Welcome

Hello! Are you ready to learn all about C programming? Great! You have come to the right place.

If you don't know me, my name is Caleb and I make programming tutorials on my YouTube channel, Caleb Curry.

This eBook goes along with the videos on my YouTube channel over C Programming. These notes are meant to go along with those videos, so watch them here as you read through this book! You can also connect with me on Twitter here.

These videos and notes are for complete beginners! If you know nothing about C Programming, don't worry. I gotch you! If you do know about C programming, don't leave yet! We will be going more in-depth as time goes on!

New notes will be uploaded every 15 days or so until we get to tutorial 100! So, please check back every now and then to receive more notes!

Also keep in mind that these are just notes. I did not spend tons of time writing these out with the thought that they would be a published eBook (which it's not). So bare with me! If something is confusing, please leave me a comment on one of my videos or send me a Facebook message. You're feedback is greatly appreciated!

Okay… I'll stop blabbering. Let's get started!

Tutorial 1 - Introduction - What is C?

This is your series on C programming and computer science for beginners. If you are completely new to C, this series is for you! What is C?

C is an example of a programming language. What is a programming language? Essentially, we as humans want to communicate with a computer to tell it to do something or to compute something for us, but computers don't understand English, they only understand **machine code**. Machine code is a sequence of ones and zeroes that the computer's processor can understand. This allows us to tell the processor to do simple instructions.

The problem is that writing software in machine code is very complex and very error prone. A programming language is something humans can understand that can also be understood by computers. The computer doesn't understand the language directly, but what we tell the computer to do in a programming language is translated to something the computer's processor can understand.

C is a very good programming language to learn. In fact, it was one of the first really well established programming languages and is one of the most popular languages used today. In fact, many other programming languages are based off of the beauty of C. You may have heard of C++, C#, Objective C, Java…These all have similarities with C. Learning C helps you learn how to become a better software developer.

When we communicate to a program in C, we essentially create a file where we type all of our commands. The stuff we write in this file is called **code,** or **source code**. When you can read C, this code makes sense. But the code cannot be read by the computer, it still has to be translated for the computer to understand. This process of translation is called **compiling**. How do we compile code? We use a special software known as a **compiler.** We hand the source code file to the compiler and it outputs the compiled version of the code and stores that in another file. Think of the compiler as the translator between us humans and the computer.

So we write code in a file (you can think of it as just a text file), and then we compile this code into what is known as a **program,** or **software.**

Once we have a program, we can **run** or **execute** the program and it will start doing the things we originally asked the computer to do when we were writing our code.

This is the very basics. The more you study, the more you will see how beautiful and fun computer science can be.

Tutorial 2 - Getting Started - Installing GCC

In order to get started programming with C, we need to have an operating system to work on.

Now, in order to be on the same page, I highly recommend being on a UNIX or Linux operating system. An operating system is a big piece of software that is always running on your computer. It's what allows you to open apps, type, print, and use anything on your computer.

Mac is an example of a UNIX operating system. Windows on the other hand, is not. If you are in Microsoft Windows, you can actually get a copy of a Linux operating system to install absolutely free. Here is a video I made on how to install Linux as a virtual machine:

Now, don't get me wrong, if you want to use windows, that's fine. You can use windows, but it is a bit more work to get everything set up and not everything is the same as will be in these tutorials. Personally, I think it is helpful to familiarize yourself with Linux.

Open the terminal. We are going to try to get comfortable using the terminal. That's because it is very universal. Every UNIX operating system will use the same commands. Another reason is that you're not always going to be able to do everything visually. For example, if you access a computer remotely or if you are writing scripts.

Now, the very first thing we need to install is a compiler. The compiler we are going to be using is called GCC.

To do this all we have to do is type gcc in the terminal. Mac is going to prompt us if we would like to install Developer Tools. Select Install. This is going to install GCC. GCC is the tool that will take our C code and compile that down to the appropriate machine code that can then be ran by the computer.

Now, another tool we are going to use is called VIM. This is a text editor that we are going to familiarize ourselves with. You guys are more than welcome to use another text editor, but once again the benefits of VIM are that you are going to be able to find it everywhere.

To use VIM, type vim followed by a file that you want to create.

```
vim hello.c
```

Now, VIM is going to seem a bit strange at first and honestly it might cause some frustration, so take it easy. The first thing that is probably going to drive you crazy is “how the crap do I close this program?” The trick to closing the program is to type :q.

Now we have all of the tools we are going to need to get started working with C. That was simple! In the next video we are going to be writing our first program.

Tutorial 3 - Writing Our First Program - Hello World

In this video we are going to write our first computer program. If you are new to programming, you should know what a Hello World program is. Essentially, a hello world program is a program that says "Hello World" on the screen.

The point of this is to have the confidence that you have all of the proper tools installed, everything is set up correctly, you're able to write the most basic program, compile it, and execute it. Essentially, a Hello World program will take you from beginning to the end of writing a program.

The very first thing we have to do is create the file that we are going to write all of our code in.

```
vim hello.c
```

The .c at the end of the file is called a file extension. The C means that it is a C file. Make sure you end all of your C files with .c.

Now once we have the editor open, we want to start typing. VIM has different modes, so we actually have to switch to Insert mode. Press i.

Now, you can start typing. To get out of insert mode we can press the Esc key.

Now to move your cursor around in your program, you can use your arrow keys. Later on we'll learn some more fancy tricks to navigate, but this is a great way to adjust to using VIM as it's very similar to any other text editor. Just don't tell anybody I told you that.

Let's go back into insert mode.

Now basically everything I type in this video may be new to you. That is ok!! In the next video I will go over what everything means and by the end of this series this will be a piece of cake.

Once your program is done you can exit vim by typing :wq while not in insert mode. This will write your changes to disk and then quit.

Now, we need to compile:

```
gcc hello.c
```

Now, let's take a look at the folder we are in by typing ls

You can see that we have hello.c, and a.out. a.out is the executable that gcc created. We can now run this to see it in action. To run the program type:

./a.out

The ./ is to say that we want to execute something in the current location, and then the name is all we need to run the program.

There you go! Congratulations!

Now, if you're not so lucky, you'll have some problems with your code and the program will refuse to execute.

Always try again! Just remember…if at first you don't succeed, don't try sky diving!
Let's see what that looks like by changing some of our code.

Now, when we run gcc, we get errors pop up. A lot of this can look like gibberish sometimes, but read it closely. Often it will give you a hint as to where and what is wrong.

Tutorial 4 - How a C Program Works – Part 1

The goal of this video is to break apart some of what we wrote in the last video and to understand some more concepts.

The way a C program works is by writing out what we want the computer to do in very specific terms. Imagine that the computer is super dumb and you have to write down every single instruction.

Imagine you are a computer and I told you to like this video. Seems simple enough, right? WRONG! If you were a computer, I would have to really spell it out. I mean reaallllyyy spell it out. This means that to tell you to like his video the instructions might look something like

1. Move your hand to the mouse,
2. Grab the mouse,
3. Move the mouse until the cursor hovers over the like button on the screen,
4. Click the left mouse button.

The idea of explaining everything in detail is called an **algorithm**. This is an algorithm to like to my video. Now, I want to modify the algorithm a bit so that it can be used to like *any* video on my channel:

1. Move your hand to the mouse,
2. Grab the mouse,
3. Move the mouse until the cursor hovers over _____ (video).
4. Click the left mouse button.
5. Move the mouse until the cursor hovers over the like button on the screen,
6. Click the left mouse button.

Another thing to realize is that computers don't have a memory quite like we do…. I mean they are really dumb. This means that to tell you to like another video, I have to repeat all of the instructions. This means that there are essentially 12 commands now. 6 for the first video, 6 for the second. What if we could just take all of these commands and extract them as one command? Well, we actually can. This is known as a **function**. Let's create a function called likeVideo().

likeVideo(){

1. Move your hand to the mouse,
2. Grab the mouse,
3. Move the mouse until the cursor hovers over _____ (video).
4. Click the left mouse button.
5. Move the mouse until the cursor hovers over the like button on the screen,
6. Click the left mouse button.

}

Now, each time I want you to like a video, I can just say likeVideo(). This is called **calling** or **executing** the function.

What are the () for? That is where I tell you what video I want you to like. likeVideo("C Programming Tutorial 10"), for example. This thing I'm passing in is known as an **argument**.

Now, what if I added to the function steps for you to look at how many people have liked the video and ask you to tell me. So I give you a request, and then you **return** a number to me. This is known as a **function return**.

Why am I telling you all of this? Well in C, we start every program with a function called **main**. When our program is run, the main function is executed.

In the next video we will be discussing more specifics about the code we wrote in the hello world program.

What is Modular Programming?
Modular programming is a way of creating software by breaking it down into smaller, independent pieces called modules. Each module is like a mini-program that handles a specific part of the bigger program. Here are some key points about modular programming:

Abstraction - Modules focus on doing a specific job and hide how they do it from the rest of the program. This means you can use a module without knowing the details of how it works.

Encapsulation - Modules bundle together everything they need to do their job, like data and procedures. This keeps things tidy and under control.

Information hiding - Modules only show the necessary parts to the outside world, keeping their inner workings private. This helps prevent other parts of the program from relying too much on how a module does its job, which can change.

Modular programming started in the late 1960s to help manage growing and more complex software systems. It's a smart way to organise programs, making them easier to build, test, fix, and reuse. Nowadays, it's a fundamental part of designing and building software.

Tutorial 5 - How a C Program Works - Part 2

This video will be dissecting the hello world program that we wrote in an earlier video.

```
#include <stdio.h>

int main()
{
	printf("Hello World\n");
	return 0;
}
```

The first thing we will look at is the main function. It is important to realize that the definition of every function looks similar to this. We have the **return data type**, the name or **identifier**, parenthesis for any **arguments**, and then a **code block** designated by curly braces. Within the code block, we have what are known as **statements**. This is when we are telling the computer to do something. We always, always, always use semicolons. It will be clearer soon when you are supposed to use a semicolon and when you are not. What you need to do is look at this program and try to memorize all of the syntax here. If you can write this all without looking at anything, you're doing very good.

Everything inside of the curly braces belongs to the main function. You can see that this function does not require that we pass anything into it. At the end of the function we have return 0. This is how we say that the program worked. 0 = good. We can use this to denote whether the program worked or not. For example, we could use a -1 to say “bro something went wrong.” 0 is an int, which is short for integer. That is why the word int comes before the name of the function.

Inside of the function we are calling another function called printf(). This shows that functions can call functions. This function is used to output something on the screen. The thing inside of quotes is the argument. We give this function some data and it will spew it on the screen. Now, this \n is known as the **newline character**. This will make the program go down to the new line.

Now I have a question for you…How does the C compiler know what this printf() is? We haven't defined it anywhere! Well, that brings up the concept of includes. C actually has a bunch of functions that are already available to us and we can include certain files to gain access to new functions. This means that the printf function comes from stdio.h.

