TOPICS IN C PROGRAMMING

By Bob Hain (ME Net)

Introduction

This document is not intended to be a text on C programming. Because many of you may not have had the opportunity to use or practice C programming, we are attempting to provide a brief description of some of the elements of C which you will need in your laboratory work. We will leave out many topics but will try to provide simple, although sometimes incomplete, explanations of some of the basic elements of C.

Why C?

The computer industry is changing rapidly. Although changes in hardware are easier to observe, changes in the software environment are no less striking. The FORTRAN and BASIC programming languages have served the scientific community for many years. These language are highly optimized for numerical calculation and are still in wide-spread use. But with the introduction of small powerful computers, software needs began to change. These computers were applied to many tasks not solely based on numerical manipulation. Two examples of such applications are the acquisition of experimental data and control of the experimental process. The FORTRAN and BASIC languages were extended to address many of these changing needs, but modern languages such as C began to spring into use. The speed with which C has developed has made it impossible for the University to introduce it early in the curriculum and to build on it throughout your education. Because of C's acceptance in industry and research institutions, we feel that you should be exposed to it. We do not expect you to become an expert in C programming. In fact, we stress that this course is about experimental techniques for heat transfer studies. An introduction to C programming falls within the scope of such a course, but it is not its main objective. We hope that you will find this experience pleasant and rewarding.

A Simple C Program

    main()
    {
    }

It has no input, no output, and does nothing. I didn't claim it was a useful program, but it is the simplest one I could think of. What is demonstrated here are the minimum requirements of a C program. All C programs must have one function called main. The syntax of a function consists of a name followed by a set of parentheses and a set of braces. The braces delimit a group of statements (null in this case). We will encounter many braces.
The program above could have been written as:

```c
main() { }
```

This is equally acceptable to the C compiler, but it is not good style. While I will not make many explicit comments about style, try to be aware of the issue.

**Formatted Output**

The next C program is often the first one people ever see. It is the first program presented in The C Programming Language by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978).

```c
main()
{
    printf("hello, world\n");
}
```

The output of this program is:

```
hello, world
```

While this program is only slightly more utilitarian than the last (it at least has output), it demonstrates a few more features of C programming. The printf() statement provides output to the screen. All statements must end in a semicolon! The most common error in C programming is to omit the semicolon. Such an omission causes the compiler to go berserk, reporting some sort of nonsensical error message a line or two later in the program. The first thing to do when things go wrong is to check that all your statements end with a semicolon. Another point to observe is that, ignoring all the stuff in the parentheses and the semicolon on the end, the statement reduces to printf(). This, of course, is a function. C is a very simple language that has very few built-in features. This does not imply that it is a limited language. The simplicity of C is its strength. A great amount of C programming consists of calling functions. These functions usually are written in C and serve to make programming easier. The concept of building ever more complex programs by assembling lots of simple statements is common in the computer world, and C embraces this concept.

Because output and formatting are so important in computer programming, it would be good to explain the printf() function and the related functions fprintf() and sprintf(). All three of these functions perform similar tasks. printf() formats output and writes it to the screen, fprintf() writes to a file, and sprintf() stores its output in an area (array) of memory. For now, the important thing to understand is how to specify how the output will be formatted. We will examine how this is accomplished using the printf() function as an example. The function's syntax is:
printf("control", arg1, arg2, ...);

The idea is that printf() will format the arguments according to the control string. After a look at the control string syntax, we can try an example or two. Most of the characters in a control string represent themselves, as in the "hello, world\n" example. But what about the \n part?

- \n stands for a new line
- \t stands for a tab
- \b stands for a backspace
- \0 stands for a null character (the utility of this will be explained later)

Another important character to watch out for is the % symbol. In the control string, this character introduces a conversion for one of the arguments. There is a one-to-one correspondence between the % characters and the arguments.

