Introduction to C Programming

Lesson 16: **Reading/Writing Files**

- Opening a File
- fopen Modes
- fprintf
- fscanf
- EOF
- fgets
- What is Recursion?
- Be Careful and Why?

Getting Started

Welcome to the O'Reilly School of Technology's C Programming Course!

Course Objectives

When you complete this course, you will be able to:

- analyze and compile C programs.
- declare variables and data types.
- perform mathematical functions in C.
- input and output data.
- create loops, conditional statements, arrays, and functions using C syntax.
- demonstrate understanding of pointers.
- declare and manipulate strings.
- create structures in C.
- read and write files recursively.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.
- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

About CodeRunner

Before we get started with C, you need to get familiar with the environments you'll be working in. You'll be learning C in the Unix environment. Why? Unix allows you to write C programs on our computers so you don't need to worry about having any special software on your own computer. We have an entire OST course on Unix, but I'm just going to go over the few things you'll need to get started learning C.

On the screen below is a Web-distributed editor and development tool we call *CodeRunner*. With it you can program in a number of different languages (including C) and you can enter a Unix shell and a MySQL database shell. In this course we'll use the editor to create, save, and compile C files (we'll define what we mean by 'compile' later on). We'll also use the Unix shell to execute and test programs.

For this lesson, we'll use the Unix shell to learn some simple Unix tasks. In the next lesson, we'll show you how to use CodeRunner to edit and save (and later, how to compile) C files. To put CodeRunner into a Unix Shell, simply click the **Unix** button at the top of CodeRunner:

You will be asked for your username and password, then you will be asked by the shell for your password. Click on the shell and retype your password.

Getting familiar with Unix

Go ahead and put CodeRunner in Unix mode now.

Note Below is a gray "Observe" box. Gray boxes simply highlight code for you to observe; no other action is needed.

Observe the following:

```
Trying cold.useractive.com ...

Connected to cold.useractive.com.

Linux 2.2.16 (cold.useractive.com) (ttyp4)

cold login: username
Password:
```

The **username** will be your login name. Type your password to finish logging in, then you'll be at a Unix prompt.

Observe the following:

```
NoMad ...


Last login: Mon Jul 24 10:31:29 -0500 2000 on ttyp3 from
hottub.useractive.com.
No mail.

cold:~$
```

Listing Files

Alright. We've got a Unix prompt: **cold:~$** Now what? Let's explore a little. We can use the **ls** command to list the files in the current directory.

Note Below is a white "listing" box. White boxes show code or commands for you to type. We usually show text for you to type or add in **blue**. Action *is* required of you.

Type the command in blue into CodeRunner below:

```
cold:~$ ls -a
.    .bash_history    .emacs*   .ssh/
..   .bash_login*     .ncftp/
```

The **-a** flag makes **ls** show all of the files--otherwise, the files that start with a dot (.) won't appear. Your directory listing probably looks different from the one above, especially if you've taken other courses with us. We can also list files using a wildcard (*), like so:

Type the command in blue into CodeRunner below:

```
cold:~$ ls -a .bash*
.bash_history  .bash_login*
```

Directories

Let's make a new directory named **stuff** within your home directory. We do this with the **mkdir**, or *make directory*, command.

Type the command in blue into CodeRunner below:

```
cold:~$ mkdir stuff
cold:~$
```

Now if we list our files we can see the new directory.

Type the command in blue into CodeRunner below:

```
cold:~$ ls
stuff
```

Again, you may have a lot more files. Let's move into the new directory by using the *change directory* command, **cd**.

Type the command in blue into CodeRunner below:

```
cold:~$ cd stuff
cold:~/stuff$
```

List the files in the new directory

Type the command in blue into CodeRunner below:

```
cold:~/stuff$ ls -a
.    ..
```

What!? How can there be two files in a new directory? Not only that, but the names are just dots!

These are actually directories that are part of the Unix file structure. The single dot stands for the current directory, and the double dot stands for the previous, or "parent" directory. This allows us to **cd** back "up" to that directory.

Type the command in blue into CodeRunner below:

```
cold:~/stuff$ cd ..
cold:~$
```

Copying and Removing Files

You can copy and remove files with the **cp** and **rm** commands, respectively.

Type the command in blue into CodeRunner below:

```
cold:~$ cp /etc/issue .
```

What this means is to copy the file named **issue** from the **/etc** directory into the current directory (**.**).

Type the command in blue into CodeRunner below:

```
cold:~$ ls
issue    stuff
```

Excellent. But we don't really need that file, so let's remove it.

Type the commands in blue into CodeRunner below:

```
cold:~$ rm issue
cold:~$ ls
stuff
```

Logging Out

Now the file is gone again. That should be all we need to know for now. When you're done in Unix mode, just type **exit** and click the CodeRunner button at the bottom of the screen to get back.

Type the command in blue into CodeRunner below:

```
cold:~$ exit
logout

Connection closed.
```

See you in the next lesson!

Editing Files

Lesson Objectives

When you complete this lesson, you will be able to:

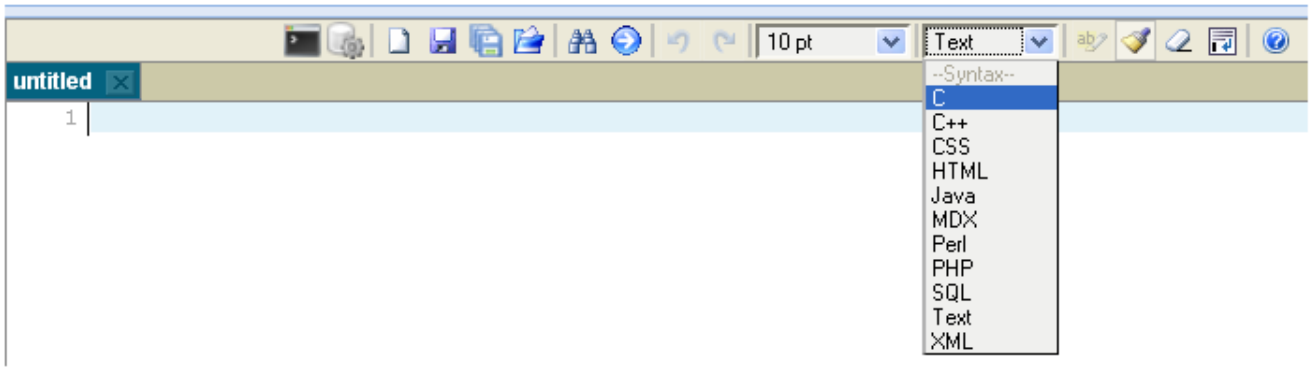
- edit C files in the CodeRunner editor.
- use the Unix prompt to run C files and review their output.

Editing Files

In this course, we will need to edit C files in the CodeRunner editor below and then switch to the Unix prompt to run them and review the output. You can use CodeRunner to edit and save your files. You will learn how to do this in this lesson.

Editing Files with CodeRunner

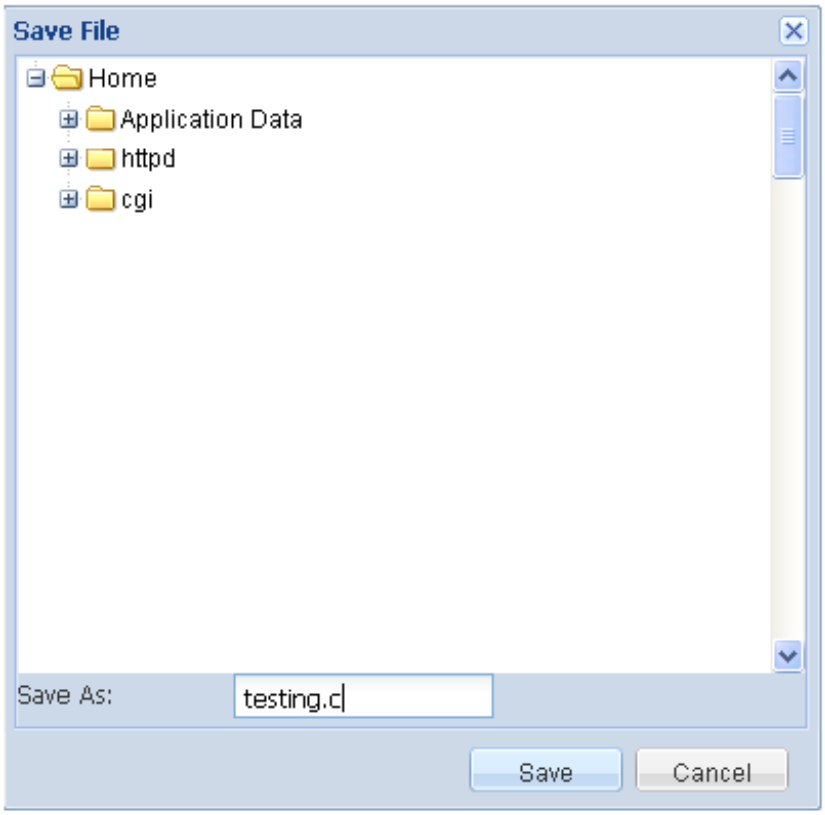
Look at the lower half of this window. By now you've figured out that it's a text editor called *CodeRunner*. Since this is a course in the C programming language, you'll want to use the **C mode**. By default, CodeRunner starts out in **Text mode**. To put CodeRunner into C mode, select the drop-down menu that contains "Text" and select **C** like this:



Now you are ready to try saving and retrieving some files. In the editor's text area type anything you want, maybe even write a story about yourself. Once you have some text written, let's save the file as "**testing.c**".

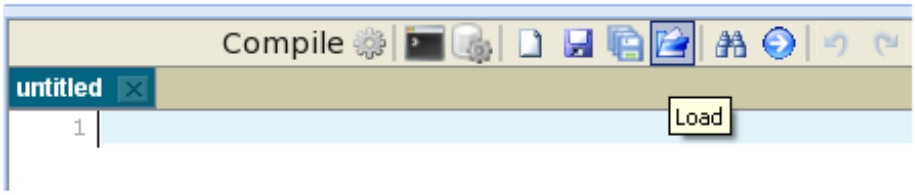
To save a file: you can select the **Save** button at the top of the CodeRunner window:

After you've done this, you'll be prompted to name your file and save it:

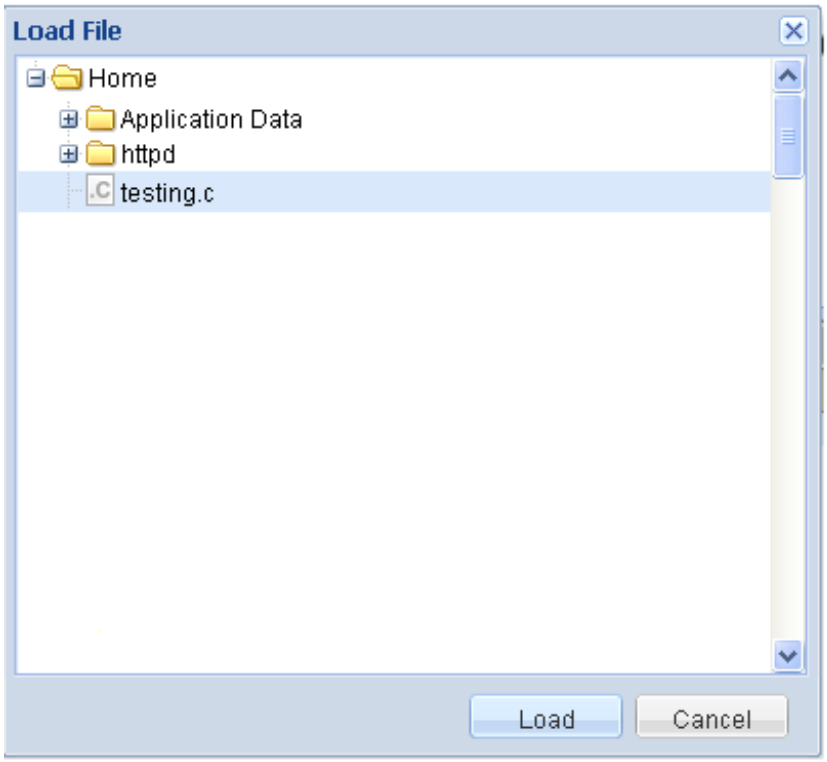


Save your file as **testing.c**.

Now, if you've saved a file and some time goes by and you need to retrieve the file from the server, simply select the **Load** button:



Select the file you wish to upload and click on the **Load** button.



We've completed the introduction to CodeRunner for now. In a later lesson we'll discuss using the Compile feature of CodeRunner to quickly compile your C programs.

That's it for the introductory stuff; let's get started with C, shall we?

Introduction to C

Lesson Objectives

When you complete this lesson, you will be able to:

- compile a program in C.
- execute programs from within a Unix shell.
- save files in your directory.
- print out a line of text.

What is C?

C was derived from the B programming language, which was built from BCPL. It was first used at Bell Labs in 1972. Now C is used everywhere. For example, the Unix operating system, the Apache web server, and even other programming languages (like Perl) are all written in C.

C is what's known as a high-level language. This means that it's a little closer to the flow of human understanding than the *machine language* your computer understands.

There are a lot of different versions of C, mostly due to different operating systems (DOS, Windows, Linux, and such). For the most part, they are all the same with just a few minor differences. You'll be learning C that uses glibc (The GNU C Libraries).

A basic C program

The easiest way to start learning C is to jump in head first and start doing some examples. I'm going to give you a simple program that prints something to the screen and then I'll explain how it works. You'll need to type the whole thing in for it to work, but for now, I only want you to pay attention to the stuff in **blue**.