Now, in the next video, I'm going to go over some of the most important Linux commands so we actually know how to do things in the command line. It's going to be awesome and educational, so catch you there!

Tutorial 6 - Intro to UNIX/Linux - Part 1

The goal of this video is to teach you how to do the most basic things inside of a terminal. Now, these commands are going to be the same for UNIX operating systems, like Mac, and also Linux operating systems, like Ubuntu.

This is in a C Programming series, but we are not going to be doing much C for right now. That's because in order to be successful with C programming, we ought to familiarize ourselves with Linux. In fact, one of the biggest reasons C became so popular is because it was the standard programming language when UNIX gained popularity. So even if we are not talking directly about C, this content is still extremely relevant.

Now, to start off, we should go over what UNIX and Linux are. As a reminder if you are a beginner, an operating system is just software that allows you to work with a computer and does things like manage input and output, memory, etc. Two popular operating systems include Windows 10 and Mac OS. UNIX and Linux are categories of operating systems. This means that there are multiple UNIX operating system, one of which is Mac OS, and there are multiple Linux operating systems, one of which is called Ubuntu. The difference between UNIX and Linux is not so important right now because Linux is a UNIX-like operating system.

Because they are so similar, often people will be taking about Linux but really it can apply to UNIX operating systems as well. In these videos I am using Mac OS, which comes from UNIX, but it is going to be nearly identical for Linux operating systems. This means that if I say UNIX, you can assume Linux as well. Or if I say Linux, you can assume Unix as well.

Now one of the biggest thing Linux is known for is the terminal. If we are talking about people who use windows, very few people know how to use the command prompt, but with Linux nearly everybody uses the terminal.

The first thing you should learn how to do is the basic things you do with a mouse.

Just like you can navigate folders visually, you can navigate folders in the command line as well.

To figure out what folder you are in:

```
pwd
```

This location is known as your working directory. PWD stands for print working directory. This gives you the location of where you are, but how do you see what is in this folder?

```
ls
```

By the way, a more common name for folders in Linux is **directories**.

You can get more information with this command:

```
ls -l
```

The l is known as a **flag** and it changes the output of the program.

There is also:

```
ls -a
```

This is another flag. You can actually combine them:

```
ls -la
```

Think "list long all"

The very first thing you'll notice is that there are more files that show up. The files starting with a . are known as hidden files and don't show up by default. They're usually hidden for a reason so be careful touching them. The first section of this will list permissions. Don't worry about that for now, but notice the first letter. A D means it is a directory. That is how you tell if something is a folder or a file. Some terminals will also color code.

How do you change your directory?

```
cd directoryname
```

The directoryname can be considered an argument to the cd command. Arguments are separated by spaces.

One more thing I wanted to teach y'all. When you run ls -la you see two special cases. One is a dot and the other is two dots. The single dot refers to the current directory and the double dot refers to the parent directory. If you run pwd, you can see that the parent directory comes right before the current directory. So in order to move up a directory, you can run:

```
cd ..
```

Alternatively, you can pass in what is known as an **absolute path**:

```
cd /absolute/path
```

In the next video we will learn be learning some more useful Linux commands.

Tutorial 7 - Intro to UNIX/Linux - Part 2

In the last video we talked a bit about navigating around Linux. In this video I wanted to go in a bit more depth. You can imagine the Linux filing system as a tree. When you run pwd, you get the absolute path of where you are. There are important terms relating to paths that you should know.

When someone says the root directory, they are referring to the very first directory. To get to the very first directory you can type:

```
cd /
```

Another important directory is called the **home directory**.

This is the directory for your user. This is the default directory that opens when you open a terminal. There is an easier way to go to the home directory though:

```
cd
```

The tilde is a trick you can use that refers to your home directory. You can also reference stuff relative to the home directory.

```
ls
```

```
cd /
```

```
cd ~/folderInHomeDirectory
```

This introduces the concept of **relative directories**. We can reference a directory in relation to other directories. We actually already did an example of this when we did:

```
cd ..
```

This is saying to move one directory up. We don't necessarily have to start from the root directory. We can also get more complex… such as:

```
cd ~/directory
```

```
ls ../..
```

```
cd ../../directory
```

We are going a bit out of scope of intro to Linux, so let's get back on track.

Let's go to the home directory:

```
cd
```

We can create a file using:

```
touch filename.c
```

```
ls -la
```

We can move a file using the mv command. For example:

```
mv filename.c /Directory
```

We can rename a file using this same command:

```
cd /Directory
```

```
mv filename.c test.c
```

We can delete a file with:

```
rm test.c
```

I think we are going beyond the scope of this video though, so that's all I'm going to say for now.

Tutorial 8 - Intro to UNIX/Linux - Part 3

This is going to be the last video about Linux. The first thing I want to talk about is how to make directories. We make directories like this:

```
mkdir test
```

```
ls
```

This is easier to remember than some commands because it is short for “make directory.”

The next thing I want to talk about is **auto completion**. The easiest way to explain this would be to create a bunch of directories:

```
mkdir pumpernickel kombucha
```

Now when you type cd p and press tab, it will jump to pumpernickel. If you have two words that start with P, it will go for the only one that matches.

```
mkdir plump
```

Now if you type cd pu and press tab, it will go to plump. If you type cd and press tab, nothing happens. That's because it doesn't know which one you want. If you press tab again, it will display all of the options available to you.

The next thing we are going to talk about is **echo**. Echo is a command that can be used to get things to display in the terminal. This is strictly Linux here so this is different than how we did this in C.

```
echo "Hello World"
```

We can direct this output to a file:

```
echo "Hello World" > output
```

```
cat output
```

When we do it again though, you can see cat shows the same thing. That's because every time we run this command it deletes everything in the file. If we want to append to the end of the file, we need to use >>.

```
echo " Nice to meet you" >> output
```

```
cat output
```

Sometimes commands can use the < symbol to get input from a file. We're not going to discuss this but I thought it would be useful to share in case you are interested in researching it.

We can also use the greater than symbol to direct other commands:

```
ls -la > lsOutput
```

```
cat lsOutput
```

You can see that this displays a ton of stuff at once, so if we have a huge text file we can actually scroll through the content using less.

```
less lsOutput
```

Now you are scrolling using the arrow keys, but how do you get out? You have to use q.

The last thing that I have to share is that if you are in the terminal and it seems to break and won't accept any commands, I'm going to give you a fix. For example, try typing in cat by itself and press enter.

You can see that it keeps going and going and going. That's because the command is waiting for extra information. Well we don't want to give the terminal extra information. So to get out of this never ending trap, hold control and press c.

This is going to come in super handy when you start writing code and writing programs that you can't get out of.

Tutorial 9 - Variables, Expressions, Statements

We started off our C programming with the hello world program. Before you move on, you really want to familiarize yourself with this basic structure. Don't freak out if you don't have it memorized!

The point is not so much to memorize it because that will come with time. Just have a rough idea of where to put what and where semicolons go. Also know that the spacing is not a concern to C. You can say that C is **whitespace insensitive**.

The next few videos are going to be designed to give you the basics of C programming. These basics are going to be various pieces of information that are going to help you, but I'm not going to go in a ton of depth. That's because after we go over the basics I am going to go in depth in many topics. I want you to be able to start working with C and not have to wait till you get to video 50 to know how to do some of the most basic things.

The first thing I am going to introduce to you is the concept of a **variable**. Now, the chances are you've worked with variables before either in some programming language or in math. We can create a variable in C very simply.

```
int x;
```

This is called the **variable declaration**. We are saying that the type of data we want to store in this variable is an integer. That means any whole number. Additionally, this entire line is known as a statement. A **statement** is when we tell the computer to do something. Think of it sort of like a command to the computer. There are tons of different kinds of statements, a variable declaration is just one type.

We've declared this variable. That means that it exists, but how do we give it a value? Giving a value to a variable is called **variable initialization**. We can initialize a variable like this:

```
x = 10;
```

The x is the variable, the = is called the assignment operator, and 10 is the integer we want to assign to x.

Now, anytime we use the value x, image just replacing it with the value 10.

Also, a very important word to know is **syntax**. This is the format of each command we can do in a programming language.

We can also do math in C.

```
int y = x/2;
```

This shows two important things. First, we can declare and initialize a variable in one command, and two, we can use the variable x anywhere the value 10 would be appropriate. This will give y the value 5. The way I think of variables is as a box containing some value. So we are putting the value of x/2 into y. That means if we were asked the value of y, we would say in this situation it is 5. Not x/2.

We could just have easily done int y = 10/2. The problem with this though is that if we use x a lot, we are not always going to want to use the value 10, or we might not always know the value of x. In this situation we are giving a value directly to x, but there will come a situation when we might ask the user for the value x.

The last thing I wanted to teach you in this video is the concept of an expression. An **expression** is anything that can be evaluated to one value. For example, x/2 is an example of an expression. This is important because variables can only contain one thing. Whatever expression you give it must be **evaluated**. You are going to hear the word expression all of the time.

In the next video we are going to learn about outputting the value of a variable into the console when we run the program.

Tutorial 10 - Print Variables using Printf()

In the last video we created variables x and y and gave them values, and I told you what they would store. But what if you didn't believe me and wanted to see for yourself? The easiest way to do this is to use the pritntf() function.

Now, we've used the printf() function to print a sentence in the terminal that said “hello world”, but now we are going to be learning how to print other things to the screen.

Let's first start with printing a number. What if we pass in just the value 9001 in to printf?

```
printf(9001);
```

In order to run this to see what happens, we actually have to recompile our program. Every single time we make any change to our program we have to recompile.

You can see that when we run it we get a compiling error. Let's go back to our code.

The reason we are getting a compiling error is because printf does not know how to work with numbers like this. In order to tell printf we want to print a number, we have to give the printf function what is known as a **format string**. A string is anything inside of quotes, usually consisting of characters, such as

```
"Hello World!\n"
```

```
printf("%i", 9001);
```

We can also add a newline character in here:

```
printf("%i\n", 9001);
```

This works!

Now, we can actually do the same thing with variables. That's because, if you remember me saying, an int variable can be used anywhere you are expecting an integer.

```
printf("%i\n", x);
```

This works great! The only problem is the program is not very descriptive when it runs. All it does is print the value. What if we want to do a bit more? We can actually add some of our own text inside of our string too. The only part that gets interpreted is the %i\n.

```
printf("The value of x is: %i\n", x);
```

If you want to print both variables, of course you could use two printf functions, but you can also use multiple variables inside of one printf function:

```
printf("The value of x is: %i\nThe value of y is: %i\n", x, y);
```

Each thing passed into the printf function is known as an **argument**. In this situation we have three arguments: our string, and two integers.

Later there will be a video devoted to format strings, conversion/format characters, and format specifier (begins with a % followed by a conversion character).

Tutorial 11 - Taking Input from User

This video we are going to discuss how to get a value for our variables from the user of our program.

Right now, this example is pretty simple. As we get into more complex programs, we are often going to want to ask the user for some value. Now, taking input can be a very complex and complicated topic. Because of this, I'll be giving you just the basics to get you started.

