Contents

Some thoughts about programming language tutorials and books

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 Barnes and Noble 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.

C, C++, Python, Ruby, Perl... A language for every day of the year.. but why? Which one is best?

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, any where, 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 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.

010011111010; Err.. I mean: 1,274. Computers count differently than we do. Lets explore that.

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, so it must become zero, and the column over to the left must now become a 1. So: 0, 1, 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.

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.

Alright now I can count in binary! Other than to impress my girlfriend (or scare her away), why do I have to know this?

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.

How to begin a career in programming

Before we continue to the next lesson, I want to talk about what I am sure is an important topic for many people in this course.

If you listen to half the people commenting on this subject, you would think that deciding to be a programmer means signing your soul to the devil and living in hell until you retire. Every time I read such a horror story I ask myself the same question, "Why doesn't the guy just quit?"

If you are planning to take the first job that comes along, work for less than you are worth, and not be willing to leave if the situation changes - that may very well be the case. However, this is as true for programmers as it is true for engineers or any field which involves building something as part of your job.

You must be patient, and evaluate every prospective position. Remember, you are interviewing them too! Don't take a job that entails you sitting in a cubicle for 10 hours a day if that is not what you want. Be patient, and set high standards for yourself.

If you set low standards for yourself, then expect to be treated like dirt. If you are treated like dirt, quit. There are always companies looking for highly skilled programmers - always.

How do you get a job without a college education?

Credentials, references, and an impressive portfolio of what you have built. If a company would hire someone fresh out of college with no experience but the same company would not hire someone with a few years of experience with a great portfolio, that is not a company you want to work with.

Many companies understand this, and that is why on many job postings you will see something like, "BS in Computer Science or 5 years experience", or similar wording.

I find that a self taught programmer who has actually built stuff is a far better fit for a programming position than a college graduate who has only the knowledge they gained from college. Many companies feel the same and this situation is getting better and better for the self-taught programmer. All of that said, it is still best to have a degree. If you are serious about a career in programming, you should seek to advance your education.

With so many jobs outsourced, how can I get paid a competitive salary?

If all you know is html, a little PHP, and how to make some basic web apps - then you are dead in the water on this one. There will always be some guy in a third world country willing to do the job cheaper.

You must build skills that go above and beyond the basics, and establish yourself especially in areas that companies will not want to outsource. The more skilled you are, the more likely you can work with highly proprietary and sensitive information.

No company wants to send their trade secrets to some third world country, and no company is going to let someone work on those types of projects who doesn't have a strong enforceable NDA in place. These are the positions that pay well and that give you the opportunity to grow.

More about counting like a computer

In lesson 3, we went over the basics of binary and I explained how a base-two system is different from a base-ten system. Please make sure you understand lesson 3 completely before beginning this lesson.

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 doesnt' 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

Include statements

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.

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.

How programming languages work with data

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.

Some basics about RAM

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.

Programs are data too

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.