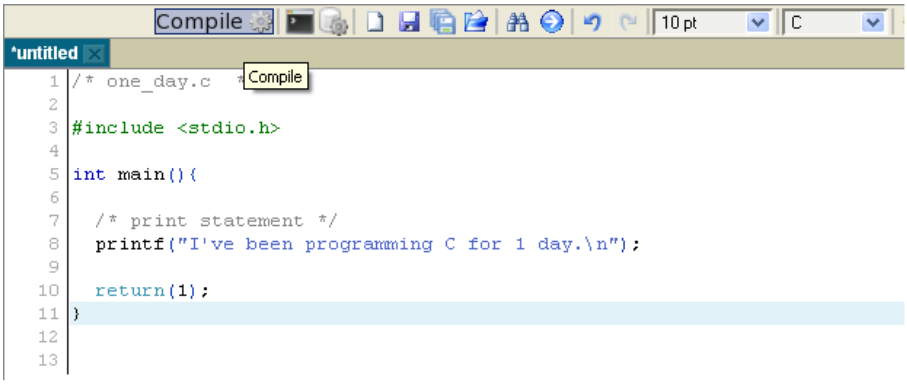
Put CodeRunner in **C** mode, and then type the program exactly as it is here:

CODE TO TYPE:

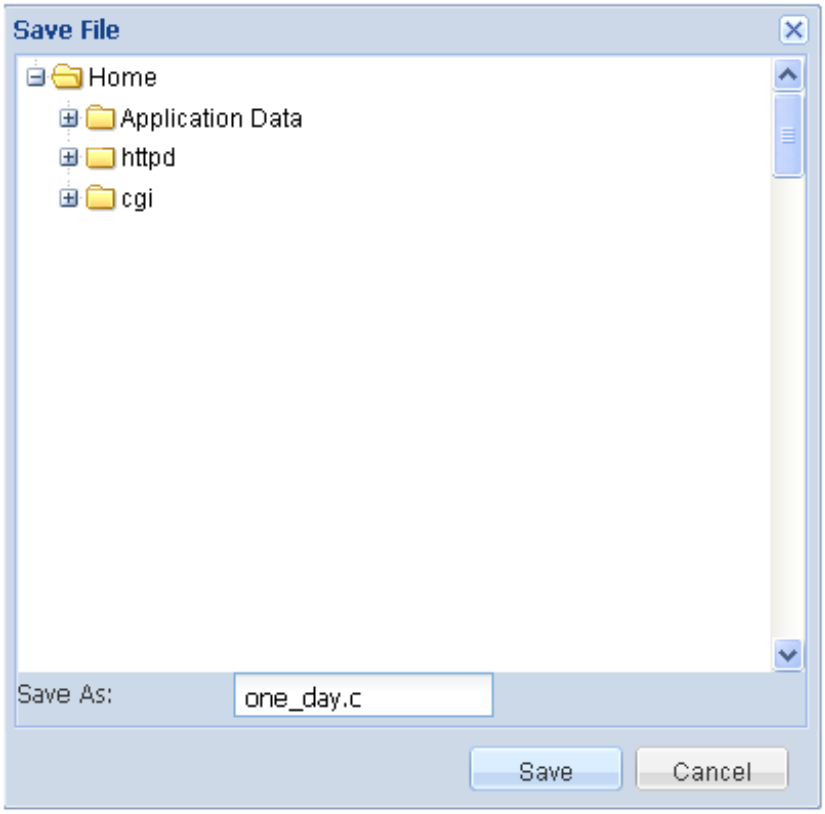
```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  printf("I've been programming C for 1 day.\n");

  return(1);
}
```

Now we'll compile this program. If you don't know anything about compiling, don't worry--that's why you're taking this class! We'll tell you more about compiling later. For now we just want to get a program working and see what it does.

When you finish typing the text into CodeRunner, select the **Compile** button:



Enter the filename **one_day.c** as shown, and then click **Save**.



Now that we've compiled the program, we can use it. For now, think of compiling as "getting the program ready."

As we mentioned before, we are going to execute programs from within a Unix shell. To enter your Unix shell, click the **Unix** button at the bottom of the screen:

You'll need to type your password in again to enter your shell.

Now that you're in your shell, be sure that the compiled program is in the same directory that you're in right now. Type the Unix listing command to ensure that the saved file and the compiled file are in your directory:

CODE TO TYPE:
`cold:~$ ls`

You should see two files, one called **one_day.c** and another called **one_day.exe**. If you don't see both files, use the **cd** command to enter the directory where you stored the one_day.c file.

The one_day.c file is the text file we created with the code in it, and **one_day.exe** is the *binary executable* file that was compiled from the one_day.c file. The binary executable is the software we created and that we can use. Of course, this software doesn't do very much. In fact, let's see what it actually does. At your unix prompt, type the following:

CODE TO TYPE:

```
cold:~$ ./one_day.exe
```

You should see the following output after you press the **Enter** key:

Output from one_day.exe

```
I've been programming C for 1 day.
```

Notice the **./** you typed before the file name. That tells Unix not to look in standard places for command files, and that the command we are executing (in this case "one_day.exe") is right here in the current directory. Try typing **one_day.exe** *without* the **./** and see what happens.

You've actually made a Unix Command. The command prints out "I've been programming C for 1 day." Not a very useful command I'll admit, but we've made progress!

Let's analyze your first program

We've made a C program and executed it. Now let's see what we actually did when we typed that stuff. Look at the program again:

Observe: one_day.c

```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  printf("I've been programming C for 1 day.\n");

  return(1);
}
```

In the first line, **/* one_day.c */**, the **/*** and ***/** tell the program to ignore everything in between them. All text between these symbols is considered comments; we've *commented out* that line. We do this to write ourselves notes about the program we are writing so that when we or others look at it later, it's clear what's going on. In this case we just repeated the name of the file so we know what file we are looking at.

The code in **red**, you'll see in every C program. For now you don't need to worry what it means. Just be sure and put the code you write inside the curly braces **{}**, where the **blue** lines are now. I promise I'll tell you later what that line does.

The first line in blue is just another example of a comment in C. Again, comments don't do anything, but it's a good idea to include them to document what each part of your program does. Think of them as notes you can write to yourself and others about the program. The start of the comment is marked with **/*** and the end is marked with ***/**. Anything after **/*** and before ***/** is part of the comment.

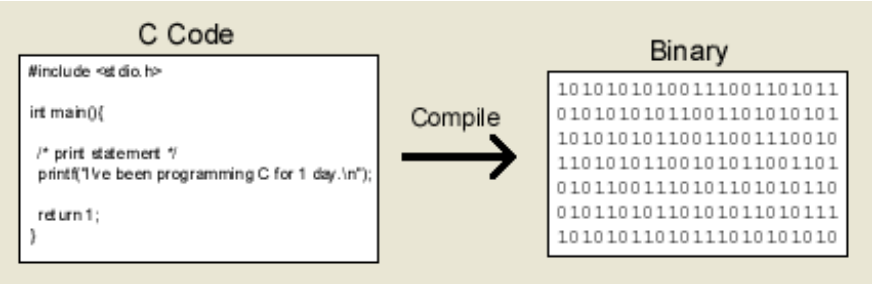
The second blue line is the meat of the program. This line tells the computer what we want it to do. In this case, we want to print the message, "I've been programming C for 1 day." to the computer screen. The **printf** function requires that you have the parenthesis and the quotations around the text you want to print. You also need to have a semi-colon at the end of the **printf** function.

Compiling

Now let's get a feeling for what it means to **compile** a program. There are two types of programming languages: *interpreted* and *compiled*. Interpreted languages convert your code into instructions for the computer every time it reads the file. This makes it faster to test your program because you don't have to go through a compilation step, but overall,

it's going to run slower than a program that's pre-compiled in a language like C. Compiling your program translates it to machine language (what your computer understands) before you run it. Additionally, the compiler can catch and report syntax errors for you.

A **compiler** is a program that converts your text into the computer instructions given by binary code (zeros and ones).



Before there were compilers, in the early days of computers, people had to program with zeros and ones! Now that we have languages like C and C compilers we have it much easier.

Common Compile-time Errors

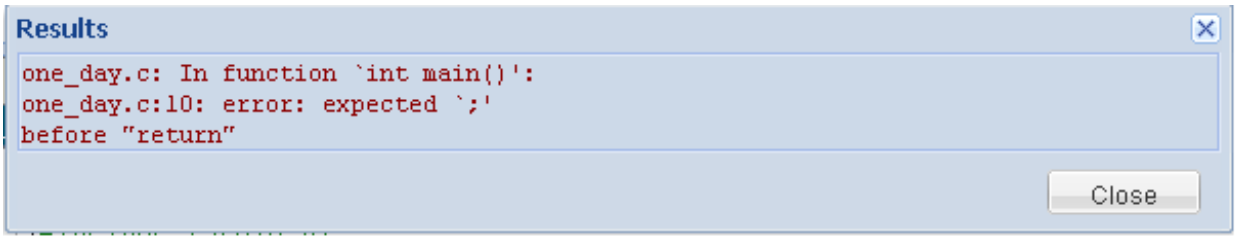
Let's try leaving off the semicolon after the **printf** function. Do this by re-opening the **one_day.c** file (if you don't have it open already) and removing the semicolon (shown in **red**) at the end of the printf line. Let's see if the compiler tells us we made a mistake. Make sure your file looks like this:

Remove the red semicolon:

```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  printf("I've been programming C for 1 day.\n");

  return(1);
}
```

Now, compile your program again by clicking **Compile**.

After you click **Save**, you should see the following message:



Instead of creating an executable file, we get an error message. Just in case your program consisted of more than one .c file (we'll see this more later), it tells you the error was in **one_day.c**. Then, as you can see, the error was inside of the **main()** function. Next, the compiler tells you that there's a syntax error before line 10. So if we go find line 10 of our code, we should notice the missing semicolon on the line before it. Put the semicolon back where it belongs.

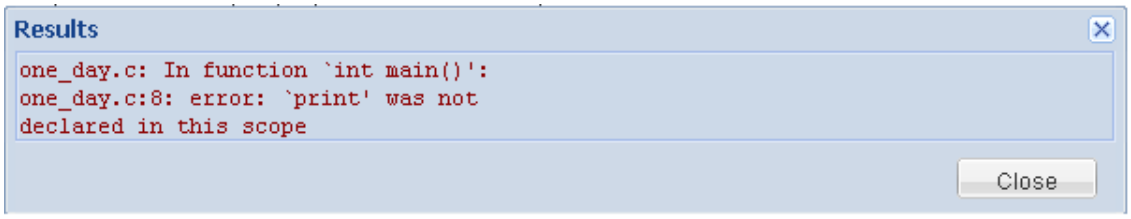
Note Even though we got errors and a new version of **one_day.exe** wasn't created, the old one still exists. You can still run the old executable file and it will still work. You should be careful to make sure there are no errors before trying to run the program. That way, you know for sure it's the newer version.

Let's try another one. Misspell **printf** as **print** (I do this all the time) and add the semicolon back:

Remove the red f and add the blue semicolon

```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  printf("I've been programming C for 1 day.\n");

  return(1);
}
```

Try compiling it again by clicking the **Compile** button. You should see errors like this:



What is all that stuff? Okay, so the compiler isn't always as helpful as we'd like it to be. But there are still a couple clues here. It tells you the error is in the **main()** function again and it's a problem with the reference to **print**. That should be enough to help you figure out the problem and correct your code. But what does the rest of that mean?

What Happens When You Compile

In broad terms, compiling your program converts it from the English-like code you write into something your computer understands. The compiler collects all of the parts and puts them together to make a single file that the computer can run. That's really all you need to know about compiling programs in C.

You don't have to need the rest of this unless you're curious about what's going on. *You don't need to understand compiling in much detail to be a great programmer.*

This doesn't have much to do with writing C, but I think it's useful to understand. It can help you debug your (and other people's) programs when they get more complicated. Every computer that you write C on has a set of C libraries. The libraries are just a bunch of C functions (like **printf**) that have been put together so you don't have to do everything yourself. However, you can't use a function without first telling C about it. So how did we use **printf**? Simple--the first line of our code included a file named **stdio.h**. This is called a *header file* because it contains information about the functions in the library. There are a lot of functions you might use in C that we'll include header files for later on. Let's quickly go over how this all ties together.

When you compile, a bunch of stuff is going to happen. First, a preprocessor is going to go through and add stuff to your code where you told it to include stuff. It'll even modify it sometimes as well. Once all the code is ready, it's actually compiled into machine language. At this point it's called object code. The compiler has **ld** link the object code to any C libraries that it will need to run. **ld** is a separate utility called the *linker*. Finally, the compiler creates the executable file to run later. This is the file that is used to see the actual code results.

A very loose analogy can be made by thinking about your lunch. I'm not kidding! Say you want to make a sandwich. You decide what you want to include in it (perhaps bread, mayonnaise, turkey, cheese, lettuce, and tomato). The "program" you write is deciding what order to put them on the bread. You're not actually going to bake the bread or make the mayonnaise from scratch, just like you aren't going to make all the functions you use from scratch. More likely you'll go to the refrigerator and get some mayonnaise out of the jar. That's just like (well, not exactly) the compiler using printf from its library.

I hope this helps you understand a little bit about compiling. We'll discuss this topic again in a later lesson when we go through *command line compiling*.

See you at the next lesson!

Variables and Data Types

Lesson Objectives

When you complete this lesson, you will be able to:

- store information in variables.
- print multiple variables.

Variables

In previous lessons we learned how to print out a line of text. That's fine and good, but we need to be able to change things. In C, as in most languages, we store changeable information in *variables*. It might be useful to think of a variable as a box. You can store stuff in a box and you can change what's in it as well--you refer to it by the box name, so it doesn't matter if the contents change. Before we can use a variable in C, we must *declare* it. Essentially we must make a box that we'll put things in later.

What we're going to do next is declare a variable named **days**. We'll use this variable to store a number and then print it out.

Open your one_day.c program in the CodeRunner editor, and click the Save As icon (). In the Save File dialog box, type the filename **prog_days.c**. Then, edit the file as shown in **blue** below:

CODE TO TYPE:

```
/* prog_days.c  */
#include <stdio.h>

int main(){
  int days;

  days = 1;

  printf("I've been programming C for %i day(s)\n",days);

  return(1);
}
```

Save and Compile this code as **prog_days.c** by clicking on **Compile**. Switch to **Unix** and run your program by typing **./prog_days.exe**. You should see this output, almost the same as before:

Output from prog_days.exe:

```
I've been programming C for 1 day(s).
```

Here we modified the first program to include some new code (the stuff that was in blue). I'll go over the new stuff one line at a time.

int days; declares the variable called **days**. We're creating the box to use later. **int** stands for integer. C keeps track of what kind of data each variable can contain (whether the box will store apples or oranges). We'll go over integers and other variable types a little later. Note how the line ends in a semicolon, just like most statements in C.

days = 1; is a variable *assignment*. We're setting **days** equal to 1. So now we've got an integer box with a 1 in it.

printf("I've been programming C for %i day(s)\n",days); might look more complicated than it really is. Since there isn't anything to distinguish a variable from any other word, we have to tell **printf** that we want to include the value of the **days** variable, rather than the word "days," in the output. The **%i** says, "place an integer here," and "days" after the comma tells where to get the integer. So **printf** gets the value for %i from the **days** variable.

Now let's change the value of **days**.