The goal of the application we are going to write is to calculate the area of a circle. So we are going to change some of our variables.

One thing that will become a big concern later is **data validation**. Let's say we ask a user for an integer and they give us something like "tacos." That can break our program. The topic of data validation is to make sure that the user inserts the correct kind of data. We are going to basically ignore data validation for now because it will require a lot of other knowledge, but I wanted to start off teaching you how to get input so you can work on building some cool programs.

We've already discussed the printf() function. This is a function for output. In order to use this function, you must include stdio.h. When we include that, we also get access to another function called scanf. This is the function we will start off with to get user input.

```
int radius;

scanf("%i", &radius);
```

We have to say what type of data to expect, which is why we use the %i. There are other format strings available, but this is the one we need for this situation.

The & behind the radius is called the **address-of operator**. This gets into some more advanced features of C, but what you need to know is that if you passed in radius by itself, and the compiler was okay with that, the function would get a copy of the value of radius and wouldn't actually be able to change the value of radius. We need it to change radius because this function is going to ask the user for a number and then assign that value to radius. It would not work without the &.

The reason printf() does not need the address-of operator is because it doesn't change the value, it just displays it on the screen, so it's okay to have a copy of the value given to the function.

We will get into all of that later because I don't want to distract from the purpose of this video. When we run a scanf(), it is going to just pause at the screen. Let's give it a try.

You can see that it doesn't tell the user what to do, so it's super confusing. Let's add a prompt.

In the next video we will worry about calculating the area. For now, let's just output this value to the screen to make sure it got the right value and that it is working.

```
printf("The given radius is: %i\n", radius);
```

Congratulations! Now you know how to do input and output.

Now we need to calculate the radius and output the value to the screen. That's what we are going to discuss in the next video.

The Address-of Operator (&)
The most critical and unique concept for input is the address-of operator (&), also known as the reference operator.

Syntax: When using scanf, you must place the & symbol directly before the variable name where the input will be stored (e.g., scanf("%i", &radius);).

Functionality (Why it's needed):

The & operator allows scanf to receive the memory address of the variable.

This is necessary because scanf needs to change the value of the variable in the program's memory by assigning the user's input to it.

Contrast with printf: The printf function does not need the & operator because it only reads the variable's value to output it; it does not change or modify the variable.

Tutorial 12 - Arithmetic Expressions

In the previous video we learned how to take input and put it on the screen. Now, let's create a new variable, area, and make a formula.

Before we start working on that formula, we need to discuss operators. An **operator** is a thing that takes some value and does something with it. We've discussed the assignment operator. The **assignment operator** takes a value and assigns it to a variable.

When it comes to writing formulas, we have to use arithmetic operators. You've seen some of these I'm sure, but just to make sure you know the most basic important ones:

+, -, /, and *.

We also have to consider the order of operations. / and * happen first from left to right, and then + - happen next.

We can also use parenthesis to force certain operators to go first.

Now, let's write that formula.

```
int area = 3.14159 * (radius * radius);
```

Now, do you guys notice anything wrong with this code? This is a perfect example of what some may consider to be a logical error. A **logical error** is when your program compiles, but it doesn't do what you expect it to do.

Take a moment to see if you can spot the problem. Let's run it and see what happens.

The problem is that we are storing this data as an integer. This is giving us results we are not expecting because values are being truncated to be an integer. **Truncation** is when we cut off the decimal values and do not round. so if we have 1.999999, it gets truncated to 1.

To fix this, we have to introduce some new data types. This int here is an example of a data type. A **data type** is a description of data that tells C how to interpret that data. To use decimals, we want to use a new data type called **double**.

When we set everything to double, we are a step closer, but there is another problem. These format strings, the %i is telling C that we are expecting integers. We are also going to have to change that to a new format which uses %d. Now everything should work as expected.

This is our very first useful program! It takes a number, runs it through a fancy pants mathematical algorithm and spits it back out. You can take what you've learned from this video and write software to solve some of your math homework.

Tutorial 13 - Basic Type Casting

Let's create another program to solve another math problem. This one is going to be a little less serious, because serious math can be boring sometimes.

We work for a chicken company who wants us to write software that tells them how many dozen eggs they produce in a day. So each night, a guy enters in how many eggs they have, and then the output is how many dozens that is. A dozen is just 12, so all we really have to do is divide the number by 12. Pretty simple, but it will illustrate a good programming point.

We are going to make a variable to store the number of eggs, but what data type do we want it to be? Well double would allow us to have a fraction of an egg, which doesn't make a lot of sense because they are not going to sell partial eggs. So integer will work fine.

```
printf("The number of eggs for the day: ");

int eggs;

scanf("%i", &eggs);
```

Now to calculate the dozen, all we have to do is divide by 12.

```
double dozen = eggs / 12;
```

The reason we want dozen to be of the data type double is because we could have fractions of a dozen. For example, we could have a half dozen (.5 dozen).

```
printf("You have %f dozen eggs.\n", dozen);
```

We can try this with a number like 24 and you can see that we get 2. This is correct! So it seems like our program is working. But wait…what happens if I put 18 eggs?

The answer is 1 dozen! What the heck?!

The problem here is that when we look at the line of code: double dozen = eggs / 12; we are dividing an integer by an integer. Whenever we divide two integers in C, we are always going to get an integer as a result.

There are two ways we can fix this. The first, is add .0 to the twelve. In this situation we no longer have two integers, so C will do this operation and give us a double value. So the lesson learned is that whenever you are doing arithmetic with an integer and a double, the double takes precedence. Meaning, C makes the result a double, not an integer.

Try it and see what you get.

The second option is using what is known as type casting. **Type casting** is the process of changing the data type of something. This is the more sophisticated way of changing data types and will work in more situations, but sometimes the previous method works fine!

```
double dozen = (double) eggs / 12;
```

In this situation we are casting the eggs to be of double data type. Now, the result will be 1.5.

You need to be careful with type casting. If we do something like this:

```
double dozen = (double) (eggs / 12);
```

The division gets evaluated as integer division, which gives us 1. Then, we convert 1 to a double and get 1.0.

So be careful and do lots of testing when working with type casting.

Tutorial 14 - Working with Strings

When we first started this series, we created a program where all it did was say “hello world” on the screen. It looked something like this:

```
#include <stdio.h>

int main()
{
	printf("Hello World!\n");
	return 0;
}
```

Well, what if we wanted this to say something like our name?

We could modify the program by replacing the printf() line with this:

```
printf("Hello %s, You look nice today.\n", "Caleb");
```

We are passing in an argument what is known as a **string**, which always has double quotes. You can think of a string as a bunch of characters strung together.

This is cool and all, but what if we want to ask the user using our program what his or her name is?

In order to do that, before we print we need to create a variable. We'll discuss the specifics of how this variable works in a future video, but we are going to be creating what is known as a char array.

```
char name[31];
```

This allows the user to put in up to 30 characters. You actually don't want to use the last character because that character is used to say it is the end of your name. If you don't reserve that last character, the computer might think your name goes on longer than it really does. The last character in that string is called the **null terminator**.

Now, we can get input using scanf();

```
printf("Yo gurl enter yo name: ");
scanf("%s", name);
```

Now, in previous videos we had to use an ampersand before the variable name, but we actually don't have to do this with arrays, as we will discuss more in the future when we start working with arrays and pointers.

scanf() only accepts one word, so you can't put your entire name, just use your first name.

If we run the program, it still does the same thing. We are getting the input but not doing anything with it.

Lastly, we can modify our final printf():

```
printf("Hello %s, You look nice today.\n", name);
```

By the time you get through this series, this stuff will be easy. I just wanted to give you some of it now so you could start making your own cool programs!

The Role of Whitespace as a Delimiter

When scanf() attempts to read a string using the %s format specifier, its built-in behaviour is: It skips any leading whitespace. It starts reading and storing characters until it encounters the next whitespace character .When it hits that space, tab, or newline, it stops reading the string, even if there are more characters in the input buffer. It then automatically adds the null terminator \0 to the end of the characters it successfully read, creating a complete C string (the first word).

Tutorial 15 - Using Functions in C

We've had a small number of videos so far covering C in general. The goal of the previous videos was largely to introduce you to a number of the beginning features of C. As we move on, we'll go into a bit more detail.

One thing I will discuss in a lot of detail later on is called functions. I thought I would give a video to introduce them to you so as we use them you are not totally confused.

A **function** is essentially a section of code that can be called numerous times. For example, let's say you are outputting three things to the console, and you are going to do that every minute or so to see how certain data changes over time. So let's say we are outputting the value of x, y, and also the current time.

Well, the code we would use would look something like: printf(…); printf(…); printf(…);. Then, when we need to output the data again, we can call the same three statements again.

This can work, but it is not the best. For example, let's say you output this data a few times, but now you changed your mind and would like to update the format that is being outputted. Now you have to go back and change it in every location. This is very error prone.

Instead, we could write what is known as a function, such as printData(), and we would define all of the code to be inside of that function. Later on we will get into how to make functions, but for now let's just say we made that function and everything is working. Everywhere that used to have the repetitive code can be replaced with a call to that function.

If you want to update the format now, all you have to do is change it in one place, reducing errors and making coding easier.

This is obviously a very simple example, but functions can be used to do a lot of cool complex things. You will see functions every day of your life when programming. In fact, printf() is a function itself. Somewhere there is code that knows how to output stuff to the console, so all you have to do is give some data to the printf() function.

So, this video is more about focusing on using functions rather than creating them, and printf() is a common function that you are going to use. It allows you to give data to be printed. When you give a function data, the computer sciency term is called **passing** data. Specifically, you can say you are passing data as an **argument**.

So, let's say we have something like this:

```
printf("%d\n, 55);
```

We are passing two arguments to the printf() function.

Don't feel like you have to understand everything about functions. Usually functions aren't really focused on until you have a good understanding of the basics of C programming. Later on we'll talk about creating function, taking data through parameters, and returning data.

Tutorial 16 - Comments

This video is going to introduce you to the concept of comments. In general, comments are pretty easy to understand. I'm going to try not to waste your time by giving you some extra information relating to comments that may come in useful for you when you work on bigger projects.

To do a single line comment, use //
Anything after that will be ignored.

The comment allows you to type in whatever you want and the compiler does not care if it is here. Run the program and see that it does exactly the same thing.

A multiple line comment allows you to put in comments as long as you want and it will continue to the next line…

```
/*This
is a multiple
line comment  */
```

These are useful for when you want to make a block comment that is usually used for describing a program or something:

```
/************************************
Author: Caleb Curry
Description: Calculates how many dozen eggs
Date: 03/17/2017
************************************/
```

These actually do not have to be on multiple lines. I'll show you an example where these comments might come in handy.

```
if (eggs < 0 /* Invalid value */)
{
	//Do something
}
```

We haven't discussed if statements, but here is a sneak peak of what is going to be coming up in the future. You can see that I was able to throw a comment in the code and then continue coding on the same line. Run this and see that it does exactly what you expect it do.