- %c prints a single character
- %s prints a string of characters
- %d prints the decimal value of the integer argument
- %o prints the octal value of the integer argument
- %x prints the hexadecimal value of the integer argument
- %f prints the floating point argument as [-]mmm.nnnnn
- %e prints the floating point argument as [-]m.nnnnnnE[/-]xx+
- %g uses %e or %f, whichever is shorter

Using what we already know, we can produce quite a bit of formatted output. One can embellish the output further by controlling the field width and the precision of the conversion as follows:

   %field_width.precisionf

Confused yet? Well, maybe some examples will help.

  main()
  {
    int i;
double f;

i = 16;
printf("The decimal value of i is %d\n",i);
printf("The octal value of i is %o\n",i);
printf("The hexadecimal value of i is %x\n",i);

f = 100.0 / 3.0;
printf("f = %f\n",f);
printf("f = %e\n",f);
printf("f = |%10.3f|\n",f);
}

The output of this program is:

The decimal value of i is 16
The octal value of i is 20
The hexadecimal value of i is 10
f = 33.333333
f = 3.333333e+01
f = |    33.333|

The vertical bars in the last formatting example surround the output field.

Declaration of Variables

In addition to the formatting examples, several new constructs have been added to the last program. Immediately after the first brace are two lines of code that state what variables are going to be used and what type they are. The variable i is an integer and the variable f is a double precision floating point number. Because of the way C handles mathematical functions, use double instead of float. (You don't want me to explain this point ...Trust me.) The C compiler will force you to declare all variables.

Mathematics

What do you think of a computer language that only knows how to add, subtract, multiply and divide? This may seem rather limiting at first but, as always, there is a way around this apparent limitation. The last program presented showed how the built-in divide (/) operator worked. But what about things like logs, powers and other mathematical operations?

#include <stdio.h>
```c
#include <math.h>
main()
{
    double f, result;

    f = sqrt(2.0);

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

    result = pow(f, 2.0);
    printf("f raised to the power of 2 = %f\n", result);

    result = exp(2 * log(f));
    printf("exp( 2 * log (f)) = %f\n", result);
}
```

The output of this program is:

```
f * f = 2.000000
f raised to the power of 2 = 2.000000
exp( 2 * log (f)) = 2.000000
```

**The Preprocessor**

The last example is fairly self explanatory, except for those funny looking lines at the top of the program that begin with `#include`. These lines tell the preprocessor to include other files (squirreled away with all the compiler parts) into your file before actual compilation. Header files (can you guess why they end in .h?) contain definitions of variables, macros, and functions that might be needed by your program. In the case of math.h, the file contains the information pertinent to the use of the math functions. The stdio.h file should have been included all along, but the C compiler was clever enough to get it when it saw the printf() function. This omission was a little slight of hand, but I didn't want to overwhelm you in the first example. The preprocessor has some other useful features that you should keep in mind. Let's look at another example.

```c
#include <stdio.h>
#include <math.h>
#define START 0
#define END 90
#define STEP 10
#define DEG2RAD (3.14159 / 180)
main()
```
double f;

for (f = START; f <= END; f = f + STEP){
    printf("Sin of %2.0f deg. is %f\n", f, sin( f * DEG2RAD));
}
}

The output of this program is:

Sin of 00 deg. is 0.000000
Sin of 10 deg. is 0.173648
Sin of 20 deg. is 0.342020
Sin of 30 deg. is 0.500000
Sin of 40 deg. is 0.642787
Sin of 50 deg. is 0.766044
Sin of 60 deg. is 0.866025
Sin of 70 deg. is 0.939692
Sin of 80 deg. is 0.984808
Sin of 90 deg. is 1.000000

In addition to the #include statements, there are several #define statements. The syntax of the define statement is:

#define identifier replacement-string

The preprocessor replaces all subsequent instances of identifier with the replacement-string.

