
Doing Stuff
With
C
Lee S. Barney



## Dedication

This book is dedicated to my wife and children. They make everything worth while. Also to those students who learn because learning is fun.

## Other

Information has been obtained by Author from sources believed to be reliable. However, because of the possibility of human or mechanical error by these sources, Author, the editor, or others, Author and the editor do not guarantee the accuracy, adequacy, or completeness of any information included in this work and are not responsible for any errors or omissions or the results obtained from the use of such information.

Question and Arrow Left icons used were created by snap2objects and used under license.
Pig image used was created by Martin Berube and used under license.
Sausage image created by Daily Overview, http://www.dailyoverview.com, and used with permission.
Green check icon was created by VisualPharm and used under license.
White Zombie image used with permission of Plasmaboy Racing.
Snow goose image used with permission of Steve Perry, http://www.backcountrygallery.com.

Snow goos flock image

# READ ME. NO, REALLY!

The book is not designed as an exhaustive API or software engineering book, a book of code script-lets to copy, nor a book full of detailed definitions of every possible word since this type of information is readily available on-line. Use the dictionary in your electronic book reader, Google®, and maybe even WikiPedia for definitions that are not explicitly given in the text for unfamiliar words.

The first chapter consists of a series if things you will need to know BEFORE you go on. Other chapters follow a consistent pattern.

**Chapter Content Description**

Each chapter consists of:

1. A brief description of the material to be covered and why knowledge of this information is important to you.
2. A series of points for consideration as you read and absorb the detailed information. These points are excellent items to discuss with co-workers or team members. Such discussions between peers yields greater depth of understanding and speeds learning.
3. A video that is a non-programming analogy of the information presented. These videos are not screen captures where you watch someone write, compile, and run something. They are designed to help you relate the new information to something that you probably already know.
4. A detailed and well organized description of the material to be learned. This material may have full color images and short videos interspersed within it. Each of these images and videos has been carefully created to clarify difficult or complex concepts.
5. A series of self-check questions to help you evaluate your understanding of the material to be presented.

CHAPTER 1

# Stuff You Need to Know

The sections of this chapter present information that you, as a reader, need to understand in order to be successful in learning the remainder of the information presented. Read them and use them as the road to more learning.

# Required Tools

## Get and Install These

1. A laptop or desktop computer
2. The latest version of a C compiler for your computer


*C and other programming tools can come from many sources*

Many tools used to create C applications are open source and free. You can edit and compile your applications by hand using a command line terminal and a text editor or you can use Xcode™ for Macs™ or Visual Studio™ for Windows™ PCs. The examples in this book use the command line terminal approach. This makes it possible to create examples that work on all platforms from Windows to RaspberryPi.

Regardless of what environment you chose to work in, you will need to download and install it. For this book we suggest Cygwin™ (http://sourceware.org/cygwin/) for Windows™ PCs. For Macs™ get the Xcode™ com-

mand line tools that are part of the Xcode™ download from the app store. Get gcc for linux PCs using your package manager.

To share with your teammates and to allow potential employers see what you know and can do, you will be using git and gitHub (http://www.github.com) as a content management system. If you don't already have a gitHub account go create one.

Learn git. It is a skill that employers of programmers currently want to see in potential employees. Oh... and by the way.... they want to see that you know how to use it from the command line. Don't use a GUI. Pro Git is a good resource book and is available for free in PDF, mobi, and ePub formats from the git website.

Example source code for this book can be retrieved from https://github.com/yenrab/doing-stuff-with-c.

The editor being used in the example videos is Sublime 2. You can choose to use any editor you like.

If it is impossible for you to install a compiler on you own machine, you can use an online compiler like compileOnline. This is not as good an option as having your own compiler on your own machine.

# Think First

### Steps to Success

1. Think - find out what you don't know and learn it by playing with it (sandboxing).
2. Design - use what you have just learned to layout a solution to the problem.
3. Test - create a series of expectations for behavior that will indicate if you have been successful.
4. Create - write the code!

*Think before you act*

Throughout history, a common approach has been used to create any type of object in the quickest and easiest way. Why would developing software be any different? It should not.

This historic approach is cyclic. This means that you take a stab creating your item and then go back and revise one or more decisions made using the approach and then continue on.

This historic approach is composed of four simple steps.

**Think** - What does the customer want? What types of knowledge are required to create the thing the customer wants? Do I already have this knowledge? If not, where can I find this knowledge? Playing with the new knowledge in a simple sandbox code set allows you to see how each portion of the new knowledge behaves and can be used. Example: If you don't know what a 2X4 wooden board is and how it behaves you should not attempt to build a house. Maybe you should play around with a few of boards instead of instantly starting to build. There is no replacement for experience.

**Design** - Decide how the thing should behave, look, and interact with other things. This is true of buildings, cars, toasters, baseball bats, and any other physical item you can think of, complex or simple. Planning is also vital to software development. How can you create an application if you have not yet decided what it is going to do, what it will look like, and how it will interact with other pieces of software or hardware?

**Test** - In this step you decide what standards your product will meet when it is done. For buildings this step would be the national and local building codes. In software this may consist of User Interaction testing, Unit testing, Component testing, System testing, Installation testing, or just figuring out, at a detailed level, what new data should come out of your application when it is given specific sets of data. You may be thinking, "How can I create a test for something that doesn't exist yet? I don't know how it should work." If this thought or one like it passes through your mind, then you have not sufficiently completed step 1, 2, or maybe both. Go back and do them first.

**Create** - Begin building. Notice that this step description starts with begin. Once you have started, it is likely that you will find weaknesses in your knowledge, design, tests, or all of these. When you do, go back and revise your results for steps 1 - 3 as needed. This does not mean 'throw away and start all over again'. It means make modifications and try again.

The standard approach taken by most software programmers goes like this:

1. A customer asks for something.
2. The programmer starts to create it.
3. It doesn't work.
4. The programmer throws it out.
5. The programmer repeat at step 2 until it works.
6. The company ships something the customer didn't want.

I call this the 'oh crap!' or 'think last' approach since nearly every time the developer hits step three 'oh crap!', or words like them, are heard. The 'think last' approach all but guarantees that the software ships late, is over budget, and doesn't have the features the customer wants. It is a proven failure method. Don't use it! Use the method that is known to work. **Think, Design, Test, Create**. Until you use it it may seem that it will take longer than the 'Oh crap!' method but it does not. Guaranteed.

# Helpful Resources

### Steps to Success

1. Install the tools found in the Required Tools section of this chapter.
2. Learn to launch your terminal (Cygwin terminal for Windows™ users).
3. Learn to use vim

*Books such as this one are not your only resource. This section has links to resources that you will find helpful as you begin programming in C.*

A quick video of some basic terminal commands

Another quick video of some basic terminal commands

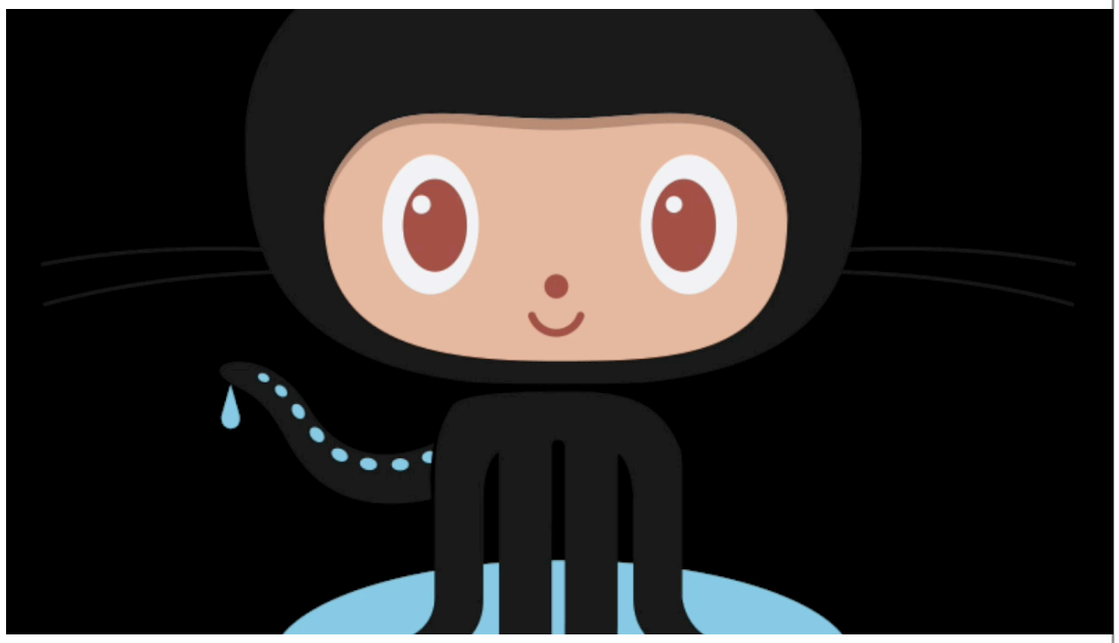
Turning on syntax highlighting and line numbering in vim

A using vim video

Another vim tutorial video

One API listing for C, there are many.

