one", you jump forward by only one.

Let's re-consider the array element: [2][5]. You could also understand this by saying: two tens, and then five ones.

Now consider this array:

```
char 3d_array[10][10][10];
```

We are still left with two luxuries. First, we are still within our own base ten counting system and we immediately understand this three dimensional array without having to resort to some sort of 3D grid. Secondly, all the array elements share the same maximum size of ten.

With the above array, how would we understand: [4][2][1] ? Four hundreds, two tens, and a one. This is true only because the original array dimensions are: [10][10][10].

Now imagine you are standing on a zero. There is a straight line that extends out in front of you towards infinity with numbers marked off at intervals starting at zero and incrementing by one.

How do you represent [4][2][1] from this situation? The answer is, you take four very large jumps (each one a hundred units in size), then you take two large jumps (each one ten units in size), and finally you take one step.

At this stage you should be perfectly comfortable visualizing even an eight dimensional array provided that each array index has a size of ten.

Lesson 3.4 : Understanding Multi-Dimensional Arrays Better (Part Three)

I split this into two halves and added some material to make it easier to grasp.

In the last lesson I showed you that our base ten counting system is itself a form of a multi-dimensional array. This is how we represent tens, hundreds, thousands, and so on.

Now, consider that instead of our array being [10][10][10], we made it: [10][5][10]

First, let's talk about what this means. Here are various ways of understanding: [10][5][10]

1. It means that we have ten array elements, each element consisting of five elements. Each of those five elements consists of ten elements.
2. It means that we have ten 5x10 grids.
3. It means that we have ten elements where each element is a set of five words and each word can consist of up to 10 characters.
4. It is a "half cube" with a width of ten, height of five, and depth of ten.

Any of these methods of visualizing should help you understand this better.

We want to convert the array index [4][2][5] to a pointer offset. Remember that our array dimensions were defined as: [10][5][10]. Here is what now happens:

We have 10x5x10 (500) elements. First, we take 4 large jumps, each one being fifty units in size; not one hundred units any more. Why fifty? Because [5] (the 2D Size) times [10] (the 1D Size) is fifty. Then we take 2 small jumps each one being ten units in size. Finally, we take 5 steps each one unit in size.

In this way we have done: [4] large jumps, [2] small jumps, [5] steps. Each large jump was 50 units. Each small jump was ten units in size. Each step was one unit in size.

Now let's consider: [20][40][10] as our starting array dimensions. In this case, to find [4][2][5] we would first take four very large jumps each one four-hundred units in size. Why? 40 x 10 = 400. Then we would take two large jumps each one ten units in size. Finally, we would take five steps each one being one unit in size.

Don't worry if you are confused. It will be clear in a minute.

At this point all you really need to know is this: Any multi-dimensional array can be visualized by imagining that you have a pointer offset in memory which starts at 0. That offset will "jump" by the largest amount for the first set of brackets. Then it will jump by a smaller amount for the next set, and it will continue to jump by smaller and smaller amounts until it is eventually arrives at the location specified by the array index. This is not merely a visualization, it is exactly what the pointer is doing in such a case.

All that is left now is to convert this to a mathematical formula. Before we do that, I want to go back briefly to our base ten example.

Consider an array created like this:

```
char five_d_array[10][10][10][10][10]
```

Even though this is a five dimensional array, it shouldn't scare you. It is just a representation of how we count up to the ten-thousands place.

Now, if we consider [3][2][9][2][1], we know the actual offset is thirty-two thousand, nine-hundred and twenty-one. Let's now convert that into a mathematical formula.

```
( 3 * 10,000 ) + (2 * 1,000) + (9 * 100) + (2 * 10) + 1
```

Remember that if you were standing on a zero, to visualize this you would basically fly over ten thousand units three times, take enormous leaps over one thousand units twice, large jump over a hundred units nine times, two large steps over ten units, and then one small step to reach your destination.

Now the only thing that remains is understanding how we got our numbers for this formula. First, we have to consider the array when it was created. Notice that I name each index with a capital letter starting at A and increasing as we get to higher dimensions.

```
[ E][ D][ C][ B][ A]
[10][10][10][10][10]
```

Now, we have to look at the actual index we are considering:

```
[e][d][c][b][a]
[3][2][9][2][1] = 32,921
```

For the array index I used lower case letters that correspond to the upper case letters. The upper case letters refer to the maximum dimensions of the array based on how it was created. The lower case letters refer to the actual index we are considering.

Before we can construct our formula, we need to know the true size of each index in our array.

```
char multi_dimensional_array[10][10][10][10][10];
or...
char multi_dimensional_array[ E][ D][ C][ B][ A];
```

```
E_Size = (DCBA) = 10 * 10 * 10 * 10 = 10,000
D_Size = (CBA)  = 10 * 10 * 10      = 1,000
C_Size = (BA)   = 10 * 10           = 100
B_Size = A                          = 10
A_Size = 1                          = 1
```

Notice that A_Size is always equal to one unit. Notice that B_Size is always equal to A.

If that is confusing, think about it another way: You get the size of any index (E) by multiplying all the indexes to the right of it (DCBA) together. Remember that the uppercase letters determine how big a jump will be for that element (by multiplying the elements to the right of it together). The lowercase letters determine how many jumps to make.

Now, we look at the actual array we are considering:

```
[3][2][9][2][1]
```

Now we can calculate this offset very easily:

```
( 3 * E_Size) + (2 * D_Size) + (9 * C_Size) + (2 * B_Size) + 1

(3 * 10,000) + (2 * 1,000) + (9 * 100) + (2 * 10) + 1
30,000 + 2,000 + 900 + 20 + 1
32,921
```

Now consider an array created like this:

```
char three_d_array[5][3][9] <-- dimensions of the array when it was created
```

In this case:

9 = A
3 = B
5 = C

The size of C is simply (BA) or 3*9. The size of B is simply (A) which is 9.

How would we then identify the offset for: 3d_array[2][2][1] based on the above dimensions? Like this:

```
( 2 * C_Size ) + (2 * B_Size) + 1

(2 * 27) + (2 * 9) + 1
54 + 18 + 1
```

```
73
```

And that is the answer. [2][2][1] in that array corresponds to element #73.

One more example:

Consider an array created like this: [12][6][4][8], and the index you want is: [4][2][3][1]

Largest Jump: 6*4*8 = 192
Next Largest: 4*8 = 32
Next Largest: 8

so, the answer is:

```
(4 * 192) + (2 * 32) + (3 * 8) + 1
768 + 64 + 24 + 1
```

Therefore, we are talking about element #857

You should now be able to take any multi-dimensional array and convert it to a pointer offset.

Lesson 3.5 : Demonstrating a 2x5 array with pointer offsets

Now that we have finished the preceding lessons, you should have a much better understanding concerning how arrays are stored in memory. Therefore, we should continue our series on the different ways that ten bytes of memory can be used.

I had also planned doing some lessons on how to use the ten bytes as a data structure, but I have decided against doing this. In an earlier lesson I have already shown how a structure can be represented in memory, and there is no point in repeating that lesson.

Now, let's begin.

First, let's create our ten bytes to be used as our 2x5 array:

```
char *our_pointer = malloc(10);
```

Now the funny thing is, we are already done. We have our 2x5 array simply because ten bytes is a 2x5 array. The only thing really left to do is to store our two words, and use printf() to show that everything works.

In order to store the data into our array, we need to know where to put it. Because this is a 2x5 array, we are planning on storing two words that are each a maximum of five characters (including NUL).

Remember from the last set of lessons that a 2x5 array will work like this:

```
5 : 1D Component
2 : 2D Component
```

Because 5 is our 1D component, any time we increase our 2D component we will increase our pointer offset by 5 characters. In other words:

```
our_pointer[0][0] = Byte: 0
our_pointer[1][0] = Byte: 5
```

Now we can use strcpy() to store words (four characters or less) at these locations.

```
strcpy(our_pointer, "two");
strcpy(our_pointer + 5, "word");
```

We can use printf() to show that this worked as expected:

```
printf("The first word is: %s", our_pointer);
printf("The second word is: %s", our_pointer + 5);
```

There is of course one problem. We cannot use our array indexing the way we could if this had been created as a true char[2][5] array. Why is that? Remember from the last lessons I showed you that for any array greater than one-dimension, that having the array index alone is not enough to know what element is being referred to.

In our two-dimensional array, we cannot just say: our_pointer[1][1] for example. C has no way of understanding how big each 2D component is. Notice that in our code we have at no point said that our array is 2x5. We are simply treating it like that using our pointer offsets.

The truth is, this is perfectly fine. As long as you do not mind not having the luxury of being able to use brackets, you can get along just fine with this method. However, this lesson would be incomplete if I did not show you how you could also use brackets to index this.

We will explore that in a few lessons. But first, here is an example program demonstrating what I just showed you.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(void) {

    char *our_pointer = malloc(10);
    strcpy(our_pointer, "two");
    strcpy(our_pointer + 5, "word");

    printf("First Word: %s  ", our_pointer);
    printf("Second Word: %s  ", our_pointer + 5);

    // Simulate array[1][1], array[0][3], array[1][2] where each 2D element is 5 chars

    int B_Size = 5;  // Remember that B_Size is always equal to A. (5 in this case)
    int A_Size = 1;  // Unnecessary, but helps to demonstrate this.

    printf("array[1][1] would be: %c  ", *(our_pointer + (B_Size *1) + 1)); // same as array[1][1]
    printf("array[0][2] would be: %c  ", *(our_pointer + (B_Size *0) + 2)); // same as array[0][2]
    printf("array[1][3] would be: %c  ", *(our_pointer + (B_Size *1) + 3)); // same as array[1][3]

    // simulate array[0] and storing a word using strcpy()

    strcpy((our_pointer + (B_Size * 0)), "test");
    printf("array[0] string is: %s  ", (our_pointer + (B_Size *0)));
```

Don't let (our_pointer + (B_Size * 0))) scare you. Anything times zero is zero. Therefore, we are adding zero to our_pointer. Which means we are getting back our_pointer. I did not have to do this. I could have just as easily have written our_pointer. If I had done so though, then you would not be able to see the simulated array syntax.

This is the same thing as:

```
    strcpy(our_pointer, "test");
    printf(" ... %s", our_pointer);

    // simulate array[1] and storing a word using strcpy()

    strcpy((our_pointer + (B_Size * 1)), "ing");
    printf("array[1] string is: %s  ", (our_pointer + (B_Size *1)));
```

Don't let (our_pointer + (B_Size * 1))) scare you. B_Size * 1 is just B_Size, which is just 5. Therefore, we are just saying: our_pointer + 5. The reason we are saying B_Size * 1 is just to illustrate that this would be the same thing as if you had a char array[2][5] array and were to write: array[1].

This is the same thing as:

```
    strcpy(our_pointer + 5, "ing");
    printf(" ... %s", our_pointer + 5);

    return 0;
}
```

Output:

```
First Word: two
Second Word: word
array[1][1] would be: o

word

array[0][2] would be: o

two

array[1][3] would be: d

word

array[0] string is: test
array[1] string is: ing
```

Lesson 3.6 : The practical use of functions (part one)

In the last lesson I showed you how to demonstrate a simple 2x5 array using 10 bytes of memory allocated using a malloc() operation. Later in the course, we will come back to that, as I want to show you some other creative ways to do this. Later I will show you a way to do this where you can even use proper array indexing just as if it had been created as a normal array.

At this point in the course though, it is time for us to go back to functions. It is critical for anyone wishing to be a programmer to know how to properly create and use functions.

Earlier in the course I taught you that variables were a way to give a name to a memory address where some data was stored. In this lesson, I want you to learn that a function is in part a way to give a name to some actual code, so that you can refer to that code from now on with a simple name instead of by writing that code out by hand. In this way, a variable and a function are quite similar.

A variable gives you a way to give a simple name to a complex memory address. Similarly, a function gives you a way to give a simple name to some complex code or operation.

In this lesson, I want to introduce to you four key reasons why you should use functions:

1. Functions make your code easier to read.
2. Functions allow for more organized and structured code.
3. Functions make fixing problems and bugs simpler.
4. Functions reduce redundancy and allow you to re-use code.

Now, let's see this in practice.

In the last lesson, we demonstrated a simple array using pointer offsets. Part of that demonstration involved using the strcpy() function to copy some text into a given array element, following by a printf() statement to show that this worked as expected.

Recall that this code looked like this:

