

Cam's Computer Programming Guide

Chapter 1 - Intro

Version 1.0

About:

I have decided to write this for many reasons. First of all, I love computers, and I love programming. Secondly, I love to teach people things; it makes me all warm and fuzzy inside! If you have ever considered programming, you've come to the right place. If you don't know what programming is, you've come to the right place. Why? Well, I write my tutorials casually, unlike many of the boring formal computer manuals and programming guides... So in a nutshell, this guide will teach you all about computers, and how to make your own software.

About me:

As you can tell, my name is Cam. My nickname is LiceFork; you can just call me Lice. How did I get that nickname you ask? Long story... Well I love computers, and computers love me. They are fun, and you can do just about everything on one! What's even better, is if you can make your own programs that do whatever you want. Personally, I enjoy making things. I love the feeling when you sit back and can say, "I made that!" I do not claim to know everything about computers, programming, and electronics; but I do have a large knowledge about computers, programming, and electronics. I wish to share this with you, so you can make your own programs!

In this chapter:

In the introduction chapter of my manual/book/tutorial, you will learn about computers and how they work, and how it all relates to programming. I am going to assume you know nothing about computers, besides how to run programs, use the mouse, keyboard, etc. If you feel you are needlessly reading information useless to you, feel free to skip around!

How do computers work?

Ah, such a refreshing question... Computers are made up of two separate divisions: hardware, and software. Hardware is everything you can physically touch, such as your hard drive or CD-ROM drive. Software, on the other hand, is the programs, or the little icons you click on that pop up and do something, such as Internet Explorer, Microsoft Word, Notepad, AIM, e-mail clients, and even the operating systems! The operating systems are coded "environments" to let the user have an easy time working on a computer. These consist of Windows, Mac OS X Tiger, Linux, Unix, BeOS, X-box, X-box 360, PS2, PSP, Gamecube, etc. Basically a "shell." It's also called a GUI or Graphical User Interface. There may be many layers of these GUI's, such as ROMS. Games may also contain GUI's to give maximum ease of use to the user.

First off, let's start on the hardware side of computers. The basic set of hardware needed to successfully run a computer is the following: motherboard, microprocessor, memory, power supply. The motherboard is a really important part; it connects everything together, and makes sure everything gets power. The microprocessor is the computer's "brain." It does all of the calculations. Computer memory is also really important, if you add 1+1, how can a computer store the answer? Memory comes in many shapes in sizes. I'm not going to go too deep in this subject, but computers have RAM and of course your regular storage memory unit, which in other words, your hard drive. RAM stands for Random Access Memory. This is where your computer stores the answer to 1+1. It's a temporary memory space used for calculations and etc. You would NOT store your programs, MP3's, and other files in the RAM, that's what your hard drive is for! Anyway, power supply should be pretty obvious, how can electronics work without electricity to power it?

There are of course other standard parts to your computer, but these are not required to run a really simple computer. These can be any of the following: graphics cards, I/O (input/output) ports, disk drives, modems, monitors, sound cards or other sound devices, speakers, keyboard, mouse, etc. But honestly, you wouldn't need a sound card to run a calculator, would you?

Microprocessor binary math:

Let's get deeper into microprocessors. Ever wondered how they do the fancy addition and subtraction? If so, let's find out, shall we?

Microprocessors are those little tiny thin chips smaller and thinner than a penny. They make calculations using binary numbers. In other words, lots of 1's and 0's. What's up with all the 1's and 0's you ask? Well, 1 stands for "on" and 0 stands for "off." Combinations of these make code understandable to the microprocessor. These "binary numbers" are calculated in bits. If you've had a computer for at least a day, you would have seen the word "bits" or "bytes." These are file sizes, calculated in binary numbers. Binary numbers can only be 1 or 0, so what about 2-9? You would write those in a binary number combination. Lost? Let me explain...

Decimal numbers are calculated at "base-10." If you know your algebra, this should be pretty easy for you. Binary numbers are calculated at "base-2." Why? Well, it's a lot less expensive.

A decimal number is any number times 10, since it's a base of 10. So $(1*10^1)$ would be 1. $(1*10^2)$ is 10. Catch my drift? Any number to the base of 10 is a decimal.

Binary numbers on the other hand, are at base-2. So $(1*2^3)$ is 8, while $(1*2^4)$ is 16. So, how do I calculate 10010 then? Well, you would read the first (all the way to the right) digit in the "one's" place and the second digit in the "tenths" place, third in the hundredths, fourth in the thousands, and so on.