**Movie 1.1** Downloading the example files



*The examples for this book can be downloaded. This video is also available for viewing in your browser.*

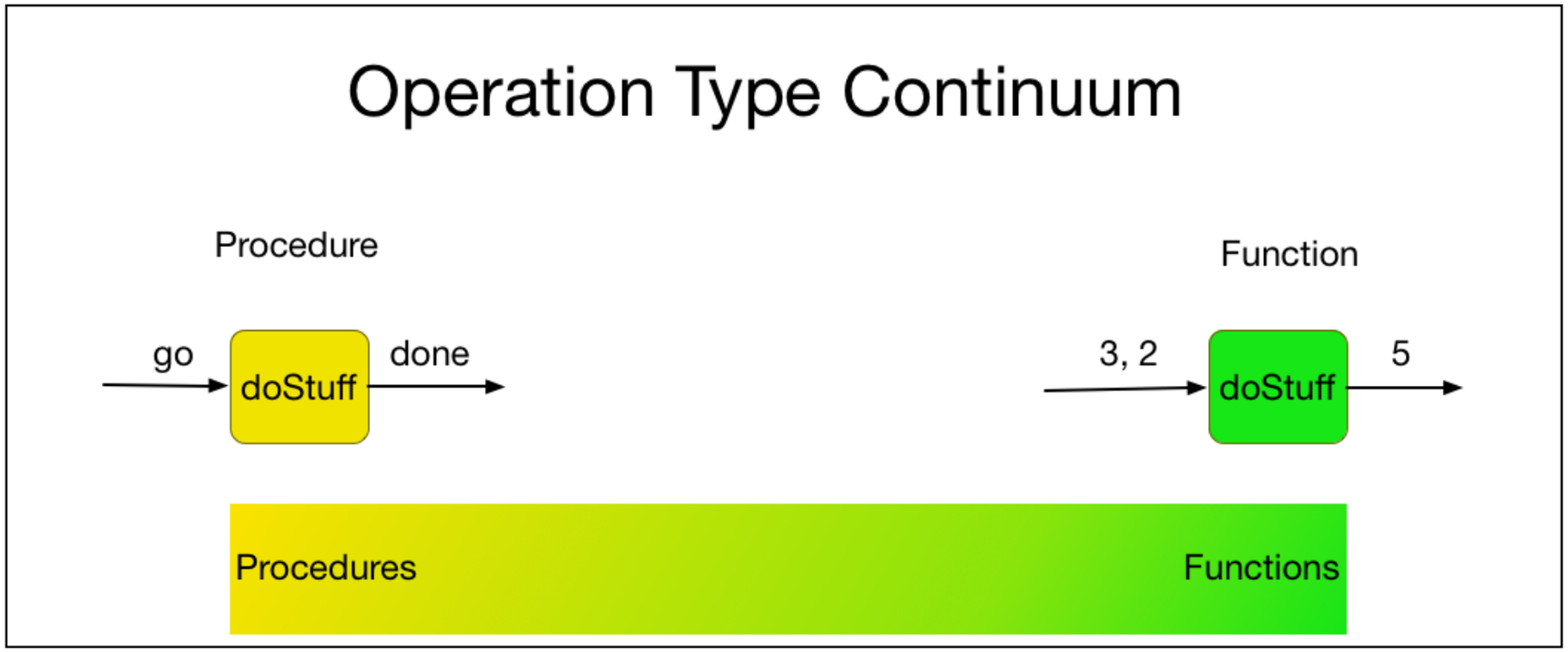
The types of variables that can be used in C.

The operators available in C.

# Operations, C, and Languages

**Lorem Ipsum**

1. What is an operation?
2. What types of operations are there?
3. How can common computer languages be categorized?




*Operations can be a pure procedure, a pure function, or somewhere in between.*

I've heard some people say they like procedural programming and they don't like Object-Oriented programming. I'm not sure what they mean. It seems they're saying they like 'step-by-step' programming.

All programming is 'step-by-step.' A program is a series of operations; things to be done. Each operation is executed one after the other regardless of the type of the language.

There are two basic types of operations, procedures and functions. Procedures are things to be done that require no data to begin with, do stuff, and let you know when they are done but don't give you back any computed results. A function

requires data to do its job, does stuff, and then gives you a result back when it's done.

A physical example of a procedure is the request to clean their room given to a teenager. There is no additional data needed. They clean their room. When they are done is obvious because they come out of their room. At least that is how it would be in an ideal world.

An example of a function would be an oven baking something. If you put in a bread dough mixture and a pan inside you get bread back out when the bake function is done. If you put cake dough and a pan in you get cake back. Either way, you get back some type of baked good.

With procedures and functions available, it is common to classify programming languages as procedural, using only procedures, or functional, using only functions, even though no programming language is purely procedural or functional. Some may be more procedural and others more functional.

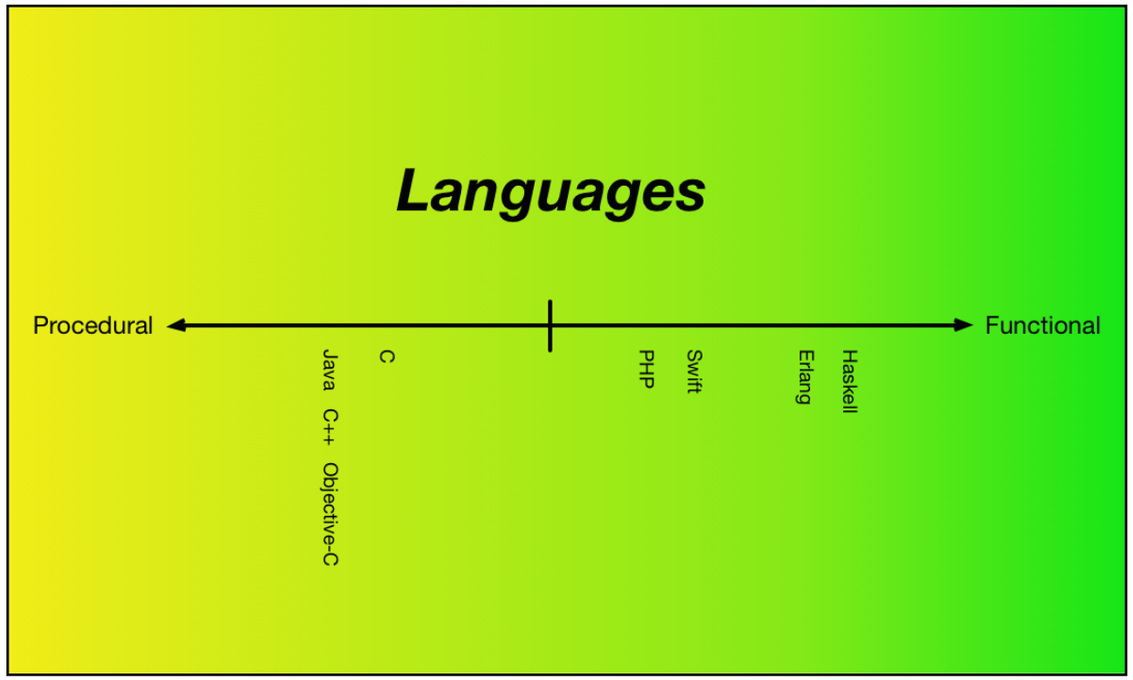



**Figure 1.** *A classification of some common programming languages.*

Based on the frequency of functions and procedures in the standard C libraries, C tends to be more procedural than functional. Using the same frequency scheme, Java and C++ are more procedural than C. Languages such as Haskell, Erlang, Swift, and even PHP tend to be more functional than procedural.

Regardless of which type a language is, and this can vary depending on who you ask, all languages combine the features and use of both procedures and functions.