```
Figure (a)

strcpy((our_pointer + (B_Size * 0)), "test");
printf("array[0] string is: %s  ", (our_pointer + (B_Size * 0)));
```

Does this appear difficult to read? If so, it is only because of its complexity. There is nothing especially difficult in those two lines of code, however simply because we have so much detail packed into so little space, it appears difficult.

Now because the code in Figure (a) seems difficult to understand, let's instead describe it. What is that code doing? Well, it is demonstrating an array.

Instead of writing the above 2 lines of code, why don't we make things easier by writing this instead:

```
demonstrate_array();
```

In other words, let's make these two lines of code into a simple function.

The first step to creating any function is to decide on what that function is actually doing. This enables you to give a name to the function that is descriptive and easy to understand.

It is poor practice to give cryptic names to functions and/or variables. It is generally poor practice to call a variable a or a function x(). Always name a variable or a function something that can be easily understood by not just you, but by anyone who will later read your code. If you are writing a variable such as for a loop, or something on these lines, then it is alright to use a variable name such as i, j, k, etc. This is because those reading your code will understand what these variables mean.

Do this even if you are 100% sure that the only person who will ever read your code is you. Why? Because you may just find yourself a year or two later going back to something you have written, and find yourself utterly unable to understand any of it. You will then have to throw it all away simply because the time it would take you to understand it is less than the time it will take you to write it over again from scratch.

The more easily understood the code you write, the more valuable it is both to you and also to anyone who will ever read it. About ten years ago I had the opportunity to sell the source code for some software I had written. The program itself worked great, but the code itself had few comments, and only I could read it. The fact that I could read it perfectly meant little to the buyer.

I ended up having to go through and re-write large chunks of the program, add many comments, and create proper documentation. This was tedious and frustrating work, and I wouldn't have had to do any of it if I had simply done this to start with. I encourage you not to make the same mistake I did.

Now we have created a function with a name, demonstrate_array and we know the two lines of code that are going to go into that function. The next step is to make the function actually work.

That will be the topic of the next lesson.

Lesson 3.7 : The practical use of functions (part two)

First, recall from our previous lesson that we are planning to create a function called demonstrate_array() using the code in Figure (a) below.

```
Figure (a)

strcpy((our_pointer + (B_Size * 0)), "test");
printf("array[0] string is: %s  ", (our_pointer + (B_Size * 0)));
```

Sometimes you will know ahead of time that you want to create a function and you will write it from scratch. Other times you will want to take some code you have already written and convert it to a function. In this lesson I will show you how to take code you have already written and turn it into a working function.

To create any function like this, you should follow these five steps:

1. Create a blank function and simply copy and paste the code into it that you will use to build your function.
2. Determine what arguments you will need for the function.
3. Convert the code in the function to use those arguments.
4. Decide on a return value, if any.
5. Test the function to ensure it works as expected.

Now let's create our demonstrate_array() function. First let's create a blank function with no arguments and no return value. Therefore, the final function will look like this, after the main() function:

```
int main(void) {
    ... main() code goes here ...
}

void demonstrate_array(void) {
    strcpy((our_pointer + (B_Size * 0)), "test");
    printf("array[0] string is: %s  ", (our_pointer + (B_Size *0)));
}
```

Notice that this function is created after the main() function ends. This should be the case for any functions you write at this stage in the course.

This is the first step for creating a function. I have simply cut and pasted the code I intend to use into a "blank" function. This will not work yet however.

A common beginner source of frustration is trying to make functions, and then finding that they simply do not work. It is easy to make the code work in the main() function, but when you take that same code and try to make a function out of it, invariably you will get some strange C compiler errors.

Understand that as frustrating as these errors can be, especially for a beginner, you must know how to read them if you are to be a successful programmer. You will experience compiler errors, and you should be glad when you get them.

Why? Because when you get an error, your code will not compile. That means it will not break, there will be no bugs. The best thing that can happen to you as a programmer when you make a mistake is that the program refuses to compile and gives you a reason why.

The worst thing that can happen to you is that the program compiles, appears to work, but has some bug which causes the program to break because of some mistake you made. In this case, finding and fixing the problem is a lot harder. For this reason, you should be glad when your compiler gives you an error.

So before we go on, what would happen if I tried to compile the program with the above function, as is? This would happen:

```
/home/carl/oct22.c: In function ‘demonstrate_array’:
/home/carl/oct22.c:33: error: ‘our_pointer’ undeclared (first use in this function)
/home/carl/oct22.c:33: error: (Each undeclared identifier is reported only once
/home/carl/oct22.c:33: error: for each function it appears in.)
/home/carl/oct22.c:33: error: ‘B_Size’ undeclared (first use in thisfunction)
```

It is impossible to learn C (or any language) without being able to understand these kinds of error messages. Therefore, let's begin with the first error message received:

```
/home/carl/oct22.c:33: error: ‘our_pointer’ undeclared (first use in this function)
```

First, understand this exact format and message may differ between different C compilers. However, notice that you will see the file name in question. In this case, oct22.c. Also, notice that the line number which created the error is given. In this case, line 33.

A common beginner mistake concerning errors is to look at the line where the error is reported to have occurred, and to assume that this line and only this line must be the problem. This is not always the case. The line number reported is only the line where the Compiler realized there was a problem. The problem may in fact have actually taken place earlier.

Therefore, the correct approach is to look at the line number in question and ask yourself, "What is it about this line that caused the compiler to realize there is a problem?"

The next thing I should point out is that often one or two problems can generate hundreds of error messages. Just because you see five hundred errors in a program you try to compile does not mean you have to go and fix five hundred different things.

It is therefore always advisable to start fixing errors by addressing the first error message you see. Often you will find that each error you fix will cut drastically the total number of error messages there are.

First, let's look at the line in question:

```
  strcpy((our_pointer + (B_Size * 0)), "test");
```

This is my line #33. This is where I should start looking in order to fix any problems.

That is our first error message. This type of error message simply means that we are using a variable we have not declared. In this case, C is complaining that we are using a variable called our_pointer but that we have never declared it.

Well, this is wrong. We have declared it. We did so in our main() function, right here:

```
char *our_pointer = malloc(10);
```

So we have declared it, why then is C complaining saying that we have not?

The answer to this question is the topic of the next lesson.

Lesson 3.8 : The practical use of functions (part three)

In the last lesson we created our demonstrate_array() function, but C generated an error message saying that our_pointer was undeclared. The reason for this has to do with something called "scope".

What is scope? Scope is a term that refers to the visibility of information within certain boundaries. Do not worry if this is confusing. It will be clear soon enough.

In most programming languages, you have some way in which can you set up boundaries in which information can or cannot be seen. In C, one of the ways this is done is through the use of functions.

The first thing you must know is that variables you create in one function, such as main() cannot be seen by any other function. This is extremely important, so remember this. Any variable you create in any function is invisible to any other function.

That means that even though I created our_pointer in main(), it is entirely invisible in the function we just created called demonstrate_array(). So therefore, the first real problem with our new function is right here:

```
void demonstrate_array(void) {

strcpy(( our_pointer + (B_Size * 0)), "test");
```

The problem is that our_pointer does not exist in this function. For our function to work properly, we need to provide some way that our_pointer can exist.

At this point, you might wonder why are things done this way. Why not just make it so that any function can see any variable created anywhere? There are many good reasons for this.

Imagine a program with hundreds of functions. If this were the case, any time you created a new variable for any of your functions, you would have to first of all make sure that you haven't already used that variable in some other function.

Worse still, if you were to accidentally use this variable without declaring it, you would end up with the value that some other function gave that variable. This would certainly cause your program to not work the way you expected.

This one reason should be enough to convince you that letting all functions see all variables is a bad idea. You will learn more reasons later in the course.

Now that I have established that our_pointer doesn't exist in the function demonstrate_array(), I want to give you another way of thinking about this problem. The problem is not that the variable doesn't exist, it is that it was created outside of the scope of the demonstrate_array() function.

Any time therefore that you create a function by copy-pasting code from somewhere else in your program, you must be mindful that any variables inside the code you are copying were created outside the scope of the function you are now creating.

This then leads us to a problem: How do we get the variable to be visible to the function? This brings us to step two of our five step process.

2. Determine what arguments you will need for the function.

Remember that an argument is the term for information that you send to a function. In this case, I need to send some information from my main() function to the function I am creating.

There are two variables which were created outside of the scope of our demonstrate_array() function that I need to be concerned about.

1. our_pointer
2. B_Size

Both of these variables were created in our main function.

our_pointer is of the data type: char *
B_Size is of the data type: int

Whenever you determine arguments for a function that was created by copy-pasting code from somewhere else, it is usually a good idea to name the arguments of that function exactly what that function expects them to be. Doing this is actually very easy.

Step one, specify the data types for the arguments:

```
void demonstrate_array(char *, int) {
```

Step two, specify the names:

```
void demonstrate_array(char *our_pointer, int B_Size) {
```

And already, I am done. I now have a usable working function.

When you become experienced in writing programs, doing what I just showed you becomes something you do without even thinking about it. You create the new function, you copy paste the code into it, and you change the arguments of the function.

Usually you know ahead of time what those arguments are, and you just type them in as soon as you create the function. Now however I have shown you how this process works.

You will notice that there is a step 3 in our five steps, which reads like this:

3. Convert the code in the function to use those arguments.

Notice that by naming the arguments according to the variables used within the function. I have completed steps 2 and 3 together with a single act.

Now that we have a working function, Inside our main() function, we just put this:

```
int main() {
    ... some code ...
    demonstrate_array(our_pointer, B_Size);
}
```

In this way we are now sending our_pointer as well as B_Size to the function. This will now cause the function to work exactly as we expect.

Have we forgotten anything? In a previous lesson I explained that any time you create a function, it is good practice to write that function definition at the top of your program. In the next lesson I will show you some additional steps we can take to make this function more useful.

Here then is a complete sample program demonstrating what we just did:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

// Function Definitions
void demonstrate_array(char *, int);

int main(void) {

  char *our_pointer = malloc(10);

  int B_Size = 5;

  demonstrate_array(our_pointer, B_Size);

  free(our_pointer);

  return 0;
}

void demonstrate_array(char *our_pointer, int B_Size) {

  strcpy((our_pointer + (B_Size * 0)), "test");
  printf("array[0] string is: %s  ", (our_pointer + (B_Size * 0)));

}
```

Lesson 3.9 : Basics of Project Management

When I first introduced that we would be doing a tic-tac-toe program, I did not realize that there was quite so much material that still needed to be covered before we could actually do this. I feel now at this stage that we have covered all of the material we needed to, and therefore we are ready to begin.

This will also give me an opportunity to show you some of the methods and thinking processes which go into designing a program in general. As much as it is possible, I want you to understand not only what code I write, but why I write it and more importantly what thought processes lead me to make the decisions that I make.

However, before we can start with this project, I need to tell you this:

The first step to writing any program, or indeed to completing any task is to define that task well. This applies especially when you are writing a program designed to solve a problem, or writing a complex algorithm. You must always remember that step one is to clearly and accurately define exactly what it is you are trying to achieve.

Your program, whatever it is, is finished when it meets the "finish criteria" that you define. The finish criteria is a list of statements of fact that are true once the program is completed. It is not a series of todo's, or a series of desired features. It is a list of statements of fact.

To those considering a career in programming, and especially to those considering becoming freelancers, I offer the following advice:

The reason I refer to this as "finish criteria" is because it leaves no doubt in anyone's mind that once these statements are true, the project is entirely finished, and your obligation regarding that project is fulfilled. Statements of fact are easily verified, and there is nothing left to debate.

Always remember the following:

1. You should not write your first line of code or even agree to do any project until the finish criteria is written and agreed upon by you and whoever it is who has hired you to write the program, whether for a job or as a contractor. This should also make sense because until you have truly stated what that finish criteria is, you really do not know for sure what is required. Therefore, how could you agree to do it?

2. Once the finish criteria is in place, do not change it. Do not let someone else change it. If there are new desired features or functionality, plan them for a future release. If the project will only take you a week to finish, then great: finish the project, then work on adding the new features.

3. Sometimes #2 is not possible, such as when a project would be worthless without a previously unknown feature. In these cases, always take a step back, and re-write the new finish criteria. Make sure it is agreed upon by everyone. Remember that any finish criteria should have a "back out" option for you if that criteria changes. If the finish criteria changes, you should be the one who is able to agree or disagree on being able to finish the newly defined project.