Loops

The last example included a loop. Looping is a basic computer programming technique, regardless of the language. For C language, the most common type of loop is the for loop. Let's examine its syntax.

for(initializations; test; end of loop processing){
    statement_1;
    .
    .
    statement_n;
}
The initialization part of the loop is done before the loop is started. The test portion of the loop is actually a while test. The end of loop processing occurs at the end of each loop, but before the test. In the last example program, f was initialized to START (which had been replaced by the preprocessor with a 0). At the end of each loop, STEP (10) was added to f. Then the while test was applied: while f was <= (less than or equal to) END (90), the loop was executed again. Remember from your language studies that languages have idioms. C language is not an exception. A C idiom worth noting is:

```c
for(;;){
    statement_1;
    ...
    statement_n;
}
```

This is a loop with no initializations, no test and no end of loop processing. It is a forever loop, and must be broken by one of the statements inside. Here is an example.

```c
#include <stdio.h>
main()
{
    int i;
    i = 0;
    for(;;){
        i = i + 1;
        if ( i > 3 ){
            break;
        }
        printf("The loop is still running. i = %d\n",i);
    }
}
```

The output of this program is:

```
The loop is still running. i = 1
The loop is still running. i = 2
The loop is still running. i = 3
```

The last example also introduced the if construction and the break statement.

**Arrays and Pointers**

I am going to introduce several related topics at once. You will use the concepts presented here in your laboratory work. The most common communication method between the computer and the instruments you will be using is sending and receiving
series of characters. Often times the series will contain characters that are not wanted and portions of the series must be removed. The model of sending and receiving characters through a common interface (the IEEE-488 bus) allows a programmer to learn to communicate with one instrument and to generalize that knowledge to communicating with other instruments. The series of characters that you send to the various instruments and the format of the character series the instruments return is dependent on the particular instrument with which you are communicating, and will be referred to as being device dependent. The basic input/output format for sending and receiving data is device independent. In the above discussion, I have avoided the term string because I wish to use that term in a very special way. Let's jump right into the worst (or best) part of the whole discussion: pointers. Because FORTRAN and BASIC do not support pointers, this will be an new concept for many of you. One of the great strengths of the C programming language is the availability of pointers. So what are they? Pointers are variables that contain the address (location in memory) of something (most likely a piece of data). Keep in mind that a pointer does not contain anything but an address. Your house or apartment has an address, and we could find you by going to that address and knocking on your door. You do not live in your address; your address is only a pointer to where you live. Let's look at an example.

```c
#include <stdio.h>

main()
{
    int i;
    char buffer[50];
    char *cp;

    i = 5;
    sprintf(buffer,"i = %d",i);

    for(cp = buffer; *cp != '\0'; cp = cp + 1 ){
        printf("%c\n",*cp);
    }
}
```

The output of this program is:

```
i
= 5
```

Oh boy! There's a lot of new things here, and a lot of funny looking characters. Everything should be familiar until the line that contains char buffer[50]. This
construction states that buffer is an array of 50 characters. The square brackets make buffer an array. The next line of the program, char *cp, defines a pointer to a character. The star makes cp a pointer. Remember that cp is only the address of a place where a character can be found; it is not a character. sprintf(buffer;"i = %d",i); works like the printf statement except that the output goes into buffer and that the output will have a special character \0 appended to it. We will learn more about the reason for the \0 character soon but, for now, we know it is there and use it in the test portion of the for loop. Let’s tackle that for loop. Dissecting it piece by piece. First, in the initialization portion of the loop, cp = buffer; the address of buffer is stored in cp. The statement is not intuitive, but if you understand that, internally, C knows an array by its address (not its name) it should make more sense. Also notice that there is no * in front of cp. This is because cp is a pointer; the * has a special meaning that is context dependent. If the * appears in declaration, as in char *cp it notifies the compiler that you wish to use cp as a pointer. If the * appears anywhere else in the program, as it does in the test portion of the for loop, it instructs the compiler to go to the address stored in cp and do something with whatever is at that address. In the example above, the test, *cp != '\0', should be read as "While the character stored at the address held in cp is not equal to the special character \0, continue the loop." In C parlance, the * is referred to as the indirection operator. The program first looks in cp to find the address of the character for which it is looking (an indirect means of access). The last nifty thing to observe is in the end of loop processing part of the for loop. The construct cp = cp + 1 does not add one to the value stored in cp. The compiler knows cp is a pointer and interprets this as "go to the address next door". This is referred to as pointer arithmetic and is quite useful. You can use it, as we do here, to easily access the next element of an array. The only remaining part of this example to examine is the printf statement. Here, we print one character at a time (followed by a new line). Do you know why the argument to be printf is *cp instead of cp? We want to print the character found at the address contained in cp we do not want to print the address contained in cp. The blank looking lines in the output actually contain a space, but you can’t see a space unless something comes after it.

