Cuddletech Programming Series

Intermidiary C Programming

Ben Rockwood, Cuddletech
<benr@cuddletech.com>

Table of Contents

Intro

Learning C can be as easy or difficult as you want it to be. This chapter is intended to be a quick refresher and guide thru the basics of C. Hopefully this will fill in any gaps new coders have and get them on their way to EFL bliss.

Simple Variable Types

C offers a variety of diffrent variable types to suit our programming needs, the most commonly used are: int, char, double, and float. Other type of variables include short, long, and void. On some systems, usually 64bit machines you'll see other types like long long, which is just what it sounds like, a long long. The following are examples of each type of variable being delared and set:

```
int var;
var = 1;

char letter;
letter = 'a';
```

```
float var;
var = 1.323;

double var;
var = 32.23;
```

Really, these mix of types looks more complicated than it is. An int is simply an integer. An integer is typically a 16 bit value. Shorts and longs are simply diffrent types of ints, an 8 bit int for a short and a 32 bit int for a long. Int's come in two varieties, signed and unsigned. By default all integers are signed, meaning that one bit is reserved to mark whether the value is positive or negative. An unsigned integer simply doesn't have that bit reserved, which means that you can store slightly larger numbers in an unsigned int but you also can only have positive values. Because shorts and long, signed and unsigned are modifications of an integer, you can use declarations like the following:

```
long int         var;
short int         var;
long long         var;
unsigned short var;
unsigned int var;
```

The next class of vars are floating point variables, meaning that it has a decimal in the number. Doubles are double precision floating points, meaning that there are 2 significant decimal places. 1234.56778 is a float, and 1234.22 is a double. If you try to store the number 1.234 in a double the value 1.23 will be stored, the 4 will be dropped.

The final two types are void and char. As you might expect, a void typed variable has no type. And a char is a variable type that stores only a single charrector. C has no type for strings, only for single charectors. Therefore in order to utilize strings we use arrays of chars.

An array is a variable that can store multiple values. It is defined like any other variable but the varname is appended with brackets containing the number of values it can store, the array length. Each value stored is known as an array member. Therefore "int number[5]" defines a signed integer array with 5 members. And "char alphabet[24]" is a charrector array with 24 members. Here is an example of defining and accessing an array and its members:

```
int         myvar[3];
```

```
myvar[2] = 1;
myvar[0] = 10023;
```

Variables can be acted upon in diffrent ways for mathmatical operations and for logic. Charrectors in C that have special meaning are called operators. Operators that change the meaning of a variable are called unary operators. The standard list of mathmatical operators exist such as addition (+), subtraction (-), mutliplication (*), division (/), and modulus (%). [NOTE: modulus is division, but the result is only the remainder, therefore 11 % 5 = 1, because 5 goes into 11 twice, with a remainder of 1. So 10 % 5 would be 0 because its an even division.] Mathmatical operators can be used with the assignment operator (=) as well, for instance if you want to set a variable to the result of that variable plus something else you could write either "var = var + 2;" or "var += 2", it's the same thing.

Unary operators modify the meaning of the variable they are prepended to. Various unary operators are use to indicate a value is positive (+var), negative (-var), a derefrence (*var), an address (&), one more than it's value (++var), one less than it's value (--var), etc.

And yet other operators are used for grouping and logic. Be sure to differentiate the diffrence between unary and non-unary operators, because they can be easily confused. You can see that "var = vara +varb" is diffrent from "var = vara + varb".

As mentioned earlier, when preforming mathmatical operations, all variables involved must have the same type. Sometimes this isn't possible. Therefore you can "cast" a variable. Casting is a type of unary operator that tells the compiler what kind of variable it should be treated as. Look at the following example where we cast a float and a double into an int for the operation:

```
float         varA;
double varB;
int         result;

varA = 1.2345;
varB = 23.11;

result = (int)varA + (int)varB;
/* result is 24 */
```

Outside of these variable types you can create your own types, using a typedef. A typedef is exactly what is sound like, a type definition. It is used to make the re-occurance of a type easier to create and is typically used with

structures. A simple example would be "typedef int number;", which could create a new variable type called "number", which would be an integer. You would then create a new variable of type number like usual: "number var;"

A term you may hear is "operand". An operand is the entity upon which an operator acts. An operand can therefor be a variable or a function.

While it is often uncommon, you can actually set an initial value of a variable when you declare it, like this "int result = 10;". When a variable is created it is not zeroed out, and may initial have whatever was last in that memory location previous to your program, therefore you may have serious problems if you try to add 2 varables together one of which was never defined.

Pointers

Pointers are mystical confusing things, or so people believe. The first thing to unlearn about pointers is that *var is not a pointer. Too many new coders come to some how think that anytime you see *var you are looking at a pointer, and its a misconception that will only hurt you. A pointer is just that, a pointer from one thing to another. A pointer itself is simply an int that instead of containing a user defined value contains a memory address. That memory address is the address of the think to which the pointer is pointing. Two special unary operators are used to work with pointers * and &. The * unary operator is called a dereference and is used on a pointer. The & is a unary operator that returns the address of a non-pointer operand. Notice, the * unary operator is used ON a pointer and the & unary operator is used on a non-pionter, neither denotes a pointer! Because a pointer is an int with a special purpose it will look like an int. Typically convensions are used in namely of variables to help distinguish that something is a pointer, calling a pointer either "ptr_variable" or "variable_p". The * operator has two diffrent contexts, and this is where people get confused. The following statements initialize and set a pointer:

```
int * ptr_var;
int   number;
int   result;

number = 13;

ptr_var = &number;

result = *ptr_var; /* result now holds "13" */
```

In this example you can see that we first declare a new integer pointer. We

use an int type because this pointer will be pointing to an int. If it were going to point to a char, we'd use the type char ("char * ptr_var;") instead. Also notice that each of the following three declaration types are the same:

```
int         *ptr_var;
int  * ptr_var;
int* ptr_var;
```

I tend to prefer the second method because the is no way it will be confused for a derefernce. In our earlier example you see the & unary operator being used on the int variable "number", this will be replaced with the memory address of the variable "number". If you were to print the value of ptr_var it'd look something like this "-1073743696". Next, we see our pointer being used with the * unary operator. The * operator causes the true value of ptr_var (the memory address) to be replaced with the value that is stored at that address. Because ptr_var is a reference to the memory location of the variable "number" when you de-reference the pointer you display the value of "number", in this case 13.