Now, let me tell you the benefits of doing things this way:

1. Your finish criteria will double as documentation once the project is finished. Very little if any editing will be needed to describe the final product. This will also help you and others who may work on the project later. Everyone will know exactly what was expected and therefore managing and maintaining the project will be much easier. Important features are less likely to be accidentally discarded because someone didn't know that feature was supposed to be there.

2. Doing this will protect you against what anyone who has freelanced knows as the "expanding horizon". This is when whoever has hired you expects you to keep working for them, forever, until the project is "finished" which typically refers to some time between now and the Sun exploding. Always remember that people will take advantage of you given the opportunity. Don't give them that opportunity.

3. Doing this will help you to better understand the project. Writing out the finish criteria will double as a project plan. It is a simple process to convert a statement of fact into a statement of what needs to be done in order to make that fact true. By clearly writing out the finish criteria you will find that it is easy to break it down into easier and easier tasks. If you do this a few levels deep, you should have a complete project plan as well as a reliable way to estimate the cost and time that will be involved.

4. You will finish projects faster. You will understand what is required clearly and you will not find yourself in a situation where you do not know what to do in order to finish the project. You will enjoy your work more because it will require less stress. Also, the risk of you having to throw out code you have already written because you did not realize something is greatly reduced.

Hopefully in this lesson I have stressed the importance of properly defining the finish criteria of any programming task before you start it. In the next lesson, we will start writing our tic-tac-toe game. Very soon after, I am hoping to start getting into bigger and better projects.

Ok, I am back. Lessons may come a bit slow this week, but they will come.

Before I show you how to write a tic-tac-toe game, I want to teach you some of the basics concerning project management.

You may be surprised to know that programming takes up at the very maximum 40% of any work day for me. That means that most of the time I am working, I am not actually writing code. There are many days that this is probably closer to 10%.

You cannot just sit down and start writing a program. You must have a clear idea of what it is you are trying to do. This requires planning and research. Also, if you are working in any kind of professional capacity, there will be a significant amount of time spent communicating with others.

With this in mind, it would be foolish for someone to spend all of their time learning skills that will only apply about 40% of the time. If you desire to be a successful programmer, you need to learn not only how to write code but the other skills that go with this line of work.

Now, let's consider again the tic-tac-toe game we are planning to write. First you must map out the basic structure of the project.

```
[ ] Tic-Tac-Toe Game
    [ ] ... Now we break it into parts ...
```

Notice how I use [ ] to indicate a part of a project that is not yet completed, and I use [X] to indicate a part that is finished. When you do this in any kind of mono-spaced text editor, it lines up very neatly, and it is easy to keep track of your progress. You can also expand lines very easily.

I do not use any kind of paper based project manager. The problem with paper is if you write out ten lines, you cannot "insert" something between line 7 and line 8. The very nature of planning a technical project requires that you are able to expand items.

So the rule is simple, for each piece of a project, you put four spaces and then [ ] then type out that particular item. As you continue to break down a task like this, you will find it starts abstract and slowly becomes closer to actual code you can write. Here is a simple example:

```
[ ] Write first C program
    [ ] It needs to say "Hello Reddit"
        [ ] printf("Hello Reddit");
```

This is a simple example, but this helps to illustrate the point. The idea of any project plan is not merely to write out your goal for the project, but also to construct a technical means to achieve your objective. The idea is that by writing out what you want to achieve, you end up simultaneously writing out how you will achieve it. That is the hallmark of a good project plan.

Now, let's go back to our tic tac to game. If we were to break it into parts, how would we do this? Think about this in terms of statements of fact that gradually become more detailed. For example:

```
[ ] There exists a tic-tac-toe program.
```

Now, we can break this into more detail. What does this mean exactly? How can we expand it? Notice that the way we break down each step is by asking questions.

```
[ ] There exists a tic-tac-toe program.
    [ ] There is a grid of nine squares
        [ ] There is a function that will display this grid
```

Ok.. now what does it mean to "draw the grid" ?

```
            [ ] This function will clear the screen
            [ ] This function will draw out the first row of 3squares
```

```
[ ] This function will draw out the second row of 3squares
[ ] This function will draw out the third row of 3squares
```

Ok, what does it mean to "draw out a row of 3 squares" ?

```
[ ] This function will draw out the first row of 3squares
    [ ] For each of three squares :
        [ ] Determine if it is an 'X', 'O', or blank
            [ ] If 'X' :
                [ ] Draw an 'X'
                    [ ] printf("X");
```

And already you can see how this is turning into programming code. You should also start to see, that just by writing out the details of the project, we can already see for example that we need a function to draw a square, and we need a function to draw a row, and we need a function to draw the whole grid.

I do not actually need to write out "printf(X)" in the above project plan. I did so only so you can see how the process evolves from simple statements, to more detailed statements, to actual programming code. As a programmer, you should be able to see a well written project plan and simply envision the code that should make it happen. You do not write the code into the project plan. Rather, you write the code into the actual program, while simultaneously checking off the parts of the project that are then completed.

Notice that everything becomes clear as you simply write out the details of the project. When the project plan is done, a large part of the actual work is done also. Then as a programmer all you have to do is simply fill in the gaps, and start checking off [X] each piece of the project as you finish it.

Further, the project plan you write will double as documentation. We will go into this process more in the next lesson.

UNIT 4 : More on Data Structures

Lesson 4.1 : Structures contain data and information

In the last lesson I showed you the basics behind how to manage simple projects. Fundamentally this comes down to keeping track of the different tasks you are doing, and knowing how to break them into sub-tasks. Think of each of these tasks as a "requirement". When all the "requirements" are done, the project itself is finished.

Now, let's consider the tic-tac-toe example program we are working on.

First of all, we have to realize at this point that we need to have a structure to contain the tic-tac-toe board.

Let's put that down as a requirement:

```
[ ] A structure exists to contain the tic-tac-toe board
```

That is simple enough. Notice that this is a statement of fact. It is either true, or it is not true. Writing out requirements as statements of fact will help you a great deal.

Let's be more specific. What should be contained in this structure? First of all, we need a grid of 3x3 squares. One way to achieve this is a simple array, so we can write that as a requirement of our structure like so:

```
    [ ] Contains an array of 3x3 characters
```

To plan the next step, we have to consider why we are using a structure to begin with. Why not just use an array? The answer is simply this, an array can easily contain a tic-tac-toe board, but that is all it can contain.

Our goal is to contain not only the tic-tac-toe board, but also certain "information" about it. For example, we can have an integer called "winning_position" which indicates if this tic-tac-toe board is a won position.

So our next set of requirements should be "Information about the tic-tac-toe board".

Here are a few such requirements we can have:

```
[ ] Whose turn it is (X or O)
[ ] Whether the position is won for 'X'
[ ] Whether the position is won for 'O'
[ ] What "move #" this is (first move, second move, etc)
```

We could always add on later, but this should be a good starting point. You can see that our data structure consists of more than just the tic-tac-toe board itself, but it also contains information about the tic-tac-toe board. Keep in mind that when we start to apply "artificial intelligence" to our tic-tac-toe game, we will want to have additional information such as potential moves to evaluate, whether or not the position looks like X is winning or O is winning, and so on.

Containing data as well as information about that data is an important programming concept which we will explore later.

Now, let's examine this a bit closer. The idea here is that we have some "thing", in this case, a tic-tac-toe board. This "thing" in this case is a 3x3 array. But this "thing" also has information associated with it.

The tic-tac-toe board is the "thing", and the properties or information about that thing are also contained within the data structure. Let's take this concept out of our tic-tac-toe example and look at some other examples of where this may apply in programming:

Let's suppose we have a data structure for a paint-brush in some graphics program. The paint-brush is the "thing". However, there must be certain information associated with it also. This includes the color it draws in, the size of the brush, perhaps the type of the brush, and so on.

So you can see here that it is important to realize that when you construct a data structure, you are interested in not only the data itself, but also the information that is attached to that data. The information attached to the data allows you to understand the data better.

Now, let's summarize this part of our requirements like so:

```
[ ] Tic-Tac-Toe Game
    [ ] A structure exists for the tic-tac-toe board position, and contains:
        [ ] Whose turn it is (X or O)
        [ ] Whether the position is won for 'X'
        [ ] Whether the position is won for 'O'
        [ ] What "move #" this is (first move, second move, etc)
```

Notice that our requirements are not concerned with how we implement these things. We are simply stating them as goals to achieve to complete the project.

Lesson 4.2 : Using functions as questions

It is often the case when writing a program that you will need to ask some question concerning the data you are working with. For example, at some point we need to know whether or not the game is over. As a programmer understand that any time you ask a question, this often indicates you will need a conditional flow statement.

Let's consider our question "Is the game over" as a mixture of "English" as well as "programming code". It would look like this:

```
if (game is over) {
```

Whenever you write out plain-English "code" like this, it is known as "pseudo-code". Writing out pseudo-code is a great way to better understand what a program is doing. Sometimes using pseudo-code within comments makes it easier to understand complex algorithms.

You should see from our pseudo-code example that it would make sense to use a function here. Let's see how that would look:

```
if (game_is_over()) {
```

Notice how our words "game is over" can perfectly translate to a function name. Our code has now with very little effort transformed from pseudo-code to actual "C" code.

The idea then is to cause the "game_is_over()" function to return a 1 if the game is over, and a 0 if the game is not over.

If the "game_is_over()" function returns a 1, the if statement will work like this:

```
if (game_is_over()) { : becomes
if (1) {
```

Why? Because remember that if game_is_won() returns a 1, then using game_is_won() anywhere in the program is the same exact thing as using 1 anywhere in the program. The function return value will always "replace" the function itself anywhere that function exists.

So in other words, if you make a function return 1, you can use that function name very easily in an if statement. Give such functions two possible return values: 0 and 1. Return 0 only if the result of the function is contrary to the function name. Return 1 otherwise.

For example:

```
if ( player_is_out_of_ammunition() ) {
```

And you can see how easy that is to read.

Back to our tic-tac-toe-game, we will need to have a function that will check to see if the game is over. We also need to know, if the game is in fact over, who won. So, let's consider how that might look:

```
if (game_is_over()) {
    if (winner_is_x()) {
    }
    if (winner_is_o()) {
    }
    if (game_was_a_tie()) {
    }
}
```

Notice how easy this is to read.

We haven't yet decided how to make these functions, but we still understand they need to exist. Therefore, let's list them in our project plan as requirements:

```
[ ] Functions we need for our tic-tac-toe data structure
    [ ] Determine if the game is over
    [ ] Determine if X won
    [ ] Determine if O won
    [ ] Determine if the result was a tie
```

Now, let's consider the purpose of these functions. All of these functions have something in common. Their purpose is to look at a data structure and then answer a question about that data. The purpose of these functions is therefore to evaluate data to determine some fact. These types of functions include whether or not the game is over, who won, who is winning, etc. These types of functions are useful any time you need to ask any question.

So in this lesson I am showing you that functions are not merely chunks of code that can achieve some task, like we have seen up until now. I am showing you that by being creative you can create different "kinds" of functions. In this case, we are creating functions designed to answer questions.

In later lessons I will show you other kinds of functions you can create, and how to use them.

Lesson 4.3 : Introducing Function Hierarchy

We established in the last lesson that there are certain functions we need to write which have the purpose of determining if something is true or false. It is useful to give these functions names that make it easy to distinguish them from other functions we will use. A good way to do this is to put the word "is" in front of each function that is designed to be used as a question. For example: is_game_over().

Now, we cannot have a tic-tac-toe game if we do not have some means of marking a square either 'X' or 'O' in our 3x3 grid. This means that the 3x3 array data will change. Therefore, a function which "marks" a square would do so by changing the array in some way.

We need some function called "mark_square()". Let's list that as a requirement:

```
[ ] A function that can mark a square in the 3x3 grid
```

Now, let's be more specific. Our function needs to know what square to mark. Therefore, we need to give our function an X coordinate and a Y coordinate. We could say for example: mark_square(1,2) or mark_square(1,1). Lastly, we need to know whether to mark the square as an 'X' or an 'O'.

Our final function therefore will have three sub-requirements:

```
    [ ] Argument to specify X coordinate
    [ ] Argument to specify Y coordinate
    [ ] Argument to specify X/O to mark
```

Planning a project effectively requires you to put yourself in the mental state as though a given task were already finished, even though it isn't. To plan out the next step we need to imagine that all the functions we have already talked about are already built even though we haven't written them yet.

