Programming in C

Lesson 1: The basics of C

This tutorial is a port of the C++ tutorial but is designed to be a stand-alone introduction to C, even if you've never programmed before. Unless you have a particular reason to learn C instead of C++, I recommend starting the C++ tutorial instead. Nevertheless, if you do not desire some of C++'s advanced features or simply wish to start with a slightly less complicated language, then this tutorial is for you!

Getting set up

C is a programming language of many different dialects, similar to the way that each spoken language has many different dialects. In C, dialects don't exist because the speakers live in the North or South. Instead, they're there because there are many different compilers that support slightly different features. There are several common compilers: in particular, Borland C++, Microsoft C++, and GNU C. There are also many front-end environments for the different compilers--the most common is Dev-C++ around GNU's G++ compiler. Some, such as GCC, are free, while others are not. Please see the compiler listing for more information on how to get a compiler and set it up. You should note that if you are programming in C on a C++ compiler, then you will want to make sure that your compiler attempts to compile C instead of C++ to avoid small compatability issues in later tutorials.

Each of these compilers is slightly different. Each one should support the ANSI standard C functions, but each compiler will also have nonstandard functions (these functions are similar to slang spoken in different parts of a country). Sometimes the use of nonstandard functions will cause problems when you attempt to compile source code (the actual C code written by a programmer and saved as a text file) with a different compiler. These tutorials use ANSI standard C and should not suffer from this problem; fortunately, since C has been around for quite a while, there shouldn't be too many compatability issues except when your compiler tries to create C++ code.

If you don't yet have a compiler, I strongly recommend finding one now. A simple compiler is sufficient for our use, but make sure that you do get one in order to get the most from these tutorials. The page linked above, compilers, lists compilers by operating system.

Every full C program begins inside a function called "main". A function is simply a collection of commands that do "something". The main function is always called when the program first executes. From main, we can call other functions, whether they be written by us or by others or use built-in language features. To access the standard functions that comes with your compiler, you need to include a header with the #include directive. What this does is effectively take everything in the header and paste it into your program. Let's look at a working program:

```
 #include <stdio.h>

int main()
{
  printf( "I am alive!  Beware.\n" );
  getchar();
  return 0;
}
```

Let's look at the elements of the program. The #include is a "preprocessor" directive that tells the compiler to put code from the header called stdio.h into our program before actually creating the executable. By including header files, you can gain access to many different functions--both the printf and getchar functions are included in stdio.h. The semicolon is part of the syntax of C. It tells the compiler that you're at the end of a command. You will see later that the semicolon is used to end most commands in C.

The next imporant line is int main(). This line tells the compiler that there is a function named main, and that the function returns an integer, hence int. The "curly braces" ({ and }) signal the beginning and end of functions and other code blocks. If you have programmed in Pascal, you will know them as BEGIN and END. Even if you haven't programmed in Pascal, this is a good way to think about their meaning.

The printf function is the standard C way of displaying output on the screen. The quotes tell the compiler that you want to output the literal string as-is (almost). The '\n' sequence is actually treated as a single character that stands for a newline (we'll talk about this later in more detail); for the time being, just remember that there are a few sequences that, when they appear in a string literal, are actually not displayed literally by printf and that '\n' is one of them. The actual effect of '\n' is to move the cursor on your screen to the next line. Again, notice the semicolon: it is added onto the end of all lines, such as function calls, in C.

The next command is getchar(). This is another function call: it reads in a single character and waits for the user to hit enter before reading the character. This line is included because many compiler environments will open a new console window, run the program, and then close the window before you can see the output. This command keeps that window from closing because the program is not done yet because it waits for you to hit enter. Including that line gives you time to see the program run.

Finally, at the end of the program, we return a value from main to the operating system by using the return statement. This return value is important as it can be used to tell the operating system whether our program succeeded or not. A return value of 0 means success.

The final brace closes off the function. You should try compiling this program and running it. You can cut and paste the code into a file, save it as a .c file, and then compile it. If you are using a command-line compiler, such as Borland C++ 5.5, you should read the compiler instructions for information on how to compile. Otherwise compiling and running should be as simple as clicking a button with your mouse (perhaps the "build" or "run" button).

You might start playing around with the printf function and get used to writing simple C programs.

Explaining your Code

Comments are critical for all but the most trivial programs and this tutorial will often use them to explain sections of code. When you tell the compiler a section of text is a comment, it will ignore it when running the code, allowing you to use any text you want to describe the real code. To create a comment in C, you surround the text with /* and then */ to block off everything between as a comment. Certain compiler environments or text editors will change the color of a commented area to make it easier to spot, but some will not. Be certain not to accidentally comment out code (that is, to tell the compiler part of your code is a comment) you need for the program.

When you are learning to program, it is also useful to comment out sections of code in order to see how the output is affected.

Using Variables

So far you should be able to write a simple program to display information typed in by you, the programmer and to describe your program with comments. That's great, but what about interacting with your user? Fortunately, it is also possible for your program to accept input.

But first, before you try to receive input, you must have a place to store that input. In programming, input and data are stored in variables. There are several different types of variables; when you tell the compiler you are declaring a variable, you must include the data type along with the name of the variable. Several basic types include char, int, and float. Each type can store different types of data.

A variable of type char stores a single character, variables of type int store integers (numbers without decimal places), and variables of type float store numbers with decimal places. Each of these variable types - char, int, and float - is each a keyword that you use when you declare a variable. Some variables also use more of the computer's memory to store their values.

It may seem strange to have multiple variable types when it seems like some variable types are redundant. But using the right variable size can be important for making your program efficient because some variables require more memory than others. For now, suffice it to say that the different variable types will almost all be used!

Before you can use a variable, you must tell the compiler about it by declaring it and telling the compiler about what its "type" is. To declare a variable you use the syntax <variable type> <name of variable>;. (The brackets here indicate that your replace the expression with text described within the brackets.) For instance, a basic variable declaration might look like this:

```
int myVariable;
```

Note once again the use of a semicolon at the end of the line. Even though we're not calling a function, a semicolon is still required at the end of the "expression". This code would create a variable called myVariable; now we are free to use myVariable later in the program.

It is permissible to declare multiple variables of the same type on the same line; each one should be separated by a comma. If you attempt to use an undefined variable, your program will not run, and you will receive an error message informing you that you have made a mistake.

Here are some variable declaration examples:

```
int x;
int a, b, c, d;
char letter;
float the_float;
```

While you can have multiple variables of the same type, you cannot have multiple variables with the same name. Moreover, you cannot have variables and functions with the same name.

A final restriction on variables is that variable declarations must come before other types of statements in the given "code block" (a code block is just a segment of code surrounded by { and }). So in C you must declare all of your variables before you do anything else:

Wrong

```
#include <stdio.h>
int main()
{
    /* wrong!  The variable declaration must appear first */
    printf( "Declare x next" );
    int x;

    return 0;
}
```

Fixed

```
#include <stdio.h>
int main()
{
    int x;
    printf( "Declare x first" );

    return 0;
}
```

Reading input

Using variables in C for input or output can be a bit of a hassle at first, but bear with it and it will make sense. We'll be using the scanf function to read in a value and then printf to read it back out. Let's look at the program and then pick apart exactly what's going on. You can even compile this and run it if it helps you follow along.

```
#include <stdio.h>

int main()
{
  int this_is_a_number;

  printf( "Please enter a number: " );
  scanf( "%d", &this_is_a_number );
  printf( "You entered %d", this_is_a_number );
  getchar();
  return 0;
}
```

So what does all of this mean? We've seen the #include and main function before; main must appear in every program you intend to run, and the #include gives us access to printf (as well as scanf). (As you might have guessed, the io in stdio.h stands for "input/output"; std just stands for "standard.") The keyword int declares this_is_a_number to be an integer.

This is where things start to get interesting: the scanf function works by taking a string and some variables modified with &. The string tells scanf what variables to look for: notice that we have a string containing only "%d" -- this tells the scanf function to read in an integer. The second argument of scanf is the variable, sort of. We'll learn more about what is going on later, but the gist of it is that scanf needs to know where the variable is stored in order to change its value. Using & in front of a variable allows you to get its location and give that to scanf instead of the value of the variable. Think of it like giving someone directions to the soda aisle and letting them go get a coca-cola instead of fetching the coke for that person. The & gives the scanf function directions to the variable.

When the program runs, each call to scanf checks its own input string to see what kinds of input to expect, and then stores the value input into the variable.

The second printf statement also contains the same '%d'--both scanf and printf use the same format for indicating values embedded in strings. In this case, printf takes the first argument after the string, the variable this_is_a_number, and treats it as though it were of the type specified by the "format specifier". In this case, printf treats this_is_a_number as an integer based on the format specifier.

So what does it mean to treat a number as an integer? If the user attempts to type in a decimal number, it will be truncated (that is, the decimal component of the number will be ignored) when stored in the variable. Try typing in a sequence of characters or a decimal number when you run the example program; the response will vary from input to input, but in no case is it particularly pretty.

Of course, no matter what type you use, variables are uninteresting without the ability to modify them. Several operators used with variables include the following: *, -, +, /, =, ==, >, <. The * multiplies, the / divides, the - subtracts, and the + adds. It is of course important to realize that to modify the value of a variable inside the program it is rather important to use the equal sign. In some languages, the equal sign compares the value of the left and right values, but in C == is used for that task. The equal sign is still extremely useful. It sets the value of the variable on the left side of the equals sign equal to the value on the right side of the equals sign. The operators that perform mathematical functions should be used on the right side of an equal sign in order to assign the result to a variable on the left side.

Here are a few examples:

```
a = 4 * 6; /* (Note use of comments and of semicolon) a is 24 */
a = a + 5; /* a equals the original value of a with five added to
it */
a == 5     /* Does NOT assign five to a. Rather, it checks to see
if a equals 5.*/
```

The other form of equal, ==, is not a way to assign a value to a variable. Rather, it checks to see if the variables are equal. It is extremely useful in many areas of C; for example, you will often use == in such constructions as conditional statements and loops. You can probably guess how < and > function. They are greater than and less than operators.

For example:

```
a < 5  /* Checks to see if a is less than five */
a > 5  /* Checks to see if a is greater than five */
a == 5 /* Checks to see if a equals five, for good measure */
```

Lesson 2: If statements

The ability to control the flow of your program, letting it make decisions on what code to execute, is valuable to the programmer. The if statement allows you to control if a program enters a section of code or not based on whether a given condition is true or false. One of the important functions of the if statement is that it allows the program to select an action based upon the user's input. For example, by using an if statement to check a user-entered password, your program can decide whether a user is allowed access to the program.