Now, a lot of people use comments to describe what some confusing code does. It's super important though that you always keep comments up to date. Nothing is more confusing than reading a comment that says code does something when it really does something else.

When should you comment? Well, if you see any kind of confusing math or confusing syntax, leave a comment before it to explain it. Imagine you are going to read it in 5 years and you want to know what it is doing.

When there is some piece of code you need to remember to implement or something you need to remember to fix, you can start the comment with TODO. That way, when you are finishing up your software you can do a search for TODO. To search in Vim all you have to do is type / followed by the word you are searching for:

/eggs

Then press n to go to the next value of it.

When to not comment? Don't comment on anything that is obvious. Often in college they want students to leave a comment on everything. Personally, I think this is a bad thing to teach students as it teaches them to over comment on things that should be clear from the code itself, but whateverrrrrrrr.

Honestly, I would avoid using large comment blocks at the top of a program when possible. I think the only thing you should really write at the top is the purpose of the program. You can put them for now, but once you start working on more sophisticated software, you are going to use a software such as Git which is going to keep track of all dates and people anyways, so it's a waste of effort and just another thing that can be wrong. For example, you might get hired at a company and all of their code has the author of the previous guy who worked there who got moved to a new department. Are you supposed to change it to you? Add your name? It just introduces confusion.

In this video I taught you about comments, but I also touched on how to search in Vim. Vim might not be so crappy if we actually know how to use it a bit. So in the next video we are going to learn some ways to make Vim a little bit more user friendly.

Tutorial 17 - Vim Basics - Part 1

In this series we have been using Vim. Now, you don't have to use Vim if you don't want. That's fine. I too am very new to Vim, I just started using it this year. When I first started using it, I found it to be very annoying, like it made everything more complex than it had to be. If you feel that way, I understand. But once you learn some of the commands, you can see that it's not so bad. One of the huge benefits of using Vim is that it is widely available. This means that if you have to connect to another computer through the command line, the chances are you can edit text files on the remote computer using Vim. Cool.

Let's open a file in Vim.

vim tacos

The first thing I want to do with vim is make it not so ugly. The way we can do that is turn syntax highlighting on!

:syntax on

We can also add a line count:

:set number

This is great and all but watch what happens when we close and reopen Vim?

:wq

vim tacos

"Vim configuration" is the set of user-defined settings and options that control how Vim looks and behaves.

You can see that the syntax highlighting and line count goes away. To fix this, we have to edit a file for our Vim configuration. So let's quit Vim.

:q

cd
vim .vimrc

Loading on Startup:
When you open Vim (e.g., by typing vim or vim <filename>) next time:
- Vim starts up.
- It automatically searches for and reads the contents of the .vimrc file in your home directory.
- It executes the commands it finds inside, which are syntax on and set number.
- As a result, your Vim editor will now always launch with syntax highlighting and line numbers turned on, solving the original problem of the settings disappearing.

Put this in the file:

syntax on
set number

It's okay if it is a new file. It should still work as long as we are in the home directory, which we are from running cd.

The next thing I wanted to discuss is the proper way to move the cursor around in Vim. Let me open a small program to show you some commands. So far you've probably used the arrow keys to move your cursor around in Vim. This is fine starting out, but a lot of people actually do it a different way. When you use the arrow keys, you often train yourself to stay in insert mode all of the time because you can move around while in insert mode. Well, you are actually not supposed to stay in insert mode, but are supposed to only be in insert mode when you are about to type.

The recommended way to move your cursor around is with h, j, k, l. Give it a try and see if you can get used to it. Now, when you see something that needs changed and you want to move there, enter into insert mode, and then exit insert mode. Let's give it a try.

Now, whether you use the arrow keys or not is ultimately up to you, but either way you will probably want to try to get into the habit of only being in insert mode when you need to insert something and not all of the time. That's because when you are not in insert mode there are other commands you can run to make coding a lot faster.

One of the easiest ways to move around is using G. Type a number followed by G to go to that line.

4G

G by itself goes to the very end.

$ goes to the end of a line

^ goes to the beginning of a line.

a will enter insert mode after the character you are on.

w is used to move to the beginning of the next word

b is used to move to the beginning of the previous word

In the next video we are going to discuss how to copy and paste, undo and redo, and other cool stuff like that.

Tutorial 18 - Vim Basics - Part 2

In the last video we talked about some tricks when it comes to moving around in Vim. This video is going to teach you even more important things to help improve your efficiency.

The first thing I wanted to talk about was different ways to quit the application. The very first is

:q!

This is how you quit without saving. If you wanted to save your changes you could have used :wq, for write and quit, or a short cut is just ZZ without the colon. You should also know that anytime we have a : in our command it allows us to type out a larger command where we actually have to press enter. The

other commands we learned about in the last video we didn't have to press enter and they didn't show up on the screen like this.

Now let's reopen it.

What if you want to be able to temporarily close Vim without having to decide to save your changes or not? Kind of like just keeping it open in a new tab or something? Well you could always open a second terminal, which we may do at some point, ha!
But the more professional way is to use ctrl c. This will put Vim in the background and to bring it back up you type fg, for foreground.

Occasionally when working with vim, you might close the program wrong or your computer shuts down and the next time you try to open a file it gives you a warning message saying you were in the middle of working on the file. If that ever comes up, you'll probably want to hit R for recover. We haven't experienced that yet and as long as you are careful you should be able to avoid it as well.

Now, whenever you mess something up big time in Vim, you can undo using

u

You can redo using

ctrl r

You can cut using

dd

This will put whatever into the clipboard. The **clipboard** is the place where things go when you copy or paste them. yeah, I have no idea where the clipboard actually is, but it's there.

To paste after the cursor

p

You could also use a capital P to paste before the line.

To delete the remainder of the line, use

d$

Remember from the last video, $ is the end of the line.

To copy the line you are on, use

yy

(think yank)

You can often mix these commands with numbers. So to copy 4 lines, use

4yy

That should be enough commands to get you started. Now, like I said, I'm still a complete beginner with Vim, so be patient with me. I'll share a page with you that has been helpful to me.

https://vim.rtorr.com/

Tutorial 19 - Intro to Data Types - Part 1

This video is going to introduce the concept of data types. First, let's define data. Data is any piece of information. Think of a variable. Let's say we have a variable called tacos, and the value of this variable is 5. The 5 is a piece of data (datum). Now, each piece of data has an associated data type.

A data type defines how the computer should treat a certain value.

As we learned before, integer division is completely different than double division.
5/2 = 2, but 5.0/2 = 2.5.

We can group data types into groups that help us understand what they are. The first grouping I would say is called **primitive data types**. These data types can be considered the building blocks. They cannot be broken down further into new data types.

Inside of the primitive data types we have numeric (number) data types, a char data type, and a _Bool data type. Numeric data types are used to store numbers, the char data type is used to store a single character, such as ";" or " ", and the _Bool data type is used to store data in one of two states. These values usually represent true or false, on or off, high or low, yes or no… anything that is binary (two states).

The primitive data types are foundational to C programming, but once we get further into things you are probably going to want to build more complex data types.

So the second category is **complex data types**. That is where structs come in. A struct allows you to define your very own data type. This means that we could make a more complex data type such as date.

Now we can actually create a variable that stores a date. So we could do something like:

```
date today;
```

similar to how we can do:

```
int x;
```

We can take all of these puzzle pieces and build things up to more complex things. For example, we can combine multiple structs to make a struct containing structs! A common application of this is a linked list.

So starting off the data types are pretty simple, but once we get into things we can make some very cool stuff with some custom data structures.

Some languages have what are known as objects, which are very similar to structs, but C does not have object oriented programming, so you don't even have to worry about that. Learn structs well and they will help you understand objects when you learn other languages like C++ or objective C.

Tutorial 20 - Intro to Data Types - Part 2

There are also other data types you will come across. One example is an array. An **array** can be used to store multiple values in one data type. So we could make an array of integers, which would be a collection of integers, where each are called **elements** of the array.

We can also create a character array. A character is just an individual letter, number or symbol from the keyboard.

The null terminator's sole purpose is to indicate where the string officially stops. When you declare a string literal like "Hello", the compiler automatically appends this \0 character.

In C and C++, a string is fundamentally treated as an array of characters in memory. Unlike other data structures, the string itself does not inherently know its own length.

Now, an important type of data that every programmer needs to know is that of string. A **string** is a group of characters surrounded by double quotes. "I like cats" is an example of a string. How do we go about storing something like this in a variable in C? Well we actually use an array of characters. So think of the string as a collection of characters in a sequence. There is something special about storing strings in that you have to end them with what is called a **null character**. We'll get into all of the details later on as this is just an intro, but essentially, we have to make the array one element longer to store a fancy \0 character, which is called the null character, or null terminator.

The null terminator is often represented as \0 or a byte with a value of zero (0).

Finally, another extremely important data type category we have are pointers. Pointers are an incredibly powerful tool in C that I hope to discuss with you guys soon.

We also have an important term known as **constant**. A constant is any value that does not need to be evaluated. So 5 is a constant, it's not some function call or some value that is retrieved from the user.

Another thing you are going to want to be familiar with is how to write constants. Now, the term constant has many meanings, so let me explain. Think of a constant as something that you can write the value of down. Certain constants are going to be interpreted different ways. When you declare a variable, you have the privilege of giving it a data type, so there is no confusion, but if you just write down a number, C has to interpret what type of data that is.

For example:

3/2 and 3/2.0 produce two entirely different results. That's because C treats the data a certain way depending on the data type of the constant value.

If you want to make a character constant, you have to single quote it, like 'k', but if you want to make a string constant, you use double quotes, like "pi$$a". You could also do something like "H", as this is a string with one character, not just a character…THERE'S A DIFFERENCE!

What's the difference between '3', "3", and 3? The first is a char, the second is a string constant, and the third is an int.

With numbers it's a bit more vague because there are actually a lot of different number data types and you don't have the difference between single or double quotes, because number constants do not use quotes. There are ways to be very specific when you want to tell C a constant is of a certain data type. For example, you can end a number with the letter F to say that it is specifically a float. Otherwise C assumes the constant is a double. But those terms may be foreign to you, so that's what we are going to be discussing in the next video!

The float data type is called single precision and the double data type is called double precision primarily because of the number of bits used to store each value, which directly dictates their precision (number of significant digits) and range.

1. Bit Size and Storage:
The terms relate directly to the memory occupied by the data type:
- A float typically uses 32 bits (4 bytes) of memory. This is considered the "single" standard size for a floating-point number.
- A double typically uses 64 bits (8 bytes) of memory. This is double the number of bits used by the float, hence the term "double precision."

2. Precision and Range:
The extra bits in double are primarily used to increase the size of the significand (mantissa) and the exponent, which enhances both the precision and the range
- Precision: The ~15-17 significant decimal digits available with double is more than double the ~7 digits available with float, thus providing double the precision in a practical sense for real-world calculations.
- Range: The larger exponent field in double allows it to represent a significantly broader range of magnitude (larger and smaller non-zero numbers).

