

Functions and Prototypes

Objectives:

Having read this section you should be able to:

1. program using correctly defined C functions
2. pass the value of local variables into your C functions

Functions - C's Building Blocks:

Some programmers might consider it a bit early to introduce the C function - but we think you can't get to it soon enough. It isn't a difficult idea and it is incredibly useful. You could say that you only really start to find out what C programming is all about when you start using functions.

C functions are the equivalent of what in other languages would be called subroutines or procedures. If you are familiar with another language you also need to know that C only has functions, so don't spend time looking for the definition of subroutines or procedures - in C the function does everything!

A function is simply a chunk of C code (statements) that you have grouped together and given a name. The value of doing this is that you can use that "chunk" of code repeatedly simply by writing its name. For example, if you want to create a function that prints the word "Hello" on the screen and adds one to variable called total then the chunk of C code that you want to turn into a function is just:

```
printf("Hello");  
total = total + 1;
```

To turn it into a function you simply wrap the code in a pair of curly brackets to convert it into a single compound statement and write the name that you want to give it in front of the brackets:

```
demo()  
{  
  printf("Hello");  
  total = total + 1;  
}
```

Don't worry for now about the curved brackets after the function's name. Once you have defined your function you can use it within a program:

```
main()  
{  
  demo();  
}
```

In this program the instruction `demo ();` is entirely equivalent to writing out all of the statements in the function. What we have done is to create a new C function and this, of course, is the power of functions. When you are first introduced to the idea of functions, or their equivalent in other languages, it is easy to fall into the trap of thinking that they are only useful when you want to use a block of code more than once.

Functions are useful here but they have a more important purpose. If you are creating a long program then functions allow you to split it into "bite-sized" chunks which you can work on in isolation. As every C programmer knows, "functions are the building blocks of programs."

Functions and Local Variables:

Now that the philosophy session is over we have to return to the details - because as it stands the demo function will not work. The problem is that the variable `total` isn't declared anywhere. A function is a complete program sub-unit in its own right and you can declare variables within it just as you can within

the main program. If you look at the main program we have been using you will notice it is in fact a function that just happens to be called "main"! So to make demo work we have to add the declaration of the variable total:

```
demo()
{
  int total;
  printf("Hello");
  total=total+1;
}
```

Now this raises the question of where exactly total is a valid variable. You can certainly use total within the function that declares it - this much seems reasonable - but what about other functions and, in particular, what about the main program? The simple answer is that total is a variable that belongs to the demo function. It cannot be used in other functions, it doesn't even exist in other functions and it certainly has nothing to do with any variable of the same name that you declare within other functions.

This is what we hinted at when we said that functions were isolated chunks of code. Their isolation is such that variables declared within the function can only be used within that function. These variables are known as local variables and as their name suggests are local to the function they have been declared in. If you are used to a language where every variable is usable all the time this might seem silly and restrictive - but it isn't. It's what makes it possible to break a large program down into smaller and more manageable chunks.

The fact that total is only usable within the demo function is one thing - but notice we said that it only existed within this function, which is a more subtle point. The variables that a function declares are created when the function is started and destroyed when the function is finished. So if the intention is to use total to count the number of times the >demo function is used - forget it! Each time demo is used the variable total is created afresh, and at the end of the function the variable goes up in a puff of smoke along with its value. So no matter how many times you run demo total will only ever reach a value of 1, assuming that it's initialised to 0.

Making The Connections:

Functions are isolated, and whats more nothing survives after they have finished. Put like this a function doesn't seem to be that useful because you can't get data values in, you can't get data values out, and they don't remember anything that happens to them!

To be useful there has to be a way of getting data into and out of a function, and this is the role of the curved brackets. You can define special variables called parameters which are used to carry data values into a function. Parameters are listed and declared in between the () brackets in the function's definition. For example:

```
sum( int a, int b)
{
  int result;
  result=a + b;
}
```

defines a function called sum with two parameters a and b, both integers. Notice that the result variable is declared in the usual way within the body of the function. Also, notice that the parameters a and b are used within the function in the same way as normal variables - which indeed they are. What is more, they are still local variables and have nothing at all to do with any variables called a and b defined in any other function.

The only way in which parameters are any different is that you can give them initial values when the function starts by writing the values between the round brackets. So

```
sum(1,2);
```

is a call to the sum function with a set to 1 and b set to 2 and so result is set to 3. You can also initialise parameters to the result of expressions such as:

```
sum(x+2,z*10);
```

which will set a equal to whatever x+2 works out to be and b equal to whatever z*10 works out to be.

As a simpler case you can also set a parameter to the value in a single variable - for example:

```
sum(x,y);
```

will set a to the value stored in x and b to the value stored in y.

Parameters are the main way of getting values into a function, but how do we get values out? There is no point in expecting the >result variable to somehow magically get its value out of the sum function - after all, it is a local variable and is destroyed when sum is finished. You might try something like:

```
sum(int a, int b, int result)
{
  int result;
  result = a + b;
}
```

but it doesn't work. Parameters are just ordinary variables that are set to an initial value when the function starts running - they don't pass values back to the program that used the function.