I want to reemphisis some points at this point. First, notice how unary operators work. When a unary operator is used on an operand (variable) it changes the meaning of that operand. So even though you see "result = *ptr_var;" the code is really saying "result = 13" in this case. And secondly, that when you use the * operator in variable declaration it is not being used as a unary operator, it is being used as a type decloration, just like "signed int" instead of "int", describing what type of int it really is. In the begining, make sure you don't confuse the two uses of *.

Lets expand on the previous example, to illistrate that it really is a pointing:

```
int * ptr_var;
int   number = 13;
int   result;

ptr_var = &number;

result = *ptr_var;     /* result now holds "13" */

number = 20;

result = *ptr_var; /* result now holds 20 */
```

We'll descuse pointers more with functions.

Complex Variables Types

There are 3 more types of varaibles we haven't discused: structures, unions. and enumerations. Structures are variables that store multiple variables, each with diffrent types. In a single struct we could hold ints, floats, chars, etc. Structures are typically used to group diffrent variables that serve a similar purpose together, which especially makes passing variables to a function much easier because you just pass a single struct instead of 10 diffrent variables. The following is any example of a struct being declared:

```
struct typename {
        int         member1;
        int         member2;
        char         string[];
        float member4;
} varname1, varname2;
```

In this creation we have the type struct, followed by a name for the structure, but note this name is to describe this style of struct, not the variable struct name. After the final brace (}) you see two variable names, varname1 and varname2, which are the actual names of your variables. Inside the braces are the members of the struct. Each member can be access by using the name of the struct var followed by a . and the member varname.

```
struct videocard {
        int         modelnumber;
        char maker[];
        char name[];
        char chipset[];
        double cost;
} radeon, gforce, matrox;

struct videocard  sun;

radeon.modelnumber = 1234;
matrox.cost = 200.45;
sun.cost = matrox.cost;
```

You can see that we've defined a videocard structure, with 5 members. Following the close brace we create 3 diffrent structs: radeon, gforce, and matrox. On the following line we create yet another structre called sun, which will be just like the first 3 structs. Notice that it's syntax is the same as the first struct declariation if you just take the braces away. The following lines

set the value of various struct members.

Unions are rare to see.

Functions

C is a procedural programming lanuage. It reads from the top of the file downward, processing line by line unless told to go some place else, and once it's done with that it comes back to where it left off and keeps going. Every C program must have at least one function: main(). main() is always the first function called. Functions can take arguments but don't have to. Functions also can return a value, but doesn't have to. The main function typically looks like this:

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

 /* code goes here */

 return 0;
}
```

The first line here describes the function, it's called main, it returns an int to the caller and takes 2 arguments as enclosed in (), an int and a char array. Main is typically called by your shell when you execute a program, and the shell passes along these two arguments, argc is the number of arguments you gave on the command line and argv is a pointer to an array of arguments themselves. Thus if you started this program with "testprog -c argument", argv[0] would be the first argument which is actually the command name itself "testprog", and argv[1] is the next argument "-c", and argv[2] would be "argument", thus the value of argc would be 3 because there are 3 arguments on the command line. The code in the function is enclosed in curly brackets called a block. Variables in C can only be called at the start of a block or start of the file before a block is entered. Variables are only usable within the function in which they were created, this is called a variables scope. This is why you must pass arguments to functions, there are no global variables in C, except that variables declared before the main block at the top of the file will be availbe to all functions thereafter. The "return 0;" statement returns a value to the calling function, the shell in this case, which we can see in the function definition is an int that gets returned. You can use the return function at any time to leave the block and return the desired value to the caller. In the UNIX shell, this return value tells the shell if the program exited cleanly, a return value of 0 means there were no problems. You can see the return value of a program after it finishes executing by viewing the $? enviromental variable

(echo $?).

Here is an example of main calling another function, passing arguments and taking a return value:

```
#include <stdio.h>

int sum(int a, int b){
        int         c;

        c = a + b;

        return(c);
}

int main(){
        int number1;
        int number2;
        int result;

        number1 = 20;
        number2 = 10;

        result = sum(number1, number2);

        printf("The sum of %d and %d is %d\n", number1, number2,

        return 0;
}
```

This code uses 3 functions: main(), sum(), and printf(). In this case main accepts no arguments, because we don't need argc and argv for this code. The printf() function is provided by the standard C library, and it is used to print text to STDOUT, in most cases the terminal it was called from. We'll discuse printf() more in a moment. The sum() function defined in our code returns an int and takes two arguments both ints. The variable name following the type is the name of the value that will be used in the function it is defining. Therefore when sum() is called the values number1 and number2 are passed to the function, and then inside the sum() function int a will have the value 20 and int b will have the value 10.

Notice that the sum() function is defined before main(), this is because all functions must be declaired before they are used.