Without a conditional statement such as the if statement, programs would run almost the exact same way every time, always following the same sequence of function calls. If statements allow the flow of the program to be changed, which leads to more interesting code.

Before discussing the actual structure of the if statement, let us examine the meaning of TRUE and FALSE in computer terminology. A true statement is one that evaluates to a nonzero number. A false statement evaluates to zero. When you perform comparison with the relational operators, the operator will return 1 if the comparison is true, or 0 if the comparison is false. For example, the check 0 == 2 evaluates to 0. The check 2 == 2 evaluates to a 1. If this confuses you, try to use a printf statement to output the result of those various comparisons (for example printf ("%d", 2 == 1);)

When programming, the aim of the program will often require the checking of one value stored by a variable against another value to determine whether one is larger, smaller, or equal to the other.

There are a number of operators that allow these checks.

Here are the relational operators, as they are known, along with examples:

```
>     greater than              5 > 4 is TRUE
<     less than                 4 < 5 is TRUE
>=    greater than or equal     4 >= 4 is TRUE
<=    less than or equal        3 <= 4 is TRUE
==    equal to                  5 == 5 is TRUE
!=    not equal to              5 != 4 is TRUE
```

It is highly probable that you have seen these before, probably with slightly different symbols. They should not present any hindrance to understanding. Now that you understand TRUE and FALSE well as the comparison operators, let us look at the actual structure of if statements.

The structure of an if statement is as follows:

```
if ( statement is TRUE )
    Execute this line of code
```

Here is a simple example that shows the syntax:

```
if ( 5 < 10 )
    printf( "Five is now less than ten, that's a big surprise" );
```

Here, we're just evaluating the statement, "is five less than ten", to see if it is true or not; with any luck, it's not! If you want, you can write your own full program including stdio.h and put this in the main function and run it to test.

To have more than one statement execute after an if statement that evaluates to true, use braces, like we did with the body of the main function. Anything inside braces is called a compound statement, or a block. When using if statements, the code that depends on the if statement is called the "body" of the if statement.

For example:

```
if ( TRUE ) {
  /* between the braces is the body of the if statement */
  Execute all statements inside the body
}
```

I recommend always putting braces following if statements. If you do this, you never have to remember to put them in when you want more than one statement to be executed, and you make the body of the if statement more visually clear.

Else

Sometimes when the condition in an if statement evaluates to false, it would be nice to execute some code instead of the code executed when the statement evalutes to true. The "else" statement effectively says that whatever code after it (whether a single line or code between brackets) is executed if the if statement is FALSE.

It can look like this:

```
if ( TRUE ) {
  /* Execute these statements if TRUE */
}
else {
  /* Execute these statements if FALSE */
}
```

Else if

Another use of else is when there are multiple conditional statements that may all evaluate to true, yet you want only one if statement's body to execute. You can use an "else if" statement following an if statement and its body; that way, if the first statement is true, the "else if" will be ignored, but if the if statement is false, it will then check the condition for the else if statement. If the if statement was true the else statement will not be checked. It is possible to use numerous else if statements to ensure that only one block of code is executed.

Let's look at a simple program for you to try out on your own.

```
#include <stdio.h>

int main()                            /* Most important part of
the program!
*/
{
    int age;                          /* Need a variable... */

    printf( "Please enter your age" );  /* Asks for age */
    scanf( "%d", &age );                 /* The input is put in
age */
    if ( age < 100 ) {                  /* If the age is less than
100 */
     printf ("You are pretty young!\n" ); /* Just to show you it
works... */
  }
  else if ( age == 100 ) {            /* I use else just to show
an example */
     printf( "You are old\n" );
  }
  else {
    printf( "You are really old\n" );     /* Executed if no other
statement is
    */
  }
  return 0;
}
```

More interesting conditions using boolean operators

Boolean operators allow you to create more complex conditional statements. For example, if you wish to check if a variable is both greater than five and less than ten, you could use the Boolean AND to ensure both var > 5 and var < 10 are true. In the following discussion of Boolean operators, I will capitalize the Boolean operators in order to distinguish them from normal English. The actual C operators of equivalent function will be described further along into the tutorial - the C symbols are not: OR, AND, NOT, although they are of equivalent function.

When using if statements, you will often wish to check multiple different conditions. You must understand the Boolean operators OR, NOT, and AND. The boolean operators function in a similar way to the comparison operators: each returns 0 if evaluates to FALSE or 1 if it evaluates to TRUE.

NOT: The NOT operator accepts one input. If that input is TRUE, it returns FALSE, and if that input is FALSE, it returns TRUE. For example, NOT (1) evalutes to 0, and NOT (0) evalutes to 1. NOT (any number but zero) evaluates to 0. In C NOT is written as !. NOT is evaluated prior to both AND and OR.

AND: This is another important command. AND returns TRUE if both inputs are TRUE (if 'this' AND 'that' are true). (1) AND (0) would evaluate to zero because one of the inputs is false (both must be TRUE for it to evaluate to TRUE). (1) AND (1) evaluates to 1. (any number but 0) AND (0) evaluates to 0. The AND operator is written && in C. Do not be confused by thinking it checks equality between numbers: it does not. Keep in mind that the AND operator is evaluated before the OR operator.

OR: Very useful is the OR statement! If either (or both) of the two values it checks are TRUE then it returns TRUE. For example, (1) OR (0) evaluates to 1. (0) OR (0) evaluates to 0. The OR is written as || in C. Those are the pipe characters. On your keyboard, they may look like a stretched colon. On my computer the pipe shares its key with \. Keep in mind that OR will be evaluated after AND.

It is possible to combine several Boolean operators in a single statement; often you will find doing so to be of great value when creating complex expressions for if statements. What is !(1 && 0)? Of course, it would be TRUE. It is true is because 1 && 0 evaluates to 0 and !0 evaluates to TRUE (ie, 1).

Try some of these - they're not too hard. If you have questions about them, feel free to stop by our forums.

```
A. !( 1 || 0 )         ANSWER: 0
B. !( 1 || 1 && 0 )    ANSWER: 0 (AND is evaluated before OR)
C. !( ( 1 || 0 ) && 0 )  ANSWER: 1 (Parenthesis are useful)
```

If you find you enjoyed this section, then you might want to look more at Boolean Algebra.

Lesson 3: Loops

Loops are used to repeat a block of code. Being able to have your program repeatedly execute a block of code is one of the most basic but useful tasks in programming -- many programs or websites that produce extremely complex output (such as a message board) are really only executing a single task many times. (They may be executing a small number of tasks, but in principle, to produce a list of messages only requires repeating the operation of reading in some data and displaying it.) Now, think about what this means: a loop lets you write a very simple statement to produce a significantly greater result simply by repetition.

One caveat: before going further, you should understand the concept of C's true and false, because it will be necessary when working with loops (the conditions are the same as with if statements). This concept is covered in the previous tutorial. There are three types of loops: for, while, and do..while. Each of them has their specific uses. They are all outlined below.

FOR - for loops are the most useful type. The syntax for a for loop is

```
for ( variable initialization; condition; variable update ) {
  Code to execute while the condition is true
}
```

The variable initialization allows you to either declare a variable and give it a value or give a value to an already existing variable. Second, the condition tells the program that while the conditional expression is true the loop should continue to repeat itself. The variable update section is the easiest way for a for loop to handle changing of the variable. It is possible to do things like x++, x = x + 10, or even x = random (5), and if you really wanted to, you could call other functions that do nothing to the variable but still have a useful effect on the code. Notice that a semicolon separates each of these sections, that is important. Also note that every single one of the sections may be empty, though the semicolons still have to be there. If the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

Example:

```
#include <stdio.h>

int main()
{
    int x;
    /* The loop goes while x < 10, and x increases by one every 
loop*/
    for ( x = 0; x < 10; x++ ) {
        /* Keep in mind that the loop condition checks
```

```
            the conditional statement before it loops again.
            consequently, when x equals 10 the loop breaks.
            x is updated before the condition is checked. */
        printf( "%d\n", x );
    }
    getchar();
}
```

This program is a very simple example of a for loop. x is set to zero, while x is less than 10 it calls printf to display the value of the variable x, and it adds 1 to x until the condition is met. Keep in mind also that the variable is incremented after the code in the loop is run for the first time.

WHILE - WHILE loops are very simple. The basic structure is

while (condition) { Code to execute while the condition is true }
The true represents a boolean expression which could be x == 1 or while (x != 7) (x does not equal 7). It can be any combination of boolean statements that are legal. Even, (while x ==5 || v == 7) which says execute the code while x equals five or while v equals 7. Notice that a while loop is like a stripped-down version of a for loop-- it has no initialization or update section. However, an empty condition is not legal for a while loop as it is with a for loop.

Example:

```

#include <stdio.h>

int main()
{
  int x = 0;  /* Don't forget to declare variables */

  while ( x < 10 ) { /* While x is less than 10 */
      printf( "%d\n", x );
      x++;             /* Update x so the condition can be met
eventually */
  }
  getchar();
}
```

This was another simple example, but it is longer than the above FOR loop. The easiest way to think of the loop is that when it reaches the brace at the end it jumps back up to the beginning of the loop, which checks the condition again and decides whether to repeat the block another time, or stop and move to the next statement after the block.

DO..WHILE - DO..WHILE loops are useful for things that want to loop at least once. The structure is

```
do {
} while ( condition );
```

Notice that the condition is tested at the end of the block instead of the beginning, so the block will be executed at least once. If the condition is true, we jump back to the beginning of the block and execute it again. A do..while loop is almost the same as a while loop except that the loop body is guaranteed to execute at least once. A while loop says "Loop while the condition is true, and execute this block of code", a do..while loop says "Execute this block of code, and then continue to loop while the condition is true".

Example:

```
#include <stdio.h>

int main()
{
  int x;

  x = 0;
  do {
    /* "Hello, world!" is printed at least one time
      even though the condition is false*/
    printf( "%d\n", x );
  } while ( x != 0 );
  getchar();
}
```

Keep in mind that you must include a trailing semi-colon after the while in the above example. A common error is to forget that a do..while loop must be terminated with a semicolon (the other loops should not be terminated with a semicolon, adding to the confusion). Notice that this loop will execute once, because it automatically executes before checking the condition.