CHAPTER 2

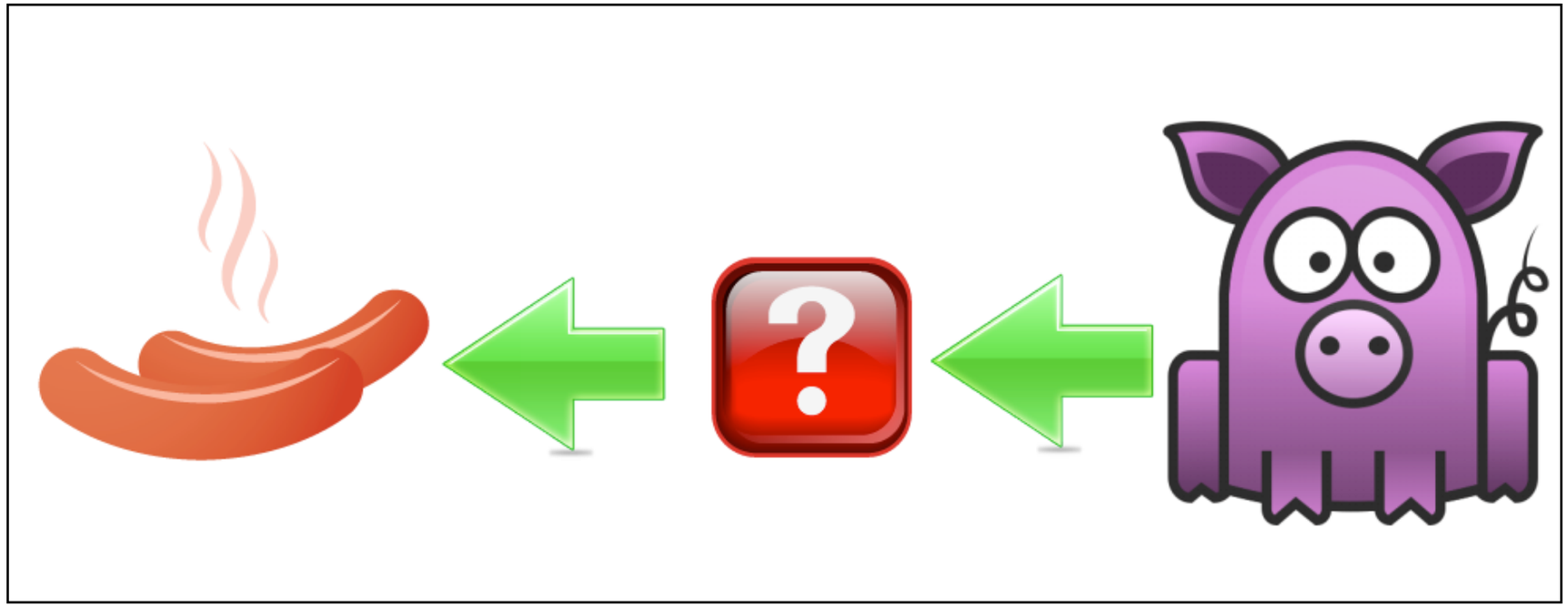
# Ready...Set...Go!


White Zombie is the worlds fastest, electric, street legal, drag racing car. It REALLY goes. Some of it is custom assembled but not all. The body, frame, tires, wheels, and other components were created by people other than Plasmaboy racing. Applications you write are built the same way. Some parts are pre-made for you and others you will assemble yourself from pre-made parts.

# Functions

**Points to Ponder**

1. What is a function and what is its purpose?
2. What do functions look like in the C language?



*Functions are one of the basic building block of all applications written using the C language. Stuff goes into a function and stuff comes out. A sausage factory is a physical manifestation of a makeSausages function.*

When you say you are writing an application in C what you are really saying is that you are creating a bunch of operations. You are telling the person you are speaking with that you are going to arrange operations so you can effectively communicate with the user of your application and help them do something.

Do NOT fall into the common error of thinking your application will talk to the computer. The computer is the medium of your communication with the user just as air is the medium when you use English or some other language to speak with someone.

Since functions are a type of operation and one of the foundations of everything you will do when

you use the C language we begin there. Answers to the two questions for this section will be provided and the details of functions will be examined. Hold on to your hat. At times you may feel like a pig in a sausage factory but I promise, you won't get ground up and spit out.

The grammar of the C language, usually called 'the syntax' by techies, declares that functions all have a similar structure, they have a name, stuff that goes into the function, stuff that comes out of the function, and stuff that makes up what the function does.

For a pork sausage factory function the structure would consist of

1. a name - makePorkSausages,

2. stuff that goes in - pigs,

3. stuff that comes out - pork sausages, and

4. stuff that makes up what the factory does - chopping, filling, and a bunch of stuff no one really wants to know about.

Take the time right now to start thinking of your everyday tasks. Each of them could be described as a function. An example could be taking a shower or bath. You, soap, shampoo, water, a tub or shower, and the grime you have built up on you over time goes in. What comes out? What do you do that makes up the process of taking a bath or shower?

If you start to examine what you do during any given day you will find functions everywhere. I strongly advise you start doing this NOW. In every spare moment you have look around you. See what people, animals, buildings, cars, tools, bugs, plants, clouds, etc. are doing. Truly look and think.

Give what you see going on a name. Figure out what, if anything, is going in, what, if anything, is coming out, and what may be happening as part of what is being done. Feel free to write down what you observe. It may help you later if you start becoming confused as we talks specifics. If you start examining your world now in every spare moment in this way, learning the C language will be MUCH easier.

Before reading this book any further go do this for at least two days. ***STOP HERE. PLEASE.*** Don't make your life harder than it has to be.

## Where To Begin?

Now that you've spent a couple of days looking for functions and their parts, let's start up again by taking a look at a very special function that is part of the C language. Its name is 'main'. In the C language, the main function exists for one purpose only. When you tell a computer to start an application it has to know where to begin executing the program. In the C language this is the purpose of the main function. If it wasn't there the computer wouldn't know where to begin.

Let's see what this special function looks like.

```
int main()
```

Like I said earlier, the C language's grammar, its syntax, states that a function must have 'stuff that comes out'. The `main` function, according to what you see above, has an integer (that is what `int` means) that comes out. I'll show you how to make stuff come out later.

The other strange things that you see are the parentheses, `()`, after the `main` function's name. These parentheses are used to hold the 'stuff that goes in'. This example of main has nothing going in so there is nothing between the parenthesis.

All functions in C follow this same grammar rule. All C functions look like this.

whatComesOut theFunctionName(whatGoesIn)

You can depend on this rule always being true. What you haven't seen yet is 'the stuff that makes the function do what it does'. We'll add that in now.

```
int main(){
        return 3 + 5;
}
```

### Details:

In the C language's grammar, the characters `{` and `}` are used to contain 'the stuff that makes the function do what it does.' They are used to help us humans see, understand, and remember what this function does compared to other functions that may be hanging around in the same location.

Great debates still swirl around the internet about exactly where is the 'best' place to put the `{` and `}` characters in spite of the C language being over 40 years old. Forty years is WAY too long for debating such an unimportant thing. On the internet and in books you will see examples with these characters in different places based ONLY on the author's opinion. All of the examples you will see from me will follow the pattern you see above.

Now that that's out of the way, lets examine each of the parts that make up what this version of the main function is going to do when it runs.

Remember when I told you I'd show you how to make stuff come out later? It's later. `return` is a special word in the C language. It means 'take what is coming next and make it be what comes out of my function'. In the example above, the integer `8` is what will come out of the `main` function.

There you have it. You now have seen all the basic parts of a function. All of the other functions you'll ever write using the C language will have these same parts and follow the same grammar. Go download this example and see it all together by getting it from the gitHub repository of examples.

**Movie 2.1** Compiling and Running the ***first_hello*** application

*Compiling what you have written in the C language is required in order for your computer create and then run your application.*

Now I'll give you a freebie. This video shows you how to compile this source code into an application and run it.

Enough of this using other people's special functions. Let's create one of our own. Let's do something that you have been doing since you were tiny. Let's multiply two numbers together by creating a function to do it for us.

```
/*
* Here is the brand new function 'multiply'.
*/

1 int multiply(int firstNum, int secondNum){
2  return firstNum * secondNum;
3 }
4
5 int main(){
6   int product = multiply(3, 4);
7   printf("3 * 4 = %d\n",product);
8   return 0;
9 }
```

*Code Sample - first_function*

Let's start by describing our new 'multiply' function. It has the all the parts of functions we talked about before on line number 1. What comes out (an int), the function name (multiply), and what goes in (two ints, firstNum and secondNum).


What's an int?

## Details:

The line number 2 says "After you multiply what ever firstNum and secondNum happen to be, send the result back out of the function." Using the *multiply* function we just created is much like using the first_hello example. Line 6 shows the *multiply* function being used. it reads "multiply three and four and keep what is sent back as an int called *product*.

Line 7 uses the standard library printf function. It is ugly. What it ends up printing out is “3 * 4 = 12” but “what the heck does all that %d\n stuff do?” In order for printf to print out the integer held in product we have to tell it what what kind of thing product is. That is the purpose of *%d*. The d stands for decimal. Yep, your right. Integers don’t have decimal points in them so why isn’t it %i for integer? Queue the old guys with the beards again. They chose decimal for its other meaning. I can almost hear them again. “Decimal numbers are base 10 numbers. Let’s use %d that way we could also do cool stuff like hexidecimal numbers!”