CODE TO TYPE:

```
/* prog_days.c  */
#include <stdio.h>

int main(){
  int days;
  days = 1;

  printf("I've been programming C for %i day(s)\n",days);

  days = days + 1;

  printf("Tomorrow it will be %i day(s)\n",days);
  return(1);
}
```

Compile and run this code (you should know the routine by now). What output did you get? Can you figure out what the code is doing? Try to look at the code and predict what it's doing. Try CHANGING THE CODE ANY WAY YOU SEE FIT! Take some time to experiment. You can't hurt anything.

Let's break down the new code.

days = days + 1; is simply another assignment operation. The only difference is the expression **days + 1** will be evaluated *before* a new value is assigned. Since this isn't an argument to printf, we don't have to do the %i stuff this time. **days** (right of the = sign) is still equal to 1, so days + 1 = 2. Now the assignment takes place and **days** (left of the = sign) is 2.

Printing Multiple Variables

We're not limited to printing one variable at a time either. **printf** allows us to print multiple variables simply by adding more arguments to the statement.

Type the code in blue into CodeRunner below:

```
/* prog_lang_days.c */
#include <stdio.h>

int main(){
  int days=3;
  char mychar='C';

  printf("I've been programming %c for %i days.\n",mychar,days);
  return 1;
}
```

Compile and test this code after saving it as **prog_lang_days.c**. As always, experiment with the code a little. Try to get it to print a few different things.

Output from prog_lang_days.c

```
I've been programming C for 3 days.
```

Characters are printed using **%c**. The variables given to **printf** should be in the same order that they appear in the output string. Play around with this a little; try printing 3 or 4 variables at a time.

Data Types

C is very particular about keeping track of what *kind* of data is stored in all of your variables. Also, each variable in C only keeps track of one thing. Different variable types are like boxes designed for different kinds of objects. There are three main data types that we'll be using throughout this course: **int**, **float**, and **char**. Integers (**int**) can only store numbers like -12, 0, 43, 32589. They can't store decimals or fractions. If you need to store numbers that have a decimal point, use a floating point variable (**float**). The last of the three types is for storing characters. The limitation of **char** variables is that they can only store a single ASCII character; no words or strings. We'll get around this later with arrays. There are other variable types, but the only variations are in the maximum value of the data being stored.

Note Integers might seem a little inferior; you can store an integer in a float variable, so why bother using the int data type at all? Integers take up less memory than floating point numbers and even though that's not a big problem with today's computers, it's still good programming practice to use them when a floating point number isn't necessary.

More on Declaring Variables

In C, we have to declare what variables we want to use before we can use them. This is because C has to allocate some of the computer's memory for each variable we want to store. Because of this, the first part of your C program should declare all the variables. You can always add more variables to the top as you need them. When we declare a variable, all we're really doing is specifying what type of data will be stored and what the variable name is.

We also need to assign values to our variables. We can either do that at the same time we declare them, or we can do it separately. Look over the following example, and later you can use it as a reference if you need to.

OBSERVE:

```
#include <stdio.h>

int main(){
  /* variable declarations */
  int a, b;
  float x;
  char mychar1;

  /* variable declarations and assignment */
  int c=5;
  char mychar2='d';

  /* assigning values to variables */
  a = 3;
  b = 4;
  x = -4.57;
  mychar2 = 'd';

  return 1;
}
```

In the first part of the example (in **blue**), we declare a few variables of different types. Conveniently, we can use commas to include multiple variables of the same type. That keeps us from needing two separate lines for **a** and **b**.

In the next section (in **dark green**, we give variables values at the same time as we declare them. The equals sign is known as the *assignment operator*. The value on the right side of the equals sign is assigned to the variable on the left side.

The next section (in **purple**) shows how to assign values to variables that have already been declared. Note that when assigning a character value, we must use single quotes. Single quotes are used for single characters and double quotes are used for strings--we'll discuss that later.

You've learned quite a lot! Now I think you're ready for... MATH!

Math

Lesson Objectives

When you complete this lesson, you will be able to:

- use arithmetic in C.
- use the modulus operator.
- execute trigonometric functions in C.

Doing Math

Yeah, I know--Math. It's not everyone's favorite activity, but we've gotta do it. Who knows you may like it afer all.

Arithmetic in C follows basically the same rules you've always used. If you have a string of numbers like 1 + 3 * 2, the numbers 3 and 2 will be multiplied first, and the number 1 will be added to the result to yield 7. Multiplication and division are done before addition and subtraction. However, if you have a string of numbers like (1 + 3) * 2, the numbers 1 and 3 will be added, and the number 2 will be multiplied by the result to yield 8. You can use parentheses to change precedence.

Type in the following example as a file called **math.c**. Again, don't worry too much if you don't understand everything yet.

Type the following into CodeRunner below:

```
/* math.c */
#include <stdio.h>

int main(){
  int a=1, b=2, c, d;
  float x=1.1, y=2.2, z;

  c = a + b;
  z = (x + y) * b;
  d = b / x;

  printf("a + b = %i\n",c);
  printf("(x + y) * b = %f\n",z);
  printf("b / x = %i\n",d);

  return 1;
}
```

Compile and run the program. If you don't remember how, go back to lesson 3 for a reminder. The output should look like:

Output from math.c

```
a + b = 3
(x + y) * b = 6.600000
b / x = 1
```

Notice that the first thing we do in the program is declare variables and values we'll use later. The variables **a**, **b**, **c**, and **d** are all integers while **x**, **y**, and **z** are floating point numbers. We're assigning initial values to some of the variables when they are declared; the others will be used to store results. To get the result of the math, we simply assign the math expression to a variable. The mathematical expression is evaluated, and the result is stored in the variable. Since **a** is equal to 1 and **b** equals 2, the value of **c**, after the addition and assignment, is 3.

Cool, huh? But let's look at the last line of output. **b** is 2 and **x** is 1.1, so the result of the division should be about 1.82. But we got 1--what's going on? The result of the division was indeed 1.82; however, we're trying to assign that number (which has a decimal point) to a variable that is an integer. As a result, C just ignores everything after the decimal point and we get 1. You have to be very careful when you do math with both integers and floating point numbers.

This is all fine and good if you want to add, subtract, multiply, and divide, but that's not going to get you very far in the world of a C programmer. There are more operators available to use besides +, -, *, and /. For example, there is an operator called the modulus (**%**). The modulus gives the remainder of a division operation. For example, 21 divided by 2 equals 10 with a remainder of 1. If we want to know the value of that remainder we'd use the modulus (**%**):

answer = 21 % 2;

After the modulus is taken, **answer** will be set to 1. This is really useful for finding out if a number is even or odd.

How about calculating 5^2? Well **5 * 5** works, but what if it was 5^{20}? I don't know about you, but I'd rather not type that in 20 times! Lucky for us, there's a header file (math.h) that defines a lot of common math functions for us. Do you remember how to include a header file? (If not, go back to lesson 2 for a reminder.) Now we can use the **pow()** function to do our calculation for us. It's as simple as **answer = pow(5,20);**.

Here's a list of other functions that are defined in **`math.h`**:

Common Functions in math.h	
Function	**Result**
pow(x,y)	x^y
fabs(x)	absolute value of x
sqrt(x)	square root of x
exp(x)	e^x
log(x)	natural log of x
log10(x)	log base 10 of x
ceil(x)	rounds up to the nearest integer
floor(x)	rounds down to the nearest integer
cos(x)	cosine of x
sin(x)	sine of x
tan(x)	tangent of x

Note All of the trigometric functions assume that you enter your angle in radians.

When you start doing math with large numbers, you may exceed the bounds of the normal data types. If this happens, you can use **double** instead of **float** and **long** instead of **int**. **double** and **long** are the same as their conterparts, except they take up more memory and allow you to store larger numbers. If you exceed the bounds of a data type, C will either complain about an overflow, or print **inf** instead of the value you expected (inf stands for infinity).

That wasn't so bad, was it? See you in the next lesson.

Output and Input

Lesson Objectives

When you complete this lesson, you will be able to:

- format output.
- do calculations based on actual input rather than hardcoded variables.
- use the **getchar()** function to read input from the buffer.

Printf

It's about time I explain all of those **printf** statements I've had you use in earlier lessons. As you may have already guessed, the **printf** statement is used to print "stuff" to the screen. Printing out just a string of text is straightforward:

A simple printf statement

```
printf("my line of text\n");
```

You're probably wondering what the **\n** is at the end of the string of text. The backslash character is called the *escape* character. It tells C to treat the next character differently from normal. For example, **\n** prints a new line. The escape character is also used if you want to print special characters such as quotes or a backslash itself. In the CodeRunner editor, in C mode, create the following program, compile and save it as **print_formatting.c**, and run it.

CODE TO TYPE:

```
/* print_formatting.c */
#include <stdio.h>

int main(){

  printf(" \" \\ hello\tthere \' \? \n");

  return 1;
}
```

You should see the following output when testing it:

Output from print_formatting.exe

```
 " \ hello      there ' ?
```

You can see how the backslash delimiter lets you print out quotation marks and other special characters that C would otherwise interpret as part of the syntax.

Printing Variables

We wouldn't get anywhere if all we could print was text, so obviously there has to be a way to print out the values of our variables as well. We saw in a previous lesson how to use **%i** to print out variables with integer values. Recall the following statement:

Printing an integer-type variable

```
printf("Tomorrow it will be %i day(s)\n",days);
```

The **%i** tells **printf** that we want to print out the value of the variable (an integer) to the screen. The second thing we gave **printf** was the name of the variable we want to print. You can print out a bunch of variables at the same time as well.

Type the following into CodeRunner below:

```
/* print_formatting_b.c */
#include <stdio.h>

int main(){
  int a=1;
  float x=2.345, y=-6.78;
  char chr1='p';

  printf("%i %f %f %c\n",a,x,y,chr1);

  return 1;
}
```

Here, **%f** and **%c** are for printing floats and characters. The variables at the end are in the order they will be printed. An integer (a), two floats (x and y), and a character (chr1). Save this as **print_formatting_b.c** and compile and run it.

Output from print_formatting_b.exe

```
1 2.345000 -6.780000 p
```

Notice that the floating point numbers, when printed, are padded with zeros up to the sixth decimal place. That's because floating point numbers have a precision of six decimal places.

Formatting Output

Let's say you've taken a bunch of data floating point numbers that have information out to the sixth decimal place. However, you don't need to be that accurate when printing a report of your data. For example, let's take 123.456789 stored in a variable called **mynum**. So try this:

Type the following into CodeRunner:

```
/* print_formatting_c.c */
#include <stdio.h>

int main(){
  float mynum=123.456789;

  printf("%10.2f\n",mynum);


  return 1;
}
```

Compile that and try to guess what happened before reading my explanation. This is **print_formatting_c.c**

The first number, **10**, is called the *field width*. This means that, if necessary, it will add blank spaces before the number represented by **f** to make it take up 10 spaces. (You can also use -10 to add the spaces after the number.) The **.2** specifies the precision. It will round up or down accordingly. The output would look like this:

OBSERVE:

```
    123.46
```

Four spaces were automatically inserted left of the 1 to make the total length 10.

Scanf

Printing stuff is great, but it's a lot more helpful to be able to do calculations based on actual input rather than hardcoded variables. That's where **scanf** comes in. As you can probably tell from the name, it works similarly to **printf**. Here's an example program that reads in two numbers entered by the user and multiplies them together:

Type the following into CodeRunner:

```
/* multiplier.c */
#include <stdio.h>

int main(){
  float num1,num2,result;

  printf("Please enter two numbers separated by a space: ");

  scanf("%f %f",&num1,&num2);

  result = num1 * num2;
  printf("The result is %f\n",result);
  return 1;
}
```

Call this one **multiplier.c**. Save, compile, and run it to see how it works.

Output from multiplier.exe

```
Please enter two numbers separated by a space: 3.4 1.2
The result is 4.080000
```

First we want to print out a line prompting the user for their input. Otherwise it would just sit there and they wouldn't have a clue what was going on. The space behind the colon is just there to make it look pretty. So what about that **scanf()** line? When the user types information at the prompt and presses Enter, it goes into a standard input (STDIN) buffer (also referred to as a stream). It stays in the buffer until the program decides what to do with it. The first argument, **"%f %f"**, tells scanf to look in the buffer for two floating point numbers separated by *white space*. Next, we tell scanf where to store the two numbers. Remember that when you declare a variable, C allocates a spot in memory to store the value. We need to tell **scanf** where that memory is. This can be done by using the *address operator*, **&**. For now, just remember to use this in all of your **scanf** statements. We'll learn more about it later.

Note *White space* refers to any number of spaces, tabs, or newlines in a row. So, in essence, you could enter your first number, press **Enter** twenty times, and then enter your second number and this program would still work.

Now the two floating point numbers that the user gave us are stored in the variables **num1** and **num2**. They can be used later in the program just as if we had assigned values for them ourselves.

Normally it's very important to test to make sure the input is valid, but since we don't know how to do that yet, we'll just assume that the user entered two floating point numbers correctly.

A very common error is forgetting the address operator in the **scanf()** statement, so if your output doesn't make any sense, you might want to check that first.

Getchar

Another very useful function for processing user input is called **getchar()**. **getchar()** is used to read input from the buffer just like **scanf**. The difference is that **getchar()** only reads in one character at a time and returns it as an integer. The following example illustrates this:

Type the following into CodeRunner below:

```
/* get_a_char.c */
#include <stdio.h>

int main(){
  char a;
  a = getchar();
  printf("%c\n",a);

  return 1;
}
```

The **getchar()** function reads a single character from the input buffer and stores it in variable **a**. Save this program as **get_a_char.c**, then compile and test it out.

So what's this business about **getchar()** returning an integer? How are we comparing character values if it returned an integer? Well, it's about time I told you something. Every character corresponds to an integer value according to the ASCII chart. Basically, this is because your computer only understands numbers. It doesn't really matter in most cases because the use of **%c** with **printf()** displays a character for us.

The problem is that the *characters* 0-9 do not correspond with the *integer values* 0-9. The characters 0-9 are actually the ASCII integers 48-57. So if you plan to do any math with the numbers you get from **getchar()**, you need a way to convert them to their real integer values first. We can do this with a function called **atoi()**, which is part of **stdlib.h**. **atoi()** requires the use of the address operator, **&**, like **scanf()** did.

Type the following into CodeRunner:

```
/* get_an_int.c */
#include <stdio.h>
#include <stdlib.h>

int main(){
  char a;
  int i=0;

  printf("Enter an integer between 0 and 9: ");
  a = getchar();
  printf("%c\n",a);

  i = atoi(&a);
  printf("%i\n",i);

  return 1;
}
```

Both of the print statements produce the same output to the screen, but **a** and **i** are different kinds of variables.

Save this program as **get_an_int.c** and test it out.

See you in the next lesson!