Consider now that we have a working function that can mark a square as an X or O anywhere we want. We also have from our previous lesson functions that can determine if a game is won, who the winner is, etc.

Let's take this lesson and the last lesson and write a small bit of pseudo-code that helps us to understand how our program will work in general whenever a player makes a move.

```
mark_square(...);
if (game_is_over() ) {
    if (winner_is_x() ) {
    ...
    if (winner_is_o() ) {
    ...
    if (game_is_tie() ) {
    ...
}
...
```

You should be able to see that every time we mark a square, or "make a move" in our tic-tac-toe game, we need to be able to see if the game is over, who the winner is, etc.

Think about this not as a programmer, but as a player of the game. When you first look at the tic-tac-toe game, you make a move. Then your opponent makes a move. What do you then do next? You check to see if the game is over, and if so who won.

Now, consider the natural hierarchy to this process:

Every time a move is made, we need to check to see if the game is over. Every time we check if the game is over, we need to see who won. Every time we need to see who won, we have to check for X, then O, then a tie.

Let's write out the above paragraph slightly different:

```
move is made
    -> is_game_over()
        -> is_winner_x()
        -> is_winner_o()
        -> is_game_tie()
```

You can see that a natural hierarchy forms without any effort on our part. Observe that I do not create a hierarchy of functions. The hierarchy creates itself, I simply must recognize it. Each time one task leads into another, you should structure your program with that hierarchy in mind. We can therefore build this into our functions as we write them.

For example, since every time we make a move we have to run the is_game_over() function, why not actually put the is_game_over() function into the mark_square() function?

Consider these two examples:

```
Figure (a) : Running all functions one after the other

mark_square(...);
if ( is_game_over() ) {
   if ( is_winner_x() ) {
      ...
   ...
}
```

The problem with this approach is that anywhere we use mark_square() in our program, we have to write out all that extra code related to checking if the game is over, etc. Similarly, every time we write out is_game_over() we have to write out all the code related to who the winner was. If at some point later on we had to change any of that code, we would have to manually change it everywhere we had done this. This would be tedious and frustrating.

However, we can construct these functions with this in mind very easily, thus saving us a great deal of work:

```
Figure (b) : Taking advantage of function hierarchy

int mark_square(int x, int y, ...) {
   .... code to mark the square goes here ...

   if ( is_game_over() ) {
      if ( winner_is_x() ) {
         ...
      }
   ...
   }
}
```

Notice that this is only possible because we are first planning out our project. If we had started the tic-tac-toe game by just writing code, we would not have become aware of the hierarchies that exist between functions until after we had already written those functions.

Observe what I have done. I have taken the mark_square() function and I have caused this function to call the is_game_over() function, and run the tests related to determining a winner.

Here you can see a simple example of function hierarchy. One function calls another. That function calls another. It is possible to write more powerful and complex programs by being able to take advantage of functions you have already written. By having these "layers" of functions, you can simply find new and creative uses for the functions you have already written.

Now I will show you a similar example taken from another application. If you have a function that can draw a single pixel, you could then have a function that can draw multiple pixels. If you have that function, you could have a function that can draw a single character on the screen. If you have that function, you can write a word, and if you have that function, you can write a paragraph. And so on.

Whenever you cause one function to call another, you are adding depth to the program you are writing. That depth will enable you to perform more complex tasks with less effort simply because you are using the power of the functions you have already written to achieve new tasks.

We will explore this more later in the course.

Lesson 4.4 : Introducing a new use for the While loop

One of the first kinds of loops we learned about was the while loop. In the last lesson I showed you that every time you mark a square in the tic-tac-toe game, you have to perform various actions such as checking if the game is over, etc.

Now, this next concept I am about to show you is very important in every application and game you will write. First, put yourself in the mental state of actually playing a tic tac toe game. Here is what happens:

```
Start tic-tac-toe game
Think about move <------------------------.
Make a move                               |
Wait for opponent to make a move          |
Run checks related to game over, etc.     |
Think about next move ---------------------'
```

Notice how there is a loop inherent in this process. It is a natural part of what it means to be playing a game, or really doing anything. If we were to write out this loop in pseudo-code, it would appear like this:

```
while (game is in progress) {
    think about next move;
    make move;
    wait for opponent to make move;
    check if game is over, who won, etc
}
```

And the final closing brace simply indicates to return back to the start of the loop. Let's examine the start of the loop again now:

```
while (game is in progress) {
```

Using what we learned a couple lessons ago, you should clearly see that we can re-write this as:

```
while (is_game_in_progress() )  {
}
```

Now we are using a function for this purpose. Therefore, we can construct a function whose job is simply to determine if the game is still in progress. If the game is still in progress, a whole set of processes can take place, repeat, and keep repeating until finally the game is over.

Let's list this as a requirement:

```
[ ] A function to determine if the game is still in progress, foruse with a while loop
```

This applies for applications as well as games. Any time you start any program, a similar loop is created. Until you exit out of that program, there is a continual process that is effectively saying "While the program is active, do this"

A program should not be thought of as merely a set of instructions to perform a task. Rather, you should also consider a program as a live process, that will stay "alive" until it is over. It therefore makes sense to have a loop which executes indefinitely until some condition is met where the program itself is over.

These kinds of loops can be thought of as the mechanism that keeps a program alive. In most applications, these kinds of loops exist within each other. For example, you could have the following:

```
while ( is_game_running() ) {
   start_level_1();

   while ( is_level_1_in_progress() ) {
      ...
      introduce_enemy_unit();

      while ( enemy_unit_is_alive() ) {
```

The above example works for games, but here is a similar example which works for let's say a drawing application:

```
while ( is_program_running() ) {
   new_drawing();

   while ( is_drawing_active() ) {
      load_paint_brush();

      while ( is_paint_brush_active() ) {
```

And so on. By created "nested" loops such as these, you can have processes that will continue to execute until a user chooses to stop them, or some condition is met.

This concept is also useful for algorithms that are designed to complete a complex task. Consider a sorting routine:

```
while ( is_data_sorted_yet() ) {
   ...
}
```

So the idea is that the "process" of sorting the data will remain "alive" until some point is reached where the data is finally sorted. At this point, the algorithm will stop.

Again just as with functions you can see that there are different "kinds" of loops. I am here introducing you to a while loop whose purpose is to keep the program itself, or some process within that program alive.

Notice also as I show you these concepts that writing a program is largely about recognizing where to apply the correct tools. It is not about "forcing a tool to work." The nature of the program will dictate what kind of tool you need. Planning a project is simply recognizing what tools you need at various points within the project.

Now, just as I showed you that you can have "kinds" of functions, such as functions to answer questions, I am also showing you that you can have "kinds" of loops. As far as a programming language is concerned, one while loop is really no different than any other. But as a programmer, you can be creative and apply different purposes to a loop.

In this case, any time you say "I need to keep this process alive until the user chooses to end it, or some condition is met" then a while loop is called for. Indeed, the very word "alive" may be enough to indicate the need for this kind of loop.

Lesson 4.5 : The basics of Rendering and Displaying data

It is possible to write a tic-tac-toe game that never actually displays a tic-tac-toe board. Similarly, it is possible to write a chess engine which never actually displays a chess board.

Data exists only within the computer as a sequence of 1s and 0s. Nothing says that a sound file has to be played or that a graphics file has to be displayed on the screen. Indeed, there are many applications which work with sound or graphics files that never have any need to display graphics or play sound.

The act of creating a "usable" image from raw data is known as "rendering". For example, you may have a 3D graphics object that exists in your computer's memory as data. That data would contain everything that can be known about that 3D object including all of its dimensions, colors, textures, and so on. However, until it is rendered it will remain just data. Rendering will convert that data into an image that can be displayed on your monitor.

Notice therefore that any data has to go through some rendering process before that data can be displayed or visualized. There are various ways to achieve this.

Suppose that you were writing a chess game. You therefore need to have some data format which stores the actual chess position at any time. With this data, it is possible to make moves, calculate positions, and everything else you may need to do. However, you cannot display the raw data.

You could however go into your favorite graphics program and draw out a chess board complete with texture, lines separating the squares, the proper colors, and so on. Next you could similarly draw out all of the chess pieces. Finally, you could have a function which reads your raw data of the chess position and then starts inserting chess piece graphics into the graphic of the blank chess board. The final result would be a fully rendered version of your chess position. This perfectly illustrates what I am trying to explain.

Now, we could choose to write our tic-tac-toe board in a way that it is already "display friendly". For example, we could write it like this:

```
_XO
_XX
X_O
```

Notice because of the \n characters, our tic-tac-toe board could easily be placed into a printf() and it would work just fine. It would be rather crude however, and there is not much more we can do with this data. It is much better to learn how to do this properly using the method I just described.

We could define our raw data instead like this:

```
_XO__XX_O
```

Now, let's create our display model for the tic-tac-toe board:

```
[ ][ ][ ]
[ ][ ][ ]
[ ][ ][ ]
```

There it is. When we display our tic-tac-toe board, we will be using this simple display model to do it. Think of this as the "blank chess board" I was talking about a few paragraphs ago. Let's construct our display model briefly in "C" :

```
char tictactoe_display_model[] = "[ ][ ][ ] [ ][ ][ ] [ ][ ][ ] ";
```

We have here created data which is ready to display correctly. In this case, it is a character array consisting of 30 characters. We can load this data exactly as is into a printf() statement and it will work fine.

Now all we need is a process which can take our tic-tac-toe board raw data and combine it with our display model to create what we will actual display to the screen. Let's visualize this process:

```
_XO__XX_O => [   ][X][O]
             [   ][   ][X]
             [X][   ][O]
```

Notice that the "data" itself is only 9 characters in size. I do not need to include any \n characters. Our display model is 30 characters in size. We can perform various manipulations on the data without affecting the display model. When we are ready to display the tic-tac-toe board, we can do so using a function.

Here is something to consider. The same tic-tac-toe board data: _XO__XX_O can just as easily be rendered into actual graphics. You could for example easily create a graphics file of a blank grid of 3x3 squares for a tic-tac-toe game. Then you could write a function which goes and draws actual X and O graphics into those squares. It is not difficult. We may in fact visit this later in the course.

This same concept applies with web-based applications. You can write out a web page in simple HTML which has no "moving parts", and just have "place holders" where the actual data goes. Consider this simple example:

Hello {name}, and welcome to our website!

That is your "display". You could easily have a function which converts {name} into the person's actual name by doing a lookup from some database. We will go over this later in the course.

Lesson 4.6 : A simple rendering algorithm

In the last lesson I showed you that we need to have some function which can combine our "raw data" with our "display model" to create a usable final result that can be displayed.

Here is how this process is intended to work:

```
_XO_XXX_O => [  ][X][O]
             [  ][X][X]
             [X][  ][O]
```

Now, let's redefine this task in a different way. We need to create a function which when given this as input:

```
_XO_XXX_O
```

Produces this as output:

```
      [ ][X][O]
      [ ][X][X]
      [X][ ][O]
```

How do we convert the raw data to a usable version that can be displayed.

Let's think of this process another way. We are to some extent creating a sort of mathematical operation, that looks like this:

```
_XO_XXX_O + [ ][ ][ ] = [  ][X][O]
            [ ][ ][ ]   [  ][X][X]
            [ ][ ][ ]   [X][  ][O]
```

Let's rewrite this as:

A + B = C

A is our "raw data". B is our rendered blank tic-tac-toe board. C is the final result which is achieved by "joining together" A and B.

Let's first evaluate the data format itself: _XO__XX_O

Every character in this simple text string corresponds to a square on the final tic-tac-toe board. We know there are three possibilities. An underscore character means that we "do nothing". That is to say, we leave the square blank. An 'X' means that we are going to place an 'X' into that square, and an 'O' means that we are going to place an 'O' into that square.

We know that we have to do this one at a time for each character. We can write out a simple loop for this in pseudo-code like this:

```
for each character :
    if 'X' : place 'X' into proper square
    If 'O' : place 'O' into proper square
    proceed to next character.
```

Notice that I do not have any operation to take place if the square is blank. Whenever you are writing an algorithm, speed is important. By not writing some action to take if the square is blank, I am saving time.

Now, from what you learned a couple of lessons ago, you should understand that we have a process here that needs to stay "alive" until it is finished. In this case, we are saying "While there are still squares left to render... render the next one."