Oohh.. Ahh... not really all that cool.

The *\n* part stands for “make the next thing printed out in the terminal show up on its own line. If you leave this out everything will come out on the same line and your printout will look funky. Make sure you don’t forget it.

So there is our first created function. It doesn’t do anything important but I hope you get the idea. Don’t forget what functions are or how they work. We need them all the time when working in C.

“But wait a minute!” you say. If printf is a function it is supposed to give us something back! Be calm. It does. I gives us back how many characters is printed out. In this case it gives us back the number 10. We are choosing to ignore it so we don’t assign it to an int. We just let it go away.

Now, not leaving functions behind let’s move on to something else, procedures.

# Remembering User's Info

**Points to Ponder**

1. What is a procedure and what is its purpose?
2. What does a procedure look like in C?
3. How can I store data so I don't loose it when I stop the application?
4. How do I get data back into my application once it has been stored?
5. What can I do to allow the user to communicate?


*Information is vital to our lives. Even small choices like where to go on a hike require information. Be ready for a workout if you go here.*

When you write an application you are communicating with the user. For there to be true communication it must be bidirectional. The user needs some way to speak to you.

Another point to remember is computers are stupid. They can sometimes seem smart because they are so fast at being stupid. They can make all kinds of stupid mistakes, evaluate and reject them, and then finally make a good choice.

Because of their stupidity, computers can't remember anything. When you stop an application

anything it has been working on is gone...unless the application is designed to store what it has done.

In computer terminology the information stored or shared with the user is called output. Anything coming in from where it was stored or from the user is called input. Together these are called IO, Input-Output.

C has a whole bunch of procedures that look similar to the printf function you've already seen. These procedures have been created for you by others. They make it easier to do Input-Output and are stored in a library called stdio (standard input-output).

One way to generate output is using the printf function you saw in the last chapter (**HINT: If you haven't read chapters 1 and 2, GO BACK!!!**). Being the highly intelligent person that you are, you have assumed if there is a way to generate output there must be a way to get input; and you would be right. When you want to get knowledge out of your head and into a book for later use by yourself and others you get the information printed. When you want to get information back out of a book and into your brain you read it.

So being intelligent you would assume that the input partner to printf would be readf. But the white haired, long bearded guys that created C's IO library 40 years ago weren't as bright as you. I can almost hear them. "Nobody reads books anymore and they certainly won't in the future! At best we just scan 'em and look for interesting things."

So they decided to call printf's partner scanf. I know...I know. Stupid decision but we are stuck with it.

Not thinking that choosing a bad name was enough, they decided to make scanf confusing to use. They did manage to make it read in only one thing at a time, thank heavens.

Here is an example that reads in two ints that are be passed to the multiply function you created in the last chapter (See...I warned you to go back and read that one. )

```
1  int main(){
2    int firstInt;
3    int secondInt;
4    printf("enter an integer: ");
5    scanf("%d",&firstInt);
6    printf("enter another integer: ");
7    scanf("%d",&secondInt);
8    int product = multiply(firstInt, secondInt);
9    printf("%d * %d = %d\n",firstInt,secondInt,product);
10    return 0;
11 }
```

*Code Sample - first_from_user*

## Details:

OK, OK... don't panic. That's the most important thing right now. DON'T PANIC. I'll step you through this. Lines 2 and 3 look familiar from the last chapter. They create two integers named *firstInt* and *secondInt* but don't actually give them a number, **YET**. The code will give them one later. Lines 4 and 6 are familiar too. They print out instructions for the person running the app. Lines 5 and 7

are the nasty ones. The scanf procedure works kinda like the printf function. We have to tell it the type of thing it should read in. That is the purpose of the %d. It says the app should expect to read in an integer just like we used %d to print out integers in the previous example.

Since we're reading something in, C makes us keep it somewhere. That's why we didn't give firstInt and secondInt number values on lines 2 and 3. We do that on lines 5 and 7. The example code has something strange in front of firstInt and secondInt in lines 5 and 7 -- the & character. Like all things C that character has a specific meaning. Line 5 says "Read in an integer and give it to firstInt to hold." This means that the & allows scanf to hand off what it reads in to firstInt in line 5 and do this again with secondInt in line 7.

Why not just have scanf return the value read in? Well...that would make scanf a function. scanf was designed to be a procedure. Procedures don't return anything. That's why we had to tell scanf where to store the int it read in. If you're shaky with the idea of procedures take a look at Section 4 of Chapter 1 again.

Line 9 uses multiple %d's to print out the two integers the user typed in, held by firstInt and secondInt, and the calculated result held by product.

This is all fine and dandy, but what about storing the users info? That's what this chapter's about right?

In C information is stored using what are called files. 40 years ago keeping paper files in file cabinets was very common. The writers of C decided to use an analogy to those files and file cabinets that everybody used to help them relate to what the computer was doing. The analogy made sense 40 years ago but not so much today. Most of us have never had a paper file that we put it in a filing cabinet. Today we might choose a different analogy for information storage. Documents are what we tend to work with today. If you can remember that a file in C means a document we are good to go.

In C, each file is somewhere on the computer. It would be nice if someone could point out to us where that file is. In C it can happen.

**Movie 2.2** Pointer Video Place Holder

*Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do tempor incididunt ut labore et dolore magna aliqua.*

Something that points to a file is called a file pointer. Thank you old guys with beards. You got this one right.  In C this is written as FILE*. Remember, a FILE* points to a file. It keeps track of where it is in the computer.

Let's write our very own procedure. Let's make it save the information we got from the user. Yep...its going to use a file pointer.

```
4 void saveNumbers(int aNumber, int anotherNumber){
5    FILE* userNumbersFile = fopen("user.data","w");
6    fprintf(userNumbersFile, "%d %d", aNumber, anotherNumber);
7    fclose(userNumbersFile);
8 }
```

*Code Sample - write_to_file*

## Details:

Line **4** says, "Create a procedure named *saveNumbers*. It should expect to be passed two ints, aNumber and anotherNumber." It is creating a procedure since it returns nothing. That's what *void* means at the beginning of the line; return nothing.

Line **5** of the write_to_file example reads, "Open a file named user.data for writing and give me back something pointing at the file." Line **5** uses the standard C function fopen, *file open*. Remember...fopen is a function because it returns something.

fopen needs two things to work correctly. The first makes sense. It is the name of the file to open. The second is a little strange. the w must stand for something. In fact, it stands for 'write' if w is sent to fopen a file is created and is waiting to receive information for storage. Be careful! Using w won't add to an existing file. It will remove any information in the file and start it over from scratch. If you want to see how to add to what is in a file take a look here.

The *fprintf* function used on line **6** is **very** similar to the printf function. Instead of writing to the terminal as printf does it writes to a file (*fopen* means file open and *fprintf* means file printf). In order to know where to print the information *fprintf* needs the file pointer *userNumbersFile*. It also needs the information to be printed just like printf. Thankfully the creators of this function followed the exact same pattern they used when creating *printf*.

One more line to go and were through the worst of it. Since we opened a file we should be good people and close it. If we don't then all kinds of weird things will happen when someone else tries to use it. Remember, anything you open you must close. The concept of 'file open' yields *fopen* so, following the same pattern, the concept of 'file close' yeilds *fclose*.

fclose, like fopen needs to know which file to close. In some situations you may have opened a whole bunch of files so which one should it close? You send *fclose* a file pointer. That is the one it will close. In our example it is *userNumberFile*.

Now let's use our *saveNumbers* procedure and store some numbers. To keep things simple we'll reuse the code from the previous example where we read in two integers from the user.

```
10   int main(){
11     int firstInt;
12    int secondInt;
13    printf("enter an integer: ");
14    scanf("%d",&firstInt);
15    printf("enter another integer: ");
16    scanf("%d",&secondInt);
17    saveNumbers(firstInt, secondInt);
18    return 0;
19 }
```

*Code Sample - write_to_file*

## Details:

Line 17 contains the call to *saveNumbers* instead of the call to *printf* and multiply that were there before. That's it! We've saved information from the user!

OK. Now how do we get information back? That is the purpose of the *fscanf* function. It looks just like the *scanf* function we used to get the original information from the user except it needs a FILE* to know which file to read from. This is like the relationship between *printf* and *fprintf*. Good thinking old bearded guys.