Conditional Statements

Lesson Objectives

When you complete this lesson, you will be able to:

- use conditional statements in C.
- use basic boolean algebra.
- embed If staments and Identation.
- use Switch statements.

If Statements

Sooner or later, you will need programs that can make decisions. They will need to take different actions depending on the value of the variables and they will need to check for errors to prevent possible bugs. This is accomplished using conditional statements. There are two main ways to do this in C and we'll cover them both. The first type we'll look at is called an **if** statement. Chances are, if you've programmed in any other language, you've seen **if** statements before. The basic format of an **if** statement is to test if something is true, and if so, perform a series of commands. Let's do an example to see this in action.

Type the following into CodeRunner below:

```
/* integer_test.c */
#include <stdio.h>

int main(){
  int var1;

  printf("Enter an integer: ");
  scanf("%i", &var1);

  /* testing variable 1 */
  if(var1 < 10){
    printf("var1 is less than 10\n");
  }

  return 1;
}
```

Save this as **integer_test.c**, and then compile and run it, entering **3** when prompted.

Observe the following:

```
Enter an integer: 3
var1 is less than 10
```

The **scanf** statement reads an integer from the user, just like we read the two floating point numbers in the previous lesson. It assigns the integer to &var1.

The **if** statement checks the result of **var1 < 10**. If the result of that expression is true, then the code inside the braces is run and we display a line stating that var1 is less than 10. However, if the result is false, the code is skipped. For example, if var1 is 20, the expression is false and the program goes on to the rest of the program after the closing brace. However, if var1 is -2, the expression is true.

The braces aren't actually required if there's only one command to be run inside of the if statement, but I usually include them so I don't get confused.

The following test operators are available C:

Test Operators	
<	Less than

>	Greater than
<=	Less than or equal
>=	Greater than or equal
==	equal
!=	does not equal

Note Be very careful not to confuse the equality operator, (**==**) with the assignment operator (**=**). Also, the exclamation mark is commonly used as the boolean NOT. All it really does is reverse whether the statement is TRUE or FALSE. For example: **if(!(var >= 2))** is the same thing as **if(var < 2)**.

Now, modify your integer_test.c program as shown in **blue**:

Type the following into CodeRunner below:

```
/* integer_test.c */
#include <stdio.h>

int main(){
  int var1;

  printf("Enter an integer: ");
  scanf("%i",&var1);

  /* testing variable 1 */
  if((var1 >= 1) && (var1 <= 10)){
     printf("var1 is between 1 and 10\n");
  }

  return 1;
}
```

Compile and test this one as well. Play with it a little to get a different range of numbers to check.

Observe the following:

```
Enter an integer: 6
var1 is between 1 and 10
```

Boolean Algebra

The following is just a little aside to explain the basics of boolean algebra. I'll be using letters to represent individual expressions such as `myvar <= 5`. Additionally, since I'm discussing this in relation to the C language, I'll use C type syntax for my boolean expressions. If you open a computer science book they'll look a bit different, but the concepts are still the same.

C	Meaning
&&	AND
||	OR
!	NOT
1	TRUE
0	FALSE

As you learned, an `if` statement checks to see if a condition is TRUE (equal to 1). Each individual conditional expression can return either a 1 or 0 (TRUE or FALSE). Let's take two expressions, `A` and `B`, and put them together using an AND.

OBSERVE:
`A && B`

This new expression, will return 1 only if both `A` and `B` are equal to 1. If either of them are 0, the expression is FALSE. How about OR?

OBSERVE:
`A || B`

An OR will return 1 if *either* of the expressions is 1. A good way to see this is by using what are called truth tables. A truth table shows all of the different combinations for the inputs and the resulting output. Below are the truth tables for AND and OR.

AND		
A	B	Result
1	1	1
1	0	0
0	1	0
0	0	0

OR		
A	B	Result
1	1	1
1	0	1
0	1	1
0	0	0

We can reverse the result of any expression by using the NOT operator. For example, the following expression returns 1 only if both of the inputs are 0.

OBSERVE:
`!(A && B)`

This would be pronounced: "Not A and B." However, from that some pronunciation we could make this expression:

OBSERVE:
`!(A) && B`

Here are the truth tables for those last two examples.

!(A && B)		
A	B	Result
1	1	0
1	0	0
0	1	0
0	0	1

!(A) && B		
A	B	Result

1	1	0
1	0	0
0	1	1
0	0	0

Below we're testing to see if var1 is greater than or equal to 1 AND less than or equal to 10. We could do it with two separate statements, but it's just as easy and requires less code to do it with one. This expression returns true only if both of the tests are true. The two **&&** stand for AND. Two vertical lines ("pipe" symbols), **||**, stand for OR.

Modify your integer_test.c again, as shown in **blue** below.

Type the following into CodeRunner below:

```
/* integer_test.c */
#include <stdio.h>

int main(){
  int var1;

  printf("Enter an integer: ");
  scanf("%i",&var1);

  /* testing variable 1 */
  if((var1 < 1) || (var1 > 10)){
     printf("var1 is not between 1 and 10\n");
  }else{
     printf("var1 is between 1 and 10\n");
  }

  return 1;
}
```

After compiling, test it out to get both of the possible outcomes.

The above statement illustrates the use of OR. The test cases are reversed as well, so we get the same result. In addition to the **if** statement there's also an **else**. If the condition in the **if** statement is FALSE, then the code inside of the **else** is run. It's the same thing as putting another **if** statement that has the opposite conditions.

You can also string together if/else statements as follows:

Type the following into CodeRunner below:

```
/* integer_test.c */
#include <stdio.h>

int main(){
  int var1;

  printf("Enter an integer: ");
  scanf("%i",&var1);

  /* testing variable 1 */
  if((var1 >= 1) && (var1 <= 10)){
     printf("var1 is between 1 and 10\n");
  }else if((var1 > 10) && (var1 <=20)){
     printf("var1 is between 11 and 20\n");
  }else{
     printf("var1 is not between 1 and 20\n");
  }

  return 1;
}
```

We already know that code inside an **else** statement is executed when the condition for the **if** is false. So what if the first thing after the **else** is another **if** statement? It really just allows for another possible set of conditions. Using **else**

if is sort of like having a giant OR, but the different conditions result in different actions.

Be sure to compile and test this example. Try it multiple times and get it to give you all of the different outputs.

> **Note** Compile-time errors will start to get more complicated now. Forgetting a parenthesis or bracket might not cause an error until much later in your program. Just backtrack until you find it. For example, removing the closing brace might cause **gcc** to report a parse error at the end of input.

Embedded Ifs and Indentation

Now suppose we have a situation where we only want to test for certain cases when other cases are true or false. When this happens, we can embed any number of **if** statements in other statements, like in the following example:

Type the following into CodeRunner:

```
/* integer_test.c */
#include <stdio.h>

int main(){
  int var1, var2, var3;

  printf("Enter three integers (0, 1, or 2) separated by spaces: ");
  scanf("%i %i %i",&var1,&var2,&var3);

  /* testing variables */
  if(var1 == 1){
    if(var2 == 1){
      printf("var1 is 1 and var2 is 1\n");
    }else{
      if(var3 == 2){
        printf("var1 is 1, var2 is not 1, and var3 is 2\n");
      }
    }
    if(var2 == 0 ){
      printf("var2 is 0\n");
    }
  }
}
```

Save, compile, and run this program to see how it works.

In this case, we only test **var2** if **var1** is 1. Also, we only test **var3** if **var1** is 1 and **var2** is not 1. Review this example very closely. Try switching things around and rerunning it.

Note how the *indentation* helps to indicate the grouping of ifs and elses and their braces. Proper indentation while writing your program can make debugging a lot easier. The next bit of code is functionally exactly the same as the one above, but notice how improper indentation can make it a lot harder to read.

Observe the following:

```
  if(var1 == 1){
if(var2 == 1){
    printf("var1 is 1 and var2 is 1\n");
    }else{
  if(var3 == 2)
      printf("var1 is 1, var2 is not 1, and var3 is 2\n");
}
if(var2 == 0 ){
    printf("var2 is 0\n");
    }
        }
```

Note Emacs Tip: If you're using the emacs editor, you can use the **Tab** key to correct the indentation. Place your cursor at the beginning of the incorrect code, press **Tab** and the down arrow repeatedly in succession until the indentation is fixed. When emacs is in C mode, it recognizes the syntax and will try to correctly indent it for you. You may notice it auto-indenting some lines after you finish typing them.

Let's switch gears a little. Instead of testing a lot of different variables, we can test the same one for a lot of different cases.

Observe the following:

```
if((var == 1) || (var == 2) || (var == 4) || (var == 8) || (var == 16)){
 printf("var is either 1, 2, 4, 8, or 16.\n");
}else if((var == 3) || (var == 5) || (var == 7)){
 printf("var is either 3, 5, or 7.\n");
}else if(var == 6){
 printf("var is 6.\n");
}else{
 printf("var isn't any of those numbers.\n");
}
```

You can see how this could be tedious, especially if you need to test for more cases than I show here. A better way to go about this is by using a *switch* statement.

Switch Statements

Switch statements are used to perform different operations depending on the value of one variable or function. The example below shows how to use **switch**, and is the equivalent of the **if** statement above.

Type the following into CodeRunner below:

```
/* integer_test.c */
#include <stdio.h>

int main(){
  int var;

  printf("Enter an integer: ");
  scanf("%i",&var);

  /* testing variable */
  switch(var){
    case 1: case 2: case 4: case 8: case 16:
      printf("var is either 1, 2, 4, 8, or 16.\n");
    break;
    case 3: case 5: case 7:
      printf("var is either 3, 5, or 7.\n");
    break;
    case 6:
      printf("var is 6.\n");
    break;
    default:
      printf("var isn't any of those numbers.\n");
    break;
  }

  return 1;
}
```

Save, compile, and run this program. Test it out to make sure it does what you think it will do.

At first, this just seems longer to write out, but some people find it easier to read when there are a lot of cases. It all boils down to personal preference in the end. However, you should be familiar with how both **if** and **switch** work in case you find yourself modifying someone else's code.

The argument, **var** in this case, is the variable or function that is tested in each **case**. Each instance of **case** is a test for equality with what we're testing for. If one of the cases match, everything until the first occurrence of **break** is executed. If none of the cases match, the **default** case is run. If you don't need a **default** case, you can omit it.

See you in the next lesson!

Loops

Lesson Objectives

When you complete this lesson, you will be able to:

- use various Loops in C.
- interrupt Loops.
- avoid Infinite Loops when necessary.

Loops

Loops are among the most powerful tools available in programming. The idea behind loops is to keep doing the same thing over and over until a certain condition is met. Consider what happens when you wash dishes by hand. You wash and dry a dish, then you go on to the next one. The process is repeated, washing and drying, until there are no more dishes left. This is a dish washing loop.

I'm going to start by teaching you **while** loops because they are the most similar to the **if** statements we've already covered. Type the following code in the CodeRunner editor.

Type the following into CodeRunner:

```
dishes=0;
if(dishes < 10){
   printf("done ");
}
```

That's a simple enough **if** statement, right? It just prints "done " if **dishes** is less than 10. Now let's replace the **if** with a **while**. (Do ***not*** try to compile and run this yet!)

Type the following into CodeRunner:

```
dishes=0;
while(dishes < 10){
   printf("done ");          /* Do not try this code */
}
```

The only difference is that an **if** will only execute once and a **while** will keep going as long as the condition "dishes < 10" is true. You can use multiple conditions for a **while** just like you can with **if** statements. That's really all there is to **while** loops, but before I have you try them I want to go over the very important topic of infinite loops.

Infinite Loops

Look again at the previous code example. Do you see what's wrong? If **dishes** is less than 10, "done" will be printed to the screen until the value of **dishes** is no longer less than 10. But because the value of **dishes** never changes, the **while** will keep running... and running... and running. Forever. That's obviously not what we want, but it does happen (mostly by accident) and you need to know what to do when it does. Lucky for us, Unix has a built-in way to stop your program. In an emergency like this, you can always press **Ctrl+c** (Control and c at the same time), to stop your program from running.

Note You only need to press **Ctrl+c** once, even though your program may appear to keep running. What's actually happening is that your infinite loop will probably execute thousands of times before you realize it and stop your program. Your terminal isn't fast enough to display everything as quickly as your program is writing to STDOUT. You just have to wait a few seconds for it to catch up.

Avoiding Infinite Loops

Avoiding infinite loops isn't that hard, it's all in how you use them. Let's add 1 to **dishes** every time we go through the loop. Essentially we'll end up with a loop that executes ten times. It'll run once for each value of **dishes** from 0 to 9.

Modify your program as shown in **blue** below and make sure you get the output you expect:

Type the following into CodeRunner:

```
/* dish_washer.c */

#include <stdio.h>

int main(){
  int dishes;

  dishes=0;

  while(dishes < 10){
    printf("done ");
    dishes++;
  }

  printf("\nDarn, time to do the dishes again.\n");

  return 1;
}
```

Now it's safe to compile and test your first loop. :-) Save the file as **dish_washer.c** and go at it.

The line **dishes++;** is a shortcut for **dishes = dishes + 1;**. This means that the value of **dishes** increases by 1 each time the **while** loop repeats. The first time through the loop, the value of **dishes** is 0, the second time 1, the third time 2, until the end of the tenth time through the loop, when the value of **dishes** is incremented so that it equals 10. Then, the eleventh time will be attempted; however, because **dishes** is no longer less than 10, the loop terminates.

This is what you should get:

Observe the following:

```
cold:~$ ./dish_washer.exe
done done done done done done done done done done
Darn, time to do the dishes again.
```

Let's try another example. Type this new program in CodeRunner:

Type the following into CodeRunner:

```
/* even_odd.c */

#include <stdio.h>

int main(){
  int var=20,ans;

  while(var < 26){

    ans = var % 2;
    if(ans == 1){
      printf("%i is odd\n",var);
    }else{
      printf("%i is even\n",var);
    }

    var++;
  }
  return 1;
}
```

Play with **even_odd.c** a little too and see what you can come up with.

Here we're using basically the same strategy as in the previous example. This time, **var** starts at 20 and the loop

continues while it's less than 26. The modulus (%) is used to get the remainder of dividing **var** by 2. The remainder will either be 1 or 0 depending on whether or not the number is even or odd. Don't forget to compile and test the examples!

OBSERVE:

```
cold:~$ ./even_odd.exe
20 is even
21 is odd
22 is even
23 is odd
24 is even
25 is odd
```

Do-While Loops

The only difference between a **do-while** loop and a regular **while** loop is that the **do-while** will always be executed at least once.