Let's re-write this as a while loop in pseudo-code:

```
while (there are squares left to render) {
    if 'X' : mark square as 'X'
    if 'O' : mark square as 'O'
}
```

So this algorithm is going to "stay alive" until the last square is rendered. Now all we need is a mechanism to "mark the square". Remember what I said in an earlier lesson: You always understand an algorithm by starting with the first iteration.

The same applies when designing an algorithm as it applies when reading one. Therefore, let's consider only the FIRST iteration of this algorithm. Here is our raw data _XO__XX_O. The first iteration is only going to be concerned with the FIRST character. In this case, a _ character.

You should clearly see then that this will be skipped, and we will then proceed to the next character, an X. So let's examine that.

Our 'X' character needs to be rendered in our display model. Let's again look at our display model:

```
[ ][ ][ ]
[ ][ ][ ]
[ ][ ][ ]
```

Alright, but now let's look at it the way C looks at it:

```
[ ][ ][ ]  [ ][ ][ ]  [ ][ ][ ]
```

The key point to understand here is that each space within the brackets is an "insertion point" where a rendered X or O can be placed. It is therefore important to identify all of these points. I am going to place a hash mark inside all insertion points to make this clearer.

```
[#][#][#]  [#][#][#]  [#][#][#]
```

Now, after our first iteration (which was an underscore), we will have left the first of these insertion points alone. It would have stayed blank. Therefore, after the first iteration, our display model would look like this:

```
After 1st iteration : [ ][#][#]  [#][#][#]  [#][#][#]
```

Notice that the remaining # characters indicate the insertion points not yet processed. Now we are on our second iteration. For this next insertion point, we are going to render an X into that square.

```
After 2nd iteration : [ ][X][#]  [#][#][#]  [#][#][#]
```

Now, let's identify the position where we just placed the 'X'. Remember that array[0] is the first character of the array, which in this case is an opening bracket character: '[' . array[1] is the first insertion point. array[2] is going to be a ']' character, then array[3] is a '[' character, and array[4] is the insertion point we just placed an X into.

If that was confusing, compare what I just said to this:

```
0 1 2 3 4 5
[    ] [  X ]
```

In other words, our second iteration came down to executing this instruction:

```
array[4] = 'X';
```

Now, let's identify the actual position for all of these "hash marks" Just to make counting the positions easier, I have replaced the "\n" with '$' characters. This way we do not accidentally count a \n as two characters.

```
[#][#][#]$[#][#][#]$[#][#][#]$
```

The locations of each hash mark above are:

```
#1 : array[1]
#2 : array[4]
#3 : array[7]

#4 : array[11]
#5 : array[14]
#6 : array[17]

#7 : array[21]
#8 : array[24]
#9 : array[27]
```

Do you notice a pattern here? Each set of three increases by exactly ten from where the previous set began. We start at 1, then 11, then 21. Similarly, within each set of three, the next hash mark is located exactly three characters ahead of the previous character.

Whenever you notice a pattern, you should consider ways to incorporate that into your algorithm design. Here we basically need two loops inside each other that will hit each hash mark:

```
for (i = 0; i <= 2; i++) {
    for (j = 1; j <= 7; j+=3) {
        hash is array[ (i * 10) + j ]
    }
}
```

Understanding this algorithm is easy if you start with the first iteration. On the first iteration, this is all you have to consider:

```
i = 0
   j = 1
      hash is array[ (i * 10) + j ] OR
      hash is array[0 + 1] OR

      hash is array[1]
```

So the first time this executes, it will hit the first hash mark, which is array[1]. Let's consider the next iteration:

```
i = 0
   j = 4
      hash is array[4]
```

Notice that with any algorithm you process the inner most loop first. Therefore, i will remain zero. j will be 4 because you have added three to what j used to be. That is the meaning of j+=3, it means "j becomes j plus three". Let's now go to the third iteration:

```
i = 0; j = 7
array[7]
```

And now the fourth. Here we have reached the condition of the inner loop (j is now <= (which means less than OR equal to) seven). So now we can say the following:

```
i = 1; j = 1;
   array[ (i * 10) + j ] OR...
   array[10 + j] OR...
   array[11]
```

Now the instruction "i++" (which means add 1 to the variable 'i') executes. Therefore i changes from 0 to 1.

Notice that j also gets reset to 1. Any time the inner loop finishes, it will be reset to the starting point. Observe how easy it is to understand this algorithm if you simply take it one iteration at a time, without worrying about the complex "for" loop syntax.

Remember that our for loop is only dealing with two simple variables: i and j. They each follow a set pattern. Finally, we are using a basic mathematical formula that is only: (i * 10) + j. Therefore, it is quite easy to understand this algorithm through all of its iterations:

```
iteration #1 : i=0; j=1;     array[1]
iteration #2 : i=0; j=4;     array[4]
iteration #3 : i=0; j=7;     array[7]
iteration #4 : i=1; j=1;     array[11]
iteration #5 : i=1; j=4;     array[14]
iteration #6 : i=1; j=7;     array[17]
iteration #7 : i=2; j=1;     array[21]
iteration #8 : i=2; j=4;     array[24]
iteration #9 : i=2; j=7;     array[27]
```

Now looking at this, consider the for loop earlier.

```
for (i = 0; i <= 2; i++) {
```

That should make sense to you as you look at the values for the variable 'i' in the above table. The variable 'i' starts at zero. Then each time that loop finishes, 'i' increases by one (the meaning of i++). This proceeds so long as the variable 'i' is less than or equal to 2. Similarly:

```
for (j = 1; j <= 7; j+=3) {
```

When you look at the values for 'j' above, that should make sense to you. The variable 'j' starts at 1. Then each time that loop finishes, 'j' increases by three (the meaning of j+=3). This proceeds so long as the variable 'j' is less than or equal to 7.

Here you can see that we have constructed an algorithm which is capable of going through and precisely hitting each insertion point that we will be replacing with either an 'X' or an 'O'. If this process still seems a bit like black magic, let me describe to you exactly how this was done:

1. First you must look closely at the display model. In this case, our display model was: [ ][ ][ ][ ][ ][ ][ ][ ][ ].

2. Then you must identify all of the points in that model which will need to be "hit" by the algorithm you are designing. In this case we found they were: 1, 4, 7, 11, 14, 17, 21, 24, 27.

3. Next, you look for patterns. There will certainly be a pattern simply because each insertion point is a set distance from another one. Also, each set of three is a set distance from the others. These two facts should tell you that you need a for loop consisting of two variables.

4. Then, you work through the algorithm yourself as though you were the program.

5. Finally you write the actual algorithm, and mentally test it through each iteration.

In the next lesson I will show you how to take this algorithm and create a function that can render and display our raw tic-tac-toe board data.

Lesson 4.7 : Our final tic-tac-toe board display function

In the last lesson I showed you the basic algorithm we need in order to locate all of the different points that we want to write an 'X' or a 'O' into our "display model". In this lesson, I am going to show you the complete function to do this.

First, let's recall how this process works.

```
_XO_XXX_O + [ ][ ][ ] = [ ][X][O]
            [ ][ ][ ]   [ ][X][X]
            [ ][ ][ ]   [X][ ][O]
```

Now, let's go ahead and write these two strings out in "C":

```
char raw_data[] = "_XO_XXX_O";
char display_model[] = "[ ][ ][ ] [ ][ ][ ] [ ][ ][ ] ";
```

Now we already know the algorithm which will "hit" all of the spaces inside our display model, so let's write it out:

```
for (i = 0; i <= 2; i++) {
    for (j = 1; j <= 7; j+=3) {
        ... array[ (i * 10) + j ] ...
    }
}
```

Now we are ready to begin.

In order to make this algorithm effective, we only need a way to map the correct location in the raw data with the correct location in the display model. We already have from the previous lesson exactly how this works.

The first character from our raw data will go in position array[1] with our display model. In this case of course, we would not say array[1] we would say display_model[1]. Now, the second character of our raw_data would go in position display_model[4] and so on, just like we saw in the last lesson.

Let me draw a simple table showing this:

```
raw_data[0] => display_model[1]
raw_data[1] => display_model[4]
raw_data[2] => display_model[7]
raw_data[3] => display_model[11]
raw_data[4] => display_model[14]
raw_data[5] => display_model[17]
raw_data[6] => display_model[21]
raw_data[7] => display_model[24]
raw_data[8] => display_model[27]
```

Seeing patterns is absolutely a critical skill for a programmer. Here you should see three distinct patterns. The raw_data has some number that is continually increasing by one. The display_model has two variables, such that you can always say [(i * 10) + j]. Therefore, this algorithm is going to require three variables.

We have already taken care of i and j. Now we need a third variable which will simply increase by one with each iteration. Let's look again at our for loop structure:

```
for (i = 0; i <= 2; i++) {
   for (j = 1; j <= 7; j+=3) {
      ... array[ (i * 10) + j ] ...
      ... somehow here we need a third variable for raw_data ...
   }
}
```

Now, let's call this third variable k. Then it is easy to see that the inside part of this for loop will look like this:

```
for (i = 0; i <= 2; i++) {
   for (j = 1; j <= 7; j+=3) {
      display_model[ (i * 10) + j ] = raw_data[k];
   }
}
```

The last thing we need to do is simply create k. Well, k is going to be a variable that will start at zero. It will increase by one with every iteration. That gives us two of the key questions we need for any loop. However, when do we know to stop? Do we write "where k is less than 9" ? We could, but we do not really need to.

You see, k will automatically stop when i and j stop. You will be surprised how simple this is to implement:

```
int i = 0;
int j = 0;
int k = 0;

for (i = 0; i <= 2; i++) {
   for (j = 1; j <= 7; j+=3) {
      display_model[ (i * 10) + j ] = raw_data[k++];
   }
}
```

And we are done. By placing "k++" inside of the raw_data index, we have achieved everything we need. The key point to remember here is that we are setting a starting point for k as zero. We do not need to create a conditional statement for k because this is already taken care of simply because of i and j. Meaning, that the for loop will already execute the correct number of times. The last thing we need to do is make sure that k increments with each iteration, and this is done by saying k++.

Note that:

```
display_model[ (i * 10) + j] = raw_data[k++];
```

is the same as:

```
display_model[ (i * 10) + j] = raw_data[k];
k++;
```

Now let's observe the final process:

```
#include <stdio.h>

int main(void) {

    char raw_data[]        = " XO XXX O";
    char display_model[]   = "[ ][ ][ ] [ ][ ][ ] [ ][ ][ ] ";

    int i, j, k; k=0;

    for (i = 0; i <= 2; i++) {
        for (j = 1; j <= 7; j+=3) {
            display_model[ (i * 10) + j ] = raw_data[k++];
        }
    }

    printf("%s ", display_model);

}
```

You will notice that I used a simple shortcut for creating our i, j, and k variables. Even though you should usually initialize a variable before you use it, you can make exceptions for simple loops. This is because I am initializing i and j in the very next lines of code. Also, notice I set k to zero.

I also made one other small change. I removed the underscores and replaced them with spaces. This removes the need to have some kind of process to check if there is an underscore, or an X, or an O. In other words, instead of an underscore character meaning "nothing", we are using the space for the same purpose. It changes nothing concerning the process involved, it simply speeds it up.

It would be trivial to modify this function to use underscores instead of spaces. You could just add an if() statement that would skip over that iteration if an underscore were present, and just add one to the k variable. For completeness, here is the algorithm with that modification:

```
    for (i = 0; i <= 2; i++) {
        for (j = 1; j <= 7; j+=3) {
            if (raw_data[k] != '_') {
                display_model[ (i * 10) + j ] = raw_data[k++];
            } else {
                k++;
            }
        }
    }
```

This simply translates to "skip to the next k iteration if this is an underscore."

Now we can easily transform this algorithm into a function and we have a proper way to display our tic-tac-toe board based on the raw data, like so:

```
#include <stdio.h>

int main(void) {

    char raw_data[]         = " XO XXX O";

    display_board(raw_data);

    return 0;

}

void display_board(char *raw_data) {
    char display_model[]    = "[ ][ ][ ] [ ][ ][ ] [ ][ ][ ] ";

    int i, j, k; k=0;

    for (i = 0; i <= 2; i++) {
        for (j = 1; j <= 7; j+=3) {
            display_model[ (i * 10) + j ] = raw_data[k++];
        }
    }

    printf("%s ", display_model);
}
```

Lesson 4.8 : Function to evaluate a won position

There are exactly three ways that a tic-tac-toe position can be won: Horizontal, Vertical, or Diagonal.