Break and Continue

Two keywords that are very important to looping are break and continue. The break command will exit the most immediately surrounding loop regardless of what the conditions of the loop are. Break is useful if we want to exit a loop under special circumstances. For example, let's say the program we're working on is a two-person checkers game. The basic structure of the program might look like this:

```
while (true)
{
    take_turn(player1);
    take_turn(player2);
}
```

This will make the game alternate between having player 1 and player 2 take turns. The only problem with this logic is that there's no way to exit the game; the loop will run forever! Let's try something like this instead:

```
while(true)
{
    if (someone_has_won() || someone_wants_to_quit() == TRUE)
    {break;}
    take_turn(player1);
    if (someone_has_won() || someone_wants_to_quit() == TRUE)
    {break;}
    take_turn(player2);
}
```

This code accomplishes what we want--the primary loop of the game will continue under normal circumstances, but under a special condition (winning or exiting) the flow will stop and our program will do something else.

Continue is another keyword that controls the flow of loops. If you are executing a loop and hit a continue statement, the loop will stop its current iteration, update itself (in the case of for loops) and begin to execute again from the top. Essentially, the continue statement is saying "this iteration of the loop is done, let's continue with the loop without executing whatever code comes after me." Let's say we're implementing a game of Monopoly. Like above, we want to use a loop to control whose turn it is, but controlling turns is a bit more complicated in Monopoly than in checkers. The basic structure of our code might then look something like this:

```
for (player = 1; someone_has_won == FALSE; player++)
    {
        if (player > total_number_of_players)
        {player = 1;}
        if (is_bankrupt(player))
        {continue;}
        take_turn(player);
    }
```

This way, if one player can't take her turn, the game doesn't stop for everybody; we just skip her and keep going with the next player's turn.

Lesson 4: Functions

Now that you should have learned about variables, loops, and conditional statements it is time to learn about functions. You should have an idea of their uses as we have already used them and defined one in the guise of main. Getchar is another example of a function. In general, functions are blocks of code that perform a number of pre-defined commands to accomplish something productive. You can either use the built-in library functions or you can create your own functions.

Functions that a programmer writes will generally require a prototype. Just like a blueprint, the prototype gives basic structural information: it tells the compiler what the function will return, what the function will be called, as well as what arguments the function can be passed. When I say that the function returns a value, I mean that the function can be used in the same manner as a variable would be. For example, a variable can be set equal to a function that returns a value between zero and four.

For example:

```
#include <stdlib.h>   /* Include rand() */

int a = rand(); /* rand is a standard function that all compilers
have */
```

Do not think that 'a' will change at random, it will be set to the value returned when the function is called, but it will not change again.

The general format for a prototype is simple:

```
return-type function_name ( arg_type arg1, ..., arg_type argN );
```

arg_type just means the type for each argument -- for instance, an int, a float, or a char. It's exactly the same thing as what you would put if you were declaring a variable.

There can be more than one argument passed to a function or none at all (where the parentheses are empty), and it does not have to return a value. Functions that do not return values have a return type of void. Let's look at a function prototype:

```
int mult ( int x, int y );
```

This prototype specifies that the function mult will accept two arguments, both integers, and that it will return an integer. Do not forget the trailing semi-colon. Without it, the compiler will probably think that you are trying to write the actual definition of the function.

When the programmer actually defines the function, it will begin with the prototype, minus the semi-colon. Then there should always be a block (surrounded by curly braces) with the code that the function is to execute, just as you would write it for the main function. Any of the arguments passed to the function can be used as if they were declared in the block. Finally, end it all with a cherry and a closing brace. Okay, maybe not a cherry.

Let's look at an example program:

```
#include <stdio.h>

int mult ( int x, int y );

int main()
{
  int x;
  int y;

  printf( "Please input two numbers to be multiplied: " );
  scanf( "%d", &x );
  scanf( "%d", &y );
  printf( "The product of your two numbers is %d\n", mult( x, y )
);
  getchar();
}

int mult (int x, int y)
{
  return x * y;
}
```

This program begins with the only necessary include file. Next is the prototype of the function. Notice that it has the final semi-colon! The main function returns an integer, which you should always have to conform to the standard. You should not have trouble understanding the input and output functions if you've followed the previous tutorials.

Notice how printf actually takes the value of what appears to be the mult function. What is really happening is printf is accepting the value returned by mult, not mult itself. The result would be the same as if we had use this print instead

```
printf( "The product of your two numbers is %d\n", x * y );
```

The mult function is actually defined below main. Because its prototype is above main, the compiler still recognizes it as being declared, and so the compiler will not give an error about mult being undeclared. As long as the prototype is present, a function can be used even if there is no definition. However, the code cannot be run without a definition even though it will compile.

Prototypes are declarations of the function, but they are only necessary to alert the compiler about the existence of a function if we don't want to go ahead and fully define the function. If mult were defined before it is used, we could do away with the prototype--the definition basically acts as a prototype as well.

Return is the keyword used to force the function to return a value. Note that it is possible to have a function that returns no value. If a function returns void, the retun statement is valid, but only if it does not have an expression. In otherwords, for a function that returns void, the statement "return;" is legal, but usually redundant. (It can be used to exit the function before the end of the function.)

The most important functional (pun semi-intended) question is why do we need a function? Functions have many uses. For example, a programmer may have a block of code that he has repeated forty times throughout the program. A function to execute that code would save a great deal of space, and it would also make the program more readable. Also, having only one copy of the code makes it easier to make changes. Would you rather make forty little changes scattered all throughout a potentially large program, or one change to the function body? So would I.

Another reason for functions is to break down a complex program into logical parts. For example, take a menu program that runs complex code when a menu choice is selected. The program would probably best be served by making functions for each of the actual menu choices, and then breaking down the complex tasks into smaller, more manageable tasks, which could be in their own functions. In this way, a program can be designed that makes sense when read. And has a structure that is easier to understand quickly. The worst programs usually only have the required function, main, and fill it with pages of jumbled code.

Lesson 5: switch case

Switch case statements are a substitute for long if statements that compare a variable to several "integral" values ("integral" values are simply values that can be expressed as an integer, such as the value of a char). The basic format for using switch case is outlined below. The value of the variable given into switch is compared to the value following each of the cases, and when one value matches the value of the variable, the computer continues executing the program from that point.

```
switch ( <variable> ) {
case this-value:
  Code to execute if <variable> == this-value
  break;
case that-value:
  Code to execute if <variable> == that-value
  break;
...
default:
  Code to execute if <variable> does not equal the value following
any of the cases
  break;
}
```

The condition of a switch statement is a value. The case says that if it has the value of whatever is after that case then do whatever follows the colon. The break is used to break out of the case statements. Break is a keyword that breaks out of the code block, usually surrounded by braces, which it is in. In this case, break prevents the program from falling through and executing the code in all the other case statements. An important thing to note about the switch statement is that the case values may only be constant integral expressions. Sadly, it isn't legal to use case like this:

```
int a = 10;
int b = 10;
int c = 20;

switch ( a ) {
case b:
  /* Code */
  break;
case c:
  /* Code */
  break;
default:
  /* Code */
  break;
}
```

The default case is optional, but it is wise to include it as it handles any unexpected cases. It can be useful to put some kind of output to alert you to the code entering the default case if you don't expect it to. Switch statements serve as a simple way to write long if statements when the requirements are met. Often it can be used to process input from a user.

Below is a sample program, in which not all of the proper functions are actually declared, but which shows how one would use switch in a program.

```
#include <stdio.h>

void playgame();
void loadgame();
void playmultiplayer();

int main()
{
    int input;

    printf( "1. Play game\n" );
    printf( "2. Load game\n" );
    printf( "3. Play multiplayer\n" );
    printf( "4. Exit\n" );
    printf( "Selection: " );
    scanf( "%d", &input );
    switch ( input ) {
        case 1:            /* Note the colon, not a semicolon */
            playgame();
            break;
        case 2:
            loadgame();
            break;
        case 3:
            playmultiplayer();
            break;
        case 4:
            printf( "Thanks for playing!\n" );
            break;
        default:
            printf( "Bad input, quitting!\n" );
            break;
    }
    getchar();

}
```

This program will compile, but cannot be run until the undefined functions are given bodies, but it serves as a model (albeit simple) for processing input. If you do not understand this then try mentally putting in if statements for the case statements. Default simply skips out of the switch case construction and allows the program to terminate naturally. If you do not like that, then you can make a loop around the whole thing to have it wait for valid input. You could easily make a few small functions if you wish to test the code.

Lesson 6: An introduction to pointers

Pointers are an extremely powerful programming tool. They can make some things much easier, help improve your program's efficiency, and even allow you to handle unlimited amounts of data. For example, using pointers is one way to have a function modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle nearly unlimited amounts of data on the fly--you don't need to know, when you write the program, how much memory you need. Wow, that's kind of cool. Actually, it's very cool, as we'll see in some of the next tutorials. For now, let's just get a basic handle on what pointers are and how you use them.

What are pointers? Why should you care?

Pointers are aptly name: they "point" to locations in memory. Think of a row of safety deposity boxes of various sizes at a local bank. Each safety deposity box will have a number associated with it so that you can quickly look it up. These numbers are like the memory addresses of variables. A pointer in the world of safety deposit box would simply be anything that stored the number of another safety deposit box. Perhaps you have a rich uncle who stored valuables in his safety deposit box, but decided to put the real location in another, smaller, safety deposit box that only stored a card with the number of the large box with the real jewelery. The safety deposit box with the card would be storing the location of another box; it would be equivalent to a pointer. In the computer, pointers are just variables that store memory addresses, usually the addresses of other variables.

The cool thing is that once can talk about the address of a variable, you'll then be able to go to that address and retrieve the data stored in it. If you happen to have a huge piece of data that you want to pass into a function, it's a lot easier to pass its location to the function that to copy every element of the data! Moreover, if you need more memory for your program, you can request more memory from the system--how do you get "back" that memory? The system tells you where it is located in memory; that is to say, you get a memory address back. And you need pointers to store the memory address.

A note about terms: the word pointer can refer either to a memory address itself, or to a variable that stores a memory address. Usually, the distinction isn't really that important: if you pass a pointer variable into a function, you're passing the value stored in the pointer--the memory address. When I want to talk about a memory address, I'll refer to it as a memory address; when I want a variable that stores a memory address, I'll call it a pointer. When a variable stores the address of another variable, I'll say that it is "pointing to" that variable.