Type the following into CodeRunner below:

```
/* do_while.c */
#include <stdio.h>

int main(){
  int i=0;

  do{
    printf("%i ",i++);
  } while(i<10);

  return 1;
}
```

Save this one as **do_while.c**. What do you think the output will be? Try it out and see if you're right.

Here, **i++;** is used inside of another function. What happens is that 1 is added to **i** *after* the **printf** runs. The first time through the loop, the **printf** statement prints 0 before **i** is incremented to 1. You could also use **++i;** to add 1 *before* **printf** executes. In that case, the first time through the loop, **i** would be incremented to 1 before **printf** printed its value.

For Loops

Loops are often used with a variable, such as **i**, that is incremented each time until a condition isn't met, as we've seen with the previous **while** loop examples. The variable needs an initial value, a test it needs to pass, and a statement that increments it. **for** loops contain all of these things at the very beginning. This reduces the number of lines of code required for a simple loop.

Type the following into CodeRunner below:

```
/* for_loop.c */
#include <stdio.h>

int main(){
  int i;

  for(i=0;i<10;i++){
    printf("bye ");
  }

  return 1;
}
```

Compile and run this example.

Notice the semicolons between the three parts of the **for** loop. The first part initializes **i** to 0, the second part tests **i** to make sure it's less than 10, and the last part increases **i** by one each time through the loop. The variable **i** will start at 0, and increase by 1 each time through the loop until it equals 10.

Observe the following:

```
cold:~$ ./for_loop.exe
bye bye bye bye bye bye bye bye bye bye
```

This simple example of a **for** loop is actually the most common way it's used. In the next lesson you'll see how this loop can make short work when traversing through an array.

Interrupting a Loop

There are two other C statements that allow you to alter the flow of loops: **break** and **continue**. Both are usually used for special cases that aren't covered by the normal conditions of the loop.

Observe the following:

```
var=0;
while(var < 20){
  printf("%i",var);
  var++;
  if(var < 10)
 continue;
  var++;
}
```

The **continue;** statement forces the loop to go to the next iteration and skips any remaining code within the loop. The result is that we'll add 1 to **var** while it's small, and 2 when it gets bigger. Let's say **var** is currently 8. First, "8" is printed out, then **var** is increased to 9. Since 9 is less than 10, the **continue** statement runs. Now we go back to the beginning of the loop. **var** is less than 20 so we print it out, 9. Then we add one to make **var** 10. It's not less than 10 anymore, so we add one again to make it 11. Back to the start again and we print out 11, 13, 15, 17, and 19 before stopping when **var** equals 21. The **break;** command is used the same way, but it stops the loop altogether instead of continuing at the top. We saw **break** working in much the same way as part of a **switch** statement.

That's all for now, see you in the next lesson!

Arrays

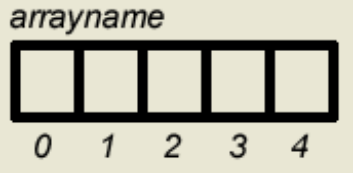
Lesson Objectives

When you complete this lesson, you will be able to:

- use basic arrays.

Array Basics

Let's say you want to remember some information you have stored in your head. You use your memory to recall it later. When you declare a variable, like an integer or character, C allocates a single block of memory to store it in. An *array* is nothing more than a series of consecutive blocks in memory that all store the same type of information. But why would we ever want to do that? The simplest reason is if you need to store a bunch of different numbers. It's easier to remember the name of one array, rather than a bunch of separate variables. I'll use the following type of diagram to illustrate data stored in arrays throughout the rest of this course.



In the diagram, *arrayname* is the name of our array. Each box represents a block of memory and the number below each box is called the *index*. The *index*, or position in the array, always starts from zero.

Declaring Arrays

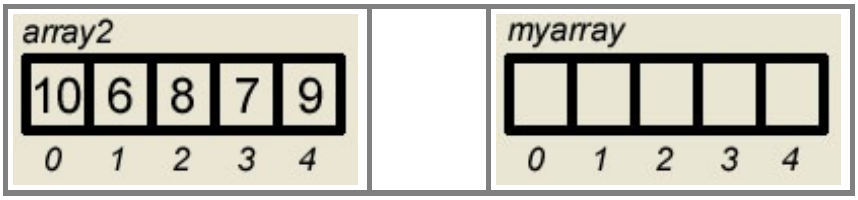
The declaration of an array is very similar to that of a normal variable. The difference is that you either have to declare initial values or a size for the array. Without values or a size, C doesn't know how much memory to allocate.

Examples of array declarations

```
int myarray[5];
int array2[] = {10, 6, 8, 7, 9};
```

Putting a number inside the square brackets, as with the declaration of **myarray**, tells C we want to allocate that number of memory blocks to store integers. The alternative way is to assign values at the same time as the declaration. By giving **array2** five values, we automatically allocate five blocks.

After initialization, the two arrays look like this in memory:



Note Even though we declare the size of the array as 5, the indices still go from 0 to 4.

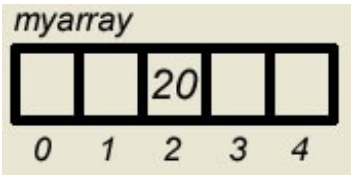
Traversing Arrays

Now that we have our array, we need to be able to change its values. To do this, we have to tell C the name of the array and the index of the position that we want to change.

OBSERVE:

```
myarray[2] = 20;
```

The array now looks like:



We assigned the third element of **myarray** the value of 20 (remember that we count the indexes starting at zero--that's why the index of 2 is actually the third element). Now we'll put this all together in an example.

Type the following into CodeRunner:

```
/* two_arrays.c */
#include <stdio.h>

int main(){
  int myarray[3];
  char array2[] = {'a','b','c','d','e'};

  myarray[0] = 700;
  myarray[1] = 800;
  myarray[2] = 900;

  printf("index 0 of array2 is %c\n",array2[0]);
  printf("index 1 of array2 is %c\n",array2[1]);
  printf("index 2 of array2 is %c\n",array2[2]);
  printf("index 3 of array2 is %c\n",array2[3]);
  printf("index 4 of array2 is %c\n",array2[4]);

  printf("index 0 of myarray is %i\n",myarray[0]);
  printf("index 1 of myarray is %i\n",myarray[1]);
  printf("index 2 of myarray is %i\n",myarray[2]);

  return 1;
}
```

Save it as saving it as **two_arrays.c**, compile it, and run it.

The first thing we're doing here is setting up a three-element integer array named **myarray**. Next, we create a character array named **array2** that contains five characters. Then we give values to the elements in **myarray**. The **printf** statements show what each element of each array contains. Now let's do some looping!

Type the following into CodeRunner:

```
/* two_arrays.c */

#include <stdio.h>

int main(){
  int myarray[3];
  char array2[] = {'a','b','c','d','e'};
  int i;

  myarray[0] = 700;
  myarray[1] = 800;
  myarray[2] = 900;

  /* illustrating the usefulness of loops */
  for(i=0;i<5;i++){
 printf("index %i of array2 is %c\n",i,array2[i]);
  }
  for(i=0;i<3;i++){
 printf("index %i of myarray is %i\n",i,myarray[i]);
  }

  return 1;
}
```

Save, compile, and run your code. The output is exactly the same as from the previous program, but we accomplished it with six lines of code instead of eight. And that's the power of loops! Even if you wanted to print arrays with hundreds of elements, you could still do it with six lines of code. It's not hard to see that a simple loop is a lot easier to write than tons of print statements.

Here, instead of using a **printf** statement for every element of the array, we have a **for** loop. The **for** loop is used to look at each element of **array2**, one at a time. The first time through, **i** equals 0, which can also be used as the index of the array. So the statement "index 0 of array2 is a" is printed out. The next time through, **i** equals 1, and so on.

OBSERVE:

```
  cold:~$ two_arrays.exe
index 0 of array2 is a
index 1 of array2 is b
index 2 of array2 is c
index 3 of array2 is d
index 4 of array2 is e
index 0 of myarray is 700
index 1 of myarray is 800
index 2 of myarray is 900
```

Compiling: Revisited

Lesson Objectives

When you complete this lesson, you will be able to:

- link to a library.
- name your compiled files.
- use **gcc** to report errors.

Compiling Revisited

We covered what happens when you compile a C program. So far you've just been saving and compiling with CodeRunner by clicking on **Compile**. That works great, but you might not always have CodeRunner to do your compiling for you!.

GCC

Let's go back and look at **one_day.c**:

OBSERVE:

```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  printf("I've been programming C for 1 day.");

  return(1);
}
```

If you're not already in CodeRunner's Unix Mode, switch to it now. Let's make sure our file is there.

Type this into CodeRunner:

```
cold:~$ ls one_day*
```

You should see this:

OBSERVE:

```
one_day.c*  one_day.exe*
```

Excellent! To compile the program we'll use a command called **gcc**. GCC stands for **G**NU **C** **C**ompiler.

Type this into CodeRunner:

```
cold:~$ gcc one_day.c
```

If everything worked, you'll see the cold prompt again; you won't see any output from the **gcc** command. If you got any errors, look at the code again to make sure it's correct. Then try to compile it again. If it still doesn't work, look below for some common errors like the ones discussed in lesson 3. If no errors occurred, you should now have a file called **a.out** in your directory.

Type this into CodeRunner below:

```
cold:~$ ls a.out
```

You should see this:

OBSERVE:

```
a.out*
```

gcc gives this name to a compiled, or executable, file. I find it really boring and not very descriptive. Also, **gcc** will overwrite **a.out**, if it already exists, without asking. It's not very useful if it keeps overwriting your previous programs. For these reasons, it is a good idea to give the compiled file a name other than **a.out** to use. We could do this by using **cp** to make a copy with a different name, but there's a better way.

Type the following into CodeRunner:

```
cold:~$ gcc one_day.c -o one_day
```

Notice that we added **-o one_day** to the compile command. The **-o** option tells **gcc** that we want the output file to be named something other than a.out. The new name, **one_day**, follows the **-o** option. Check to make sure you have a file called **one_day** now:

Type the following into CodeRunner:

```
cold:~$ ls
```

You should see this:

OBSERVE:

```
  a.out*  one_day*  one_day.c*  one_day.exe*
```

Now, run your new executable:

Type the following into CodeRunner:

```
cold:~$ ./one_day
```

You should see this:

OBSERVE:

```
I've been programming C for 1 day.
```

Great! Now let's put an error in the C code again so you can see how **gcc** reports errors. Try leaving off the semicolon after the **printf** function. Do this by re-opening **one_day.c** and removing the semicolon:

Type the following into CodeRunner:

```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  /* printf("I've been programming C for 1 day."); */
  printf("I've been programming C for 1 day.")

  return(1);
}
```

We simply commented out the old printf statement and put in a new one that's missing a semicolon. Now save your file and compile your program again.

Type the following into CodeRunner:

```
cold:~$ gcc one_day.c -o one_day
```

You should see something like this:

OBSERVE:

```
one_day.c: In function `main':
one_day.c:11: error: syntax error before "return"
```

Instead of creating the executable file, gcc gave us an error message. This is the same error message you saw earlier.

Change **one_day.c** back, removing the code shown in **red** below so it will compile correctly:

Edit one_day.c as shown:

```
/* one_day.c  */

#include <stdio.h>

int main(){

  /* print statement */
  /* printf("I've been programming C for 1 day."); */
  printf("I've been programming C for 1 day.")

  return(1);
}
```

The -lm flag

There is a special case when compiling programs using the **math.h** library. Type the following C program in the CodeRunner editor:

Type the following into CodeRunner below:

```
/* pow.c */

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

int main(){
  float a=5.0, b=10.0, c;

  c = pow(a,b);

  printf("%f\n",c);

  return 1;
}
```

(We covered the **pow()** function in Lesson 5.) Save this file as **pow.c**. Let's try to compile it with **gcc**.

Type the following into CodeRunner below:

```
cold:~$ gcc pow.c -o pow
/tmp/ccJextG3.o: In function `main':
/tmp/ccJextG3.o(.text+0x27): undefined reference to `pow'
collect2: ld returned 1 exit status
```

There's nothing wrong with the code. Why is there an undefined reference to **pow**? Hmm...

This is the special case I was talking about. When using **math.h**, you must use the **-lm** flag with gcc. Here's how:

Type the following into CodeRunner:

```
cold:~$ gcc -lm pow.c -o pow
cold:~$ ./pow
9765625.000000
```

Excellent. **-l** means to link to a library, and **m** indicates the math library, so the **-lm** flag simply indicates the use of **math.h**.

Next up: Functions!

Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- write your own functions.
- separate the .c and .h files.

What's a Function?

A function is a self-contained segment of code that performs operations on some input and returns the result. It might sound complicated, but you've been using one this whole time. Let's look again at the very first program we wrote:

OBSERVE:

```
/* one_day.c */
#include <stdio.h>

int main(){

  /* print statement */
  printf("I\'ve been programming C for 1 day.");

  return(1);
}
```

main() is a function. It's a special function, in that C always starts executing withit, and as a result, it must always exist. Notice that **main()** is declared with a variable type as well. This variable type tells C what kind of variable the function will return. This means that, when we call a function, we can get back a result. The result can be anything from the result of a math function to simply an indication that the function ran okay. In this case, the **return** function returns a 1 (TRUE). This is the same thing as saying that it exited successfully.

All functions require parentheses () after the name, where you specify what type of input, if any, the function can receive. The input can be any information that the function needs to manipulate to get a result. In this case, there isn't any input so we don't put anything in the parentheses. We'll look at this more when we write our own function. A function definition also must include braces {} around the function's contents.

Why Functions?

There are three good reasons why you'd want to write your own functions. First, if you need to do the same thing multiple times throughout your code, but with different variables, functions save a lot of typing and reduce clutter. Second, the use of outside functions clean up **main()** so the flow of the program is more easily understood when looking back. Without other functions, **main()** could get really long. Lastly, functions can be re-used in future programs. An excellent example of this is the **pow()** function. We could do the same thing with a loop, but it's much easier to use **pow()**. To prove this, let's write our own version of **pow()** that we'll call **power**. Save this in a file called **exponentiator.c**.

Type the following into CodeRunner below:

```
/* exponentiator.c */

#include <stdio.h>

int main(){
  int var1, var2;
  double result;
  double power(int x, int y);

  /* get input from the user */
  printf("Please enter two integers separated by a space: ");
  scanf("%i %i",&var1,&var2);

  /* call our function and store the returned value in a variable */
  result = power(var1,var2);

  printf("result = %f\n",result);

}

double power(int x, int y){
  int i;
  double answer=1;

  /* multiply by x, as many times as there are y */
  for(i=0;i<y;i++){
    answer = answer * x;
  }
  return answer;
}
```

Save this example as **exponentiator.c**, then compile and run it to see it in action.