Let's consider the raw_data format we used in the previous lesson. The first and most obvious won position would look like this:

```
XXX => raw_data = "XXX______";
___
___
```

Let's look at the array index positions for our tic-tac-toe board:

```
012
345
678
```

In other words, if raw_data[0], [1], and [2] would be set to 'X' (or 'O'), then a win exists.

There are three possibilities for a horizontal win for 'X'. They are:

```
raw_data[0], [1], [2] == 'X'
raw_data[3], [4], [5] == 'X'
raw_data[6], [7], [8] == 'X'
```

Do you see any patterns? If we think of this table as having three rows, then the start of each row is exactly three greater than the previous row. Inside each row, the next position is exactly one greater than the previous.

We could write this out in pseudo-code like this:

```
for (i = 0; i <= 6; i+=3) {
    ... test: raw_data[i], raw_data[i + 1], raw_data[i + 2] ...
}
```

That would go through all three rows and test to see if we have a horizontal won position. All it is really saying is this:

```
start at position [0] (the start of the first row)
check to see if position [0], [1], and [2] are set
jump to the next row (by adding three) and repeat
```

So to check to see if a horizontal win exists, we simply need to test position [0], [1], [2] and then repeat this test after adding 3. We do this for all three rows and we have finished. Now this test can be easily done using an if statement for the first row, like so:

```
if (raw_data[0] == 'X' && raw_data[1] == 'X' && raw_data[2] == 'X'){
    ... horizontal win exists for 'X' ...
}
```

This would test the first row for a horizontal win. Notice that the nature of the '&&' operation means that raw_data[2] == 'X' will only take place if the other two tests confirmed. Remember that we learned this in a previous lesson. Therefore, this algorithm will not perform all three tests every time it runs, but only those times where the first two are already set to 'X'.

Now we just need to convert that if() statement to something that works for all three rows, like so:

```
int i;

for (i = 0; i <= 6; i+=3) {
    if (raw_data[i] == 'X' && raw_data[i+1] == 'X' && raw_data[i+2]== 'X') {
        ... horizontal win exists ...
    }
}
```

Now, how can we do the same thing for a vertical win? Again we have to clearly define what a vertical win is. Let's look at a simple example:

```
X__ => raw_data[] = "X__X__X__"
X__
X__
```

So here we see that a vertical win exists where there are 3 'X' (or 'O') that are exactly 3 apart. If you find an 'X' on the first row, you can simply add 3 and see if you have another 'X'. If you repeat this process one more time and find an 'X', you know that a vertical win exists.

Let's convert this to an algorithm. First, let's consider the first iteration:

```
if (raw_data[0] == 'X' && raw_data[3] == 'X' && raw_data[6] == 'X'){
    ... vertical win exists ...
}
```

Now, let's convert it to a full algorithm with 3 iterations, one for each column we are testing:

```
for (i = 0; i <= 2; i++) {
    if (raw_data[i] == 'X' && raw_data[i+3] == 'X' and raw_data[i+6]== 'X') {
        ... vertical win exists ...
    }
}
```

And we are done.

Lastly, we need to evaluate a diagonal win. In this case, it is easiest to just define with a single if statement what we want. There are only two possibilities for a diagonal win, and there is no need for a for loop:

```
012 => the raw_data[] positions for our tic-tac-toe board
345
678
```

```
if (raw_data[0] == 'X' && raw_data[4] == 'X' && raw_data[8] == 'X'){
if (raw_data[2] == 'X' && raw_data[4] == 'X' && raw_data[6] == 'X'){
```

Notice that both of these if statements have raw_data[4] in common. Therefore, it makes sense to test raw_data[4] first. Remember that raw_data[4] corresponds to the center square on the 3x3 grid.

```
if (raw_data[4] == 'X') {
    if (raw_data[0] == 'X' && raw_data[8] == 'X') {
        ... diagonal win exists ...
    }
    if (raw_data[2] == 'X' && raw_data[6] == 'X') {
        ... diagonal win exists ...
    }
}
```

Here we are saying that we test the center square first. If the center square is not marked, there is no need to test further.

All we have to do now is put all of this together into a simple function. First we should consider what parameters do we need to send to this function? Certainly we need to send raw_data, the actual tic-tac-toe board position to evaluate. But we also should send whether we are testing 'X' or 'O'. We could therefore create the function like this:

```
int is_winning_position(char *raw_data, char player) {
    int i;

    // Test for horizontal win
    for (i = 0; i <= 6; i+=3) {
        if (raw_data[i] == player && raw_data[i+1] == player &&raw_data[i+2] == player) {
            return 1;
        }
    }

    // Test for vertical win
    for (i = 0; i <= 2; i++) {
        if (raw_data[i] == player && raw_data[i+3] == player &&raw_data[i+6] == player) {
            return 1;
        }
    }

    // Test for diagonal win
    if (raw_data[4] == player) {
        if (raw_data[0] == player && raw_data[8] == player) {
            return 1;
        }
        if (raw_data[2] == player && raw_data[6] == player) {
            return 1;
        }
    }

    return 0;

}
```

Do not let the == player confuse you. We are simply using the word "player" in place of whatever character was sent. If 'X' was sent, then player becomes 'X'. If 'O' was sent, then player becomes 'O'.

I used one other trick here. I return 1 every time a win is confirmed. I return 0 at the end. This means that the only way this function will return 0 is if it has exhausted all possibilities of a win. In other words, this function will test all possibilities for a "win" for the player we give it ('X' or 'O'). If any win is found, it will return 1 (true). Then after all such possible wins, it will return 0 only if no win is found.

This means we can use our function as a question in an if statement like this:

```
if (is_winning_position(raw_data, 'X')) {
    printf("X has won!");
}
```

You can very easily read that as: "If this is a winning position for 'X'. If and only if the function returns a 1 then this if statement will evaluate as true.

Lesson 4.9 : About Functions and Return Values other than 1 and 0

In the last lesson I showed you how to write a simple function to evaluate a won position for any tic-tac-toe position. This function returns 1 every time a position is won. It achieves exactly the purpose we gave it in that it can tell us perfectly if a position is won for either 'X' or 'O'.

What it cannot tell us however is how the position was won. Was it won vertically, horizontally, or diagonally? We could write another function that is designed to tell us this detail, but it turns out there is a much easier way.

Remember earlier I told you how the "zero flag" works. Really, your computer cares either that the zero flag is set or that the zero flag is not set. There is no other option.

In other words, there are two real possibilities so far as your computer is concerned: "It is zero." or "It is not zero."

Put another way, there are two possibilities: zero and non-zero. If a conditional statement returns any non zero value then it will be treated as true. Observe these examples:

```
if (5) { printf("Five  "); }
if (-3) { printf("Negative Three  "); }
if (0) { printf("Zero  "); }
```

Output:

```
Five
Negative Three
```

Notice then that the only result which does not output is "Zero". If an if statement evaluates as "zero" then it is false. If it evaluates to any non zero value it will evaluate as true. We can take advantage of this to get more details from our function.

Instead of writing return 1; each time a win is confirmed, we could return any non zero integer value. We could therefore create a simple mapping between the possibilities:

0 : No win is detected
1 : A horizontal win is detected
2 : A vertical win is detected
3 : A diagonal win is detected

Keep in mind that if we return a 1, or a 2, or a 3, the same if statement we used earlier will evaluate:

```
if (is_winning_position(raw_data, 'X')) {
    ... this will work for ANY non-zero value ...
}
```

That means inside of that if statement, we can add additional if statements to see the type of win that resulted:

```
if (int return_value = is_winning_position(raw_data, 'X')) {
    if (return_value == 1) {
        printf("A horizontal win resulted for X!");
    }
    if (return_value == 2) {
        printf("A vertical win resulted for X!");
    }
}
```

This is useful. However, we can get even more detailed if we want. There are exactly three possibilities for a vertical win. Three possibilities for a horizontal win, and two possibilities for a diagonal win. That makes 8 total possibilities.

We could therefore create a map of all eight possibilities. However, that would require we have to significantly modify our "is_winning_position" function. That means we would have to slow down the algorithm to test for these eight possibilities.

Is there another way we can do this? Remember that each test uses a for loop to go through columns and rows. We therefore know what number of row (0, 1, 2) or what column (0, 3, 6) we are dealing with. Observe now how we can put all of this information together to get much more detailed information about won positions:

```
int is_winning_position(char *raw_data, char player) {

    int i;

    // Test for horizontal win
    for (i = 0; i <= 6; i+=3) {
        if (raw_data[i] == player && raw_data[i+1] == player &&raw_data[i+2] == player) {
            return 10 + i;
        }
    }

    // Test for vertical win
    for (i = 0; i <= 2; i++) {
        if (raw_data[i] == player && raw_data[i+3] == player &&raw_data[i+6] == player) {
             return 20 + i;
        }
    }

    // Test for diagonal win
    if (raw_data[4] == player) {
        if (raw_data[0] == player && raw_data[8] == player) {
            return 31;
        }
        if (raw_data[2] == player && raw_data[6] == player) {
            return 32;
        }
    }

    return 0;

}
```

Notice that 10 + i; will translate to 10 for the first row, 13 for the second, and 16 for the third. Similarly 20 + i will translate to each column. Finally 31 for the upper left to lower right diagonal, and 32 for the other diagonal.

All of these return values will evaluate as "True" by our if statement simply because they are non-zero values. We can now see not only if a position is won, but exactly how the win occured. Also by using 10, 20, and 30 as "starting points", we can easily determine whether or not the win was horizontal, vertical, or diagonal.

Lesson 4.10 : Introducing Switch and Case

In the last lesson I showed you that it is often useful to cause a function to return additional values other than 1 and 0. You should realize that it would be quite tedious to write individual if statements for each possible return value. Also, so many if statements creates code that is somewhat difficult to read.

It turns out there is a short-hand method for doing this, and it is present in just about every programming language. This method is known as "switch" and "case". The idea is simple: Rather than you having to write individual if statements to test specific values for a given data item, you can write a "switch" statement instead. Here is how this works:

```
if (i == 1) {
    printf("The value is one  ");
} else
if (i == 2) {
    printf("The value is two  ");
} else
if (i == 3) {
    printf("The value is three  ");
}
```

Can become:

```
switch (i) {
    case 1 : printf("The value is one  "); break;
    case 2 : printf("The value is two  "); break;
    case 3 : printf("The value is three  "); break;
}
```

So the syntax is simple. You write the word "switch" followed by the data item you are going to perform all the tests on. For each test, you use the "case" statement. In our previous example, we might do something such as this:

```
int won_position_return_value = is_winning_position(raw_data, 'X');

switch (won_position_return_value) {
    case 10 : printf("Horizontal win on Row #1"); break;
    case 13 : printf("Horizontal win on Row #2"); break;
    case 16 : printf("Horizontal win on Row #3"); break;
}
```

Of course we could replace the printf() statements with a block of code. The idea is simple. With a "switch" statement we can replace a lot of "if" statements with something that is much easier to read.

What happens if none of the "case" statements apply? Then a "default" statement is used. Here is an example of default in action:

```
int i = 5;

switch (i) {
    case 2 : printf("This won't print  "); break;
    case 4 : printf("This won't print either  "); break;

    default : printf("This will print because the other cases failed.  "); break;
}
```

Now with this in mind we can create another interesting function for our tic-tac-toe game. This function would be designed to display additional winning information based on the return value received from the "is_winning_position" function. It would work like this:

```
void show_win_details(int win_value, char player) {

    switch (win_value) {

        // Horizontal
        case 10 :
            printf("Horizontal win on first row for Player: %c  ",player);
        break;
        case 13 :
            printf("Horizontal win on second row for Player: %c  ",player);
        break;
        case 16 :
            printf("Horizontal win on third row for Player: %c  ",player);
        break;

        // Vertical
        case 20 :
            printf("Vertical win on first column for Player: %c  ",player);
        break;
        case 21 :
            printf("Vertical win on second column for Player: %c  ",player);
        break;
        case 22 :
            printf("Vertical win on third column for Player: %c  ",player);
        break;

        // Diagonal
        case 31 :
            printf("Diagonal win upper left to lower right for Player: %c  ", player);
        break;
        case 32 :
            printf("Diagonal win lower left to upper right for Player: %c  ", player);
        break;

        default: printf("Some error occurred.  "); break;

    }
}
```

Lesson 4.11 : Calculating a winning move

In the last lesson I showed you how to write a function that can detect whether or not a position is won for either 'X' or 'O'.