Pointer Syntax

Pointers require a bit of new syntax because when you have a pointer, you need the ability to both request the memory location it stores and the value stored at that memory location. Moreover, since pointers are somewhat special, you need to tell the compiler when you declare your pointer variable that the variable is a pointer, and tell the compiler what type of memory it points to.

The pointer declaration looks like this:

```
<variable_type> *<name>;
```

For example, you could declare a pointer that stores the address of an integer with the following syntax:

```
int *points_to_integer;
```

Notice the use of the *. This is the key to declaring a pointer; if you add it directly before the variable name, it will declare the variable to be a pointer. Minor gotcha: if you declare multiple pointers on the same line, you must precede each of them with an asterisk:

```
/* one pointer, one regular int */
int *pointer1, nonpointer1;

/* two pointers */
int *pointer1, *pointer2;
```

As I mentioned, there are two ways to use the pointer to access information: it is possible to have it give the actual address to another variable. To do so, simply use the name of the pointer without the *. However, to access the actual memory location, use the *. The technical name for this doing this is dereferencing the pointer; in essence, you're taking the reference to some memory address and following it, to retrieve the actual value. It can be tricky to keep track of when you should add the asterisk. Remember that the pointer's natural use is to store a memory address; so when you use the pointer:

```
call_to_function_expecting_memory_address(pointer);
```

then it evaluates to the address. You have to add something extra, the asterisk, in order to retrieve the value stored at the address. You'll probably do that an awful lot. Nevertheless, the pointer itself is supposed to store an address, so when you use the bare pointer, you get that address back.

Pointing to Something: Retrieving an Address

In order to have a pointer actually point to another variable it is necessary to have the memory address of that variable also. To get the memory address of a variable (its location in memory), put the & sign in front of the variable name. This makes it give its address. This is called the address-of operator, because it returns the memory address. Conveniently, both ampersand and address-of start with a; that's a useful way to remember that you use & to get the address of a variable.

For example:

```
#include <stdio.h>

int main()
{
    int x;            /* A normal integer*/
    int *p;           /* A pointer to an integer ("*p" is an
integer, so p
                       must be a pointer to an integer) */

    p = &x;           /* Read it, "assign the address of x to p"
*/
    scanf( "%d", &x );          /* Put a value in x, we could also
use p here */
    printf( "%d\n", *p ); /* Note the use of the * to get the
value */
    getchar();
}
```

The printf outputs the value stored in x. Why is that? Well, let's look at the code. The integer is called x. A pointer to an integer is then defined as p. Then it stores the memory location of x in pointer by using the address operator (&) to get the address of the variable. Using the ampersand is a bit like looking at the label on the safety deposit box to see its number rather than looking inside the box, to get what it stores. The user then inputs a number that is stored in the variable x; remember, this is the same location that is pointed to by p. In fact, since we use an ampersand to pass the value to scanf, it should be clear that scanf is putting the value in the address pointed to by p. (In fact, scanf works becuase of pointers!)

The next line then passes *p into printf. *p performs the "dereferencing" operation on p; it looks at the address stored in p, and goes to that address and returns the value. This is akin to looking inside a safety deposit box only to find the number of (and, presumably, the key to) another box, which you then open.

Notice that in the above example, the pointer is initialized to point to a specific memory address before it is used. If this was not the case, it could be pointing to anything. This can lead to extremely unpleasant consequences to the program. For instance, the operating system will probably prevent you from accessing memory that it knows your program doesn't own: this will cause your program to crash. If it let you use the memory, you could mess with the memory of any running program--for instance, if you had a document opened in Word, you could change the text! Fortunately, Windows and other modern operating systems will stop you from accessing that memory and cause your program to crash. To avoid crashing your program, you should always initialize pointers before you use them.

It is also possible to initialize pointers using free memory. This allows dynamic allocation of memory. It is useful for setting up structures such as linked lists or data trees where you don't know exactly how much memory will be needed at compile time, so you have to get memory during the program's execution. We'll look at these structures later, but for now, we'll simply examine how to request memory from and return memory to the operating system.

The function malloc, residing in the stdlib.h header file, is used to initialize pointers with memory from free store (a section of memory available to all programs). malloc works just like any other function call. The argument to malloc is the amount of memory requested (in bytes), and malloc gets a block of memory of that size and then returns a pointer to the block of memory allocated.

Since different variable types have different memory requirements, we need to get a size for the amount of memory malloc should return. So we need to know how to get the size of different variable types. This can be done using the keyword sizeof, which takes an expression and returns its size. For example, sizeof(int) would return the number of bytes required to store an integer.

```
#include <stdlib.h>

int *ptr = malloc( sizeof(int) );
```

This code set ptr to point to a memory address of size int. The memory that is pointed to becomes unavailable to other programs. This means that the careful coder should free this memory at the end of its usage lest the memory be lost to the operating system for the duration of the program (this is often called a memory leak because the program is not keeping track of all of its memory).

Note that it is slightly cleaner to write malloc statements by taking the size of the variable pointed to by using the pointer directly:

```
int *ptr = malloc( sizeof(*ptr) );
```

What's going on here? sizeof(*ptr) will evaluate the size of whatever we would get back from dereferencing ptr; since ptr is a pointer to an int, *ptr would give us an int, so sizeof(*ptr) will return the size of an integer. So why do this? Well, if we change ptr to point to something else like a float, then we don't have to go back and correct the malloc call to use sizeof(float). Since ptr would be pointing to a float, *ptr would be a float, so sizeof(*ptr) would still give the right size!

The free function returns memory to the operating system.

```
free( ptr );
```

After freeing a pointer, it is a good idea to reset it to point to 0. When 0 is assigned to a pointer, the pointer becomes a null pointer, in other words, it points to nothing. By doing this, when you do something foolish with the pointer (it happens a lot, even with experienced programmers), you find out immediately instead of later, when you have done considerable damage.

The concept of the null pointer is frequently used as a way of indicating a problem--for instance, malloc returns 0 when it cannot correctly allocate memory. You want to be sure to handle this correctly--sometimes your operating system might actually run out of memory and give you this value!

Taking Stock of Pointers

Pointers may feel like a very confusing topic at first but I think anyone can come to appreciate and understand them. If you didn't feel like you absorbed everything about them, just take a few deep breaths and re-read the lesson. You shouldn't feel like you've fully grasped every nuance of when and why you need to use pointers, though you should have some idea of some of their basic uses.

Lesson 7: Structures

When programming, it is often convenient to have a single name with which to refer to a group of a related values. Structures provide a way of storing many different values in variables of potentially different types under the same name. This makes it a more modular program, which is easier to modify because its design makes things more compact. Structs are generally useful whenever a lot of data needs to be grouped together--for instance, they can be used to hold records from a database or to store information about contacts in an address book. In the contacts example, a struct could be used that would hold all of the information about a single contact--name, address, phone number, and so forth.

The format for defining a structure is

```
struct Tag {
  Members
};
```

Where Tag is the name of the entire type of structure and Members are the variables within the struct. To actually create a single structure the syntax is

```
struct Tag name_of_single_structure;
```

To access a variable of the structure it goes

```
name_of_single_structure.name_of_variable;
```

For example:

```
struct example {
  int x;
};
struct example an_example; /* Treating it like a normal variable
type
                              except with the addition of struct*/
an_example.x = 33;           /*How to access its members */
```

Here is an example program:

```
struct database {
  int id_number;
  int age;
  float salary;
};

int main()
{
  struct database employee;  /* There is now an employee variable
that has
                              modifiable variables inside it.*/
  employee.age = 22;
  employee.id_number = 1;
  employee.salary = 12000.21;
}
```

The struct database declares that it has three variables in it, age, id_number, and salary. You can use database like a variable type like int. You can create an employee with the database type as I did above. Then, to modify it you call everything with the 'employee.' in front of it. You can also return structures from functions by defining their return type as a structure type. For instance:

```
struct database fn();
```

I will talk only a little bit about unions as well. Unions are like structures except that all the variables share the same memory. When a union is declared the compiler allocates enough memory for the largest data-type in the union. Its like a giant storage chest where you can store one large item, or a small item, but never the both at the same time.

The '.' operator is used to access different variables inside a union also.

As a final note, if you wish to have a pointer to a structure, to actually access the information stored inside the structure that is pointed to, you use the -> operator in place of the . operator. All points about pointers still apply.

A quick example:

```
#include <stdio.h>

struct xampl {
  int x;
};

int main()
{
    struct xampl structure;
    struct xampl *ptr;

    structure.x = 12;
    ptr = &structure; /* Yes, you need the & when dealing with
                           structures and using pointers to them*/
    printf( "%d\n", ptr->x );  /* The -> acts somewhat like the *
when
                                   does when it is used with
pointers
                                    It says, get whatever is at
that memory
                                   address Not "get what that
memory address
                                   is"*/
    getchar();
}
```

Lesson 8: Arrays

Arrays are useful critters that often show up when it would be convenient to have one name for a group of variables of the same type that can be accessed by a numerical index. For example, a tic-tac-toe board can be held in an array and each element of the tic-tac-toe board can easily be accessed by its position (the upper left might be position 0 and the lower right position 8). At heart, arrays are essentially a way to store many values under the same name. You can make an array out of any data-type including structures and classes.

One way to visualize an array is like this:

```
[][][][][][]
```

Each of the bracket pairs is a slot in the array, and you can store information in slot--the information stored in the array is called an element of the array. It is very much as though you have a group of variables lined up side by side.

Let's look at the syntax for declaring an array.

```
int examplearray[100]; /* This declares an array */
```

This would make an integer array with 100 slots (the places in which values of an array are stored). To access a specific part element of the array, you merely put the array name and, in brackets, an index number. This corresponds to a specific element of the array. The one trick is that the first index number, and thus the first element, is zero, and the last is the number of elements minus one. The indices for a 100 element array range from 0 to 99. Be careful not to "walk off the end" of the array by trying to access element 100!

What can you do with this simple knowledge? Lets say you want to store a string, because C has no built-in datatype for strings, you can make an array of characters.