Let's use *fscanf* and create another procedure to read the users numbers back in. Hm... We called our other procedure *saveNumbers*. Let's call this one *readNumbers*. That way the name will reflect what it does and mirror the way we named *saveNumbers*.

```
4 void readNumbers(int* aNumberPointer,
                                        int* anotherNumberPointer){
5    FILE* userNumbersFile = fopen("user.data","r");
6    fscanf(userNumbersFile, "%d %d", aNumber, anotherNumber);
7    fclose(userNumbersFile);
8 }
```

*Code Sample - read_from_file*

The *readNumbers* procedure looks a lot like code we've already created. if we replace the call to *fprintf* in *saveNumbers* with a call to *fscanf,* we have made most of the changes we need. We also need to change fopen a little bit. Instead of passing it "w" for write we'll pass it "r" for read. Remember, computers are stupid. We have to tell them little fiddly thing.

There is bit of strangeness in our procedure. What the heck is the * character doing in line 4? When we understand that we will have more truth about the & character we've been using. Line 4 says, "Create a procedure called *readNumbers*. It should expect to be passed two int pointers, aNumberPointer and anotherNumberPointer." Take a look at how we use the *readNumbers* procedure on line 13 below. It will help us understand what is going on.

```
10  int main(){
11     int firstInt;
12     int secondInt;
13     readNumbers(&firstInt, &secondInt);
14     printf("%d * %d = %d\n"
                     ,firstInt,secondInt,firstInt*secondInt);
15     printf("%d + %d = %d\n"
                     ,firstInt,secondInt,firstInt+secondInt);
16     printf("%d - %d = %d\n"
                    ,firstInt,secondInt,firstInt-secondInt);
17     return 0;
18  }
```

*Code Sample - read_from_file*

Since C requires exactly matching types in all conditions, *readNumbers* expects int pointers, and the **&** character is being used in front of ints where we should expect an int* as parameters there is only one logical conclusion that we can arrive at. The **&** character must give us an int* when we put it in front of an int. This is a true statement. You will probably need to read that through a couple of times. It's a little weird. The code &firstInt says, "Create an int pointer that points at firstInt." Again, take some time with this idea. It is tricky. It is also very important to understand. Often explaining the relationship between the **&** character and pointers to someone else can really help. As you try to explain it, misunderstandings in your own head will come up and you can straighten them out.

There we go. With these examples you should be able to remember, in other words read and write, information gathered from the user or calculated as part of an application. Use this in your apps.

Nobody wants to use an app that can't remember what they already told it.

# Making Choices

**Points to Ponder**

1. How do C applications choose between different behaviors depending on the situation?
2. How do I structure the choices so my application's behavior is predictable?

*If you go out on a limb you will eventually have to choose a branch to follow.*

Choosing between things is always difficult. Choosing one thing means not choosing the other options. Studies have shown that the more options humans have the worse they feel about the choice they made. Think of the last time you bought ketchup. So many options.

Computers don't have the problem of not being satisfied with the choice they make. But then again, they're stupid.

C has a structure that allows code to choose between any number of options. Being based on English, C uses english words to express choice making. In English we could phrase a choice be-

tween multiple things by saying 'If ....otherwise if....otherwise if....otherwise.' In C it is written 'if....else if....else if....else.'

If what? How is the 'what' portion described in C? We could check to see if a number is even or odd. To do this we need to use the Modulo operator, %. It returns the remainder of an integer division. 5%2 yields 1. 4%2 yields 0. So any number %2 is even if the result is 0.

```
4  void checkInts(){
5
6        int aNumber = 5;
7        int remainder = 5 % 2;
8        if(aNumber == 0){
9              printf("0 is neither even nor odd.\n");
10       }
11       else if (remainder == 0){
12             printf("%d is even.\n", aNumber);
13       }
14       else{
15             printf("%d is odd.\n", aNumber);
16       }
17 }
18
19 int main(){
20       checkInts();
21       return 0;
22 }
```

*Code Sample - even_choices.c*

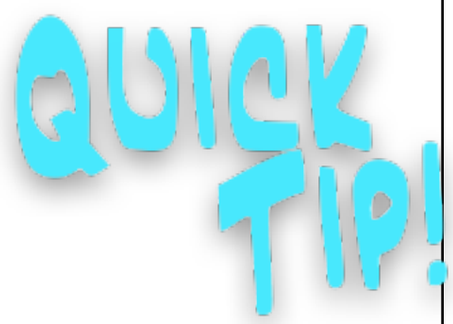
## Details:

The even_choices code sample checks a single number, 5 for being odd or even. Lines 8, 11, and 14 are the specific lines where this check occurs. Line 7 calculates the remainder. Line 8 checks to see if the full number *aNumber* is zero. This special check takes care of the number zero. It reads, "If aNumber is equal to zero." Just like with function declarations, what happens in this case is contained within the { } characters.

Line 11 reads, "else if the remainder is equal to zero." This is the second alternative. The third alternative is declared with the *else* keyword. Look carefully. The else keyword doesn't have any check associated with it! It will be where everything that doesn't match lines 8 and 11. In other words, if aNumber isn't zero and isn't even, then line 14 will kick in.

But what if I don't want to do a check for something being equal? That can be done the same way using one of C's comparison operators.

The fraction_choices example checks to see if any positive double is a fraction between -1 and 0 and between 0 and 1.




**What's a double?**

```
4  void checkDoubles(){
5
6    double aNumber = 0.25;
7    if (aNumber > 0 && aNumber < 1){
8        printf("%f is only fractional.\n", aNumber);
9    }
10   else if (aNumber < 0 && aNumber > -1){
11       printf("%f is only fractional.\n", aNumber);
12   }
13   else{
14       printf("%f has a whole number component.\n"
                        , aNumber);
15   }
16 }
17
18 int main(){
19   checkDoubles();
20   return 0;
21 }
```

*Code Sample - fraction_choices.c*

C doesn't have a between operator so you need to do two comparisons to check if something is between two numbers. Line **7** reads, "If aNumber is greater than zero and aNumber is less than 1." The && characters one of the logical operators available to you in C. It stands for 'and.' It is very commonly used so you should get to know it well. Line **10** does an additional check. This one is looking for purely fractional negative numbers. It reads, "else if aNumber is less than zero and aNumber is greater than negative 1." If aNumber doesn't match line **7** or line **10**, then line **13**, the default choice, will execute because there is a whole number part to *aNumber*. It is zero, less than or equal to -1, or greater than or equal to 1.

The fraction_choices example has a problem. Lines 8 and 11 are identical. In the world of programming this is called code duplication.

Code duplication causes problems. Not only is it inefficient to write, but it can wreak havoc when you start to fix bugs. Take my advice. Don't duplicate code.

But how can fraction_choices be done without code duplication? Put both checks in one if statement.

```
4  void checkDoubles(){
5
6    double aNumber = 0.25;
7    if ((aNumber > 0 && aNumber < 1)
                  || (aNumber < 0 && aNumber > -1)){{
8        printf("%f is only fractional.\n", aNumber);
9    }
10   else{
11       printf("%f has a whole number component.\n"
                        , aNumber);
12   }
13 }
14
15 int main(){
16   checkDoubles();
17   return 0;
18 }
```

*Code Sample - fraction_choices2.c*

## Details:

Line 7 has changed. It now reads, "if aNumber is greater than zero and aNumber is less than one **or** aNumber is less than zero and greater than -1." This is a compound check. The || characters rep-

resent the idea of **or**. The two aNumber checks have been put in parenthesis. This keeps our poor, stupid computer from getting confused. Without them it can get all mixed up. Better safe than sorry. No one likes to debug this kind of code. Write your code exactly how you mean it and when you come back to fix it later any thinking error you made should be easier to find. If you write your code vaguely, you won't remember what you meant to do and will waste a lot of time.

# Groups and Loops

**Lorem Ipsum**

1. What is an array?
2. How do you make an array?
3. How do you use an array?
4. What are C's two types of loops and when are they used?

*In C all elements of a group of things, called an array, must be of the same type just like this flock of geese.*

So what if we need to represent a bunch of numbers? Maybe we are averaging two years of monthly income from our awesome application. Is there a way to do that without creating a whole bunch of doubles? Yep. There is. You'd think someone thought through this before.

In C groups of things, like the income from your app by month, are called arrays. This is the only native grouping in C though you could write code to create other types like lists, dictionaries, and trees if you wanted to. We'll stick with arrays.

Arrays are both great and dangerous. Misuse of arrays in C causes one of the most common security issues targeted by hackers. We'll talk about that later, but remember...you MUST be very careful when using arrays.