These "digit places" have numerical values. One's would be 1, tenths is 2, hundredths is 3, and so on... So you would then take that number (counting from right to left) and raise the base to that number. You then take all of those calculated numbers and add them together. So 10010 would be:

$$(1*2^5)+(0*2^4)+(0*2^3)+(1*2^2)+(0*2^1) = 36$$

Or...

$$32 + 0 + 0 + 4 + 0 = 36$$

So the binary number 10010's value would be 36! Here is a small list of counting numbers from 1-10 and their binary value:

0 = 0
1 = 1
2 = 10
3 = 11
4 = 100
5 = 101
6 = 110
7 = 111
8 = 1000
9 = 1001
10 = 1010

Now that that has been made clear, it's time to go into bytes. Bytes consist of 8-digit bits. Why? Over the past 50 years, it was made clear this is a good method via trial and error.

0 = 00000000
 1 = 00000001
 2 = 00000010
 ...
 254 = 11111110
 255 = 11111111

Using this method, you can get a up to 256 values (0-255)

On the other hand... CD's use 2 bytes, or 16 bits:

0 = 0000000000000000
 1 = 0000000000000001
 2 = 0000000000000010
 And so on...

So, what does all of this boring math have to do with anything at all? Well, programs such as Notepad use binary values to display a certain character. These values have been standardized into an ASCII character set. Here are a few examples:

...
 62 = >
 63 = ?
 64 = @
 65 = A
 66 = B
 67 = C
 And so on...

As I said earlier, memory is calculated in bits and bytes. Millions and billions of bytes have different names. Here's a lovely chart to clarify my point:

| Name: | Abbreviation: | Size: (in bytes) |
|--------------|----------------------|--|
| Kilo | K (1kb) | $2^{10} = 1,024$ |
| Mega | M (1mb) | $2^{20} = 1,048,576$ |
| Giga | G (1gb) | $2^{30} = 1,073,741,824$ |
| Tera | T (1tb) | $2^{40} = 1,099,511,627,776$ |
| Peta | P (1pb) | $2^{50} = 1,125,899,906,842,624$ |
| Exa | E (1eb) | $2^{60} = 1,152,921,504,606,846,976$ |
| Zetta | Z (1zb) | $2^{70} = 1,180,591,620,717,411,303,424$ |
| Yotta | Y (1yb) | $2^{80} = 1,208,925,819,614,629,174,706,176$ |

Well, now you know, happy? Next!

How about that handy addition the microprocessor can do? Binary math works just like decimal math, except that the value of each bit can be only 0 or 1. First let's look at a decimal math equation:

```
452
+ 751
-----
1203
```

To add these two numbers together, you start at the right: $2 + 1 = 3$. Boom! Next, $5 + 5 = 10$, so you save the zero and carry the 1 over to the next place. Next, $4 + 7 + 1$ (because of the carry) $= 12$, so you save the 2 and carry the 1. Finally, $0 + 0 + 1 = 1$. So the answer is 1203. Easy stuff, just like first grade!

Binary addition works just as easy:

```
010
+ 111
-----
1001
```

Starting at the right, $0 + 1 = 1$ for the first digit. No carrying needed. You've got $1 + 1 = 10$ for the second digit, so save the 0 and carry the 1. For the third digit, $0 + 1 + 1 = 10$, so save the zero and carry the 1. For the last digit, $0 + 0 + 1 = 1$. So the answer is 1001. If you translate everything over to decimal you can see it is correct: $2 + 7 = 9$. So it all works out, and everybody is happy once again!

Now, as you may know, computers can do much more than addition. It has subtraction, multiplication, division, etc. It all works the same as binary addition. So let's move on to a more fancy subject, shall we?

Microprocessor boolean logic:

Here comes the fancy stuff. Computers can calculate "boolean operations" such as AND, NOT, OR, NOR, NAND, XOR, XNOR, etc. What does all this mean? Well, it's pretty east actually. Ever studied geometry? You know those "if-then" formulas? This is how those come in handy. You will see how to put these operations to use once we get into the actual coding part. Here's an example:

If $x=7$ AND $y=8$ Then explode!

So, if x is 7 and y is 8 then we will explode, how nice! These really come to great use in programming! Now, how about we figure out how these things work? Well, it was originally developed by a man named George Boole in the mid 1800s, and it adds some pretty sweet things to bits and bytes.