In order to use the **power()** function outside of **main()**, we have to declare it like we would a variable. Unlike variables, we don't do this to allocate memory, but because C needs to know what type of data the function expects as input and what type it will return. In this case, we want the **power()** function to return a double. Also, inside the parentheses we define what type of input the function is expecting. (The actual variable names aren't required at this point, only the types; but I include them to keep things straight.)

Next, we call the **power()** function to get a result. Notice that the **result** variable is the same type as we expect the function to return. In the parentheses, we pass **var1** and **var2**, which are the integer inputs that **power()** is expecting.

Finally, we have the **power()** function itself. The beginning of the function looks exactly like the declaration of it from inside of **main()**. The difference is that now the variable names (x and y in this case) are required. The variables x and y are allocated for use in the **power()** function and they store the passed values of var1 and var2 respectively. The next couple of lines declare a few variables we need to make the calculation. Next, the **for** loop does the math, and then we **return** the answer. That answer is stored into the **result** variable from **main()**.

Output from exponentiator.exe

```
Please enter two integers separated by a space: 2 3
result = 8.000000
```

Our **power()** function is a little less powerful than the real **pow()** because we're only dealing with integers, but it gets the idea across.

Note The reason I chose a double is, even though the result of our math will be an integer, the number can be rather large depending on the input. A double can store a much bigger number than an int, float, or even a long.

Separate .c and .h files

Let's imagine we have a lot of extra functions that we've written. The **exponentiator.c** file is getting a little big to work with and we'd like to break it up into smaller, more manageable, files.

We need to create two new files, one named **power.c** and one nameed **power.h**.

Type the following into CodeRunner:

```
/* power.c */

double power(int x, int y){
  int i;
  double answer=1;

  /* multiply by x, as many times as there are y */
  for(i=0;i<y;i++){
    answer = answer * x;
  }
  return answer;
}
```

Save this as **power.c**.

Type the following into CodeRunner below:

```
/* power.h */
double power(int x, int y);
```

We'll save this one as **power.h**.

Next remove the code shown in **red** from the **exponentiator.c** file and add the line in **blue**.

Code to Edit: exponentiator.c

```
/* exponentiator.c */

#include <stdio.h>
#include <power.h>

int main(){
  int var1, var2;
  double result;
  double power(int x, int y);

  /* get input from the user */
  printf("Please enter two integers separated by a space: ");
  scanf("%i %i",&var1,&var2);

  /* call our function and store the returned value in a variable */
  result = power(var1,var2);

  printf("result = %f",result);

}

double power(int x, int y){
  int i;
  double answer=1;

  /* multiply by x, as many times as there are y */
  for(i=0;i<y;i++){
    answer = answer * x;
  }
  return answer;
}
```

The new file looks like this:

exponentiator.c after changes

```
/* exponentiator.c */

#include <stdio.h>
#include <power.h>

int main(){
  int var1, var2;
  double result;

  /* get input from the user */
  printf("Please enter two integers separated by a space: ");
  scanf("%i %i",&var1,&var2);

  /* call our function and store the returned value in a variable */
  result = power(var1,var2);

  printf("result = %f",result);

}
```

Save this as **exponentiator.c**.

Notice how we replaced the declaration of the **power** function in **main()** by the inclusion of the **power.h** header file. The functions are declared in the header file. Instead of declaring all the functions separately, we only need to include the one header file. The actual function we took from **exponentiator.c** is in **power.c**. So now we have three files: **exponentiator.c**, **power.c**, and **power.h**. We have to do something special to get them all to compile into one program. Use the following command to compile it:

Type the following into CodeRunner below:

```
cold:~$ gcc -I./  exponentiator.c power.c -o power
```

The **-I./** option tells the compiler to look in the current directory as well as the default system places for header files. Without this, we'd get a message that gcc can't find power.h:

Observe the following:

```
main.c:4: power.h: No such file or directory
```

The **stdio.h** file that we've been including this whole time declares many standard functions such as **printf** and **scanf**. There are many more header files, some of which we'll look at as we progress. See you in the next course!

Random Numbers

Lesson Objectives

When you complete this lesson, you will be able to:

- generate some random numbers.
- integrate random fractions.

Random Numbers

At some point you may find yourself wanting to generate some random numbers. Perhaps you want to simulate the rolling of a six-sided die for a game.

To generate a random number, we'll use the **rand()** function. This function is declared in **stdlib.h**, so we need to include that header file in programs where we want to use **rand()**DARKGREEN>. The **rand()** function returns a number between 0 and *RAND_MAX*. RAND_MAX is a constant that is defined separately for each system. On a 32-bit computer, it's usually over 2.1 billion.

Usually we don't need numbers that large. We can use the modulus (%) to change the range of our random number. For example, **rand() % 6** would produce a number between 0 and 5. To simulate a six-sided die, all we need to do is add one to that result. Let's make a small example.

Type the following into CodeRunner below:

```
/* die_roller.c */

#include <stdio.h>
#include <stdlib.h>

int main(){
  int result;

  result = rand() % 6 + 1;

  printf("%i\n",result);

  return 1;
}
```

Save and compile this program as **die_roller.c**. Run it several times. You should notice something a little strange--you're getting the same number over and over. What? I thought it was supposed to be random. Well, it is, but it's the *same* random number every time! You have to remember that, in the world of computers, nothing is ever truly random. What **rand()** actually does is use some strange math function (that doesn't really matter to us) to generate numbers that appear to be random. The problem is that it always starts at the same place and gives the same sequence of "random" numbers. However, there is a solution.

srand

To make **rand()** start from another place, use another function, **srand()**. **srand()** takes a value called a *seed*. But if we give **rand()** a constant integer as a seed, the only result is that we'll get a different sequence of the same random numbers. So we need a seed that will be different every time we run the program.

Where can we get a number that constantly, reliably, changes? The computer's clock! Your computer keeps track of the time of day in seconds, so if we can use that number as our seed, we'll be "golden." We need to include the **time.h** header file and use the **time()** function. Here's how:

Type the following into CodeRunner:

```
/* die_roller.c */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
  int result,i;

  srand(time(NULL));

  for(i=0;i<5;i++){
    result = rand() % 6 + 1;

    printf("%i\n",result);
  }

  return 1;
}
```

Compile and run this one after saving it as **die_roller.c**.

We only need to use **srand** once at the beginning of the program. Then we can generate as many random numbers as we want.

Note

Theoretically, we could repeat our "random" number sequence by picking a time of day, and trying to run our program at exactly the same time on another day.

Also, if you run the program twice in a row fast enough (if a second hasn't gone by you'll get the same seed), you'll still get the same sequence of numbers, but for the most part we can ignore these issues.

Fractions

You may have noticed that the only thing we're getting from the random number generator is integers. However, if you want to do anything involving percentages, or perhaps a statistics simulation, you'll need random fractions. The best way I've found to do this is to take full advantage of the **RAND_MAX** constant that already exists.

Code to type: random_fract.c

```
/* random_fract.c */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(){
  float result;

  srand(time(NULL));

  result = rand() / (float) RAND_MAX;
  /* gotta make sure we do the math with all floats or it won't work */

  printf("%f\n",result);

  return 1;
}
```

Save and compile **random_fract.c**, and run it a few times.

By dividing a random number that goes from 0 to RAND_MAX, by RAND_MAX, we get a fraction from 0 to 1. Again, it's not truly random because there's only a certain amount of accuracy you can get with your fractions. However, RAND_MAX is so large that you are actually limited by the six decimal place accuracy of the floating point variable.

Pointers

Lesson Objectives

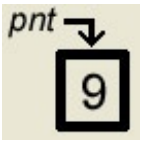
When you complete this lesson, you will be able to:

- store the location of a variable rather than its actual value.
- use pointers to return multiple values.
- use pointers with arrays.

What's a Pointer?

Sometimes in C, it becomes necessary to store the location of a variable rather than its actual value. "Huh?" you might say. Let me explain. Normally, when you declare a variable, say an integer, C allocates a specific block of memory to store the variable. The variable itself represents the value stored in that memory block. Here's a picture showing an integer variable named **num**.

Here, **num** represents the number 12. Now suppose we have a pointer to a memory block. We'll represent that as follows:



pnt is a pointer to the memory location where the number 9 is stored. **pnt** does not store the number 9, it stores a memory address, the *location* of the number 9. However, we can get at the data stored in memory by using an asterisk. The asterisk is called the dereference operator. So, in our program, ***pnt** would refer to the actual number 9. Pointers are weird until you get used to them, so don't worry if this makes absolutely no sense right now. That's perfectly normal.

Using Pointers

A pointer is declared almost the same way as a regular variable because C needs to know the type of data to which the pointer is pointing. We use the dereferencing operator (*) in the declaration as well, to indicate it's a pointer and not a normal variable. Here's a small but important example:

Type the following into CodeRunner below:

```
/* pointer.c */

#include <stdio.h>

int main(){
  int num;
  int *pnt;

  pnt = &num;

  num = 1;
  printf("num is %i\n",num);

  *pnt = 2;
  printf("num is %i\n",num);
  printf("*pnt is %i\n",*pnt);

  printf("pnt is %p\n",pnt);

}
```

Save it as **pointer.c**, compile, and run. You should get output like this:

Output from pointer.exe

```
num is 1
num is 2
*pnt is 2
pnt is 0xbffffa74
```

Declaring a pointer doesn't set up the memory location it's going to point to. As a result, we have to assign an address to the pointer. On the line where we assign **num** to **pnt**, we use the address operator (remember the address operator from when we learned **scanf**? **&**) to get the address of **num** and store it in **pnt**. From this point on, **pnt** points to the same memory block that **num** refers to.

Now the dereferencing operator allows us to use ***pnt** as essentially the same thing as **num**. When assigning a value of 2 to ***pnt**, we change the value of **num** as well. Changing one changes the other.

The last line prints out the memory address that's stored in **pnt**. You don't need to know how to read this address--that's what your computer is for. I just wanted to show you that **pnt**, as a variable, stores an address.

It might be useful to think of your mailbox as a variable, and your street address as a pointer. You can either walk out to your mailbox and put something in it directly, or you could mail it to your address. Either way, it ends up in the same place.

Notice how the second time we printed out **num**, the value was 2.

Note If you're trying to use a pointer as a regular variable and you're getting segmentation faults, try putting parentheses around the pointer and its dereferencer. For example, ***j++;** will not work correctly because **++** has a higher precedence than *. However, **(*j)++;** does work.

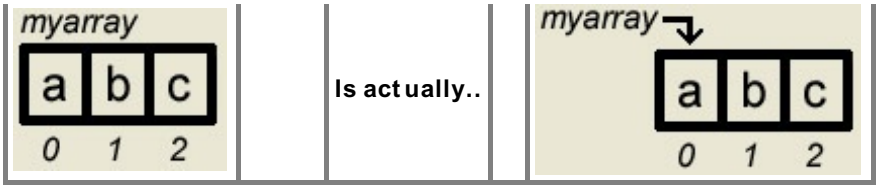
Pointers to Arrays

You may think that pointers to arrays would be trickier, but in fact, they're much easier. When you declare an array...

OBSERVE:

```
char myarray[] = {'a','b','c'};
```

...the name, **myarray**, is automatically a pointer to the first element of the array.



Since the array is made up of consecutive blocks of memory, the square brackets ([ ]) act as the dereferencer. The number inside the brackets just indicates which block of memory is being referred to. To prove this, try the following:

Type the following into CodeRunner:

```
/* array_pointer.c */

#include <stdio.h>

int main(){
  int myarray[] = {1,2,3,4,5,6,7,8,9,10};

  printf("%i\n",*myarray);
  printf("%i\n",myarray[0]);
}
```

Save it as **array_pointer.c**, and compile and test the code to make sure it works. Both of the print statements will give the same output. Both ***myarray** and **myarray[0]** refer to the first element of the array.

But Why?

Pointers seem really complicated. Why on earth would we ever want to use them? Pointers are used to get around an important problem with functions. As you know, a function can only return one value. That's no good. What happens when you need a function that returns more than one value? The answer lies with pointers. You can set up the variables you need and then pass only the addresses of those variables to the function.

Let's see how this all works.

Type the following into CodeRunner:

```
/* double_adder.c */

#include <stdio.h>

int main(){
  int number1=10, number2=20;
  int addfive(int *num1, int num2);

  printf("Before adding 5 we have %i and %i\n",number1, number2);

  number2 = addfive(&number1, number2);

  printf("After adding 5 to both we get %i and %i\n",number1, number2);
}

int addfive(int *num1, int num2){
  *num1 += 5;
  num2 += 5;

  return num2;
}
```

Examine this code closely before compiling it. See if you can figure out what's going on before reading my explanation. Don't forget to save it as **double_adder.c**.

Before we use the function we have to declare it. The **addfive()** function is going to take one pointer and one integer as input. What pointer will we give it if I didn't declare one? A pointer simply stores a memory address, right? So when we pass a pointer to our function, we can just use the address operator and pass the address of the variable directly.

We call the function and give it our two inputs. The **number1** variable will be changed by using a pointer to its memory location and **number2** will be assigned a new value that **addfive** returns.

Now comes the actual **addfive()** function. It takes an address and declares the pointer ***num1** for it, while **num2** stores the value of **number2**. Five is added to both of them and **num2** is returned. **number1** was changed at its memory location by using the pointer.

OBSERVE:

```
cold:~$ ./double_adder.exe
Before adding 5 we have 10 and 20
After adding 5 to both we get 15 and 25
```

We can do the same thing as above with two pointers instead of one while having the function return nothing. Also, the program turns out to be shorter by using pointers instead of relying on it to return a number.

Observe the following:

```
#include <stdio.h>

int main(){
  int number1=10, number2=20;
  void addfive(int *num1, int *num2);

  printf("Before adding 5 we have %i and %i\n",number1, number2);

  addfive(&number1, &number2);

  printf("After adding 5 to both we get %i and %i\n",number1, number2);
}

void addfive(int *num1, int *num2){
  *num1 += 5;
  *num2 += 5;
}
```

Giant Pointer Example

This exercise is just a larger example of using pointers with functions. It's an important concept that deserves a little extra time. This also helps to tie together a lot of the stuff you've learned so far.

Type the following into CodeRunner:

```
/* averager.c */

#include <stdio.h>

int main(){
  int myarray[] = {1,2,3,4,5,6,7,8,9,10};
  int size=10;
  float average;
  int getavg(int ary[], float *avg, int sz);

  if(getavg(myarray,&average,size)){
     printf("average is %f",average);
  }else{
     printf("average getting function was unsuccessful");
  }

}

int getavg(int ary[], float *avg, int sz){
  int i=0,total=0;

  if(sz == 0){
     /* array is empty, return failure */
     return 0;
  }

  while(i < sz){
    total += ary[i];
    i++;
  }

  *avg = (float) total / (float) sz;
  return 1;
}
```

Save as **averager.c**, and compile and run it. The result should be **5.500000**.