That is:

```
sum(1,2,r);
```

doesn't store 1+2 in r because the value in r is used to initialise the value in result and not vice versa. You can even try

```
sum(1,2,result);
```

and it still will not work - the variable result within the function has nothing to do with the variable result used in any other program.

The simplest way to get a value out of a function is to use the return instruction. A function can return a value via its name - it's as if the name was a variable and had a value. The value that is returned is specified by the instruction:

```
return value;
```

which can occur anywhere within the function, not just as the last instruction - however, a return always terminates the function and returns control back to the calling function. The only complication is that as the function's name is used to return the value it has to be given a data type. This is achieved by writing the data type in front of the function's name. For example:

```
int sum(a,b);
```

So now we can at last write the correct version of the sum function:

```
int sum(int a, int b)
{
  int result;
  result = a + b;
  return result;
}
```

and to use it you would write something like:

```
r=sum(1,2);
```

which would add 1 to 2 and store the result in r. You can use a function anywhere that you can use a variable. For example,

```
r=sum(1,2)*3
```

is perfectly OK, as is

```
r=3+sum(1,2)/n-10
```

Obviously, the situation with respect to the number of inputs and outputs of a function isn't equal. That is you can create as many

parameters as you like but a function can return only a single value. (Later on we will have to find ways of allowing functions to return more than one value.)

So to summarise: a function has the general form:

```
type FunctionName(type declared parameter list)
{
    statements that make up the function
}
```

and of course a function can contain any number of return statements to specify its return value and bring the function to an end.

There are some special cases and defaults we need to look at before moving on. You don't have to specify a parameter list if you don't want to use any parameters - but you still need the empty brackets! You don't have to assign the function a type in which case it defaults to int. A function doesn't have to return a value and the program that makes use of a function doesn't have to save any value it does return. For example, it is perfectly OK to use:

```
sum(1,2);
```

which simply throws away the result of adding 1 to 2. As this sort of thing offends some programmers you can use the data type void to indicate that a function doesn't return a value. For example:

```
void demo();
```

is a function with no parameters and no return value.

void is an ANSI C standard data type.

The break statement covered in a previous section can be used to exit a function. The break statement is usually linked with an if statement checking for a particular value. For example:

```
if (x==1) break;
```

If x contained 1 then the function would exit and return to the calling program.

Functions and Prototypes:

Where should a function's definition go in relation to the entire program - before or after main()? The only requirement is that the function's type has to be known before it is actually used.

One way is to place the function definition earlier in the program than it is used - for example, before main(). The only problem is that most C programmers would rather put the main program at the top of the program listing. The solution is to declare the function separately at the start of the program. For example:

```
int sum();
main()
{
    etc...
```

declares the name sum to be a function that returns an integer. As long as you declare functions before they are used you can put the actual definition anywhere you like.

By default if you don't declare a function before you use it then it is assumed to be an int function - which is usually, but not always, correct. It is worth getting into the habit of putting function declarations at the start of your programs because this makes them easier to convert to full ANSI C.

What is ANSI C?:

When C was first written the standard was set by its authors Kernighan and Ritchie - hence "K&R C". In 1990, an international ANSI standard for C was established which differs from K&R C in a number of ways.

The only really important difference is the use of function prototypes. To allow the compiler to check that you are using functions correctly ANSI C allows you to include a function

prototype which gives the type of the function and the type of each parameter before you define the function. For example, a prototype for the sum function would be:

```
int sum(int,int);
```

meaning sum is an int function which takes two int parameters. Obviously, if you are in the habit of declaring functions then this is a small modification. The only other major change is that you can declare parameter types along with the function as in:

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

rather than:

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

was used in the original K&R C. Again, you can see that this is just a small change. Notice that even if you are using an ANSI compiler you don't have to use prototypes and the K&R version of the code will work perfectly well.

The Standard Library Functions:

Some of the "commands" in C are not really "commands" at all but are functions. For example, we have been using printf and scanf to do input and output, and we have used rand to generate random numbers - all three are functions.

There are a great many standard functions that are included with C compilers and while these are not really part of the language, in the sense that you can re-write them if you really want to, most C programmers think of them as fixtures and fittings. Later in the course we will look into the mysteries of how C gains access to these standard functions and how we can extend the range of the standard library. But for now a list of the most common libraries and a brief description of the most useful functions they contain follows:

stdio.h: I/O functions:

- getchar() returns the next character typed on the keyboard.
- putchar() outputs a single character to the screen.
- printf() as previously described
- scanf() as previously described

string.h: String functions

- strcat() concatenates a copy of str2 to str1
- strcmp() compares two strings
- strcpy() copies contents of str2 to str1

ctype.h: Character functions

- isdigit() returns non-0 if arg is digit 0 to 9
- isalpha() returns non-0 if arg is a letter of the alphabet
- isalnum() returns non-0 if arg is a letter or digit
- islower() returns non-0 if arg is lowercase letter
- isupper() returns non-0 if arg is uppercase letter

math.h: Mathematics functions

- acos() returns arc cosine of arg
- asin() returns arc sine of arg
- atan() returns arc tangent of arg
- cos() returns cosine of arg
- exp() returns natural logarithm e
- fabs() returns absolute value of num
- sqrt() returns square root of num

time.h: Time and Date functions

- time() returns current calendar time of system
- difftime() returns difference in secs between two times
- clock() returns number of system clock cycles since program execution

stdlib.h: Miscellaneous functions

- malloc() provides dynamic memory allocation, covered in future sections

rand() as already described previously
srand() used to set the starting point for rand()

Throwing The Dice:

As an example of how to use functions, we conclude this section with a program that, while it isn't state of the art, does show that there are things you can already do with C. It also has to be said that some parts of the program can be written more neatly with just a little more C - but that's for later. All the program does is to generate a random number in the range 1 to 6 and displays a dice face with the appropriate pattern.

The main program isn't difficult to write because we are going to adopt the traditional programmer's trick of assuming that any function needed already exists. This approach is called stepwise refinement, and although its value as a programming method isn't clear cut, it still isn't a bad way of organising things:

```
main()
{
    int r;
    char ans;

    ans = getans();

    while(ans== 'y')
    {
        r = randn(6);
        blines(25);
        if (r==1) showone();
        if (r==2) showtwo();
        if (r==3) showthree();
        if (r==4) showfour();
        if (r==5) showfive();
        if (r==6) showsix();
        blines(21);
        ans = getans();
    }

    blines(2);
}
```

If you look at main() you might be a bit mystified at first. It is clear that the list of if statements pick out one of the functions showone, showtwo etc. and so these must do the actual printing of the dot patterns - but what is blines, what is getans and why are we using randn()? The last time we used a random number generator it was called rand()!

The simple answers are that blines(n) will print n blank lines, getans() asks the user a question and waits for the single letter answer, and randn(n) is a new random number generator function that produces a random integer in the range 1 to n - but to know this you would have written the main program. We decided what functions would make our task easier and named them. The next step is to write the code to fill in the details of each of the functions. There is nothing to stop me assuming that other functions that would make my job easier already exist. This is the main principle of stepwise refinement - never write any code if you can possibly invent another function! Let's start with randn().

This is obviously an int function and it can make use of the existing rand() function in the standard library

```
int randn(int n)
{
    return rand()%n + 1;
}
```

The single line of the body of the function just returns the remainder of the random number after dividing by n - % is the remainder operator - plus 1. An alternative would be to use a temporary variable to store the result and then return this value. You can also use functions within the body of other functions.

Next getans()

```

char getans()
{
    int ans;

    printf("Throw y/n ?");
    ans = -1;
    while (ans == -1)
    {
        ans=getchar();
    }
    return ans;
}

```

This uses the standard int function `getchar()` which reads the next character from the keyboard and returns its ASCII code or -1 if there isn't a key pressed. This function tends to vary in its behaviour according to the implementation you are using. Often it needs a carriage return pressed before it will return anything - so if you are using a different compiler and the program just hangs, try pressing "y" followed the by Enter or Return key.

The `blines(n)` function simply has to use a for loop to print the specified number of lines:

```

void blines(int n)
{
    int i;

    for(i=1 ; i<=n ; i++) printf("\n");
}

```

Last but not least are the functions to print the dot patterns. These are just boring uses of `printf` to show different patterns. Each function prints exactly three lines of dots and uses blank lines if necessary. The reason for this is that printing 25 blank lines should clear a standard text screen and after printing three lines printing 21 blank lines will scroll the pattern to the top of the screen. If this doesn't happen on your machine make sure you are using a 29 line text mode display.

```

void showone()
{
    printf("\n * \n");
}

```

```

void showtwo()
{
    printf(" * \n\n");
    printf(" * \n");
}

```

```

void showthree()
{
    printf(" * \n");
    printf(" * \n");
    printf(" * \n");
}

```

```

void showfour()
{
    printf(" * * \n\n");
    printf(" * * \n");
}

```

```

void showfive()
{
    printf(" * * \n");
    printf(" * * \n");
    printf(" * * \n");
}

```

```
void showsix()
{
    int i;

    for(i=1 ; i<=3 ; i++) printf(" * * \n");
}
```

The only excitement in all of this is the use of a for loop in showsix! Type this all in and add:

```
void showone();
void showtwo();
void showthree();
void showfour();
void showfive();
void showsix();
int randn();
char getans();
void blines();
```

before the main function if you type the other functions in after.

Once you have the program working try modifying it. For example, see if you can improve the look of the patterns. You might also see if you can reduce the number of showx functions in use - the key is that the patterns are built up of combinations of two horizontal dots and one centered dot. Best of luck.