Tutorial 21 - Int, Float, and Double Data Types

Now, we've discussed data types some, but this video is going to be devoted to discussing numeric data types. **Numeric data types** are used to describe numbers. So any number in our program can be classified as one of many numeric data types.

There are a ton of numeric data types, but most are just modified versions of the three main numeric data types: integer, float, and double.

int stands for integer. **Integers** are whole numbers, that is, any number with no fractional part. Integers = {…, -2, -1, 0, 1, 2, …}.

Floats and doubles are used to store numbers with numbers after the decimal point. So think .5, or .2342, or 1.2, or 430.0. These two data types collectively are called floating point data types. That can be kind of confusing because one is called float, but they are actually both called floating point. The float data type is called single precision floating point, while double is called double precision floating point.

The float data type is called single precision and the double data type is called double precision primarily because of the number of bits used to store each value, which directly dictates their precision (number of significant digits) and range.

So let's say you wanted to store how many dogs you owned… Are you going to want to use int, float or double? Probably int, because you should not have a fraction of a dog for a pet. That's just messed up.

So if there are two data types for storing numbers with fractional parts, how are you supposed to know which one you should use? It all has to do with precision. Essentially, when you declare a variable of type double, it takes up twice as much storage as float, but it allows for around twice as much precision. So if you want to store something like the square root of two, using a float will give you only so many digits, and using double will give about double as many digits.

In general, I prefer to use doubles for about everything.

So we are going to practice creating some variables and outputting them to the screen.

To start, we will output an integer. This is super easy. You can use either %d or %i. They both work exactly the same. You think of %i being short for integer, if that helps you.

You can see that declaring and outputting an integer is a piece of cake.

Now, when we output a floating point number, we need to use %f. Let's give it a try.

Super easy.

Now, there is something important you need to know about the float and double data type. Due to the way these data types are stored, they cannot represent every value perfectly. For example:

```
float pizza = 1111.1111;
printf("%f\n", pizza);
```

When we run this, the output may not be what you expect.

This has to do with the fact that certain numbers are impossible to represent with a finite number of binary digits. If you have no idea what I'm talking about…just trust me for now. We can talk about it

some other day, ha! I do think we should talk a little bit about how numbers are stored, though. They are actually stored in scientific notation. We are going to discuss how this works in the next video.

Base Conversion Issues (Decimal to Binary):
Many simple decimal fractions such as 0.1 do not have a finite or exact representation in binary (base-2). Their binary representation is a repeating, infinite sequence—just as 1/3 is 0.333... in decimal.
For example, 0.1 in binary becomes 0.00011001100110011..., repeating indefinitely.
Because a computer cannot store an infinite sequence, it must round or truncate the number to fit within the limited number of bits available. This necessary rounding introduces small errors even before any calculations are performed, and is the primary reason for inexactness in floating-point representation.

Limited Memory (Finite Precision):
Computers use a fixed, finite number of bits (like 32 or 64) to represent numbers, particularly floating-point numbers (decimals).
- Real numbers are infinite: There are infinitely many real numbers, even between two integers.
- Storage is finite: Since a computer only has a fixed amount of memory to store a number, it can only hold a certain number of significant digits. Any digits beyond that limit must be discarded, and this is where rounding error occurs.
- Approximation is necessary: The stored number is an approximation of the true mathematical value.

Tutorial 22 - Scientific Notation with Floating Point Numbers

Now that you've seen the basics, it's important for you to understand how floating point numbers are actually stored. Floating point numbers are stored in scientific notation.

If you need a refresher on scientific notation, that's fine. Essentially, we can represent a number as multiplied by 10 to some power.

So 24000 becomes 2.4 X 10^4 = 24,000.

Scientific notation is a way to write very large or very small numbers as a product of a number between 1 and 10 and a power of 10. To convert a number, move the decimal point so that only one non-zero digit is to its left. The number of places you move the decimal is the exponent for the power of 10. If you move the decimal to the left, the exponent is positive; if you move it to the right, the exponent is negative.

We can also have 10 to a negative power. So .000045 becomes 4.5 X 10^-5.

Mantissa: The part of a floating-point number which represents the significant digits of that number.

The number that is being multiplied actually has a fancy name. It's called the mantissa. They call the other part the exponent, but always remember it's times 10 raised to that exponent, not just the number itself. 2.4 to the fourth power is much different than 2.4 x 10 to the fourth power.

Sometimes it is actually easier to input our data in the form of scientific notation. The way we do that is to use an e. This basically means "times 10 to the power of" and then use a number either positive or negative. Let's try converting our number to scientific notation for the input and make sure we get exactly the same output.

```
int dogs = 2.5e4; // same as dogs = 25000
```

This is a great time for me to introduce a new keyword to you…**hardcoding**. Hardcoding is when we type in a value in the code directly rather than getting the value from the input, a database or something like that. Sometimes hardcoding is okay, but in this situation we don't want to hardcode the value of these variables. As we have it, this program can take a couple specific values and output them. But imagine if we took any input in the form of scientific notation and it converted it to decimal notation. Then our program would be legit sauce. That's what we are going to be talking about in the next video.

Floating-Point Numbers and Scientific Notation:
The first video explains the concept of how floating-point numbers (like float and double in C) are stored in memory using a variation of scientific notation.
Scientific Notation Fundamentals:
Scientific notation is a method for expressing very large or very small numbers in a compact and standardised form. A number is written as a product of two parts:
1. Significand (or Mantissa): A number with only one non-zero digit to the left of the decimal point.
2. Exponent: A power of 10.
Scientific Notation in C:
In C programming, you can directly represent floating-point values using this notation, where the letter e (or E) stands for "times 10 to the power of."
The format for hardcoding this value in C is: [Significand] e [Exponent].
* Example: To represent 25,000, you would write 2.5e4.
* Example: To represent 0.00000012, you could write 1.2e-7.

Tutorial 23 - Format Characters for Float and Double

Our program starts with a hardcoded floating point number in scientific notation and outputs that number in decimal notation. What I want to do is explore some of the other ways we can format this data and also how to take the input directly from the user rather than hard coding it.

First, let's learn some format characters.
%f - display the number in decimal notation
%e - display the number in scientific notation
%g - let the computer decide how to format it

Should be greater than or equal to 6, I think!

> Formats it using scientific notation if the exponent is less than -4 or greater than 5. Otherwise, it uses decimal notation.

We will output our number once using each of the three format characters.

Now, we are going to learn how to take floating point numbers as input.

As I mentioned before, you have the option of using either the float or double data type, but I think you should be using double for nearly everything, so that is the data type we are going to be focusing on. Use whichever floats your boat (lol).

Format characters for taking input are similar, but you need to prefix it with an l.

```
scanf("%lf", &var);
```

You also need to prefix the variable name with the address-of operator.

You can actually use this format character to take data in either decimal notation or scientific notation, so you only have to remember one thing. You can use e or g instead of the f if you prefer as they can all be used to read data in either notation.

When we print, you can use %f, %e, and %g for both float and double, but with scanf, float and double are different. If you want to specifically read in a value as a double, you need to use the l prefix. Otherwise, you are reading in a value as a float. so scanf(%f, &var); is saying that you want to read in the value as a float, which we do not want to do.

I'm going to give you some more advanced information, so if you don't understand it all yet, that is okay. We will get to it in more detail in upcoming videos.

The reason there is a difference between float and double for scanf() but not for printf() is that when we pass in a float to printf(), the function actually copies the value and stores it in a new variable of type double. This means that both float and double can be treated the same by printf().

With scanf(), we are using the address-of operator, which means we are actually passing in a pointer. Pointers stay the same in that if the original data type is float, the function will treat it as a float. This means we end up with two versions, %f for float, and %lf for double, depending on what you want to convert the input to. Let's finish the program up!

The text mentions %le or %lg as alternatives:

In the context of input (using scanf), the format specifiers %lf, %le, and %lg are all effectively identical. They all tell the program to read a double and can handle the input in either decimal or scientific notation.

The difference between %f, %e, and %g is mostly relevant for output (using printf), where they control how the number is displayed (decimal, scientific, or shortest form).

Tutorial 24 – ASCII

We are now going to begin our discussion of the char data type.

Now, char is short for character. A **character** is any number, letter, or symbol that can be used on a computer. That's not all of the characters, though. There are actually some characters that do certain things. For example, there is a tab character, there is a newline character, there is a space character, there is even a null character. A **null** is the absence of something. So yes, the null character is literally a character that is nothing. But it is still a very important character as we will see later. There are many different types of characters.

How do we know what is a character or not? Well, there is a collection of allowed characters. This is known as a **character set**. The character set that I am going to be teaching you about today is called **ASCII**.

Now, before we dive into ASCII, I wanted to discuss something called **binary**. When I say binary, you should automatically think of something that is in two states. Binary is the most fundamental way that the computer stores information.

Imagine that we have a number, and this number has only two states. The possible values we could use are 0 and 1. This is called a **bit.** a single bit can be used to represent anything that is in two states, such as on or off, high or low, hot or cold, she loves me or she loves me not, etc.

Well, computers use binary to store information. If we have a bunch of zeros or ones we can start to build more complex information. We will often have groups of bits. One of the most common sized group of bits is called a byte, which is 8 bits. For example, we could have 01000001.

Now, back to ASCII. ASCII will use this to represent 1 character. So 010000001 represents the character A.

Now, the standard ASCII character set is only going to use 7 of the 8 bits. Sometimes it will use 8 and this is known as **extended ASCII.** With 7 available bits we can have a total of 128 possible characters. This includes the entire English alphabet (capital and lower case), all numbers, a bunch of symbols, and some special characters such as the newline character.

Now, how does the computer know that 01000001 is A? This has to do with how the binary is interpreted. That's because this binary number could just as easily be a way to represent some number. In fact, if this was considered an integer, this number is 65.

So you can directly convert between binary and integer data. Every ASCII character can be represented as a number. In the next videos we will be using the char data type and learning more about converting between char and int.

Tutorial 25 - Char Data Type

We can create a char variable like this:

```
char var = 'A';
```

It is very important that we use single quotes.

That's really all there is to creating a char variable, so now let's talk printing this to the console and getting it from user input.

We are going to need to learn a new format character, specifically %c. This tells the function that we are going to be working with char data.

```
printf("%c\n", var);
```

Now, if we actually wanted to scan the data from the console, we can use scanf():

```
scanf("%c", &var);
```

If you want to know more about ASCII, check out http://www.asciitable.com/

The first section on the ASCII table are called control characters and only a few are going to be useful to us. I'd also group the last one in here, which is the DEL ASCII code. These codes were for controlling devices, such as printers. Some are still going to be useful to us, such as the new line character. The rest of the standard table are called **printable characters**. These are the characters that you can visually see. You can also see that each character has an associated integer value, 0-127.