Simple gates:

Gates? Yes, gates. Gates are a type of "check system." A good example would be going to the movies. If you don't have a ticket, the ticket taker will not let you through the gate. You can use these components in anything electronic, even your wall clock! Although, it wouldn't be much use...

NOT gate:

| A | Q |
|---|---|
| 1 | 0 |
| 0 | 1 |

As you can tell, the not gate reverses the binary value. I used A and Q so O wouldn't be thought of as a zero, so therefore; A is the input, Q is the output. Thank you.

AND gate:

| A | B | Q |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

This gate simple says "if A and B are both 1, then the output is 1; if anything else, output is 0." So it's a logical AND operation.

OR gate:

| A | B | Q |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 1 |

If A or B (or both) are 1, then the output is 1

NOR gate:

| A | B | Q |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 0 |

Combination of NOT and OR gates. If neither A or B (or both) are 1, then output is 1.

NAND gate:

| A | B | Q |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Combination of NOT and AND gates. If A and B (or both) are not the same, output is 1.

XOR gate:

| A | B | Q |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |

Commonly called "exclusive or," it's logic is "if A or B = 1, but not both, then output is 1."

XNOR gate:

| A | B | Q |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 1 | 1 |

Commonly called "exclusive nor," it's logic is "if A or B are the same (1 or 0) then output is 1."

I believe at this point, you have enough information on how a computer works, deep down inside, using binary numbers. There are a few more things you could learn, but I don't think it's absolutely necessary. Now, let's move on!

Software, and how it relates to programming:

Next up is software, of course. Software can also be called programs, applications, or whatever your little heart desires! Since this is a guide to programming, I will refer software to programs from now on. Programming? Yes, it's the art of creating a program. How do you create a program? Simply, of course!

As you may know (hopefully), everything a computer does is calculated in binary numbers. Now, how are we going to write a program using a whole bunch of 1's and 0's? It's very hard, frustrating, and takes years. Luckily, people have already done that for us, and created an easily human-understandable "language" to write programs using. This language is called, of course, a *programming language*. There are many different programming languages, each with it's own Pros and Cons. Some of these include Visual Basic, Delphi, Python, Assembly, C/C++, and many... many more. By far, Visual Basic is the easiest of the languages, and the most stable. Stable as in, you can adjust something slightly in the code and your program will still *compile*. Compile? I'll go over that later. Now where was I... oh yes! Cons of VB are very easily noticeable. The term "Visual" before any programming language means it works with Microsoft's Visual Studio .Net. What is that? I'll cover that later. But what that means is very bad. It means the code is NOT cross-platform capable. This means you cannot take the code or the compiled .exe file and run it on any platform other than Windows! I'll also cover that later.

Delphi, oh Delphi! Simply put; don't use it. It's very complicated and annoying. It can be useful, but not for our purposes. Plus, the compiler costs a pretty penny!

Python, very nice! Easy to use, fast, streamlined. Its purpose is not to create a game on it's own, but it can be used for on the fly scripting if you implement it into your program.

Assembly, very low level, hard to use, and you cannot create complex programs with it. It can be very good use though, if you can find the time and the will power to actually learn how to use it!

Other than the many... many more, this leaves us with C/C++. To date, this programming language has become the industry standard for programming. It's fast, not too hard to learn, easy to compile, and almost every program you own is written in C or C++. What does this mean? This means, you will learn it! You will love it, trust me.

Compiling?

As I said, now is the time I am going to cover this. If you "compile" something, you squish it together into a working piece. It's like a puzzle you piece together. How do we do that? Using

a C/C++ compiler of course! We are going to make our programs cross-platform, which I will cover later. First of all, let me explain what a compiler does. You see; computers cannot understand C/C++, since it's in words, and computers can only understand 1's and 0's remember? So we use a compiler to compile, or piece together, the words into "machine code," which is another word for the binary numbers we discussed earlier. The compiler takes the *source code*, or the "words," and smashes it into ".o" or "object" files. What are these for? Well, in order to create the .exe file, all code must be in binary format first. The compiler compiles all of the source code files, and THEN it creates the .exe, by linking everything together using nasty *linkers*. Linkers tell the compiler how everything should be put in order, and also includes any external library files that are needed to finish compiling the program. A little too complicated? Not a chance!

What will I need?