**String Handling**

While C does not support strings, it has a number of functions available for doing just that. This is analogous to how C handles mathematics. The string functions operate on character arrays. In general, we do not know how long a string might be, and neither do the string functions, so we must indicate the end by some means. One character that cannot be part of any string is the null character (\0). (This was something I promised to tell you about earlier.) Some string functions and sprintf automatically append the \0 character to the strings they format. Other functions like printf (when told to print a string) and strlen (string length) look for the \0 character to signal the end of the string.
```c
#include <stdio.h>
#include <string.h>
#include <math.h>

main()
{
  char buffer[100];
  double number;

  strcpy(buffer,"123");
  strcat(buffer,"ABC");
  printf("So far the buffer contains %d characters\n",strlen(buffer));
  printf("The characters are |%s|\n",buffer);

  printf("\n");

  number = 1000.0 / 3.0;
  sprintf(&buffer[strlen(buffer)],"%.10f",number);
  printf("Now the buffer contains %d characters\n",strlen(buffer));
  printf("The characters are |%s|\n",buffer);

  printf("\n");

  printf("The integer at the beginning of the buffer is %d\n",atoi(buffer));
  printf("The floating point number at the end of the buffer is %f\n",atof((strchr(buffer,'C') + 1)));
}
```

The output of this program is:

So far the buffer contains 6 characters
The characters are |123ABC|

Now the buffer contains 16 characters
The characters are |123ABC333.333333|

The integer at the beginning of the buffer is 123
The floating point number at the end of the buffer is 333.333333

In order to use the string functions, we must include string.h. To use the character-to-number conversion functions, atof and atoi, we have to include math.h. With these preliminaries out of the way, let's examine the program. The first new function we encounter (strcpy) will copy the string "123" into the buffer. strcat will add the string
"ABC" to the end of the string already contained in the buffer. The buffer is printed out so you can see its contents. Now for some tricky stuff. We use sprintf to format a number and add it to the contents of the buffer. We do not want to write at the beginning of the buffer so we use the function strlen to find the end of the string already in the buffer. Because the buffer[i] is a character and the function sprintf expects a pointer to where it can start putting characters, we resort to the & operator. This operator yields the address (equivalent to a pointer) of the item that follows. In this case, &buffer[strlen(buffer)] is the address at the end of the string contained in the buffer. The buffer is printed again so you can see the contents. That explains how to put a string together piece by piece, but how can we take it apart and extract the pieces we want? You could apply some of the techniques you have already seen in the example that was presented in the section on arrays and pointers but, because you already know those tricks, we will show you some new ones. Two new functions atoi and atof will be used to convert portions of the string contained in the buffer into an integer and a floating point number. The function atoi(buffer) reads characters from the beginning of the buffer until it finds the first character that doesn't make sense as part of an integer, and converts what it had found up to that point from a string representation to an integer. In the next line, we have to do a little more work because the number we seek is not at the beginning of the buffer. We could have used our previous knowledge and said atof(&buffer[6]) (arrays start at 0) but we might not always know the place where we want to start reading. In this case, we used our knowledge that the character 'C' immediately preceded the place we expected to find the number we seek. Because strchr returns the address of the character 'C', we had to get to the next address (remember pointer arithmetic) by adding 1. The atof function could then read as much as made sense (to the end of the string in this case) and convert the result to a floating point number.