The second table represents the extended ASCII, which gives more available options.

In the next video we are going to create a program that will convert between int and ASCII. By the end of that video, you'll see how easy working with ASCII is.

Tutorial 26 - ASCII and Int Conversion

In the last two videos I've laid the foundation for understanding character data and integer data. Be sure to check those out.

In this video we are going to be writing a program that will take a character and convert it to an integer.

First, we will cover how to convert from ASCII to an integer. Then, we will convert it back.

The next thing we will do is show that you can do math with characters. That's because they're really just numbers in disguise.

```
 #include <stdio.h>

int main()
{
        char ASCII;
      printf("Please enter a character: ");
      scanf("%c", &ASCII);
      printf("The int value of your character is: %d\n", ASCII);

        int integer;
      printf("Please enter an integer between 0 and 127: ");
      scanf("%d", &integer);
        printf("The ASCII for that int is: %c\n\n", integer);

        return 0;
}
```

Now, let's try to do some math with characters:

```
        //Math with ASCII:
        char mathz = 'A' + '\t';
      printf("A(65) + \t(11) = %c(%d)\n", mathz, mathz);
```

You can see that the \t is actually rendered. Remember how if we want to go to a newline we use \n, well we can also use \t! To actually print a backslash we need to use two back slashes.

```
        //Math with ASCII:
        char mathz = 'A' + '\t';
         printf("A(65) + \\t(11) = %c(%d)\n", mathz, mathz);
```

Tutorial 27 - _Bool Data Type – Part 1

In this video we are going to be discussing Boolean data. **Boolean** data is anything that is either true or false.

Now, if you remember back a few videos ago when I started talking to you about binary, I mentioned how a single bit can be either 1 or 0. It works pretty similar with a Boolean variable.

To create a Boolean variable, we use _Bool (the B must be capitalized):

```
_Bool calebIsFat;
```

Notice the naming convention here. I have decided to lower case my name to follow what is known as **camel-case naming convention**. That is where the first character of the first word is lower case, but every word after that is upper case. Another common convention is to use all lower case with underscores to separate the words. Whatever you use is up to you:

```
_Bool caleb_is_fat;
```

Now, the reason I named this variable calebIsFat is not because he is, but rather that he might be. I'm not saying that Caleb is fat. I'm saying that the statement "Caleb is fat" is either true or false. This is the very foundation for what is known as Boolean logic, which you will find when you study discrete mathematics, computer engineering, computer science, and a ton of other cool stuff.

Now, we need to assign a value to this variable. This means we have to ask ourselves…Is Caleb fat? Well, I wouldn't go so far as to say I'm fat, but I could use some trimming around the waist. In fact, I had to buy new dress clothes for my videos because I'm so fat. Therefore, I am going to go with yes.

```
_Bool calebIsFat = 1;
```

To assign true to this variable, use 1 (which represents true). If you want to assign false, use 0. Actually though, if you store anything besides zero, it is stored as 1. This means that anything that is not zero is stored as true. So to express my fatness, I could do this:

```
_Bool calebIsFat = 9001;
```

Now my fat level is over 9000.

How do we output these values though? Well, you can output just as if it were an integer:

```
printf("Is Caleb fat(1 is true, 0 is false)?: %i\n", calebIsFat);
```

You can see that according to my program, indeed I am fat.

Now, this is cool and all, but some of this isn't very intuitive. Like, why is it called _Bool? You also have to remember that 1 is true and 0 is false, and it's just plain hard. I have good new though! There is an easier way to represent Boolean data, and that is what I am going to talk about in the next video.

Tutorial 28 – The bool Data Type

In the previous video we learned how to use the _Bool data type. I recommend you watch that video before you watch this one as it has fundamental knowledge that will help you here.

If you've noticed, it seems that all the data types in C are lower case. We have int, double, float, char, but this _Bool had to go ruin everything for us.

Well, there is a way we can actually make this just bool. You first have to include a header file:

```
#include <stdbool.h>
```

Now, we can declare a variable like this

```
bool calebIsFat;
```

Not only does this allow us to use the bool data type, but to make our lives easer we can use the words true or false rather than 0 or 1.

```
bool calebIsFat = false;
```

Yup, I exercised since the last video.

One important thing to note here is that the word false is not in quotes. We are not giving the variable a string with the word false in it. We are giving the value false with no quotes.

The way these work is no different than _Bool, it just makes it easier to work with. In fact, the values stored are still zero or one. Consider this example:

```
bool CalebIsFat = true;
printf("%i\n", calebIsFat + 10);
```

This will print the value 11 because the value of calebIsFat is 1.

Now, there is not a format character to print true or false, so for now the easiest way to print this is to use the same thing in the previous video, where we just print it as an integer. This will still work where 1 = true and 0 = false.

Later, when we get into something called conditionals, we could write a program that would say Caleb is fat or Caleb is not fat depending on the value of this variable, but for now, this is good.

Tutorial 29 – Variables

This video is going to discuss variables.

Now, I know, we've said a lot about variables, so this video is going to be pretty short.

The first thing I want to make sure you guys are clear on is the difference between defining a variable and initializing a variable. We can define a variable like this:

```
int x;
```

We can initialize a variable like this:

```
x = 5;
```

We can do both at the same time like this:

```
int x = 5;
```

You don't always have to initialize a variable. For example, if you are going to get the value of this integer from user input in the console, you do not need to initialize it. That's because when we use scanf() a similar function, it will assign that value to the variable for us.

You always have to be careful not to use an uninitialized variable. Let's figure out why:

```
int x;

printf("%i\n", x);
```

Run this and you'll see that the value of x is completely meaningless. This is not the kind of data you want showing up in an application you write.

The next thing I want to talk about is naming variables. As you know, we as programmers are allowed to choose our variable names. But there are some rules and some guidelines that I have for you.

The first thing is the characters you are allowed to use are all letters, numbers, and the underscore character. The only thing is that you have to start with a letter or underscore, not a number.

Additionally, variables are case sensitive. What **case sensitive** means is that uppercase and lowercase characters are considered to be different when it comes to naming variables. So Tacos, tAcos, and TACOS are all different variables. Even though you are allowed to use different casing, be very cautious of having two variables with the same name but different casing because it can be very easy to use the wrong one.

As for symbols, the only one you are allowed to use is the underscore. You may get by with using some others, for example you can try creating a variable with a $ and it might compile, but that is not a universal statement and not all compilers are going to be okay with that.

Make your variable names descriptive. A variable named “amount” is super vague…amount of what? Additionally, I would recommend avoiding making your variable names too long. And please, please, please do not use shortened words unless you have a very clear system. Don't call amount amt. The variable named amountSold at least says what it is about, but using the variable name amtSld makes your code harder to read.

Another thing is that it is sometimes recommended to declare all variables at the beginning of a block (inside the main function for example), but this is less relevant today as the compilers are okay with declaring variables about anywhere. Because of this, I personally think it is more readable to declare variables that are going to be used throughout a section of code at the top, but declare variables that are going to only be used briefly right before the section where you need them. I find it annoying when I'm reading code that references a variable for the first time and the variable was declared way before then.

Tutorial 30 - Intro to Operators

In this video we are going to be discussing operators.

Now, the simplest examples of operators are the arithmetic operators. Think of something like this:

5 + 5

The + sign is called an operator, and each of the two integers are called **operands**. The entire thing is called an **expression**.

But why does it really matter? Haven't we already discussed how to do math in C like a million videos ago? Well the reason I am discussing operators more now is because there are actually many operators in C. There are so many that we are going to be spending the next section of videos discussing all of the

different operators that are available to us. And no, not all operators are used for math. When we get into more complicated math, we tend to use what are known as **functions**, rather than operators.

The easiest way to start thinking about operators is to separate them into groups. You know we gotta group everything!

We can group the operators by the number of operands they work on. So the + operator has two operands, the first number and the second number. Operators with 2 operands are called **binary operators**. The other two classifications of operators are **unary operators**, which only work on one operand, and ternary operators, which work on three operands.

As we go through the operators, I want you to resist the urge to assume that operators change the values of stuff, because only a few do.

For example, if we have the expression x + 5, the value of x does not get changed by 5. The value of x always stays the same. This may seem obvious now, but as we get into some other operators keep this in mind. The ones that change values I will explicitly tell you that they do.

Grouping by number of operands can be helpful, but there is a more useful grouping that I am going teach you about. That is grouping the operators by function. For example, the + in 5 + 5 is part of the arithmetic operators.

Arithmetic operators is just one category of many operators. These are used for arithmetic. There are also relational operators, assignment operators, logical operators, and so forth.

So throughout the next section of videos you will likely see me using both classifications. So for example, + is a binary arithmetic operator. This allows you to understand a lot about the operator before you even start learning what it does.

In the next video we are going to be discussing all of the arithmetic operators.

Tutorial 31 - Arithmetic Operators

This video is going to discuss the arithmetic operators. Now, we've discussed these some already and I'm sure you are familiar with them from math classes you've taken so this video is really not going to be that hard or confusing.

The **arithmetic operators** include the +, -, /, and *. These are all binary operators that need two operands.

The / is called a forward slash. Some people call \ a forward slash and they are wrong! The top of the slash is the reference point, not the bottom. The top is leaning forward, therefore / is called the forward slash.

The * is called an astersk. I like to call it capital 8, but I'm weird.

Now, when you have an expression like this:

```
int x = 2 + 3 * 4 / 3 – 2;
```

First the multiplication and division happens from left to right, whichever comes first. This means that 3 * 4 happens first. The expression could be simplified to 2 + 12 / 3 - 2. The next thing to happen is 12 / 3. The expression could be simplified to 2 + 4 - 2. Next, the addition and subtraction happens, whichever comes first from left to right. So we get 6 – 2 which evaluates to 4.

You may find it more clear if we did this:

```
int x = 2 + ((3 * 4) / 3) - 2;
```

You can use as many parenthesis as you want and wherever you want, as long as you remember to close any parenthesis you open. They always get evaluated from the inside out. So 3 * 4 happens first, then the result of that gets divided by 3.

Now, you can also use parenthesis to force a certain part of the expression to happen first. For example, if we wanted the subtraction to happen first, we could have set it up like this:

```
int x = 2 + 3 * 4 / (3 - 2);
```

Let's output this and make sure we get the result we expect.

Some people choose to use parenthesis for nearly everything because they believe without them it is easier to make mistakes with arithmetic, but this is up to you.

Now, I'm going to introduce you to a new operator…

Check out this expression.

```
int y = 5 % 2;
```

Now, I can tell you now that the value of y is going to be 1. Can you figure out what this operator does? We'll find out in the next video.

Tutorial 32 - Modulus Operator

This video is going to introduce you to a new operator called the **modulus operator** (%).

In the last video I gave you this example:

```
int y = 5 / 2;
```

I told you that the value of y is going to be 1. Did you figure out what this operator does? Well it actually takes the remainder. I've always been pretty bad at visualizing a remainder, so I use imagery. Imagine you have a pizza with 5 slices and you want to share the pizza with your pet rat, Pepper Jack.