Now that you know the basics on how everything is pieced together, it is time to get what you need, and get ready to start programming!

Obviously, you will need a compiler. There are many compilers. Free ones, and costly ones. It also depends on what operating system you have. For Windows/Linux users, I'd recommend using Bloodshed's Dev-C++ (www.bloodshed.net). It is both free and open source! The source is in Delphi, which is a notable example on how Delphi can be useful! I absolutely do not recommend using Micro\$oft's Visual Studio. Not only does it cost a pretty penny, but also, it SUCKS! The only good it does is compile really small and optimized programs. If you are really worried, just download the free visual c toolkit. For MacOS/X users, DON'T USE COCOA! It's NOT cross-platform, which we don't want! Instead, invest in Metrowerks CodeWarrior, or use the freely available MPW compiler.

Since we want our programs to be cross-platform, we need something that can make our code portable with hardly any extra code. This is where SDL comes in to play. SDL stands for Simple Direct media Layer. It can be completely ported to Linux, Windows, BeOS, MacOS, MacOS X, FreeBSD, OpenBSD, BSD/OS, Solaris, IRIX, and QNX. It also has unofficial support for Windows CE, AmigaOS, Dreamcast, Atari, NetBSD, AIX, OSF/Tru64, and SymbianOS. Now there is no way you can tell me that is a bad thing. Another thing, Windows programming is by far the worst and the most confusing, and it uses the most lines of code! The ratio of the amount of Windows code compared to SDL code, in my opinion, is 10:1! You can't beat that. Best of all, SDL is FREE! SDL can be downloaded at www.libsdl.org.

Since I love SDL very... very much, I will proudly tell you about the very nice and successful author. SDL was written by Sam Lantinga, who used to be the Lead Programmer for Loki Entertainment Software, and is currently working at Blizzard Entertainment. Mr. Lantinga successfully ported Maelstrom from the Macintosh to Linux, worked on port of the Macintosh emulator Executor to Win32, and ported some of the DOOM tools from DOS (DEU, DHE, etc.). You may see more of his projects at <http://www.devolution.com/~slouken/>.

Other than that, you will need your brain, spare time, and this guide. No need to mention the essentials, they are too obvious. Pretty much, as long as you can operate a computer casually, you will follow along well, hopefully...

Some history on the C programming language:

In the early 1970s, smart people were working on an OS named Multix, to give the common household easy and inexpensive access to computers, graphics, e-mail, databases, and etc. Unfortunately back then people weren't that smart, and the project was a failure.

So, a small team of engineers at Bell Labs (former name of Packard Bell, which was the former name of HP) decided to work on a smaller, single-task version of the project, and named it Unix. Sadly for them, they had multiple computers from multiple manufacturers, and they had to constantly re-write the code over and over again. So they got pissed and created a small, lightweight, yet powerful cross-platform programming language named C. Later on when technology increased, better methods of programming popped up, and C was left behind. Fortunately, the community added new features to the language (because they loved it so much) and it was called C++.

C or C++?

I bet you are wondering what's the deal with the name "C/C++." Well I'm going to explain it to you. As I explained earlier, C++ is the successor of C. C code works with C++ natively, but C++ cannot be compiled using a C compiler. So we will be using C++ ;)

Conclusion to chapter 1:

Well this concludes the first chapter to my guide. Hopefully by now you know the basics on how a computer works, and how it all pieces together. This knowledge you now know is not required, but it makes this stuff a lot easier to learn, and it makes you understand it a lot better, which is very important. If you have completely read and understood this introduction, it is now like going into a battlefield with an assault rifle rather than a pistol.

Special thanks go to:

- SoBe No Fear: for being the only energy drink that actually tastes good!
- Coffee: `nuff said!
- Angels Way: it's where I typed most of this up at ;)
- My Dad's Laptop: the way I typed it up away from home
- All NoResidue Fans: for giving me motivation
- Food: for keeping me alive
- The Internet: I would know nothing without it!
- Sam Lantinga: for inventing SDL!
- Bell Labs: for inventing C!
- Computers: for being so literal

Contact the Author:

You may contact me via e-mail: LiceFork@gmail.com

Also, check out my site for more tutorials: www.blackenedice.com

If you have AIM, my sn's LiceFork

Please leave any comments, good or bad, and tell me how my guide helped you out! (Or... destroyed your life)

Also... please report any found errors!

-Last updated: 4/8/06-