Let me point out a few things about the code. The reason we want the **average** variable to be a floating point number is purely based on the math behind getting an average. Even though we'll be adding up integers and dividing by an integer, the result is likely to be a fraction.

In **int getavg(int ary[], float *avg, int sz)**, we declare the function we'll be using and the inputs it expects. The **getavg()** function will return an integer value (success 1 or failure 0). As input, we need to send it the address of an array, a pointer to the variable that will store our average (essentially the address of our **average** variable), and an integer giving the size of the array. When we actually use the function we don't need to use a separate pointer variable. Since a pointer simply stores the address of a variable, we can just pass that address to **getavg()**.

The **if** statement checks to see if the **getavg()** function returns successfully. As a result, the **getavg()** function executes inside of the **if** statement. This might be strange the first time you see it. Think of the **getavg()** function just like you would any other conditional expression in an **if** statement. For example, if we had **varname < 1**. We're testing for TRUE (1) or FALSE (0). However, we can also just return a numerical value of 1.

Back to the **getavg()** function. We pass the name of our array (its address), and we use the address operator (&) to pass the address of the **average** variable, and then we simply give it an integer storing the size of the array.

Inside the **getavg()** function, the first thing we want to do is check and see if the array is empty or not. If it's empty, we're going to return zero indicating that the function failed to determine an average value.

Inside the **while** loop, you can see how we use the square brackets to get each value of the array and add up the total.

The last line stores the result in our **average** variable by using the pointer. The pointer is pointing to the memory address of **average** so when we change the pointer, the real variable changes as well.

But wait a second, what's up with putting **(float)** in front of the variables? That's got nothing to do with pointers. It has to do with how C does math. **total** and **sz** are both integers, so C will try to return an integer as the answer. Putting **(float)** in front of a variable tells C to treat it like a floating point number for the rest of that statement. Treating a variable as another type is called *casting*.

See you in the next lesson!

Character Arrays and Strings

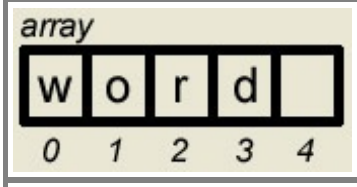
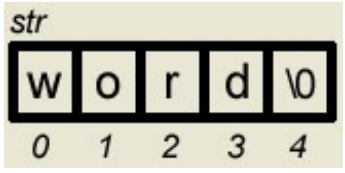
Lesson Objectives

When you complete this lesson, you will be able to:

- declare strings.
- use strings with a NULL character.
- use functions to output from strings.

Strings

I mentioned early on that strings and words couldn't be stored in a normal variable. However, with an array of characters, we have that power. The difference between just an array of characters and a string in C is the addition of a NULL character (**\0**) at the end. NULL indicates where the end of the string is located.

		This is just a character array
		This is a string

Declaring Strings

Strings can be declared using the same method used to specify values in a character array or we can use a shortcut by including a string in double quotes. When we use double quotes, the NULL character is automatically added.

OBSERVE:

```
char string1[]={'y','o','\0'};
char string2[]="hello";
char string3[6];
```

The first string ends up being a three-character array with the string "yo" stored inside (**\0** means NULL). Because it uses double quotes, **string2** automatically includes the **\0** at the end. **string3** is essentially just declared as an array of characters. In order for it to be a string, we have to remember to include the **\0** at the end.

Using Strings

Let's start using our strings!

Type the following into CodeRunner below:

```
/* string_printer.c */

#include <stdio.h>
#include <string.h>

int main(){
  char string1[]={'y','o','\0'};
  char string2[]="hello";
  char string3[6];

  strcpy(string3,"there");

  printf("%s ",string1);
  printf("%s ",string2);
  printf("%s\n",string3);

  return 1;
}
```

Save it as **string_printer.c**, and compile and run it to see it in action.

Output from string_printer.exe

```
yo hello there
```

When using strings, the first thing we need to do is include an additional header file named **string.h**. This allows us to use many of the functions designed to work with strings. The **strcpy()** statement is a way to assign a value to a string. Unlike with normal variables, we can only use the assignment operator (**=**) when declaring a string. To set a value later, we must use something like **strcpy()**. **strcpy()** can also be used to copy one string into another. In this case, we set **string3** equal to "there." Remember to keep the size of the string in mind so that it does not exceed the size of the array when a NULL character is added to the end.

It's simple to print out strings using a **%s**. That way we don't have to worry about where the end of the string is.

Note **string.h** isn't necessary on all platforms, but it's better to include it than try to remember which ones it's needed for.

For the rest of this lesson I'll briefly cover some of the functions most commonly used with strings. **Compile and test** all of the following string examples to get a feel for them. Play around a bit and see what you can come up with.

strcat

The **strcat()** function, as with all of the functions I'll go over that start with "str," requires that you include **string.h**.

Type the following into CodeRunner:

```
/* concatenator.c */

#include <stdio.h>
#include <string.h>

int main(){
  char first[40] = "The first string ";
  char second[] = "and the second string.\n";

  strcat(first,second);
  printf("%s",first);

  return 1;
}
```

The **strcat()** function appends the second string to the end of the first string. (They don't have to be called **first** and

second, but it makes it easier to illustrate.) What about the NULL character at the end of the first string? Well, luckily for us, **strcat()** handles that by putting the first character of the second string where the NULL used to be. The result is this:

Output from concatenator.exe

```
The first string and the second string.
```

We could also put **strcat()** directly in the print statement:

Observe the following:

```
printf("%s",strcat(first,second));
```

This is because **strcat()** returns a pointer to the first string, which is exactly the same thing that the name **first** refers to.

strcmp

Use the **strcmp()** function to compare two strings. It returns an integer value based on the result of the comparison. It returns less than 0 if the first string is less than the second, 0 if they are equal, and greater than 0 if the first is greater than the second. Wait a second. Less than and greater than? I thought we were talking about characters here. We are, but remember, characters in C are represented by numbers as well!

Type the following into CodeRunner below:

```
/* comparator.c */

#include <stdio.h>
#include <string.h>

int main(){
  char one[] = "abcd";
  char two[] = "abcz";
  void compare(char *str1, char *str2);

  compare(one, one);
  compare(one, two);
  compare(two, one);

  return 1;
}

void compare(char *str1, char *str2){
  int value;
  value = strcmp(str1,str2);

  if(value < 0){
    printf("%s is less than %s\n",str1,str2);
  }else if(value == 0)
    printf("%s is equal to %s\n",str1,str2);
  else if(value > 0){
    printf("%s is greater than %s\n",str1,str2);
  }

}
```

Save this one as **comparator.c**. Try it out to get a feel for what it does. Here I went an extra step and wrote another function to show all the different outcomes. The **void** type is used when we don't need the function to return anything.

strlen

The **strlen()** function is used to find the length of a string. It's useful when you need to traverse a string like an array, and you don't know how long it is. It reads through the string, counting characters, until it sees the NULL character at

the end. As a result, it's very important that NULL character is there.

Observe the following:

```
/* how_long.c */
#include <stdio.h>
#include <string.h>

int main(){
  char str[] = "Mary had a little lamb.";
  int length;

  length = strlen(str);

  printf("The string is %i characters long.\n",length);

  return 1;
}
```

The NULL character is not counted in the length of the string. The result looks like this:

Observe the following:

```
The string is 23 characters long.
```

sprintf

The **sprintf()** function works almost exactly the same as **printf()**, except that you have to give it the name of your string first, and it saves its results in a string instead of printing to standard output (STDOUT).

Type the following into CodeRunner below:

```
/* string_maker.c */

#include <stdio.h>
#include <string.h>

int main(){
  char str[50];
  int bob = 3;

  sprintf(str,"This is my new string and bob equals %i\n",bob);

  printf("%s",str);
  return 1;
}
```

Save this as **string_maker.c** before compiling and running it.

Note When working with strings, be very careful not to exceed the length of the string (one less than the size of the array). Writing into memory past the end of a string will commonly cause a lot of really weird errors.

sscanf

Just like we have **sprintf()**, we've also got **sscanf()**. This time, instead of reading from standard input (STDIN) it reads from, you guessed it, a string!

Type the following into CodeRunner below:

```
/* string_scanner.c */

#include <stdio.h>
#include <string.h>

int main(){
  char str[] = "a b c";
  char one,two,three;

  sscanf(str,"%c %c %c",&one,&two,&three);

  printf("the characters are: %c %c %c\n",three,two,one);
  return 1;
}
```

Save as **string_scanner.c**, compile, and run. As you can see, the format is exactly the same as in **scanf()**, except that you have to tell it the name of your input string first.

gets

The **gets()** function reads from STDIN, just like **scanf()**. However, the difference is that **gets()** keeps reading characters until a newline or end of file (EOF). It then stores the characters into a string. **gets()** will automatically append the NULL character to the end of the string as well.

Whenever you compile a program that uses **gets()**, you'll get an error message saying that using **gets()** is dangerous. Ignore this warning for now, but if you want it to go away you'll need to read the link below about programming strings more securely.

Note The EOF (end of file) character is found at the end of a file (as we'll see later) or an input stream.

Type the following into CodeRunner below:

```
/* string_getter.c */

#include <stdio.h>
#include <string.h>

int main(){
  char str[100];

  gets(str);

  printf("%s",str);

  return 1;
}
```

Save it as **string_getter.c** and try it out to see it in action.

Secure Strings

Functions like `strcpy`, `strcat`, `sprintf`, and `gets` don't check the size of the string they're modifying. It's really easy to write past the end of your string. When this happens, your program might crash, or it might start executing code you didn't write. This problem is known as a *buffer overflow*, and is the reason many programs and operating systems have security problems.

The solution to this problem is to be extremely careful with the size of the strings. Lucky for us, there are four replacement functions that help do this: `strncpy`, `strncat`, `snprintf`, and `fgets`.

`strncpy` and `strncat` work very similarly. They require a number as an extra input. Only the first few characters of the second string, up to that number, are considered. The problem, is that you may loose the NULL character at the end of

the first string, so you have to put one back.

Type the following into CodeRunner below:

```
/* secure.c */
#include <stdio.h>
#include <string.h>

int main(){
  char str[100];
  char str2[] = "hello there";
  int a=5;

  strncpy(str,str2,a);
  str[a] = '\0';

  printf("%s\n",str);

  return 1;
}
```

The result of the above sample follows. As you can see, only the first 5 characters of `str2` are in `str`. Since we're copying the first 5 characters, we want a NULL character at the sixth spot which just happens to be the 5th index.

OBSERVE:

```
hello
```

`strncmp` works pretty close to the same way, except that the number limits the number of characters of the first string that are compared to the second string.

The function that is the most different from it's counterpart is `fgets`. First, `fgets` was designed to read from files as well as STDIN, so we have to specify that we want to use standard input as well as the maximum number of characters to read. It will read in up to one less than the number you specify (it saves room for the NULL character it adds) or until it reaches a newline. Unlike `gets` that doesn't keep a trailing newline as part of the string, `fgets` does. Another difference, is that `gets` also considers an EOF the end of a string, while `fgets` considers an EOF to be an error. If an error occurs, `fgets` returns a NULL pointer.

Here's an example that reads in the first ten characters:

Type the following into CodeRunner:

```
#include <stdio.h>

int main(){

  char str[20];
  int i=10;

  if(fgets(str,i,stdin) == NULL){
    printf("read failed.\n");
  }

  printf("%s\n",str);

  return 1;
}
```

Try running this program, typing in a something like "Now is the time for all good C programmers."

OBSERVE:

```
Now is the time for all good C programmers
Now is th
cold:~$
```

The program only read in the first ten characters: **Now is th**.

These functions alone are not enough to ensure your program is safe from buffer overflows, but using them is definately a good idea.

We learned a lot about strings in this lesson. In the next lesson we will discuss structures. See you then!

Structures

Lesson Objectives

When you complete this lesson, you will be able to:

- define your own data types.
- access each element of a structure.
- create a pointer to a structure.

What is a Structure?

A structure allows you to define your own data types. Usually it's a collection of several other types and by declaring your new type, all of the other ones are automatically declared for you. Structures are sort of like arrays, but all of the elements don't have to be the same type. It's easiest to do an example. First I'll describe three common ways of defining a struct, and then I'll show how to use them.

OBSERVE:

```
#include <stdio.h>

struct house{
  char owner[30];
  float size;
  float stories;
  int bedrooms;
  float bathrooms;
};

int main(){
  struct house bighouse;
  struct house smallhouse;

  return 1;
}
```

The first section defines the **contents** of a structure called **house**. Now, every house structure we use will contain all of those elements. In **main**, we can then use **struct house** to define a new house with all of its elements. As you can imagine, this is a lot faster than declaring variables for each individual element for every house.

OBSERVE:

```
#include <stdio.h>

struct house{
  char owner[30];
  float size;
  float stories;
  int bedrooms;
  float bathrooms;
};

typedef struct house House;

int main(){
  struct house bighouse;
  struct house smallhouse;

  return 1;
}
```

This additional piece of code illustrates the use of **typedef**. **typedef** simply allows you to use a single name for

variable declaration instead of the full **struct house**.

OBSERVE:

```
#include <stdio.h>

typedef struct house{
  char owner[30];
  float size;
  float stories;
  int bedrooms;
  float bathrooms;
} House;

int main(){
  House bighouse;
  House smallhouse;
  return 1;
}
```

Here I've combined the **structure definition** and the **typedef** into one statement. This is how you'll usually see it. Now we have two houses declared--how do we use them?

Using Structures

To access each element of a structure, you give the structure's name, a period, and then the element name. The period is called the *dot operator*. For example, if we wanted to set the **bedrooms** element of the **bighouse** structure, we'd do the following:

Observe the following:

```
bighouse.bedrooms = 4;
```

A structure and element combination acts just like a variable of the type specified in the structure definition; in this case, an int.

Type the following into CodeRunner:

```
/* house_info.c */

#include <stdio.h>
#include <string.h>

typedef struct house{
  char owner[30];
  float size;
  float stories;
  int bedrooms;
  float bathrooms;
} House;

int main(){
  House bighouse;

  bighouse.size = 4;
  bighouse.stories = 2;
  strcpy(bighouse.owner,"John Doe");

  printf("size is %.2f square feet.\n",bighouse.size);
  printf("There are %.1f stories.\n",bighouse.stories);
  printf("%s owns it.\n",bighouse.owner);

  return 1;
}
```

Save as **house_info.c**. Compile and run it to get the following output:

Output from house_info.exe:

```
size is 4.00 square feet.
There are 2.0 stories.
John Doe owns it.
```

Here we're only using three elements of the structure: **size**, **stories**, and **owner**. Note the use of **strcpy** to assign a value for the owner of the house. I've formatted the output to make it look a little nicer. Try assigning values for the other elements and printing those out as well.