For example:

```
char astring[100];
```

will allow you to declare a char array of 100 elements, or slots. Then you can receive input into it from the user, and when the user types in a string, it will go in the array, the first character of the string will be at position 0, the second character at position 1, and so forth. It is relatvely easy to work with strings in this way because it allows support for any size string you can imagine all stored in a single variable with each element in the string stored in an adjacent location--think about how hard it would be to store nearly arbitrary sized strings using simple variables that only store one value.

Since we can write loops that increment integers, it's very easy to scan through a string:

```
char astring[10];
int i = 0;
/* Using scanf isn't really the best way to do this; we'll talk
about that
   in the next tutorial, on strings */
scanf( "%s", astring );
for ( i = 0; i < 10; ++i )
{
    if ( astring[i] == 'a' )
    {
        printf( "You entered an a!\n" );
    }
}
```

Let's look at something new here: the scanf function call is a tad different from what we've seen before. First of all, the format string is '%s' instead of '%d'; this just tells scanf to read in a string instead of an integer. Second, we don't use the ampersand! It turns out that when we pass arrays into functions, the compiler automatically converts the array into a pointer to the first element of the array. In short, the array without any brackets will act like a pointer. So we just pass the array directly into scanf without using the ampersand and it works perfectly.

Also, notice that to access the element of the array, we just use the brackets and put in the index whose value interests us; in this case, we go from 0 to 9, checking each element to see if it's equal to the character a. Note that some of these values may actually be uninitialized since the user might not input a string that fills the whole array--we'll look into how strings are handled in more detail in the next tutorial; for now, the key is simply to understand the power of accessing the array using a numerical index. Imagine how you would write that if you didn't have access to arrays! Oh boy.

Multidimensional arrays are arrays that have more than one index: instead of being just a single line of slots, multidimensional arrays can be thought of as having values that spread across two or more dimensions. Here's an easy way to visualize a two-dimensional array:

```
[][][][][]
[][][][][]
[][][][][]
[][][][][]
[][][][][]
```

The syntax used to actually declare a two dimensional array is almost the same as that used for declaring a one-dimensional array, except that you include a set of brackets for each dimension, and include the size of the dimension. For example, here is an array that is large enough to hold a standard checkers board, with 8 rows and 8 columns:

```
int two_dimensional_array[8][8];
```

You can easily use this to store information about some kind of game or to write something like tic-tac-toe. To access it, all you need are two variables, one that goes in the first slot and one that goes in the second slot. You can make three dimensional, four dimensional, or even higher dimensional arrays, though past three dimensions, it becomes quite hard to visualize.

Setting the value of an array element is as easy as accessing the element and performing an assignment. For instance,

```
<arrayname>[<arrayindexnumber>] = <value>
```

for instance,

```
/* set the first element of my_first to be the letter c */
my_string[0] = 'c';
```

or, for two dimensional arrays

```
<arrayname>[<arrayindexnumber1>][<arrayindexnumber2>] =
<whatever>;
```

Let me note again that you should never attempt to write data past the last element of the array, such as when you have a 10 element array, and you try to write to the [10] element. The memory for the array that was allocated for it will only be ten locations in memory, (the elements 0 through 9) but the next location could be anything. Writing to random memory could cause unpredictable effects--for example you might end up writing to the video buffer and change the video display, or you might write to memory being used by an open document and altering its contents. Usually, the operating system will not allow this kind of reckless behavior and will crash the program if it tries to write to unallocated memory.

You will find lots of useful things to do with arrays, from storing information about certain things under one name, to making games like tic-tac-toe. We've already seen one example of using loops to access arrays; here is another, more interesting, example!

```
#include <stdio.h>

int main()
{
  int x;
  int y;
  int array[8][8]; /* Declares an array like a chessboard */

  for ( x = 0; x < 8; x++ ) {
    for ( y = 0; y < 8; y++ )
      array[x][y] = x * y; /* Set each element to a value */
  }
  printf( "Array Indices:\n" );
  for ( x = 0; x < 8;x++ ) {
    for ( y = 0; y < 8; y++ )
    {
        printf( "[%d][%d]=%d", x, y, array[x][y] );
    }
    printf( "\n" );
  }
  getchar();
}
```

Just to touch upon a final point made briefly above: arrays don't require a reference operator (the ampersand) when you want to have a pointer to them. For example:

```
char *ptr;
char str[40];
ptr = str;  /* Gives the memory address without a reference
operator(&) */
```

As opposed to

```
int *ptr;
int num;
ptr = &num; /* Requires & to give the memory address to the ptr */
```

The fact that arrays can act just like pointers can cause a great deal of confusion.

Lesson 9: C Strings

This lesson will discuss C-style strings, which you may have already seen in the array tutorial. In fact, C-style strings are really arrays of chars with a little bit of special sauce to indicate where the string ends. This tutorial will cover some of the tools available for working with strings--things like copying them, concatenating them, and getting their length.

What is a String?

Note that along with C-style strings, which are arrays, there are also string literals, such as "this". In reality, both of these string types are merely just collections of characters sitting next to each other in memory. The only difference is that you cannot modify string literals, whereas you can modify arrays. Functions that take a C-style string will be just as happy to accept string literals unless they modify the string (in which case your program will crash). Some things that might look like strings are not strings; in particular, a character inclosed in single quotes, like this, 'a', is not a string. It's a single character, which can be assigned to a specific location in a string, but which cannot be treated as a string. (Remember how arrays act like pointers when passed into functions? Characters don't, so if you pass a single character into a function, it won't work; the function is expecting a char*, not a char.)

To recap: strings are arrays of chars. String literals are words surrounded by double quotation marks.

```
"This is a static string"
```

Remember that special sauce mentioned above? Well, it turns out that C-style strings are always terminated with a null character, literally a '\0' character (with the value of 0), so to declare a string of 49 letters, you need to account for it by adding an extra character, so you would want to say:

```
char string[50];
```

This would declare a string with a length of 50 characters. Do not forget that arrays begin at zero, not 1 for the index number. In addition, we've accounted for the extra with a null character, literally a '\0' character. It's important to remember that there will be an extra character on the end on a string, just like there is always a period at the end of a sentence. Since this string terminator is unprintable, it is not counted as a letter, but it still takes up a space. Technically, in a fifty char array you could only hold 49 letters and one null character at the end to terminate the string.

Note that something like

```
```

```
char *my_string;
```

can also be used as a string. If you have read the tutorial on pointers, you can do something such as:

```
arry = malloc( sizeof(*arry) * 256 );
```

which allows you to access arry just as if it were an array. To free the memory you allocated, just use free:

For example:

```
free ( arry );
```

Using Strings

Strings are useful for holding all types of long input. If you want the user to input his or her name, you must use a string. Using scanf() to input a string works, but it will terminate the string after it reads the first space, and moreover, because scanf doesn't know how big the array is, it can lead to "buffer overflows" when the user inputs a string that is longer than the size of the string (which acts as an input "buffer").

There are several approaches to handling this problem, but probably the simplest and safest is to use the fgets function, which is declared in stdio.h.

The prototype for the fegets function is:

```
char *fgets (char *str, int size, FILE* file);
```

There are a few new things here. First of all, let's clear up the questions about that funky FILE* pointer. The reason this exists is because fgets is supposed to be able to read from any file on disk, not just from the user's keyboard (or other "standard input" device). For the time being, whenever we call fgets, we'll just pass in a variable called stdin, defined in stdio.h, which refers to "**st**an**d**ard **in**put". This effectively tells the program to read from the keyboard. The other two arguments to fgets, str and size, are simply the place to store the data read from the input and the size of the char*, str. Finally, fgets returns str whenever it successfully read from the input.

When fgets actually reads input from the user, it will read up to size - 1 characters and then place the null terminator after the last character it read. fgets will read input until it either has no more room to store the data or until the user hits enter. Notice that fgets may fill up the entire space allocated for str, but it will never return a non-null terminated string to you.

Let's look at an example of using fgets, and then we'll talk about some pitfalls to watch out for.

For a example:

```
#include <stdio.h>

int main()
{
    /* A nice long string */
    char string[256];

    printf( "Please enter a long string: " );

    /* notice stdin being passed in */
    fgets ( string, 256, stdin );

    printf( "You entered a very long string, %s", string );

    getchar();
}
```

Remember that you are actually passing the address of the array when you pass string because arrays do not require an address operator (&) to be used to pass their addresses, so the values in the array string are modified.

The one thing to watch out for when using fgets is that it will include the newline character ('\n') when it reads input unless there isn't room in the string to store it. This means that you may need to manually remove the input. One way to do this would be to search the string for a newline and then replace it with the null terminator. What would this look like? See if you can figure out a way to do it before looking below:

```
char input[256];
int i;

fgets( input, 256, stdin );

for ( i = 0; i < 256; i++ )
{
    if ( input[i] == '\n' )
    {
        input[i] = '\0';
        break;
    }
}
```

Here, we just loop through the input until we come to a newline, and when we do, we replace it with the null terminator. Notice that if the input is less than 256 characters long, the user must have hit enter, which would have included the newline character in the string! (By the way, aside from this example, there are other approaches to solving this problem that use functions from string.h.)

Manipulating C strings using string.h

string.h is a header file that contains many functions for manipulating strings. One of these is the string comparison function.

```
int strcmp ( const char *s1, const char *s2 );
```

strcmp will accept two strings. It will return an integer. This integer will either be:

```
Negative if s1 is less than s2.
Zero if s1 and s2 are equal.
Positive if s1 is greater than s2.
```

Strcmp performs a case sensitive comparison; if the strings are the same except for a difference in cAse, then they're countered as being different. Strcmp also passes the address of the character array to the function to allow it to be accessed.

```
char *strcat ( char *dest, const char *src );
```

strcat is short for "string concatenate"; concatenate is a fancy word that means to add to the end, or append. It adds the second string to the first string. It returns a pointer to the concatenated string. Beware this function; it assumes that dest is large enough to hold the entire contents of src as well as its own contents.

```
char *strcpy ( char *dest, const char *src );
```

strcpy is short for string copy, which means it copies the entire contents of src into dest. The contents of dest after strcpy will be exactly the same as src such that strcmp (dest, src) will return 0.

```
size_t strlen ( const char *s );
```

strlen will return the length of a string, minus the termating character ('\0'). The size_t is nothing to worry about. Just treat it as an integer that cannot be negative, which is what it actually is. (The type size_t is just a way to indicate that the value is intended for use as a size of something.)