How could we write a function that can find a winning move? For example, let's assume the following tic-tac-toe board:

```
[X][  ][  ] => 012
[  ][  ][  ] => 345
[X][  ][  ] => 678
```

Here it is obvious that the winning move is "3". If we intend to have an "artificial intelligence" engine that can win vs a human player, it must be capable of playing the "final winning move".

Remember in earlier lessons I explained that you can use functions you have already built in order to make more powerful functions possible. This is such a case. Because we have a function that can evaluate whether or not a position is won or not, we can easily write a function that will play a winning move.

The way it works is simple. We just need to play all possible moves, and then evaluate if any of them are winning.

Let's look at a sample tic-tac-toe position:

```
[X][  ][X] => X_XO__O__
[O][  ][  ]
[O][  ][  ]
```

There are five possible moves that can be played: 1, 4, 5, 7, and 8.

In pseudo code, we would play the winning move for the above position like this:

```
play move 1
is it a winning position? If so, game over. If not:
play move 4
... repeat this process for 1, 4, 5, 7, and 8
```

One thing you will notice is that it will be important to "play a move" without actually playing it. In other words, the computer will evaluate the position that will result had the move actually been played, but it will not need to play the move. This is similar to how a human player would think about possible moves before actually making one.

Let's go back to the raw data:

```
"X XO  O  " (Remember we are using spaces)
```

Watch how simple this is:

```
char raw_data[] = "X XO  O  ";
char test_position[10];
```

```
int i, win_result;

for (i = 0; i < 9; i++) {
    if (raw_data[i] == ' ') {
        strcpy(test_position, raw_data);
        test_position[i] = 'X';
        win_result = is_winning_position(test_position, 'X');
        printf("The result of playing X at position %d is: %d  ", i,win_result);
    }
}
```

Output:

```
The result of playing X at position 1 is: 10
The result of playing X at position 4 is: 0
The result of playing X at position 5 is: 0
The result of playing X at position 7 is: 0
The result of playing X at position 8 is: 0
```

Now we can just cut-paste this code into a function:

```
int find_winning_move(char *raw_data) {
    char test_position[10];
    int i, win_result;

    // Go through all possible squares
    for (i = 0; i < 9; i++) {

        // Determine if that square is empty
        if (raw_data[i] == ' ') {

            // Copy the actual board into the "test_position"
            strcpy(test_position, raw_data);

            // Play 'X' at that square
            test_position[i] = 'X';

            // Check to see if this is a winning move or not
            win_result = is_winning_position(test_position, 'X');

            // Printf similar to: The result of playing X at position 1 is: 10  (non-zero = win)
            printf("The result of playing X at position %d is: %d  ", i, win_result);
        }
    }

    return win_result; // This is not quite finished yet, as you will see in upcoming lessons.
}
```

We create test_position to be a temporary tic-tac-toe board that the computer player can try various moves on without affecting the actual game. We then copy the current board position into the test_position using strcpy(). Finally we obtain the result of the "is_winning_position" function we wrote in the last lesson. We do this for each possible move, and therefore we can know if any of the possible moves are winning. Notice that we say if raw_data[i] == ' '. Remember that space ' ' means a move has not yet been played in that position. That if statement is simply saying "If the square is empty."

Because we now have a function that can calculate a winning move one level deep, we could easily create a function that can calculate a winning move two levels deep, if there is one.

```
[X][   ][   ] => 012
[O][   ][X] => 345
[O][   ][   ] => 678
```

Here there exists a winning move for 'X', two actually. If 'X' were to play at position #2 or at position #8, the game is over. Let's create a function which can calculate this:

```
char raw_data[] = "X  O XO  ";
```

How can we modify our function so that it can find a winning move two levels deep? That is the subject of the next lesson.

Lesson 4.12 : A snapshot of current progress

Before we go into the next lesson, I believe that it is important that we take a snapshot of where we are now. I am pasting a "working" program below in its entirety, and in the next lesson I will go through it piece by piece to ensure that everyone understands what is going on.

```
#include <stdio.h>
#include <string.h>

int find_winning_move(char *, char, int);
int display_board(char *);
int is_winning_position(char *, char);
void show_win_details(int, char);

int main(void) {
    int retval = 0;
    char raw_data[]        = "X  X XO  ";
    char player = 'X';

    printf("We are examining this board:  \n");
    display_board(raw_data);

    find_winning_move(raw_data, player, 1);

    return 0;
```

```
}

int find_winning_move(char *raw_data, char player, int depth) {

    char test_position[10];
    int i, win_result;

    for (i = 0; i < 9; i++) {
        if (raw_data[i] == ' ') {
            strcpy(test_position, raw_data);
            test_position[i] = player;
            win_result = is_winning_position(test_position, player);
            printf("The result of playing %c at position %d is: %d  \n",
                player, i, win_result);

            display_board(test_position);
        }
    }

    return 0;
}

int display_board(char *raw_data) {
    char display_model[]     = "[ ][ ][ ]\n[ ][ ][ ]\n[ ][ ][ ]\n";

    int i, j, k; k=0;

    for (i = 0; i <= 2; i++) {
        for (j = 1; j <= 7; j+=3) {
            display_model[ (i * 10) + j ] = raw_data[k++];
        }
    }

    printf("%s ", display_model);
}

int is_winning_position(char *raw_data, char player) {

    int i;

    // Test for horizontal win
    for (i = 0; i <= 6; i+=3) {
        if (raw_data[i] == player
            && raw_data[i+1] == player
            && raw_data[i+2] == player)
        {
            return 10 + i;
        }
    }

    // Test for vertical win
    for (i = 0; i <= 2; i++) {
        if (raw_data[i] == player
            && raw_data[i+3] == player
            && raw_data[i+6] == player)
        {
             return 20 + i;
        }
    }

    // Test for diagonal win
    if (raw_data[4] == player) {
        if (raw_data[0] == player && raw_data[8] == player) {
            return 31;
        }
        if (raw_data[2] == player && raw_data[6] == player) {
            return 32;
        }
    }

    return 0;

}

void show_win_details(int win_value, char player) {

    switch (win_value) {

        // Horizontal
        case 10 :
            printf("Horizontal win on first row for Player: %c  \n",
                player);
        break;
        case 13 :
            printf("Horizontal win on second row for Player: %c  \n",
                player);
        break;
        case 16 :
            printf("Horizontal win on third row for Player: %c  \n",
                player);
        break;

        // Vertical
        case 20 :
            printf("Vertical win on first column for Player: %c  \n",
                player);
        break;
        case 21 :
            printf("Vertical win on second column for Player: %c  \n",
                player);
        break;
        case 22 :
            printf("Vertical win on third column for Player: %c  \n",
                player);
        break;
        // Diagonal
        case 31 :
          printf("Diagonal win upper left to lower right for Player: %c  \n",
             player);
        break;
        case 32 :
          printf("Diagonal win lower left to upper right for Player: %c  \n",
             player);
        break;

        default: printf("Some error occurred.  \n"); break;

      }
   }
```

Sample Output:

```
We are examining this board:
[X][  ][  ]
[X][  ][X]
[O][  ][  ]

The result of playing X at position 1 is: 0
[X][X][  ]
[X][  ][X]
[O][  ][  ]

The result of playing X at position 2 is: 0
[X][  ][X]
[X][  ][X]
[O][  ][  ]

The result of playing X at position 4 is: 13
[X][  ][  ]
[X][X][X]
[O][  ][  ]

The result of playing X at position 7 is: 0
[X][  ][  ]
[X][  ][X]
[O][X][  ]

The result of playing X at position 8 is: 0
[X][  ][  ]
[X][  ][X]
[O][  ][X]
```

Notice therefore that this program takes any given tic-tac-toe position, and evaluates the end result of playing any of the available moves, one at a time. A non-zero result means that a win has been detected.

Lesson 4.13 : Better understanding of Analysis Functions

Before we begin this lesson, I strongly suggest you review the full sample program in Lesson 4.12.

Now, let's begin. There are a lot of different kinds of functions you might write. Some of these functions can be classed into specific roles, and one of the roles a function may have is to analyse data in order to give you information that you can use later in your program. In general, I will refer to these kinds of functions as "Analysis Functions". An "Analysis Function" is a function whose purpose is to look at some data and give you useful information.

To better understand this concept, imagine that you wish to hire a house inspector. That inspector is going to show up at your house with a clipboard full of forms he intends to fill out. Then he will go through your house filling in these forms one item at a time until all of the forms have been filled out. When he is finished, he will leave your house with several sheets of paper which will give him all of the information he needed. This is a simple analogy to how an analysis function works.

Now think about something: Your house is relatively large, and the few sheets of paper he leaves with are relatively small. Only a few sheets of paper are able to describe in great detail a much larger "thing", in this case your house. This concept of analyzing a large amount of data and producing a small useful result is useful not just in programming, but for programmers it is a fundamental concept.

Just about any useful program will require some functions to analyze data. Generally, these functions will operate in the following way:

1. They are directed to the start of the data.
2. They will go through the data and run through series of questions.
3. The answers to each question are kept.
4. Once all of the answers have been collected, a "final answer" is determined.
5. That final answer is returned.

In our last example, we were able to show how a program can evaluate a tic-tac-toe board to determine whether or not a given move would result in a win. To do this, we used the "is_winning_position" function, which is an analysis function like what I am describing. This function is able to read any tic-tac-toe board position, and return a simple two-character code which gives us all of the information we need.

However, this function does not interpret the result for us. It will return a 0 for example, but it will not explain that 0 means "no win". It may return a 13, but it does not explain that the "13" means "horizontal win". This is not a problem however, because we have another function which is able to interpret the results. In these lessons, we are going to modify our tic-tac-toe program to include these interpretations and thus make it more useful. Let's look at this function:

```
int is_winning_position(char *raw_data, char player) {

    int i;

    // Test for horizontal win
    for (i = 0; i <= 6; i+=3) {
        if (raw_data[i] == player
            && raw_data[i+1] == player
            && raw_data[i+2] == player)
        {
            return 10 + i;
        }
    }

    // Test for vertical win
    for (i = 0; i <= 2; i++) {
        if (raw_data[i] == player
            && raw_data[i+3] == player
            && raw_data[i+6] == player)
        {
             return 20 + i;
        }
    }

    // Test for diagonal win
    if (raw_data[4] == player) {
        if (raw_data[0] == player && raw_data[8] == player) {
            return 31;
        }
        if (raw_data[2] == player && raw_data[6] == player) {
            return 32;
        }
    }

    return 0;

}
```

Before we continue, I want to point out several important details about this function. First, notice that it is "stand-alone". It has a specific job that it can accomplish with or without any other functions. If you give it a tic-tac-toe board in the format it expects, it will do its job and return a useful analysis of that tic-tac-toe board.

It is good practice when writing programs to keep jobs separate. You should generally give each function a specific task, and make each function as stand-alone as possible. By following this guideline, you are less prone to problems.

Whenever you write a function designed to do many different tasks, it is easy to get lost. You might think you had accounted for one possibility, but you didn't. You might have, but then accidentally removed it without realizing it.

As a beginner, it may seem when you are reviewing a complex program that programmers must be super-human and able to understand huge amounts of program code all at once, but this is not the case. The best programmers are those who can see everything they need in a single screen worth of code. If a function spans multiple screens, then it is very likely too long and should itself be split into multiple functions. Also, programming is easier and less stressful when you can see everything you need on one screen.

Many bugs and security exploits happen simply because people do not follow this simple rule: Keep the code simple and keep each function dedicated to a single specific role.

Now, let's continue.

In an earlier lesson we wrote a function which is capable of reading the results from our "is_winning_position" function, and interpret them for a user. All we need to do is modify our program to include this function. Just as before, realize that this function is given a specific task: to interpret result codes.