OK. So lets get started.

```
6  double numbers[7] = {0.25, 11.003, -2.14, 0.14,
                                    20001.3, -.000001, -10.0};
```

*Code Snippet - group_loop.c*

Line 6 in the group_loop example shows how to make an array. It reads a little backwards. "Create an array of seven doubles called numbers and put 0.25, 11.003, -2.14, 0.14, 20001.3, -.00001, and -10.0 in the array." Wow. The left hand side of the = sign sure is mixed up. Why doesn't it look more like this?

```
   array 7 double numbers = {0.25, 11.003, -2.14, 0.14,
                                    20001.3, -.000001, -10.0};
```

*Bad Code*

C is designed to use few characters. The guys that designed C preferred what you see on line 6 of the group_loop code sample. When you write your own language you can decide how it will look. Until then you are stuck with other people's decisions.

So line six shows how to put an array together. Any time we put something in a group we eventually need to get something out of the group. When you put groups of dollars, euros, or what ever your currency is into a bank don't you eventually intend to take them back out?

How then can we get a value out of an array? Code snippet1 shows how.

```
double aNumber = numbers[1];
```

*Code Snippet1*

It reads, "the double aNumber is the same as the one'th value in the numbers array." Now you may think aNumber would now be 0.25. That is a very logical conclusion. It is actually 11.003. What?!

Notice I wrote "one'th" rather than "first." Hmm... tricky. Arrays in C represent their first position with a zero rather than a one. Code snippet2 shows aNumber becoming 0.25.

```
double aNumber = numbers[0];
```

*Code Snippet2*

This means that even though there are seven elements in the numbers array, assigning the last element of the numbers array, -10.0, to aNumber is done using a 6. Code snippet3 shows you how to do this.

```
double aNumber = numbers[6];
```

*Code Snippet3*

Weird...I know, but you get used to it if you work with C arrays long enough. With C arrays, the elements in the array are kept in the order they were added. That way we can always get back out the one we want. All we have to do is remember which one we want.

Now...lets combine arrays with the fraction_choices2 example from the Making Choices section. This means we need some way of checking more than one number. We could do it really poorly with a lot of code duplication, or we could do it with a loop.

In programming, a loop is something that happens over and over again. So if we could do the if check over and over again with a different number each time we could check each number in the numbers array. C has more than one type of loop. The type specifically designed to work with arrays is called a 'for loop.' It is called this because the keyword used to declare it is 'for.' Line **7** in code snippet group_loop.c is the *for* loop declaration that works with the numbers array in line **6**. Line **7** has the 5 distinct parts of all for loops.

```
6  double numbers[7] = {0.25, 11.003, -2.14, 0.14,
                                20001.3, -.000001, -10.0};
     1       2       3     4
7    for (int i = 0; i < 7; i++){
        .
        . 5
        .
15 }
```

*Code Snippet - group_loop.c*

The standard for loop parts when working with an array are:

1. the for keyword,

2. an int to keep track of the location we are in the array,

3. a check to see if the loop should continue,

4. an update that increases the location tracker by one, and

5. the code to execute every time we go around the loop, lines **8** though **15** in the group_loop example.

Line **7** reads, "create a for loop where i, the location tracking int, starts out as zero, the loop stops if i reaches seven, i goes up by one and the code in the {} characters is executed each time the loop goes around."

Line **7** is a standard structure in C. It is so standard that many code editors will generate most of it for you. By tradition, the location tracker is named 'i'. It stands for 'index.' No need to memorize this little bit of information. It's just some cultural trivia.

So let's put the group loop code all together now. We've looked at most of the parts before. Line 8 is the only new line.

```
4  void checkDoublesGroup(){
5
6    double numbers[7] = {0.25, 11.003, -2.14,
                          0.14, 20001.3, -.000001, -10.0};
7    for (int i = 0; i < 7; i++){
8       double aNumber = numbers[i];
9       if ((aNumber > 0 && aNumber < 1)
                          || (aNumber < 0 && aNumber > -1)){
10           printf("%f is only fractional.\n", aNumber);
11      }
12      else{
13           printf("%f has a whole number component.\n",
                                                  aNumber);
14      }
15   }
16 }
17
18 int main(){
19   checkDoublesGroup();
20   return 0;
21 }
```

*Code Sample - group_loop.c*

## Details:

Remember, i is used to keep track of where we are in the array as we go through the values. In programming speak, going through the values of an array is called 'iterating over the array.' You will hear that phrase a lot in programming books and tutorials.

Line 8 reads, "let the double aNumber be the ith element in the numbers array" where ith can be the zeroth, oneth, twoth, threeth, fourth, fifth, or sixth element. Notice again I didn't use the word first. The rest of the code executed by the loop, lines 9 through 14, come directly from the fraction_choices code sample from Section 3. If you don't understand those lines, look there.

The value of an element in a C array can be changed. For example, we could change the zeroth value of the numbers array to be 10.25. We reuse the [] characters to do this.

```
numbers[0] = 10.25
```

*Code Snippet4*

Snippet4 reads, "let the zeroth double of the numbers array be 10.25." Now we'll get 10.25 ever time we use the zeroth element.

Remember back at the beginning of this section when I said that arrays were dangerous? I'm going to show you how right now. Look at code sample group_bad.c

```
4  void crashIt(){
5
6    double numbers[7] = {0.25, 11.003, -2.14,
                                0.14, 20001.3, -.000001, -10.0};
7    numbers[7] = 45.9;
8  }
9
10 int main(){
11   crashIt();
12   return 0;
13 }
```

*Code Sample - group_bad.c*

## Details:

Line 7 is VERY bad. Even though numbers is seven elements long there is no seventh element. Remember C arrays start at zero so the sixth element is the last real element. In some other languages, like JavaScript, this wouldn't be a problem. JavaScript would add a new element to the end of the numbers array. We are not writing JavaScript. We are writing C and C doesn't have arrays that can change size. Once we said the array had seven elements it can never have more and can never have less.

The reason this code is so bad is your code will be writing a value into RAM outside of the array's location. It causes your application to crash. If this crash is caused by a hacker sending a bad image, text, or some other type of data to your app they can now insert any type of code into RAM they want and execute it. I won't show you how to do that. Just make sure you NEVER allow a loop or some other code to set or get values for arrays with indexes that are to large. I know that sounds easy to do. If it was, this type of hacking attack wouldn't be one of the most common attack vectors. Beware.

Some compilers may warn you about this bad situation. This warning will only happen in very limited conditions. Compilers aren't smart enough to catch this type of error in any sort of real code. You must always check your code yourself.

For loops are not the only kind of loop in C. There is also a 'while' loop. While loops are used in situations that are not linked directly to arrays. They are used in situations where an application experiences 'states'. In English there is a saying, "Work while the sun shines." This saying means that while the sun is shining I'll work and stop when night comes. That is exactly how a while loop in C works. Code snippet5 shows some code of how this English saying could be written.

```
    1          2

1 int sunIsShining = 1;


2 while(sunIsShining == 1){
3
4   ..work, work, work. 3
5   if(currentTime >= sundown){
6         sunIsShining = 0;
7   }
8 }
```

Line 2 shows how to declare a while loop. Unlike the for loop, while loops only have three parts, (1)the while keyword, (2)the check to see if the loop should continue, and (3)the code to execute each time we go around the loop.

One common way a while loop is used in C is to do user IO. In the Remembering User's Info section the user didn't get to enter more than one pair of ints. A calculator, like that one, which shuts down after the first calculation is not a good idea. Let's rethink that example. First, instead of multiplying lets divide two ints. Let's also create a procedure so we can get out of the main function as soon as possible. Let's use what we know about printf and scanf to do user IO. Also, computers can't divide by zero so let's use that restriction. If the user enters 0 as the number to divide by let's tell them 'Opps. There's a problem' and exit the app.

In this example I'm going to use a few math terms. Don't let them freak you out. Dividing 6 by 2 and getting 3 can be represented as $6 \div 2 = 3$. We can also think of it like a fraction and represent it as

6/3 = 2. When talking about the 6 in this example we call it the **dividend**, the number to be divided. The 3 is called the **divisor**, the number to divide by, and the 2 is the **quotient**, the answer. Don't freak. You know this from long ago when you first learned how to do division. You just don't quite remember it. That's OK. If you get confused just look back at the 6-3-2 example.

All right. Lets do some dividing!

```
4  void divideInts(){
5
6     int dividend;
7     int divisor;
8     printf("enter an integer: ");
9     scanf("%d",&dividend);
10    printf("What integer would you like to divide %d by?\n",
                                                      dividend);
11    scanf("%d",&divisor);
12    while(divisor != 0){
13        int quotient = dividend/divisor;
14        printf("%d / %d = %d\n", dividend, divisor, quotient);
15        printf("enter an integer: ");
16        scanf("%d",&dividend);
17        printf("What integer would you like to divide %d by?\n",
                                                      dividend);
18        scanf("%d",&divisor);
19    }
20    printf("Oops. There's a problem. You can't divide
                                                    by zero.\n");
21 }
22
23 int main(){
24   divideInts();
25   return 0;
26 }
```

## Details:

This example is written non-optimally. There is duplicate code. Lines **8** - **11** and **15** - **18**, our user IO code, are duplicates. While there are C ways to remove this duplication, they would have complicated the while loop example. Rather than expose you to those more advanced topics I chose to not confuse you the multiple new concepts you would need to know. I chose to keep it simple. There are huge numbers of examples of how to solve this specific duplication problem on the web so I'll leave this for you to explore.