Pointers to Structures

Just like with other variables, you can create a pointer to a structure. The only difference is how you access them.

Type the following into CodeRunner:

```
/* house_info.c */

#include <stdio.h>

typedef struct house{
  char owner[30];
  float size;
  float stories;
  int bedrooms;
  float bathrooms;
} House;

int main(){
  House bighouse;
  House *pnt2house;

  pnt2house = &bighouse;

  (*pnt2house).size = 4;
  pnt2house->stories = 2;
  strcpy(pnt2house->owner,"John Doe");

  printf("size is %.2f square feet.\n",pnt2house->size);
  printf("There are %.1f stories.\n",pnt2house->stories);
  printf("%s owns it.\n",pnt2house->owner);

  return 1;
}
```

After compiling, you should get the same result as before.

We declare the pointer and assign it an address the same way as with other variable types. The trick is that we have to dereference the pointer before we can access its elements. The dot operator (**.**) has a higher priority than the dereferencer (*****). So we have to put parentheses around the dereferencer and the pointer name. That's kind of annoying, but luckily, C provides a shortcut: the *arrow operator*. It consists of a minus sign and a greater-than sign (**->**). The arrow operator does the same thing as dereferencing the pointer and using the dot operator, but it's just a little easier to read and use.

Now we'll use all of this to do a big example. Don't let the size of this scare you; it's really not that bad. Just look at it in pieces.

Type the following into CodeRunner:

```
/* house_info.c */

#include <stdio.h>
#include <string.h>

/* Define our house structure */
typedef struct house{
  char owner[50];
  float size;
  float stories;
  int bedrooms;
  float bathrooms;
} House;

int main(){
  House house1;
  House house2;
  int benefits1=0,benefits2=0;
  void GetHouseInfo(House *tmphouse);

  /* Get the data for each of the two houses */
  printf("Now enter the information for the first house\n");
  GetHouseInfo(&house1);
  printf("Now enter the information for the second house\n");
  GetHouseInfo(&house2);

  /* Compare all of the house elements and adjust the benefits accordingly */
  if(house1.size > house2.size){
    benefits1++;
  }else if(house2.size > house1.size){
    benefits2++;
  }

  if(house1.stories > house2.stories){
    benefits1++;
  }else if(house2.stories > house1.stories){
    benefits2++;
  }

  if(house1.bedrooms > house2.bedrooms){
    benefits1++;
  }else if(house2.bedrooms > house1.bedrooms){
    benefits2++;
  }

  if(house1.bathrooms > house2.bathrooms){
    benefits1++;
  }else if(house2.bathrooms > house1.bathrooms){
    benefits2++;
  }

  /* Determine which house is better */
  if(benefits1 > benefits2){
    printf("The house owned by %s is better.\n",house1.owner);
  }else if(benefits2 > benefits1){
    printf("The house owned by %s is better.\n",house2.owner);
  }else{
    printf("Both houses are equally good.\n");
  }

  return 1;
}

void GetHouseInfo(House *tmphouse){
  printf("Enter the owner's name: ");
  gets(tmphouse->owner);
```

```
  /* This gets rid of stray newlines that could screw up "gets" */
  while(strlen(tmphouse->owner) == 0){
    gets(tmphouse->owner);
  }

  printf("Enter the number of square feet: ");
  scanf("%f",&tmphouse->size);

  printf("Enter the number of stories: ");
  scanf(" %f",&tmphouse->stories);

  printf("Enter the number of bedrooms: ");
  scanf(" %i",&tmphouse->bedrooms);

  printf("Enter the number of bathrooms: ");
  scanf(" %f",&tmphouse->bathrooms);

  printf("\n\n");

}
```

Save it, compile it (again, ignore the warning about **gets** for now), and run it.

We need to include **string.h** because we use **strlen()** later.

In the declaration of the **GetHouseInfo()** function, you can see how our defined **House** data type is used just like **int**, **char**, or anything else.

We send the address of each of our house structures to the **GetHouseInfo()** function. This way, by using a pointer to the house structure, we only have to write the code to query the user one time.

At the start of **GetHouseInfo()**, we give the name of the pointer to a house structure as **tmphouse**.

The **while** loop inside **GetHouseInfo()** helps get rid of stray newlines from the input buffer. This works because **gets()** stops on a newlines. If the first thing encountered is a newline, the size of the input string will be zero.

Reading/Writing Files

Lesson Objectives

When you complete this lesson, you will be able to:

- open a file and set up a pointer to that file from within C.
- print to files.
- use recursion to have a function call itself.

Opening a File

In order to read or write to a file, you first have to *open* it. By opening a file you are setting up a pointer to that file from within C. To do this, C provides us with a data type for files known as a file structure.

Type the following into CodeRunner below:

```
/* opener.c */

#include <stdio.h>

int main(){
  FILE *myfile;

  myfile = fopen("coutput.dat","w");
  if(myfile == NULL){
     printf("Error opening file\n");
  }else{
     printf("File opened successfully.\n");
     fclose(myfile);
  }

  return 1;
}
```

Save this as **opener.c** Compile and run it. Keep in mind that this program doesn't test to see if the file exists before trying to create it.

Here we can see that the file structure type, **FILE**, is used to declare a pointer. Now we have to establish which file we want to point to. Using the **fopen()** function, we can specify the name of the file and which *mode* we want to open it in (modes are explained below). If there is an error opening the file, **fopen()** will return a NULL pointer. We should check for a NULL pointer because we don't want to continue trying to use this file if it didn't open successfully. When we're done with the file we use **fclose()** to close it.

Note Opening a file essentially sets up a data stream for input and/or output just like **stdin** and **stdout**.

fopen Modes

When using **fopen()** to open a file, you need to indicate how you want to access the file by specifying a *mode*. In the example above, we used **w**. The **w** mode opens a file for writing, and if it already exists, its original contents are removed. Two other basic modes are **r** and **a**. The **r** mode opens a file for just reading (you can't edit or delete its contents), while the **a** mode opens it for appending (you can only add to its contents). While appending, anything new that's written to the file is put at the end.

Normal Modes	
r	read from an existing file
w	create a new file for writing or clear out an existing one
a	writes to end of an existing file

There are three additional modes that allow for reading and writing at the same time: **r+**, **w+**, and **a+**.

Read/Write Modes	
r+	modify existing file
w+	create new file or clear out existing one
a+	reads normally, appends while writing

fprintf

Very similar to **printf()** and **sprintf()**, this function is used to, you guessed it, print to files. The **fprintf()** function takes a file pointer as its first argument. Edit **opener.c** as shown in **blue** below:

Type the following into CodeRunner below:

```
/* opener.c */

#include <stdio.h>

int main(){
  FILE *myfile;

  myfile = fopen("coutput.dat","w");
  if(myfile == NULL){
     printf("Error opening file\n");
  }else{
     fprintf(myfile,"Here are the contents of my file.\n");
     fclose(myfile);
  }

  return 1;
}
```

Compile and run it to make sure it works. After you've run it, use **ls** at the unix prompt to make sure the file exists. You can open the file in your editor or use **cat** to make sure it's correct.

Observe the following:

```
cold:~$ gcc opener.c -o opener
cold:~$ ./opener
cold:~$ ls
a.out*        house.c      math1.c~     pointer.c    scanf*       struct*
coutput.dat   house.c~     openfile*    pointer.c~   scanf.c      struct.c
file.c~       main.c       openfile.c   power*       scanf.c~     struct.c~
first.c*      main.c~      pntfunc*     power.c      string*      test*
first.c~*     math1*       pntfunc.c    power.c~     string.c     test.c
house*        math1.c      pointer*     power.h      string.c~    test.c~
cold:~$ cat coutput.dat
Here are the contents of my file.
cold:~$
```

fscanf

I bet you didn't see THIS coming! :) Again, **fscanf()** works much like **scanf()** and **sscanf()** except that you have to give it a file pointer first.

fscanf syntax

```
fscanf(myfile,"%f %c",&myfloat,&mychar);
```

Note Be careful when reading and writing a file at the same time. If you start reading, and then start writing, it will write over the characters at that position in the file. This is because one of the elements of the file structure stores the position within the file.

EOF

EOF stands for *end of file*. There is an EOF marker at the end of every file. It allows you and other functions to check for the end of a data stream. C provides us with a function called **feof()** to facilitate this.

Type the following into CodeRunner below:

```
/* file_reader.c */

#include <stdio.h>

int main(){
  char blaa;
  FILE *myfile;

  /* open the file for reading */
  myfile = fopen("coutput.dat","r");

  /* make sure the open was successful */
  if(myfile == NULL){
    printf("Error opening file\n");
    return 0;
  }

  /* read from the file, one character at a time */
  /* and stop at the EOF */
  fscanf(myfile,"%c",&blaa);
  while(!feof(myfile)){
    printf("%c",blaa);
    fscanf(myfile,"%c",&blaa);
  }

  fclose(myfile);

  return 1;
}
```

Save it as **file_reader.c**. Compile and run it to make sure it displays the contents of the file correctly.

Observe the following:

```
Here are the contents of my file.
```

Here we're just reading in and printing out one character at a time until we hit the end of the file. We scan in the first character of our file and then test for EOF. While we're not at the end of the file, we print out the character we scanned and get another one.

fgets

If you want to read strings from a file, use the **fgets()** function. If you read the aside in an earlier lesson about more secure programming of strings, you'll remember it. We have to specify a file pointer as well as the maximum number of characters to read. It will read up to one less than the number you specify (it saves room for the NULL character it adds) or until it reaches a newline. Unlike **gets()**, which doesn't keep a trailing newline as part of the string, **fgets()** does. Another difference is that **gets()** also considers an EOF the end of a string, while **fgets()** considers an EOF to be an error. If an error occurs, **fgets()** returns a NULL pointer. Here's an example:

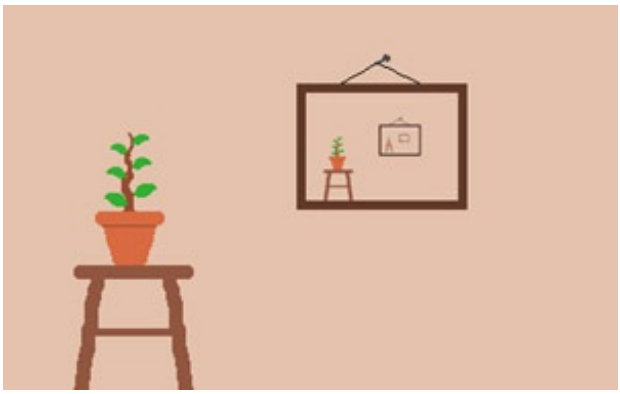
Type the following into CodeRunner below:

```
/* eof_finder.c */

#include <stdio.h>

int main(){
  char str[40];
  int i=40;
  int linecount = 0;
  FILE *myfile;

  /* open the file for reading */
  myfile = fopen("coutput.dat","r");

  /* make sure the open was successful */
  if(myfile == NULL){
    printf("Error opening file\n");
    return 0;
  }

  /* read a string from the file */
  while(fgets(str,i,myfile) != NULL){
    printf("%s\n",str);
    linecount++;
  }

  printf("read failed after %i reads.\n",linecount);

  fclose(myfile);

  return 1;
}
```

Since the **coutput.dat** file still exists, we can read the line from that file and print it out. Compile and test the above code after saving it as **eof_finder.c**.

What is Recursion?

Recursion doesn't introduce any new functions in C, but it introduces an important concept. The point of recursion is basically to have a function that calls itself. It can call itself over and over until a certain condition is met (this condition is called the *base case*). It's sort of like one of those pictures you've probably seen of a frame hanging on a wall with a picture of the wall in it.



(Sorry, I'm not an artist)

You can think of the base case as the point at which you can't draw any smaller.

Let's start by making a recursive function that doesn't really require recursion. I just want to get down the basic concepts first.

Type the following into CodeRunner:

```
/* recursor.c */

#include <stdio.h>

int main(){
  int i=0,end;
  int numbers(int *j);

  end = numbers(&i);

  printf("End is %i\n",end);
  return 1;
}

int numbers(int *j){
  int tmp;
  if(*j == 3){
    return 0;
  }

  (*j)++;
  tmp = *j;

  return(numbers(j) + tmp);
}
```

Save it as **recursor.c**. Try this code to get the following result:

OBSERVE:

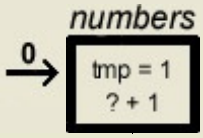
```
End is 6
```

The first section in the **numbers()** function is our base case. This provides a stopping point. It's essential that, each time **numbers()** runs, we make progress toward the base case. Here we're just adding 1 to an integer until we get to 3. If we don't make progress, it's possible to get something that's infinitely recursive and we don't want that. When the integer equals 3, **numbers()** just returns zero.

When the base case hasn't been reached, however, **numbers()** returns the value of another run through **numbers()** added to the value of the integer when it was called. Needless to say, this gets confusing.

Here is a graphic representation of recursive flow:

Next | Start Over



Step 1: This box represents the first time **numbers** is called. The **tmp** variable is set to 1 (equal to the new v
we increased a pointer to it) and the function tries to return the result of another call to **numbers** plus the valu

Consider the change below; what do you think the output will be? Is it the same thing? All I'm doing is getting rid of the temporary integer.

Type the following into CodeRunner:

```
/* recursor.c */

#include <stdio.h>

int main(){
  int i=0,end;
  int numbers(int *j);

  end = numbers(&i);

  printf("end is %i\n",end);
  return 1;
}

int numbers(int *j){
  if(*j == 3){
    return 0;
  }
  (*j)++;

  return(numbers(j) + *j);
}
```

Save, compile, and run it. Examine the output on your own and see if you can figure out any differences.

Be Careful and Why?

You have to be pretty careful when using recursion. One of the most popular examples of recursion involves the Fibonacci sequence (if you're not familiar with the Fibonacci sequence, don't worry about it). The Fibonacci recursive function that is used to find the *n*th number of the series, calls itself twice. The result is exponential growth. Sometimes it's better to use a while loop (a while loop to find the 40th number of the series took almost no time at all on the OST machines, while the recursive version took 18 seconds).

If you go on and take a data structures course in C/C++, you'll see the real power of recursion.