Here is a small program using many of the previously described functions:

```
#include <stdio.h>    /* stdin, printf, and fgets */
#include <string.h>   /* for all the new-fangled string functions
*/
```

```
/* this function is designed to remove the newline from the end of
a string
entered using fgets.  Note that since we make this into its own
function, we
could easily choose a better technique for removing the newline.
Aren't
functions great? */
void strip_newline( char *str, int size )
{
    int i;

    /* remove the null terminator */
    for (  i = 0; i < size; ++i )
    {
        if ( str[i] == '\n' )
        {
            str[i] = '\0';

            /* we're done, so just exit the function by returning
*/
            return;
        }
    }
    /* if we get all the way to here, there must not have been a
newline! */
}

int main()
{
    char name[50];
    char lastname[50];
    char fullname[100]; /* Big enough to hold both name and
lastname */

    printf( "Please enter your name: " );
    fgets( name, 50, stdin );

    /* see definition above */
    strip_newline( name, 50 );

    /* strcmp returns zero when the two strings are equal */
    if ( strcmp ( name, "Alex" ) == 0 )
    {
        printf( "That's my name too.\n" );
    }
    else
    {
        printf( "That's not my name.\n" );
    }
    // Find the length of your name
    printf( "Your name is %d letters long", strlen ( name ) );
    printf( "Enter your last name: " );
    fgets( lastname, 50, stdin );
    strip_newline( lastname, 50 );
    fullname[0] = '\0';
    /* strcat will look for the \0 and add the second string
starting at
       that location */
    strcat( fullname, name );     /* Copy name into full name */
    strcat( fullname, " " );      /* Separate the names by a space
*/
    strcat( fullname, lastname ); /* Copy lastname onto the end of
fullname */
    printf( "Your full name is %s\n",fullname );

    getchar();

    return 0;
}
```

Safe Programming

The above string functions all rely on the existence of a null terminator at the end of a string. This isn't always a safe bet. Moreover, some of them, noticeably strcat, rely on the fact that the destination string can hold the entire string being appended onto the end. Although it might seem like you'll never make that sort of mistake, historically, problems based on accidentally writing off the end of an array in a function like strcat, have been a major problem.

Fortunately, in their infinite wisdom, the designers of C have included functions designed to help you avoid these issues. Similar to the way that fgets takes the maximum number of characters that fit into the buffer, there are string functions that take an additional argument to indicate the length of the destination buffer. For instance, the strcpy function has an analogous strncpy function

```
char *strncpy ( char *dest, const char *src, size_t len );
```

which will only copy len bytes from src to dest (len should be less than the size of dest or the write could still go beyond the bounds of the array). Unfortunately, strncpy can lead to one niggling issue: it doesn't guarantee that dest will have a null terminator attached to it (this might happen if the string src is longer than dest). You can avoid this problem by using strlen to get the length of src and make sure it will fit in dest. Of course, if you were going to do that, then you probably don't need strncpy in the first place, right? Wrong. Now it forces you to pay attention to this issue, which is a big part of the battle.

Lesson 10: C File I/O and Binary File I/O

When accessing files through C, the first necessity is to have a way to access the files. For C File I/O you need to use a FILE pointer, which will let the program keep track of the file being accessed. (You can think of it as the memory address of the file or the location of the file).

For example:

```
FILE *fp;
```

To open a file you need to use the fopen function, which returns a FILE pointer. Once you've opened a file, you can use the FILE pointer to let the compiler perform input and output functions on the file.

```
FILE *fopen(const char *filename, const char *mode);
```

In the filename, if you use a string literal as the argument, you need to remember to use double backslashes rather than a single backslash as you otherwise risk an escape character such as \t. Using double backslashes \\ escapes the \ key, so the string works as it is expected. Your users, of course, do not need to do this! It's just the way quoted strings are handled in C and C++.

The modes are as follows:

```
r  - open for reading
w  - open for writing (file need not exist)
a  - open for appending (file need not exist)
r+ - open for reading and writing, start at beginning
w+ - open for reading and writing (overwrite file)
a+ - open for reading and writing (append if file exists)
```

Note that it's possible for fopen to fail even if your program is perfectly correct: you might try to open a file specified by the user, and that file might not exist (or it might be write-protected). In those cases, fopen will return 0, the NULL pointer.

Here's a simple example of using fopen:

```
FILE *fp;
fp=fopen("c:\\test.txt", "r");
```

This code will open test.txt for reading in text mode. To open a file in a binary mode you must add a b to the end of the mode string; for example, "rb" (for the reading and writing modes, you can add the b either after the plus sign - "r+b" - or before - "rb+")

To close a function you can use the function

```
int fclose(FILE *a_file);
```

fclose returns zero if the file is closed successfully.

An example of fclose is

```
fclose(fp);
```

To work with text input and output, you use fprintf and fscanf, both of which are similar to their friends printf and scanf except that you must pass the FILE pointer as first argument. For example:

```
FILE *fp;
fp=fopen("c:\\test.txt", "w");
fprintf(fp, "Testing...\n");
```

It is also possible to read (or write) a single character at a time--this can be useful if you wish to perform character-by-character input (for instance, if you need to keep track of every piece of punctuation in a file it would make more sense to read in a single character than to read in a string at a time.) The fgetc function, which takes a file pointer, and returns an int, will let you read a single character from a file:

```
int fgetc (FILE *fp);
```

Notice that fgetc returns an int. What this actually means is that when it reads a normal character in the file, it will return a value suitable for storing in an unsigned char (basically, a number in the range 0 to 255). On the other hand, when you're at the very end of the file, you can't get a character value--in this case, fgetc will return "EOF", which is a constnat that indicates that you've reached the end of the file. To see a full example using fgetc in practice, take a look at the example here.

The fputc function allows you to write a character at a time--you might find this useful if you wanted to copy a file character by character. It looks like this:

```
int fputc( int c, FILE *fp );
```

Note that the first argument should be in the range of an unsigned char so that it is a valid character. The second argument is the file to write to. On success, fputc will return the value c, and on failure, it will return EOF.

Binary I/O

For binary File I/O you use fread and fwrite.

The declarations for each are similar:

```
size_t fread(void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);

size_t fwrite(const void *ptr, size_t size_of_elements, size_t
number_of_elements, FILE *a_file);
```

Both of these functions deal with blocks of memories - usually arrays. Because they accept pointers, you can also use these functions with other data structures; you can even write structs to a file or a read struct into memory.

Let's look at one function to see how the notation works.

fread takes four arguments. Don't by confused by the declaration of a void *ptr; void means that it is a pointer that can be used for any type variable. The first argument is the name of the array or the address of the structure you want to write to the file. The second argument is the size of each element of the array; it is in bytes. For example, if you have an array of characters, you would want to read it in one byte chunks, so size_of_elements is one. You can use the sizeof operator to get the size of the various datatypes; for example, if you have a variable int x; you can get the size of x with sizeof(x);. This usage works even for structs or arrays. Eg, if you have a variable of a struct type with the name a_struct, you can use sizeof(a_struct) to find out how much memory it is taking up.

e.g.,

```
sizeof(int);
```

The third argument is simply how many elements you want to read or write; for example, if you pass a 100 element array, you want to read no more than 100 elements, so you pass in 100.

The final argument is simply the file pointer we've been using. When fread is used, after being passed an array, fread will read from the file until it has filled the array, and it will return the number of elements actually read. If the file, for example, is only 30 bytes, but you try to read 100 bytes, it will return that it read 30 bytes. To check to ensure the end of file was reached, use the feof function, which accepts a FILE pointer and returns true if the end of the file has been reached.

fwrite is similar in usage, except instead of reading into the memory you write from memory into a file.

For example,

```
FILE *fp;
fp=fopen("c:\\test.bin", "wb");
char x[10]="ABCDEFGHIJ";
fwrite(x, sizeof(x[0]), sizeof(x)/sizeof(x[0]), fp);
```

Lesson 11: Typecasting

Typecasting is a way to make a variable of one type, such as an int, act like another type, such as a char, for one single operation. To typecast something, simply put the type of variable you want the actual variable to act as inside parentheses in front of the actual variable. (char)a will make 'a' function as a char.

For example:

```
#include <stdio.h>

int main()
{
  /* The (char) is a typecast, telling the computer to interpret
the 65 as a
     character, not as a number.  It is going to give the
character output of
     the equivalent of the number 65 (It should be the letter A
for ASCII).
     Note that the %c below is the format code for printing a
single character
   */
  printf( "%c\n", (char)65 );
  getchar();
}
```

One use for typecasting for is when you want to use the ASCII characters. For example, what if you want to create your own chart of all 256 ASCII characters. To do this, you will need to use to typecast to allow you to print out the integer as its character equivalent.

```
#include <stdio.h>

int main()
{
    for ( int x = 0; x < 256; x++ ) {
        /* Note the use of the int version of x to output a number
and the use
         * of (char) to typecast the x into a character which
outputs the
         * ASCII character that corresponds to the current number
         */
        printf( "%d = %c\n", x, (char)x );
    }
    getchar();

}
```

If you were paying careful attention, you might have noticed something kind of strange: when we passed the value of x to printf as a char, we'd already told the compiler that we intended the value to be treated as a character when we wrote the format string as %c. Since the char type is just a small integer, adding this typecast actually doesn't add any value!

So when *would* a typecast come in handy? One use of typecasts is to force the correct type of mathematical operation to take place. It turns out that in C (and other programming languages), the result of the division of integers is itself treated as an integer: for instance, 3/5 becomes 0! Why? Well, 3/5 is less than 1, and integer division ignores the remainder.

On the other hand, it turns out that division between floating point numbers, or even between one floating point number and an integer, is sufficient to keep the result as a floating point number. So if we were performing some kind of fancy division where we didn't want truncated values, we'd have to cast one of the variables to a floating point type. For instance, (float)3/5 comes out to .6, as you would expect!

When might this come up? It's often reasonable to store two values in integers. For instance, if you were tracking heart patients, you might have a function to compute their age in years and the number of heart times they'd come in for heart pain. One operation you might conceivably want to perform is to compute the number of times per year of life someone has come in to see their physician about heart pain. What would this look like?

```

/* magical function returns the age in years */
int age = getAge();
/* magical function returns the number of visits */
int pain_visits = getVisits();

float visits_per_year = pain_visits / age;
```

The problem is that when this program is run, visits_per_year will be zero unless the patient had an awful lot of visits to the doc. The way to get around this problem is to cast one of the values being divided so it gets treated as a floating point number, which will cause the compiler to treat the expression as if it were to result in a floating point number:

```

float visits_per_year = pain_visits / (float)age;
/* or */
float visits_per_year = (float)pain_visits / age;
```

This would cause the correct values to be stored in visits_per_year. Can you think of another solution to this problem (in this case)?