Well you give a piece to Pepper Jack, a piece to you, a piece to pepper jack, and then another piece to you. Now, you have one left over:

```
int piecesOfPizza = 5;
int numberOfEaters = 2;
int leftOver = piecesOfPizza % numberOfEaters;
```

Now you might suggest just ripping the pizza in half, but you have to realize that piecesOfPizza is an integer. This means that we cannot split the piece in half.

This brings up the good point that the modulus operator is designed to work with integers. That's because if you were using numbers of type double here, you and your pet rat would both get 2.5 pieces of pizza and the remainder would be zero. The modulus operator would be useless working with floating point numbers.

You can do a lot of cool things with the modulus operator. One example is figuring out if a number is even or odd. if you divide a number by 2 and the remainder is anything but 0, then you do not have an odd number.

That is all for the modulus operator. In the next video we are going to learn our first unary operator.

Tutorial 33 - Unary Plus and Minus

This video is going to be talking about the minus and plus operator. But wait, don't leave yet. We're not talking about the binary minus and plus operators, rather the unary ones. As a reminder, the difference is how many operands they work on. The binary minus operator, for example, will take one operand, subtract the second operand from it, and generate a result from that. Neither of the operands are modified.

The **unary minus**, on the other hand, just takes one operand, and negates it. To **negate** something means to take the negative of. To make this clear, let's go through an example.

```
double money = 25;
double bill = 15;

double total = money - bill;
```

This example uses the binary operator. But now let's say you go to pay your bill and they say, you know what, we are going to give you some money instead:

```
double total = money - -bill;
```

Now, we essentially have 25 - -15, which becomes 40.

Now, what is the value of bill?

The value of bill is still 15. This operator does not change the value. In the next video we are going to talk about a unary operator that actually does change the operands value.

There is also a unary plus, which is less useful, so I'm not really going to talk about it. I would encourage you if you are an eager learner to look up the use of the unary plus operator.

Tutorial 34 - Increment and Decrement Operators

Let's say we have a variable and we want to add one to it.

```
int pizzasToEat = 123;
```

The hard way to do this is to go like this:

```
pizzasToEat = pizzasToEat + 1;
```

What the assignment operator does here is take the entire expression on the right and evaluate it to a value. So pizzasToEat + 1 = 124. Then, it assigns that value to the variable on the left. So now pizza will equal 124. We can output this data before and after to see this in action.

This is pretty great if you want to add a certain amount to a variable. For example, let's say the local pizza shop goes out of business and since I'm their biggest customer, they sell me 200 pizzas. We could do this:

```
pizzasToEat = pizzasToEat + 200;
```

If you are just adding one to a variable, there is actually a shortcut:

```
pizzasToEat++;
```

This operator is a **unary operator** because it only takes one operand, the variable. It is also unique in that it changes the value of the variable. pizzasToEat + 1 does not actually change the value of pizzasToEat unless you assign it back to the variable itself.

The ++ is called the **Increment operator** and it is one of the most popular operators in programming so please become familiar with it and practice using it.

Now, I'm going to introduce some trivia for you.

Let's clean up a bit…int pizzasToEat = 100; is the only thing we have in our code.

What will this output:

```
int output = pizzasToEat++;
printf("Pizzas to eat: %i \n", pizzasToEat);
```

5…4…3…2…1…

The answer is 100!

This is one of the trickiest things to get used to with the increment operator. We are actually saying we want the ++ to happen after the value is assigned. If we want to say increment the value before it is assigned, we put the ++ in front.

Now let's start with int pizzasToEat = 100 again. What will this output?

```
int output = ++pizzasToEat;
printf("Pizzas to eat: %i\n", pizzasToEat);
```

5…4…3…2…1…

The answer is 101!

We can print out the value afterwards and see that I'm not lying.

There is also a **decrement operator**:

```
pizzasToEat--;
```

This is the same as doing:

```
pizzasToEat = pizzasToEat - 1;
```

These two operator are very foreign to beginners so it is often neglected. So make sure you understand what is going on in this video! If you don't understand, keep trying examples and watch this video a few more times.

Tutorial 35 - Assignment Operators

In the previous video I showed you how you can add one to a variable. You can use the ++ operator. I also showed you what to do if you wanted to add more than one to a variable. For example, we can have something like this:

```
int pizzasToEat = 100;
pizzasToEat = pizzasToEat + 100;
```

The left side of this will evaluate to 200 and then that will be assigned to pizzasToEat. The = here is actually an operator of its own. It's a binary operator because it takes some value on the right and assigns to the variable on the left.

There is actually a collection of operators that exist to make this a whole lot easier. The syntax is a bit odd at first, but you will get used to it.

```
pizzasToEat += 100;
```

This will actually add 100 to the variable.

The name for this operator sounds just like it looks. It's called the **plus equals operator**.

There are a multitude of assignment operators. The most important ones to know are =, +=, -=, /=, *=, and %=. There are some others but they are very complicated compared to where we are right now, so we'll ignore those for now. The thing to keep in mind, though, is that these are assignment operators, so they are all going to change the value of the variable.

Let's go through some more examples.

pizzasToEat -= 100 will subtract the value of the variable by 100.

pizzasToEat /= 5 will divide the value by 5

pizzasToEat *= 2will multiply the value by 2.

pizzasToEat %= 2 will give the remainder when divided by 2.

These operators are great because once you are familiar with them they more clearly express what you are trying to do. They also reduce the chance of errors because you only have to type a particular variable one time, not twice.

Tutorial 36 - Operator Precedence

You may have heard from math class of this fancy thing called order of operations. We discussed this in a previous video so I'm not going to waste your time by repeating myself, but I am going to look at the same concept from a bigger point of view.

The point of the order of operations is to say which operators happen first and from which direction. For example, we know for the c operators, the multiplication happens first. This order in which operators happen is known as **precedence**. For example, you can say that multiplication has precedence over addition. And if we have multiple multiplications in one expression, it happens from left to right. The direction of which way a specific operator happens is known as the operator's **associativity**.

This is a good reference page that lists the C operators, their precedence, and their associativity.
http://en.cppreference.com/w/c/language/operator_precedence

A good example of left associativity is if we do something like this:

```
int x, y;
x = y = 5;
```

This is valid C and shows that we can declare two variables on one line and also assign values on one line. In general, I try to do things one step at a time, but sometimes you will encounter statements like these.

Although either way the result is going to be that x and y are both equal to 5. The order in which this happens is from the left and from the right. First, y = 5 will happen. Next, x = y will happen.

If you have two operators on one line, the first thing that happens is the operator with the highest precedence. For example:

```
int z =  20;
y = 2;
x = -y + z;
```

This could either be evaluated as:

```
x = -(y + z);  //x = - 22
```

or

```
x = (-y) + z;  //x = 18
```

The second is the one that will happen according to the operator precedence.

Tutorial 37 - Strongly Typed Vs Loosely Typed Languages

This video is going to be an introduction to how programming languages consider data types. As we know, there are many data types in C programming. C is an example of a strongly typed programming language**. Strongly typed** means that every piece of data has a specific type.

Imagine if we stored everything in strings. This would become a problem because the computer would never know how to work with certain data. For example, let's say we have "5" and "6" and we try to add them. Should the computer add the numbers to be "11", or should it combine the strings to be "56" (The process of combining two strings like this is known as **concatenation**). Concatenation doesn't work quite this way in C, but in general, these are the kinds of problems that come up when we do not have

things strongly typed. The opposite of a strongly typed language is called a loosely typed language. Now, a loosely typed language is not quite this loose.

A loosely typed language will usually have a few general types such as numbers, characters, or strings, but they are not nearly as strict with the typing.

What is a good example of this?

In C, if we do something like this: 1/3, the result is going to be zero. That's because the type of data being used are all integers. A loosely typed language is probably not going to be that obnoxious about types.

The benefit of using a strongly typed language is that it teaches you to be very discrete about what you want the computer to do. I think of a loosely typed language like water: it's flowy and forgiving. I imagine a strongly typed language as very strict. Using a loosely typed language can be more simple starting out because you don't have to worry about all of the different data types, but there is less protection as we will see soon.

For example, in a popular programming language known as JavaScript the type system is loosely typed. This means instead of having data types like integer, double, float, signed and unsigned, we just have numbers. Some people call the way JavaScript works with data types as duck typing…. Which is weird, but basically it says something like "if it looks like a duck, if it talks like a duck, it is a duck."

Let's say you write a function (which is just a section of code you can call multiple times) which expects a number. Whoever uses that function can pass in whatever they want, or pass in nothing at all.

Because JavaScript is not strongly typed, you have to do extra work inside of the function to make sure the function responds appropriately. What happens if somebody throws in an array? What happens if somebody throws in a string? What happens if somebody throws in a potato?

With a strongly typed language, you often have to pay extra attention to data types, but you have an extra layer of protection, too. Different people like different things.

In C, it is not easy to flow between data types. Because of this, we need something called type casting. That is what we will be discussing in the next video.

Tutorial 38 - Type Casting

In the previous video we mentioned how C was a strongly typed language. Imagine C as a grumpy old man who you owe a dollar bill to. Well, he wants his money now, but all you have is some change. you look at your change, grab four quarters, and find out that it adds up to exactly a dollar. But unfortunately, you owe him a dollar bill, not just 100 cents. So yeah, he beats you over the head with his cane.

That is the way you ought to look at data types in C. You could say that the guy wanted 1 dollar, but you gave him .25 + .25 + .25 + .25 = 1.00. This my friend is unacceptable.

Well, to fix this problem, you could go the bank, exchange your four quarters for an actual dollar, and then go pay the man. The bank in this situation has fixed your problem by converting your change to

whole dollars. In computer science terminology, you can say that **your number of the double data type was casted to a number of the integer data type**.

So, **type casting** is the process of converting one data type to another.

There are two main categories of type casting, implicit and explicit type casting.

Implicit type casting is when the conversion happens automatically and you do not have to worry about it.

Explicit type casting is when you have to say specifically that you want to convert some data from one data type to another. You can think of this as "explicitly" telling the computer what to do. It's not just assumed.

In general, you can think of the implicit type casting as "lossless". A good example of this is converting an integer to a double. If you convert 1 to 1.00, nothing is lost.

You can think of explicit type casting as "lossy". For example, if you convert the double value 2.54 to an integer, you truncate the number down to 2.

An example of implicit conversion is if you assign 1 to a double value. You gave the assignment operator an integer, but it gets stored as a double. There's really nothing you have to do.

The easiest way to explicitly type cast is to use an operator where you put the type you want the value to be in parenthesis before the data. For example, we could do something like (int) 2.5. This will convert the 2.5 to 2 (an integer).

In the next video we will be going over some examples of implicit and explicit conversions and some things to look out for that can cause problems.

Tutorial 39 - Implicit Type Promotion

The previous video was foundational to this video, so go watch that! Essentially, casting is when something of a certain data type is converted to a different data type.