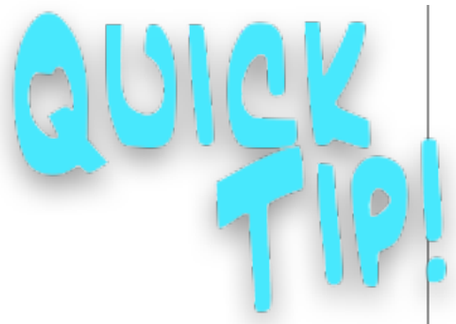
Line **12** contains the while loop and reads, "While the divisor is not zero execute the loop code." The loop code is on lines **13** to **21**. Notice that the divisor is entered by the user on line 11 so the while loop can check to see if it is zero or not.

Line **20** is executed ONLY if the user enters 0 at line **11** or line **18**. This happens because divisor is now 0 and the loop code only executes while the divisor is not 0. When the divisor is 0 the loop is 'terminated.'

So lets try some numbers. 20 / 5 = 4. Yep. That worked. -27 / 3 = -9. Yep. That worked. 12 / 5 = 2. What???? Shouldn't that be 2.4? Hmm.... Nope. It actually should be 2. Why you ask. Because we told the computer through our code to only deal with integers. Remember, computers are stupid. They aren't smart like you. C can't look at a situation and say, "Oh. There should be a fractional part to the result of this calculation. I'll add one on." Nope. It can't do it. If you want to have fractional parts to the answers you would need dividend, divisor, and quotient to be doubles. That way the computer will know you always want fractional parts.

But remember, computers are stupid. That means your user can't enter 6 and 3 to get 2. If you are using doubles they would enter 6

and 3 but would get 2.0. Try converting the code to use and display doubles.




**Printing Doubles**

# Representing Things

**Lorem Ipsum**

1. How can C represent things we see in the real world and complex ideas?
2. What is a struct?
3. How is a struct made and used?

*The world around us is made up of things. We make up groups of things all the time. This thing fits in the bird, goose, and snow goose groups and many others. It could also fit in a 'things with white coloration' group.*

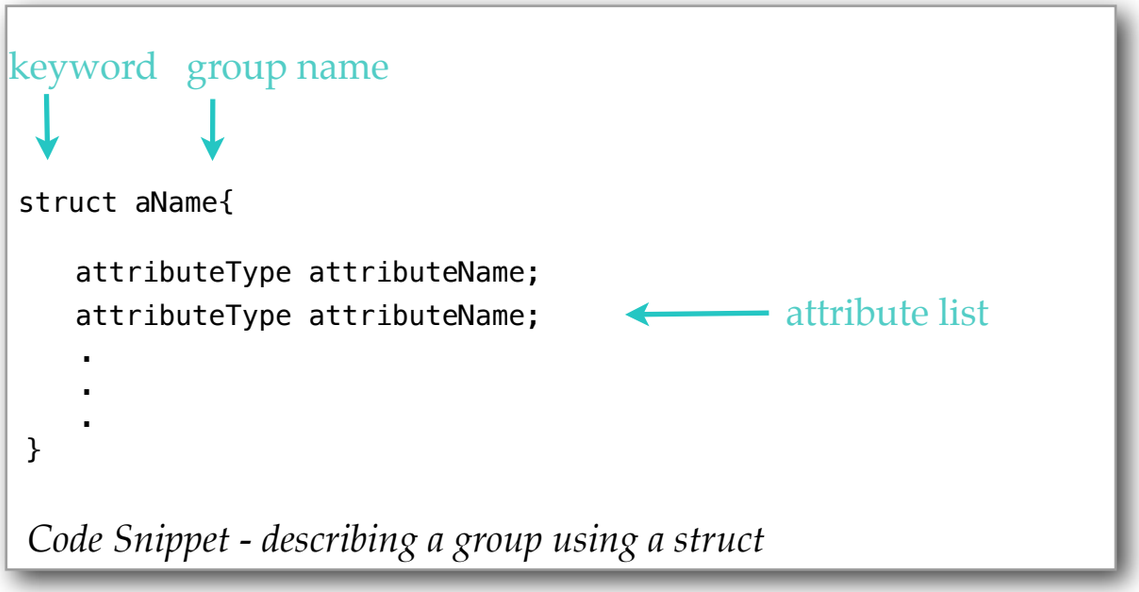
Arrays are nice, but how in the world can you represent something like a customer using arrays? A clothing customer could have a name, age, and inseam. That means for each customer would need to consist of a string, int, and double. Obviously we can't put these three types of things into one C array.

There are ways to pull this of if we write bad code, but the it would be hard to write, modify, and debug. So lets find a better way.

Because humans think in things there is a way to represent a thing in C. It is called a struct, short for structure.

Because structs represent actual things, actually the traits, or attributes, of the group to which things belong, we have to describe what the thing 'looks like.' We said that a clothing customer would need to have a name, age, and inseam. Why those? The only reason is because those pieces of information are all we need in our pretend situation. In other situations we may need hat size, sleeve length, or any number of other measurements. For our example let's keep it simple.

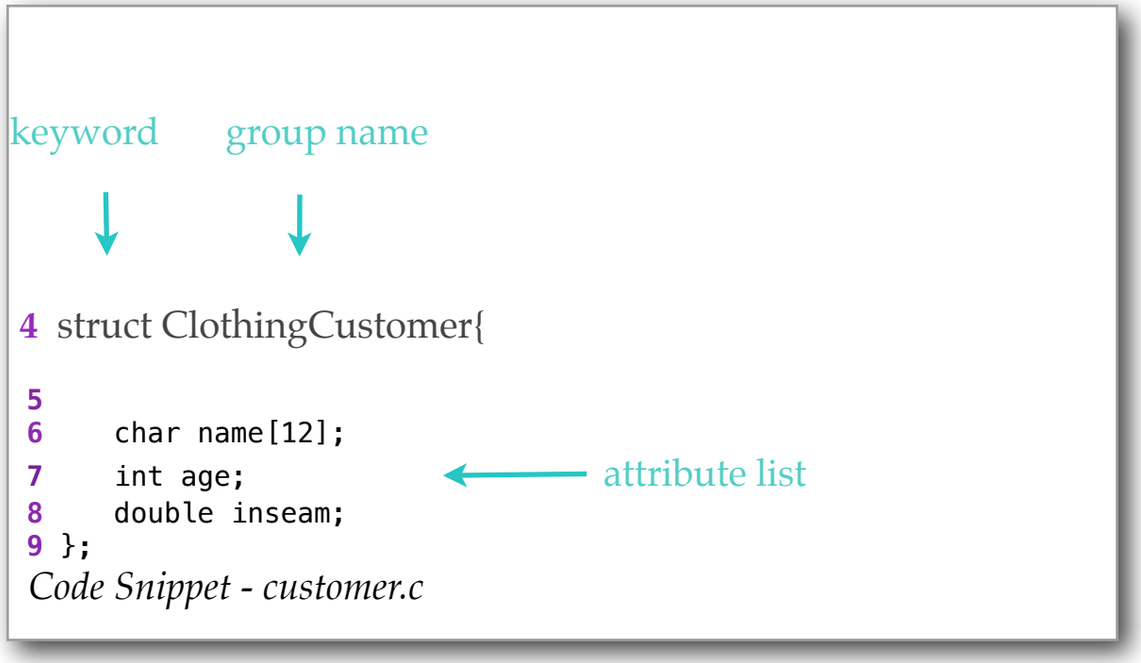
The pattern for describing something's group using a struct is always the same. The struct keyword is used to tell the computer to prepare for a group description.




*Code Snippet - describing a group using a struct*

This is always followed by the name of the group being described. Within the {} characters a list of types and names is used to describe each of the traits of the group.

So lets stop talking about patterns and how to do this in general and look at our example.




*Code Snippet - customer.c*

## Details:

Line 3 has the *struct* keyword and the name of the group of things we're describing, *ClothingCustomer.* Line 5 looks a little strange, so well come back to that in a second. Line 6 is an int attribute named *age* and line 7 is a double attribute named *inseam.* Those last two seem familiar. They are like the values we've stored and used in the other code examples but here they are used to describe the CloathingCustomer 'thing.'