Lesson 14: Accepting command line arguments

In C it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.

The full declaration of main looks like this:

```
int main ( int argc, char *argv[] )
```

The integer, argc is the **arg**ument **c**ount. It is the number of arguments passed into the program from the command line, including the name of the program.

The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available. After that, every element number less than argc is a command line argument. You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

How could this be used? Almost any program that wants its parameters to be set when it is executed would use this. One common use is to write a function that takes the name of a file and outputs the entire text of it onto the screen.

```
#include <stdio.h>

int main ( int argc, char *argv[] )
{
    if ( argc != 2 ) /* argc should be 2 for correct execution */
    {
        /* We print argv[0] assuming it is the program name */
        printf( "usage: %s filename", argv[0] );
    }
    else
    {
        // We assume argv[1] is a filename to open
        FILE *file = fopen( argv[1], "r" );

        /* fopen returns 0, the NULL pointer, on failure */
        if ( file == 0 )
        {
            printf( "Could not open file\n" );
        }
        else
        {
            int x;
            /* read one character at a time from file, stopping at
EOF, which
               indicates the end of the file.  Note that the idiom
of "assign
               to a variable, check the value" used below works
because
               the assignment statement evaluates to the value
assigned. */
            while  ( ( x = fgetc( file ) ) != EOF )
            {
                printf( "%c", x );
            }
        }
        fclose( file );
    }
}
```

This program is fairly short, but it incorporates the full version of main and even performs a useful function. It first checks to ensure the user added the second argument, theoretically a file name. The program then checks to see if the file is valid by trying to open it. This is a standard operation, and if it results in the file being opened, the the return value of fopen will be a valid FILE*; otherwise, it will be 0, the NULL pointer. After that, we just execute a loop to print out one character at a time from the file. The code is self-explanatory, but is littered with comments; you should have no trouble understanding its operation this far into the tutorial. :-)

Lesson 15: Singly linked lists (Printable Version)

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. Specifically, the programmer writes a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type (it has to be a pointer--otherwise, every time an element was created, it would create a new element, infinitely). Each of these individual structs or classes in the list is commonly known as a node or element of the list.

One way to visualize a linked list is as though it were a train. The programmer always stores the first node of the list in a pointer he won't lose access to. This would be the engine of the train. The pointer itself is the connector between cars of the train. Every time the train adds a car, it uses the connectors to add a new car. This is like a programmer using malloc to create a pointer to a new struct.

In memory a linked list is often described as looking like this:

```
----------        ----------
- Data   -        - Data   -
----------        ----------
- Pointer- - - -> - Pointer-
----------        ----------
```

The representation isn't completely accurate in all of its details, but it will suffice for our purposes. Each of the big blocks is a struct that has a pointer to another one. Remember that the pointer only *stores* the memory location of something--it is not that thing itself--so the arrow points to the next struct. At the end of the list, there is nothing for the pointer to point to, so it does not point to anything; it should be a null pointer or a dummy node to prevent the node from accidentally pointing to a random location in memory (which is very bad).

So far we know what the node struct should look like:

```
#include <stdlib.h>

struct node {
  int x;
  struct node *next;
};

int main()
{
    /* This will be the unchanging first node */
    struct node *root;
    /* Now root points to a node struct */
    root = malloc( sizeof(struct node) );

    /* The node root points to has its next pointer equal to a
null pointer
       set */
    root->next = 0;
    /* By using the -> operator, you can modify what the node,
       a pointer, (root in this case) points to. */
    root->x = 5;
}
```

This so far is not very useful for doing anything. It is necessary to understand how to traverse (go through) the linked list before it really becomes useful. This will allow us to store some data in the list and later find it without knowing exactly where it is located.

Think back to the train. Let's imagine a conductor who can only enter the train through the first car and can walk through the train down the line as long as the connector connects to another car. This is how the program will traverse the linked list. The conductor will be a pointer to node, and it will first point to root, and then, if the root's pointer to the next node is pointing to something, the "conductor" (not a technical term) will be set to point to the next node. In this fashion, the list can be traversed. Now, as long as there is a pointer to something, the traversal will continue. Once it reaches a null pointer (or dummy node), meaning there are no more nodes (train cars) then it will be at the end of the list, and a new node can subsequently be added if so desired.

Here's what that looks like:

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

struct node {
  int x;
  struct node *next;
};

int main()
{
    /* This won't change, or we would lose the list in memory */
    struct node *root;
    /* This will point to each node as it traverses the list */
    struct node *conductor;

    root = malloc( sizeof(struct node) );
    root->next = 0;
    root->x = 12;
    conductor = root;
    if ( conductor != 0 ) {
        while ( conductor->next != 0)
        {
            conductor = conductor->next;
        }
    }
    /* Creates a node at the end of the list */
    conductor->next = malloc( sizeof(struct node) );

    conductor = conductor->next;

    if ( conductor == 0 )
    {
        printf( "Out of memory" );
        return 0;
    }
    /* initialize the new memory */
    conductor->next = 0;
    conductor->x = 42;

    return 0;
}
```

That is the basic code for traversing a list. The if statement ensures that the memory was properly allocated before traversing the list. If the condition in the if statement evaluates to true, then it is okay to try and access the node pointed to by conductor. The while loop will continue as long as there is another pointer in the next. The conductor simply moves along. It changes what it points to by getting the address of conductor->next.

Finally, the code at the end can be used to add a new node to the end. Once the while loop as finished, the conductor will point to the last node in the array. (Remember the conductor of the train will move on until there is nothing to move on to? It works the same way in the while loop.) Therefore, conductor->next is set to null, so it is okay to allocate a new area of memory for it to point to (if it weren't NULL, then storing something else in the pointer would cause us to lose the memory that it pointed to). When we allocate the memory, we do a quick check to ensure that we're not out of memory, and then the conductor traverses one more element (like a train conductor moving on the the newly added car) and makes sure that it has its pointer to next set to 0 so that the list has an end. The 0 functions like a period; it means there is no more beyond. Finally, the new node has its x value set. (It can be set through user input. I simply wrote in the '=42' as an example.)

To print a linked list, the traversal function is almost the same. In our first example, it is necessary to ensure that the last element is printed after the while loop terminates. (See if you can think of a better way before reading the second code example.)

For example:

```
conductor = root;
if ( conductor != 0 ) { /* Makes sure there is a place to start */
    while ( conductor->next != 0 ) {
        printf( "%d\n", conductor->x );
        conductor = conductor->next;
    }
    printf( "%d\n", conductor->x );
}
```

The final output is necessary because the while loop will not run once it reaches the last node, but it will still be necessary to output the contents of the next node. Consequently, the last output deals with this. We can avoid this redundancy by allowing the conductor to walk off of the back of the train. Bad for the conductor (if it were a real person), but the code is simpler as it also allows us to remove the initial check for null (if root is null, then conductor will be immediately set to null and the loop will never begin):

```
conductor = root;
while ( conductor != NULL ) {
    printf( "%d\n", conductor->x );
    conductor = conductor->next;
}
```

Lesson 16: Recursion

Recursion is a programming technique that allows the programmer to express operations in terms of themselves. In C++, this takes the form of a function that calls itself. A useful way to think of recursive functions is to imagine them as a process being performed where one of the instructions is to "repeat the process". This makes it sound very similar to a loop because it repeats the same code, and in some ways it *is* similar to looping. On the other hand, recursion makes it easier to express ideas in which the result of the recursive call is necessary to complete the task. Of course, it must be possible for the "process" to sometimes be completed without the recursive call. One simple example is the idea of building a wall that is ten feet high; if I want to build a ten foot high wall, then I will first build a 9 foot high wall, and then add an extra foot of bricks. Conceptually, this is like saying the "build wall" function takes a height and if that height is greater than one, first calls itself to build a lower wall, and then adds one a foot of bricks.

A simple example of recursion would be:

```
void recurse()
{
    recurse(); /* Function calls itself */
}

int main()
{
    recurse(); /* Sets off the recursion */
    return 0;
}
```

This program will not continue forever, however. The computer keeps function calls on a stack and once too many are called without ending, the program will crash. Why not write a program to see how many times the function is called before the program terminates?

```
#include <stdio.h>

void recurse ( int count ) /* Each call gets its own copy of count
*/
{
    printf( "%d\n", count );
    /* It is not necessary to increment count since each
function's
       variables are separate (so each count will be initialized
one greater)
```

```
     */
    recurse ( count + 1 );
}

int main()
{
  recurse ( 1 ); /* First function call, so it starts at one */
  return 0;
}
```

This simple program will show the number of times the recurse function has been called by initializing each individual function call's count variable one greater than it was previous by passing in count + 1. Keep in mind that it is not a function call restarting itself; it is hundreds of function calls that are each unfinished.

The best way to think of recursion is that each function call is a "process" being carried out by the computer. If we think of a program as being carried out by a group of people who can pass around information about the state of a task and instructions on performing the task, each recursive function call is a bit like each person asking the next person to follow the same set of instructions on some part of the task while the first person waits for the result.

At some point, we're going to run out of people to carry out the instructions, just as our previous recursive functions ran out of space on the stack. There needs to be a way to avoid this! To halt a series of recursive calls, a recursive function will have a condition that controls when the function will finally stop calling itself. The condition where the function will not call itself is termed the base case of the function. Basically, it will usually be an if-statement that checks some variable for a condition (such as a number being less than zero, or greater than some other number) and if that condition is true, it will not allow the function to call itself again. (Or, it could check if a certain condition is true and only then allow the function to call itself).

A quick example:

```

void count_to_ten ( int count )
{
    /* we only keep counting if we have a value less than ten
       if ( count < 10 )
       {
           count_to_ten( count + 1 );
       }
}
int main()
{
  count_to_ten ( 0 );
}
```

This program ends when we've counted to ten, or more precisely, when count is no longer less than ten. This is a good base case because it means that if we have an input greater than ten, we'll stop immediately. If we'd chosen to stop when count equalled ten, then if the function were called with the input 11, it would run of of memory before stopping.

Notice that so far, we haven't done anything with the result of a recursive function call. Each call takes place and performs some action that is then ignored by the caller. It is possible to get a value back from the caller, however. It's also possible to take advantage of the side effects of the previous call. In either case, once a function has called itself, it will be ready to go to the next line after the call. It can still perform operations. One function you could write could print out the numbers 123456789987654321. How can you use recursion to write a function to do this? Simply have it keep incrementing a variable passed in, and then output the variable twice: once before the function recurses, and once after.