Right now, we get output that looks like this:

```
The result of playing X at position 7 is: 0
```

What we want is output that looks like this:

```
The result of playing X at position 7 is: No win detected.
```

To do this, we could modify our "show_win_details" function, accordingly:

```
default: printf("Some error occurred.  \n"); break;
```

Becomes:

```
default: printf("No win detected.  \n"); break;
```

By doing this, we are assuming that any result-code sent to this function that is not already covered by our switch/case must be a "no-win" position. However, this is risky and not best practice. You should *always* know what to expect, and you should *never* make an assumption that what is being sent is "correct". This is also how many security exploits come into being. Much better practice is to leave the "default" case as an error, and to write a new case statement for the "0" result code, like this:

```
// No win
case 0 :
   printf("No win detected for Player: %c  \n",
      player);
break;

default: printf("Some error occurred.  \n"); break;
```

Now we have modified our "show_win_result" function so that it can handle any board position, including board positions where there is no win detected. In the next lesson, I will show you how to modify our main program to use these two functions, "is_winning_position" and "show_win_details" together.

Lesson 4.14 : Using a function to interpret result codes

In the last lesson we talked briefly about "Analysis Functions". One important point I wish to bring up is that usually with these kinds of functions, you are interested in analyzing the data and producing a usable result. However, you are usually not interested in "interpreting" that result. For that, you would typically want to build another type of function.

Therefore, "analysis" functions are typically paired with "interpretation" functions.

Let me describe this a bit. When you go to a web page that does not exist, you will often get a "404" error. The interpretation of that "404" error is that the page could not be found. A function that determines a page cannot be found need only return a "404", not the interpretation of what "404" actually means. A totally different function could have a table of various codes, and how to interpret them. A function that collects facts on some data does not need to interpret those facts, only to collect them and to have a mechanism to pass that information along.

This does not apply only for errors. Any real program is going to have many functions whose job is simply to read and analyze data, sometimes in a never-ending loop. The results of this analysis is used by other functions which are able to read that result and make decisions in the program as well as to report useful information to the user.

In our example program, we have one function which is designed to analyze a tic-tac-toe board and produce a simple numeric result which contains all of the facts we are interested in about that tic-tac-toe board in order to determine whether or not a win exists, and the specifics of it. However, that function contains no code that is useful to interpret the result code.

Why not combine the analysis and interpretation into a single function?

At this point, you may be asking yourself this question: If the result code is necessary, and the interpretation is necessary, why not combine them into a single function? The answer is that doing this would lead to a much slower and inefficient program. And here is why:

First, look at the function which analyzes the tic-tac-toe board:

```
int is_winning_position(char *raw_data, char player) {

    int i;

    // Test for horizontal win
    for (i = 0; i <= 6; i+=3) {
       if (raw_data[i] == player
          && raw_data[i+1] == player
          && raw_data[i+2] == player)
       {
          return 10 + i;
       }
    }

    // Test for vertical win
    for (i = 0; i <= 2; i++) {
       if (raw_data[i] == player
          && raw_data[i+3] == player
          && raw_data[i+6] == player)
       {
           return 20 + i;
       }
    }

    // Test for diagonal win
    if (raw_data[4] == player) {
        if (raw_data[0] == player && raw_data[8] == player) {
            return 31;
        }
        if (raw_data[2] == player && raw_data[6] == player) {
            return 32;
        }
    }

    return 0;
}
```

This algorithm is very small, consisting of only two loops, and several if statements. It will process *very* quickly. Would it process this quickly if it had to also interpret the result codes? Absolutely not. The more interpretation that an algorithm has to do, the slower and less efficient it is. The goal of this type of algorithm is to be as fast as possible, and to have as little "overhead" as possible. This is because we are then able to run this algorithm many times, analyzing many different tic-tac-toe boards, without necessarily having to interpret those results.

Another reason this concept is true is that a result code can be passed along to many different functions, and it is very small. Some of these functions may not need to know how to interpret that result code. Those that do can simply call the function whose job is to interpret the result code. The less information you need to pass between functions, the faster and more efficient your program will be.

Remember, our goal is to have an "artificial intelligence" system that is capable of beating a human player. In tic-tac-toe, there are a very small number of possible positions/moves. However, if this were something more complex, such as chess, you can see why we want our algorithms to be as fast as possible. In general it comes down to this: When you have to make a decision based on analyzing many different outcomes/possibilities, the faster the analysis the better the decisions you will be able to make.

So now, let's modify our main program so that it combines our "analysis" function and our "interpretation" function in a useful way:

To do this, you only need to change one function, the "find_winning_move" function, and you only need to add one line:

```
int find_winning_move(char *raw_data, char player, int depth) {

    char test_position[9];
    int i, win_result;

    for (i = 0; i < 9; i++) {
        if (raw_data[i] == ' ') {
            strcpy(test_position, raw_data);
            test_position[i] = player;
            win_result = is_winning_position(test_position, player);
            printf("The result of playing %c at position %d is: %d  \n",
                player, i, win_result);
            // We are adding the below line, which will now be able
            // to give us more useful information.
            show_win_details(win_result, player);

            display_board(test_position);
        }
    }

    return 0;
}
```

In the next lesson we will talk more about the analysis process as a whole, as well as how a computer makes decisions.

Unit 5 : Programming and Math

Lesson 5.1 : The concept of N! (N factorial) as it applies to our tic-tac-toe game

When we run our program in its current state, we see output similar to this:

```
We are examining this board:
[X][  ][  ]
[X][  ][X]
[O][  ][  ]


 The result of playing X at position 1 is: 0
No win detected for Player: X
[X][X][  ]
[X][  ][X]
[O][  ][  ]


 The result of playing X at position 2 is: 0
No win detected for Player: X
[X][  ][X]
[X][  ][X]
[O][  ][  ]


 The result of playing X at position 4 is: 13
Horizontal win on second row for Player: X
```

```
[X][  ][  ]
[X][X][X]
[O][  ][  ]


 The result of playing X at position 7 is: 0
No win detected for Player: X
[X][  ][  ]
[X][  ][X]
[O][X][  ]


 The result of playing X at position 8 is: 0
No win detected for Player: X
[X][  ][  ]
[X][  ][X]
[O][  ][X]

Process returned 0 (0x0)   execution time : 0.006 s
Press any key to continue.
```

Here we are starting with this board:

```
[X][  ][  ]
[X][  ][X]
[O][  ][  ]
```

We are evaluating all possible outcomes that would result from playing a move on any given open square.

Fundamentally, in order to win the game, our "artificial intelligence" must be able to determine whether or not a move played on any "open" square results in an immediate win. That is our first goal. Once we have a system that can see a win one move ahead, we can then build a system that can see a win two moves ahead, or any number of moves ahead.

To better illustrate this, let's imagine the above board one move ago:

```
[X][  ][  ]
[X][  ][  ]
[O][  ][  ]
```

Now let's imagine that we are the C program examining this position. There are six possible moves. Each move when played results in a new position. Therefore, there are exactly six future positions possible.

When one of those six positions is played, there are now 5 possible moves that can be played, thus five possible positions. After that 4, after that 3, after that 2, then 1, then none. Before we proceed to the next step in writing this program, I want to discuss a bit of the math that goes into this.

First of all, we should know how many possible tic tac toe positions are there? If we start with all empty squares, then we have 9 possible moves to play. Each of these 9 moves will result in a single unique board position. Once we play one of those moves, we now have 8 left. Then 7. And so on.

Let's look at this concept in greater detail. Examining a board with one position left to play would be rather pointless. Let's therefore examine a board with two possible moves:

```
[X][O][O]
[X][X][O]
[O][  ][  ]
```

Here it is useful to apply some abstract thinking. Each square on the board can exist in exactly three states, which we are representing as: [X], [O], and [ ].

This makes it somewhat space-consuming to show you multiple boards. Therefore, let's think of each board state as:

```
_ = empty.
X = X has played this square.
O = O has played this square.
```

Our above board could therefore be understood as:

```
XOO
XXO
O__
```

Since the top rows have already been played fully, we do not need to consider them in our example, thus we can simplify this further:

```
O__
```

Let's only look at the bottom row. X can play in either of the two empty squares. O will immediately play whatever square is left. Therefore, we will have one of the following two futures:

```
O__ -> OX_ -> OXO
O__ -> O_X -> OOX
```

If the above example is hard to understand, let me clarify it a bit. We are starting with O__ which means that there are two empty squares that can be played by X. Regardless of which one is played, the very next position is inevitable. If X plays in the middle square, OXO will be the final position. If X plays in the last square OOX will be the final position.

We can therefore state that two open squares means exactly two possible future positions. What happens if we have three open squares?

If it is X's turn, we get the following possible futures:

```
___    -> X__
       -> XO_ -> XOX
       -> X_O -> XXO
___    -> _X_
       -> OX_ -> OXX
       -> _XO -> XXO
___    -> __X
       -> O_X -> OXX
       -> _OX -> XOX
```

Each "->" indicates the completion of a move. So here we say that if X plays the first square ("X__"), then O will play either the middle square or the end square ("XO_" or "X_O"). Each one leads to an inevitable outcome. For example, "XO_" must lead to "XOX" since now it will be X's turn again and X will have to play the final available square.

You should notice that for each of these *three* possible moves, that there are *two* possible futures. The total number of futures is therefore 3*2, or 6. If we were to start with four possible open squares, then for each of those four there would be three following. For each of those three there would be two. Then finally one. That means that with four starting positions there are 4*3*2*1 = 24 possible future end positions. When you start at some number and multiply it repeatedly each time subtracting one from the number, that is known as "factorial".

The number of possible positions from a given tic tac toe position is therefore that number "factorial". If we call that number "N", then the total number of possibilities is N!. The "!" simply means "factorial".

Here is a simple example. If we were wanting to calculate 5! (that is, five factorial), we would say:

5! = 5*4*3*2*1 = 120

Now, when we start the tic tac toe game there are 9 open positions. Each of those nine lead to 8, then 7, and so on. That means there are 9! (nine factorial) possible future board positions from an empty tic tac toe board. How many total positions is that? 362,880. Of course, this does not take into account wins and losses, so the actual number of positions that need to be evaluated are much smaller.

Three hundred thousand positions is a lot for a human to evaluate, but it is not a great challenge for today's computers. In the next lesson, I will show you how to calculate all possible outcomes in a given tic-tac-toe board position in order to determine whether or not a position can be won by force.

Lesson 5.2 : More on N! And Introducing trees

In the last lesson we talked about factorials, and how they are useful to calculate the total number of possibilities in a situation where you have N possibilities, and upon each of those possibilities being realized you have one less. It is important for someone who wishes to be a programmer to be able to simplify concepts. Of course, this is something that mathematics and programming have greatly in common. To be an effective programmer you need to be able to take a complex concept and simplify it, the same way as you might take a complex math problem and simplify it.

For our tic tac toe board, we start with 9 possibilities. We know that there are 9! (nine factorial) total possible outcomes. Of course, this does not take into account won positions. Once a win is realized, then there is no need to calculate further. Therefore, we know that the total number of possible boards that our artificial intelligence system (hereafter simply called "AI") will need to calculate will be less than 9!. Here I am going to introduce another concept, called the "tree".

Whenever you start with one possibility and this leads to another, you can draw out a simple diagram for this just as we did in the last example. We call these types of diagrams "trees" because if you think of the initial state as a "trunk", then each subsequent stage is similar to a branch. Each branch may have more branches. Let's look at this briefly:

```
___ -> X__
    -> _X_
    -> __X
```

Here we see a simple "one level deep" tree. This is one row of our tic-tac-toe board. Any particular location we refer to as a "node". For example, "X__" is a "node". This particular tree has 3 nodes one level deep. Now, we can expand this tree further, like so:

```
___ -> X__ -> XO_ -> XOX
           -> X_O -> XXO
    -> _X_
           -> OX_ -> OXX
           -> _XO -> XXO
    -> __X
           -> O_X -> OXX
           -> _OX -> XOX

     3  *  2  *  1  = 3!
```

Here, notice that we have a two-level deep tree. This tree now has a total of 3 nodes on the first level, and 6 nodes on the second level, and 6 nodes on the third level. Now, keep in mind that we start with 3 empty squares. We know therefore that the total # of possible outcomes is 3! which is 3*2*1, or simply 3*2, which is 6. That is why you see 6 final nodes.

You should also be able to see *why* the concept of N! works here. Each number in our equation corresponds to how many *new* possibilities are opened up. First we open up three possibilities. Then for each of those three, we open up two possibilities. Finally, each of those two possibilities leads to exactly one outcome.

Trees are a common and important concept throughout higher computing. You will use trees to plan, organize, and store data. You are often able to understand data by where it resides in a given tree.

I believe that at this stage in the lesson, you should fully understand the concept of N! as it relates to our tic-tac-toe board. You should now be ready to proceed to the next step which is to write this same concept as a program in C.