Let's take a look at line 5. The type of this attribute is char. While we haven't used this type before you should have found it in the list of C data types in the Helpful Resources section. char stands for character.

Line 5 is an attribute called *name* that is an array of 12 characters. This is how C handles words, what other languages and programming in general calls strings. An array of char's can represent any-

thing from an acronym to the text of an entire book. It's an array of characters. Good thing we've already learned about arrays in the Groups and Loops section :)

So there is our description. All ClothingCustomers will have three attributes, traits; a name consisting of an array of 12 characters, an age represented by an int, and an inseam length as a double.

A CloathingCustomer is created in C by first using the struct keyword to tell the computer to expect a type of struct. I know. I know. Didn't we use the struct keyword to begin the description? Yep. We did. But C has us reuse it. It is used to start a descriptions and when we want to create new structs. The *creating a ClothingCustomer* code snippet shows how to reuse struct and the rest of the pattern to follow; First the struct keyword, then the group name, *ClothingCustomer*, then some sort of identifier for customer being created, *cheapCustomer.* The first line of the *creating a ClothingCustomer* code snippet reads, "Create a struct of group type ClothingCustomer and identify it as cheapCustomer."

```
struct ClothingCustomer cheapCustomer;
cheapCustomer.name = "bob";
cheapCustomer.age = 45;
cheapCustomer.inseam = 35;
```

*Code Snippet - creating a ClothingCustomer*

After a customer is created its attributes can be set. This is done by listing the struct identifier, *cheapCustomer*, a . character, and then the identifier for the attribute. The second line of the code snippet reads, "Let cheapCustomer's name be 'bob'." The age is set on the next line. It reads, "Let cheapCustomer's age be 45." Get it? Let's do the last line just to make sure. It reads, "Let cheapCustomer's inseam be 35."

There we go. We've created a customer and set the customer's attributes. Great. Let's see how to do this as part of an app. Keeping it simple, the app will create a ClothingCustomer and scan to get the values for the name, age, and inseam from the user. Then it will print out a description of the ClothingCustomer. Just like in other examples this will be done in a procedure to get us out of the main method.

```
4  struct ClothingCustomer{
5
6    char name[12];
7    int age;
8    double inseam;
9  }
10
11 void createAndPrintCustomer(){
12   struct ClothingCustomer aCustomer;
13   int quotient = dividend/divisor;
14
15   printf("Enter customer name: ");
16   scanf("%s",aCustomer.name);
17
18   printf("Age: ");
19   scanf("%d",&aCustomer.age);
20
21   printf("Inseam: ");
22   scanf("%lf",&aCustomer.inseam);
23
24   printf("%s is %d years old and needs pants with an inseam
                              of %lf\n", aCustomer.name,
                              aCustomer.age, aCustomer.inseam);
25 }
26
27 int main(){
28
29   createAndPrintCustomer();
30   return 0;
31 }
```

*Code Sample - customer.c*

## Details:

Lines 4 - 9 we've seen before. This is where we describe the ClothingCustomer group. Line 12 creates a structure of the group ClothingCustomer and identifies it as aCustomer. Lines 15 - 22 is where we scan to get the values for aCustomer's attributes. Let's start by looking at line 19.

When we've scanned in other examples we passed the scan function a type of thing to read in, and a pointer to a thing of that type. Line 19 looks a little different but is doing the same thing. Instead of a pointer to a regular it being passed in line 19 passes in a pointer to aCustomer's age. It reads, "Scan in an int and assign it to aCustomer's age."

Line 22 is very similar. It scans in a double, using the %lf indicator, to assign to aCustomer's inseam. It reads, "Scan in a double and assign it to aCustomer's inseam."

Good. So those aren't so much different than scanning in values to regular ints and doubles. Line 16 is different. the & character that allows us t pass a pointer to the scan function is missing! That's because we need a string from the user, strings are arrays of characters, and arrays in C have some strange behaviors.

In C, the indicator of an array, in this case *name* is the pointer to the array. Why? Just because. So when we want to scan in an array we don't have to put the & character there. This saves us from typing so much and that's good, right? Hmm.... maybe. It can be confusing because it breaks the pattern when scanf is being used. It does come in handy in other situations.

If C is designed well, then these same kinds of changes, using aCustomer.age instead of a regular int, would be used in printf, right? It is! Line 24 prints out aCustomer's description.

But entering data and printing it out is useless if the data can't be stored. How can structs be printed to and scanned from files? Section 2 is about doing this with regular data types like ints. Now we'll figure out how to do it with structs.

You can't just pass a struct to the fprintf function. It has to be broken apart just like we did for the printf function.

In the write_customer example five customers are created and written out. A function called createACustomer gets information from the user and creates a CloathingCustomer struct. Each CloathingCustomer is then written to a file on disk.

```
4  struct ClothingCustomer{
5
6    char name[12];
7    int age;
8    double inseam;
9  }
10
11 struct ClothingCustomer createACustomer(){
12   struct ClothingCustomer aCustomer;
13   int quotient = dividend/divisor;
14
15   printf("Enter customer name: ");
16   scanf("%s",aCustomer.name);
17
18   printf("Age: ");
19   scanf("%d",&aCustomer.age);
20
21   printf("Inseam: ");
22   scanf("%lf",&aCustomer.inseam);
23   return aCustomer;
24 }
25
26 int main(){
27
28  FILE* customersFile = fopen("customers.data","w");
29  for (int i = 0; i < 5; i++){
30     struct ClothingCustomer aCustomer = createACustomer();
31     fprintf(customersFile, "%s %d %lf\n", aCustomer.name,
                              aCustomer.age, aCustomer.inseam);
32  }
33  fclose(customersFile);
34  return 0;
35 }
```

*Code Sample - write_customer.c*

## Details:

Much of what is used in this example you've seen before in the examples described in the previous secstions.. If you get confused go back and look at those. It will help.

Line 11 is the createACustomer function declaration. It reads, "Return a struct of the ClothingCustomer group from the function createACustomer." This function looks almost the same as the createAndPrintCustomer procedure in the customer.c example. The difference is that this function returns a ClothingCustomer that has all of its attributes filled in with values.

Line 28 looks almost the same as line 5 in the write_to_file.c example in Section 3. The major difference is the name of file is being opened. Line 29 begins a for loop to generate and write 5 ClothingCustomers to the customers.data file.

Line 30 shows the call to the createACustomer function. It reads, "Call *createACustomer* and keep what is sent back in a structure of the *ClothingCustomer* type named *aCustomer*."

Line 31 uses fprintf, see Section 2, much like printf was used in the customer.c example. The differences are the inclusion of the FILE pointer to the output goes to the customers.data file and the removal of the descriptive text a user would need to make sense of the data. We don't need that since the user won't directly read the customers.data file anyway.

Reading the data back in also uses the concepts described in the previous sections so the read_customer.c example should also look familiar. It consists of a procedure called readAndPrintCustomers. five customers were written out, so this example expects five customers to be read back in.

```
4  struct ClothingCustomer{
5
6    char name[12];
7    int age;
8    double inseam;
9  }
10
11 void readAndPrintCustomers(){
12
13   FILE* customersFile = fopen("customers.data","r");
14   for (int i = 0; i < 5; i++){
15      struct ClothingCustomer aCustomer;
16      fscanf(customersFile, "%s %d %lf", aCustomer.name,
                          &aCustomer.age, &aCustomer.inseam);

17      printf("%s is %d years old and needs pants with an
                                  inseam of %lf\n",aCustomer.name,
                                  aCustomer.age,aCustomer.inseam);
18   }
19   fclose(customersFile);
20 }
21
22 int main(){
23   readAndPrintCustomers();
24   return 0;
25 }
```

*Code Sample - read_customer.c*

## Details:

Line 14 starts the loop to read in the five customers. Line 16 uses *fscanf* like it was used in Section 2. When the customers were written out to the customers.data file, each customer's information was placed on a single line. This means that there are five lines of data in the file, each representing a single customer. Line 16 uses *fscanf* to read a line at a time and assign the values read to the attributes of *aCustomer.* Line 19 closes the file.

# Where Next?


*A good start is only a start. If you don't keep going did you really start or are you right where you began?*

There are still many things about C you still don't know. A few of these are listed here with links to help you continue your discovery process. Also, completing a quality C tutorial or additional classes would be a good idea.

C Arrays are pointers. There is so much more you can do when you understand this.

Applications with multiple code files. No application of any real size is written in a single file.

Variadic functions. What they are and why we need them.

Last but not least, you REALLY should browse through any good C API to see what else is possible. Talk with other interested peo-

ple about what you see there and play with code using what you see. Nothing helps you become fluent faster than playing.

Have fun!