This video is going to be showing examples of both implicit casting and explicit casting.

When you study implicit conversion in C, you will likely come across the term promotion.

This article talks about implicit conversion, and the section on promotion talks about what data types can be converted to other data types without loss of information.
http://en.cppreference.com/w/cpp/language/implicit_conversion

A common example of this is when we pass floats to functions.

```
float x = 50.0f;  // You can put the f here if you want to be specific that this is a float, but either way should work just the same).

printf("%f", x);
```

The printf function actually takes the x as a double. This means that 50.0f is **promoted** to 50.0. If we are talking about constant values, a float value always has an f. If you leave the f off, it is assumed to be of type double. You can use the %f conversion character for both floats and for doubles because they are actually always being printed as doubles… Even if you pass in a float.

Now, it's important to understand that the actual value or data type does not change. The variable x is still of type float.

Another common type of implicit conversion is when we are doing assignment. If we have something such as this:

```
double age = 60;
```

The integer 60 is implicitly casted to a double. That's because an integer can fit inside of a double without losing data.

Sometimes, it's not that easy and we risk losing data. That's what we are going to be talking about in the next video.

Tutorial 40 - How to use the Type Cast Operator

Now, let's give an example of explicit conversion:

```
int slices = 17

double halfThePizza =  slices / 2;
```

In this situation, we are splitting the pieces of pizza with our friend. We are trying to figure out how many pieces each person should get.

Both operands are integers, so the result will be an integer. There are two ways we can fix this. First, we can add .0 to the 2. This works, but it will not always work. For example, we could have this:

```
int slices = 17;

int people = 2;

double slicesPerPerson = slices / people;
```

Print this out and see that the result of the double is actually 8, not 8.5.

Let's assume that the second operand is given to us as an integer and we cannot change it for some reason. The only solution is to use an explicit cast:

```
double slicesPerPerson = (double) slices / people
```

This explicitly casts the value of slices to be a double.

Now, when you output the data, it should be 8.5.

The important thing to know here is that the cast does not affect the original value of the variable. The casting is actually just another operator. Think of it as a unary operator that does not change the value of the variable. The value of the variable is first given, and then that value is changed to a double, but the original variable stays the same. Once you define a variable as a certain type, you're stuck with it.

It's important to think of the casting as a unary operator. That means that it only works on one operand. Which operand? Is it this:

```
double slicesPerPerson = (double) (slices / people);
```

or is it:

```
double slicesPerPerson (double) (slices) / peoples;
```

It's actually the second one. It only affects the thing to the direct right.

You can cast an entire expression using parenthesis, so the first one is completely valid. The only problem is that the result will still be 8. That's because we divide two ints by one another and get 8, and then we cast 8 to a double, which stays 8. Because of this potential area of confusion, I recommend you try to avoid casting a large expression that is in parenthesis unless you know exactly how it is going to work.

TRIVIA TIME!!!

What will this output?

```
double c = 25 / 2 * 2;
double d = 25 / 2 * 2.0;

printf("c: %f\n", c);
printf("d: %f\n", d);
```

This program illustrates that even if one of the operands is floating point, the entire expression does not use floating point arithmetic. In the case with the variable d, the 25/2 evaluates to 14, then * 2 evaluates to 24.0. Both printf()'s output the same thing.

Remember that the decimal is always truncated, never rounded. So even if some division ended with .99999, it would just delete the .99999 and not round up.

Tutorial 41 - Quiz Part 1

Up next we will be covering logic, conditionals, loops, and how to make more complex programs. But before this, I think it is important to make sure we have the previous videos down well. These next few videos will quiz you on some parts of C. If you have a hard time answering these question, you might want to review your notes.

Questions:

1. So we have this: 5 + 5. What is the entire thing called?
2. What is the + sign an example of?

3. What are the two fives examples of?
4. What are the three categories of operators?
5. What is a language called when it does not care if you add extra spaces between commands?
6. If we have this: int x;What is this called?
7. What about x = 5; ?
8. What is the word used to describe the format of how you type commands, or the rules on how to type C?
9. What are printf() and scanf() examples of?
10. When we pass in a value to a function, that value is called a(n)?

Answers:

1. An expression
2. An operator
3. They are constants, but also, they are operands
4. We group them into groups depending upon how many operands they have. Unary, binary, ternary
5. White case insensitive
6. Variable declaration
7. Variable initialization
8. Syntax
9. Functions
10. Argument

Tutorial 42 - Quiz Part 2

Just as a reminder, up next we will be covering logic, conditionals, loops, and how to make more complex programs. But before this, I think it is important to make sure we have the previous videos down well. These next video and previous video will quiz you on some parts of C. If you have a hard time answering these question, you might want to review your notes.

Questions:

1. What is casting?
2. When you cast a double to an integer, is the number rounded or truncated?
3. What is the last character that you have to put in a character array to make a string?
4. What are the two ways to make comments in C and what is the difference?
5. What are primitive data types vs complex data types?

6. What is format character used for integers?
7. What is the format character used to print floating point numbers?
8. What do you need to prefix the format characters with when you are using scanf?
9. When using scanf, what is special about the way you pass in the variable?
10. What is the ASCII code for capital A?
11. What is the difference between 'A', and "A"?

Answers:

1. Casting is changing the data type of an expression
2. Truncated
3. We only briefly talked about this one so if you're not sure, that's okay
4. /* */ and //. The first is a multiline comment
5. Primitive data types are the foundational data types, such as integers. You can combine primitive data types to make more complex data types
6. Either %d or %i
7. %f, %e, and %e
8. You have to use a special operator before your name. &, the address-of operator
9. 65
10. They're both constant values, but the first is a character while the second is a single character string

Tutorial 43 - Quiz Part 3

Just as a reminder, up next we will be covering logic, conditionals, loops, and how to make more complex programs. But before this, I think it is important to make sure we have the previous videos down well. The previous videos will quiz you on some parts of C. If you have a hard time answering these question, you might want to review your notes.

Questions:

1. What is the % operator called and what does it do?
2. What is the difference between i++ and ++i?
3. What does j += 5 mean?
4. What is the word used to describe which operators happen first?
5. What is the word used to describe if you have multiple of the same operators whether or not they are evaluated from left to right or right to left?

6. What is the associativity of the division operator?
7. What is the associativity of the assignment operator?
8. When it comes to C and data types, C is said to be a _____ typed language?
9. What do you have to do to use printf() and scanf() inside of your code?
10. What do you have to include in order to use the bool data type?

Answers:

1. Modulus. Gets the remainder of division.
2. i++ increments after use while ++i increments before use.
3. Increase the variable by 5.
4. Precedence.
5. Associativity.
6. Left to right.
7. Right to left.
8. strongly.
9. #include <stdio.h>
10. stdbool.h

Tutorial 44 - Coding Challenge

You know a lot of the concepts, but now you need to practice writing some code. I am going to give you a small coding assignment. You can give it a try and then compare your answer to mine.

The assignment: You have a right triangle, and you are given the length of the smaller sides. Write a program that asks the user for two lengths of a right triangle, and outputs the length of the longest side (the hypotenuse).

To do this, you use the Pythagorean theorem. a^2 + b^2 = c^2

You will also need to know that for this assignment you will need to use the sqrt() function which requires you to include math.h. Another thing, when you are using some math functions, you have to do something special when you compile.

Normally, you compile like this: gcc py.c. Additionally, you can add -Wall, which will show warnings when you compile. Well, you have add -lm. This has to do with what is known as **linking**. We are linking

the C math library to our program when we compile. If you can't figure that part out, that's okay. I'll show you, but still try to write as much code as possible.

I'm going to let you research the details on how to use sqrt() and whatever else so I don't ruin all of the fun.

The solution:

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

int main()
{
        double a;
        double b;
    printf("This app will calculate the hipotinuse of a right triangle.\n");
        printf("Enter the first value: ");
    scanf("%lf", &a);
    printf("Now Enter the second value: ");
    scanf("%lf", &b);

        double c = sqrt((a * a) + (b * b));

    printf("The hypoteneose is %f\n", c);
        return 0;
}
```

To compile:

```
gcc - py.c -Wall –lm
```

Tutorial 45 - Intro to Logic

This video is going to be an introduction to logic. Now, the term logic can be used for a lot of different things, but in computer science it is specifically talking about things being true or false.

When we claim something, we are making what is known as a statement. A **statement** is a claim that is either true or false, but not both. This is a little different than statement in computer science which tells the computer to do something. The statement in logic is often called an expression in computer science.

So let's say we write a program, and we ask for the user's age. Let's say that the user's age is 72. Well, we can make a statement about this age. This user's age is above 21. This is an example of a logical statement. When we convert this English into something the computer can understand, we would write age > 21. Now, we have an expression. Expressions evaluate to some value. In the context of logic, this expression evaluates to true or false. Age > 21 evaluates to true, but age < 50 evaluates to false.

We've discussed the **bool data type**, which can store either true or false. This is the same concept. A bool variable can either hold true or false, but not both.

The whole concept of statements being evaluated as true or false is the foundation to what is known as **boolean logic**.

Boolean logic is foundational to everything we are going to be discussing in the next videos. Boolean logic is actually very useful for many subjects including computer engineering and discrete math. In computer science, boolean logic is the foundation to understanding branching**. Branching** is when we make a program that can do two separate things depending on the truth value of a statement.

Let's say we are making some app where you have to be at least 13 years of age to use it. Well, with logic, we can write a program that will grant access if the person's age is 13 or more, and decline access if the person is younger.

We can also create what are known as **compound conditionals**. This is when we combine multiple statements with what are known as logical operators. One of the most common logical operators is the AND operator, which is symbolized with two ampersands (&&).

So if we were making an app that required you to be a female above the age of 13, we could check the truth value of two statements (age is greater than 12 and sex is female).

It is often helpful to visualize logic. The way we do this is with what is known as a **truth table**. With a truth table we combine all possibilities of input and see what the output will be.

This is a very simple example, but you can see that the output is always either true or false:

Age greater than 12	Sex is female	Age greater than 12 and sex is female
T	T	T
T	F	F
F	T	F
F	F	F

In the upcoming videos, we will be discussing how logic is used inside of programming to create control flow statements. A **control flow statement** is something that guides the path of our program. We started off writing programs that were very linear. They take an input, calculate something, output the data to console, and then end. Now, we are going to start writing programs that branch into different

paths, have cool loops, and programs that can actually ask the user if they want to restart, or do something again (depending on the situation) rather than just abruptly ending.

This is when programming gets fun! The benefit here is that I am going to try to teach the subject in a way that can be applied to other sciences such as math or engineering. This is one of my favorite subjects because it actually helps you think about life and teaches you to truly evaluate things. I think this is important for everyone to have some experience with as it will help protect them from being brain washed in real life.

What's next?

Many more chapters are on the way! Please check back soon! I promise they will get better… This is a work in progress!

Thanks for reading and watching!

Catch ya in the next video. :)