```
void printnum ( int begin )
{
    printf( "%d", begin );
    if ( begin < 9 )         /* The base case is when begin is no
longer */
    {                           /* less than 9 */
        printnum ( begin + 1 );
    }
    /* display begin again after we've already printed everything
from 1 to 9
     * and from 9 to begin + 1 */
    printf( "%d", begin );
}
```

This function works because it will go through and print the numbers begin to 9, and then as each printnum function terminates it will continue printing the value of begin in each function from 9 to begin.

This is, however, just touching on the usefulness of recursion. Here's a little challenge: use recursion to write a program that returns the factorial of any number greater than 0. (Factorial is number*number-1*number-2...*1).

Hint: Your function should recursively find the factorial of the smaller numbers first, i.e., it takes a number, finds the factorial of the previous number, and multiplies the number times that factorial...have fun. :-)

Lesson 17: Functions with variable-length argument lists

Perhaps you would like to have a function that will accept any number of values and then return the average. You don't know how many arguments will be passed in to the function. One way you could make the function would be to accept a pointer to an array. Another way would be to write a function that can take any number of arguments. So you could write avg(4, 12.2, 23.3, 33.3, 12.1); or you could write avg(2, 2.3, 34.4); The advantage of this approach is that it's much easier to change the code if you want to change the number of arguments. Indeed, some library functions can accept a variable list of arguments (such as printf--I bet you've been wondering how that works!).

Whenever a function is declared to have an indeterminate number of arguments, in place of the last argument you should place an ellipsis (which looks like '...'), so, int a_function (int x, ...); would tell the compiler the function should accept however many arguments that the programmer uses, as long as it is equal to at least one, the one being the first, x.

We'll need to use some macros (which work much like functions, and you can treat them as such) from the stdarg.h header file to extract the values stored in the variable argument list--va_start, which initializes the list, va_arg, which returns the next argument in the list, and va_end, which cleans up the variable argument list.

To use these functions, need need a variable capable of storing a variable-length argument list--this variable will be of type va_list. va_list is like any other type. For example, the following code declares a list that can be used to store a variable number of arguments.

```
va_list a_list;
```

va_start is a macro which accepts two arguments, a va_list and the name of the variable that directly precedes the ellipsis ("..."). So in the function a_function, to initialize a_list with va_start, you would write va_start (a_list, x);

```
int a_function ( int x, ... )
{
    va_list a_list;
    va_start( a_list, x );
}
```

va_arg takes a va_list and a variable type, and returns the next argument in the list in the form of whatever variable type it is told. It then moves down the list to the next argument. For example, va_arg (a_list, double) will return the next argument, assuming it exists, in the form of a double. The next time it is called, it will return the argument following the last returned number, if one exists. Note that you need to know the type of each argument--that's part of why printf requires a format string! Once you're done, use va_end to clean up the list: va_end(a_list);

To show how each of the parts works, take an example function:

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

double average ( int num, ... )
{
    va_list arguments;
    double sum = 0;

    /* Initializing arguments to store all values after num */
    va_start ( arguments, num );
    /* Sum all the inputs; we still rely on the function caller to
tell us how
     * many there are */
    for ( int x = 0; x < num; x++ )
    {
        sum += va_arg ( arguments, double );
    }
    va_end ( arguments );                  // Cleans up the list

    return sum / num;
}

int main()
{
    printf( "%f\n", average ( 3, 12.2, 22.3, 4.5 ) );
    printf( "%f\n", average ( 5, 3.3, 2.2, 1.1, 5.5, 3.3 ) );
}
```

It isn't necessarily a good idea to use a variable argument list at all times; the potential exists for assuming a value is of one type, while it is in fact another, such as a null pointer being assumed to be an integer. Consequently, variable argument lists should be used sparingly.

Binary Trees: Part 1

The binary tree is a fundamental data structure used in computer science. The binary tree is a useful data structure for rapidly storing sorted data and rapidly retrieving stored data. A binary tree is composed of parent nodes, or leaves, each of which stores data and also links to up to two other child nodes (leaves) which can be visualized spatially as below the first node with one placed to the left and with one placed to the right. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure. It is the leaf on the left which has a lesser key value (ie, the value used to search for a leaf in the tree), and it is the leaf on the right which has an equal or greater key value. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values. More importantly, as each leaf connects to two other leaves, it is the beginning of a new, smaller, binary tree. Due to this nature, it is possible to easily access and insert data in a binary tree using search and insert functions recursively called on successive leaves.

The typical graphical representation of a binary tree is essentially that of an upside down tree. It begins with a root node, which contains the original key value. The root node has two child nodes; each child node might have its own child nodes. Ideally, the tree would be structured so that it is a perfectly balanced tree, with each node having the same number of child nodes to its left and to its right. A perfectly balanced tree allows for the fastest average insertion of data or retrieval of data. The worst case scenario is a tree in which each node only has one child node, so it becomes as if it were a linked list in terms of speed. The typical representation of a binary tree looks like the following:

```
          10
        /    \
       6      14
      / \    /  \
     5   8  11  18
```

The node storing the 10, represented here merely as 10, is the root node, linking to the left and right child nodes, with the left node storing a lower value than the parent node, and the node on the right storing a greater value than the parent node. Notice that if one removed the root node and the right child nodes, that the node storing the value 6 would be the equivalent a new, smaller, binary tree.
The structure of a binary tree makes the insertion and search functions simple to implement using recursion. In fact, the two insertion and search functions are also both very similar. To insert data into a binary tree involves a function searching for an unused node in the proper position in the tree in which to insert the key value. The insert function is generally a recursive function that continues moving down the levels of a binary tree until there is an unused leaf in a position which follows the rules of placing nodes. The rules are that a lower value should be to the left of the node, and a greater or equal value should be to the right. Following the rules, an insert function

should check each node to see if it is empty, if so, it would insert the data to be stored along with the key value (in most implementations, an empty node will simply be a NULL pointer from a parent node, so the function would also have to create the node). If the node is filled already, the insert function should check to see if the key value to be inserted is less than the key value of the current node, and if so, the insert function should be recursively called on the left child node, or if the key value to be inserted is greater than or equal to the key value of the current node the insert function should be recursively called on the right child node. The search function works along a similar fashion. It should check to see if the key value of the current node is the value to be searched. If not, it should check to see if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node, or if it is greater than the value of the node, it should be recursively called on the right child node. Of course, it is also necessary to check to ensure that the left or right child node actually exists before calling the function on the node.
Because binary trees have log (base 2) n layers, the average search time for a binary tree is log (base 2) n. To fill an entire binary tree, sorted, takes roughly log (base 2) n * n. Lets take a look at the necessary code for a simple implementation of a binary tree. First, it is necessary to have a struct, or class, defined as a node.

```
struct node
{
  int key_value;
  struct node *left;
  struct node *right;
};
```

The struct has the ability to store the key_value and contains the two child nodes which define the node as part of a tree. In fact, the node itself is very similar to the node in a linked list. A basic knowledge of the code for a linked list will be very helpful in understanding the techniques of binary trees. Essentially, pointers are necessary to allow the arbitrary creation of new nodes in the tree.

There are several important operations on binary trees, including inserting elmeents, searching for elements, removing elements, and deleting the tree. We'll look at three of those four operations in this tutorial, leaving removing elements for later.

We'll also need to keep track of the root node of the binary tree, which will give us access to the rest of the data:

```
struct node *root = 0;
```

It is necessary to initialize root to 0 for the other functions to be able to recognize that the tree does not yet exist. The destroy_tree shown below which will actually free all of the nodes of in the tree stored under the node leaf: tree.

```
void destroy_tree(struct node *leaf)
{
  if( leaf != 0 )
  {
      destroy_tree(leaf->left);
      destroy_tree(leaf->right);
      free( leaf );
  }
}
```

The function destroy_tree goes to the bottom of each part of the tree, that is, searching while there is a non-null node, deletes that leaf, and then it works its way back up. The function deletes the leftmost node, then the right child node from the leftmost node's parent node, then it deletes the parent node, then works its way back to deleting the other child node of the parent of the node it just deleted, and it continues this deletion working its way up to the node of the tree upon which delete_tree was originally called. In the example tree above, the order of deletion of nodes would be 5 8 6 11 18 14 10. Note that it is necessary to delete all the child nodes to avoid wasting memory.

The following insert function will create a new tree if necessary; it relies on pointers to pointers in order to handle the case of a non-existent tree (the root pointing to NULL). In particular, by taking a pointer to a pointer, it is possible to allocate memory if the root pointer is NULL.

```
insert(int key, struct node **leaf)
{
    if( *leaf == 0 )
    {
        *leaf = (struct node*) malloc( sizeof( struct node ) );
        (*leaf)->left->key_value = key;
        /* initialize the children to null */
        (*leaf)->left->left = 0;
        (*leaf)->left->right = 0;
    }
    else if(key < (*leaf)->key_value)
    {
        insert( key, &(*leaf)->left );
    }
    else if(key > (*leaf)->key_value)
    {
        insert( key, &(*leaf)->right );
    }
}
```

The insert function searches, moving down the tree of children nodes, following the prescribed rules, left for a lower value to be inserted and right for a greater value, until it reaches a NULL node--an empty node--which it allocates memory for and initializes with the key value while setting the new node's child node pointers to NULL. After creating the new node, the insert function will no longer call itself. Note, also, that if the element is already in the tree, it will not be added twice.

```
struct node *search(int key, struct node *leaf)
{
  if( leaf != 0 )
  {
      if(key==leaf->key_value)
      {
          return leaf;
      }
      else if(key<leaf->key_value)
      {
          return search(key, leaf->left);
      }
      else
      {
          return search(key, leaf->right);
      }
  }
  else return 0;
}
```

The search function shown above recursively moves down the tree until it either reaches a node with a key value equal to the value for which the function is searching or until the function reaches an uninitialized node, meaning that the value being searched for is not stored in the binary tree. It returns a pointer to the node to the previous instance of the function which called it.