C Primer 2

by Nathan Lane
nathandelane.com

Introduction.

Similar to the last document, this document contains my opinion and my opinion only. It may contain false-hoods which are not true in every context. I do not attempt to teach you the *best* way to program in C, rather I teach simply how to program in C. I have tested all programs using MinGW's ISO C89/C99 compliant gcc compiler, version 3.4.5 (mingw-vista special r3) on Windows XP Professional, Service Pack 2, version 5.1.2600. If you obtain different results then it is because you are using a different compiler, a different environment, or you typed something differently from that way I wrote it. That said, let us begin.

Brief Recap of C Primer.

In C Primer I talked about the standard C libraries, basic console output using the printf-function, control structures like loops and conditional-statements, functions, and pointers. In this document I will discuss basic console input, more about pointers, streams, file input and output, and encapsulation methods.

Standard Input.

In the previous document I discussed and used standard output in many of my programs in order to illustrate the result of the functions, expressions, and statements I was using. But in all actuality ISO standard C does not include functionality for output. Though output is obviously needed to interact with a human, it is not necessarily needed to interact with a computer. The library, or header file which we will call them from here on out, stdio.h has been included in C packages since the dawn of C so as to make human-interactive software. Although it is not part of the standard, Stdio.h is a very useful addition to C in my opinion. Let's take a brief look again at our last non-pointer variation of the program with the power function.

```
#include <stdio.h>

int power(int n, int r);

int main(void)
{
    int n = 2;
    int r = 8;

    printf("The result of %d raised to the power of %d is %d.\n", n, r, power(n,
r));
    return 0;
}
```

```
int power(int n, int r)
{
    int base = n;
    int result = base;
    int i;

    for(i = 1; i < r; i++)
    {
        result = result * base;
    }

    return result;
}
```

This program outputs the following

```
The result of 2 raised to the power of 8 is 256.
```

That's nice, but what if we wanted to change the output of the program? We would just change the variables n and r to different values, right? Then we'd recompile, and run it again to get the different output. Well that's a lot of wasted time if you ask me. So why not ask the user for input? A novel idea. Standard input in C is, as you should have guessed, part of the header file stdio.h, hence the *i* in the title. There are two methods of getting input from the user at the console level, the gets-function and the scanf-function. We'll come back to the gets-function in a minute.

Scanf-function

Scanf is the little sister to printf. Like the *f* in printf means *formatted*, it means the same thing in scanf. Scanf allows you to ask the user for specific input, down to the character. Here is a brief example out of context of a scanf-function with a format string. Assume that all variables have been declared correctly.

```
...
    scanf("%s", &name);
...
```

In this example, %s refers to a string- or char[]- (those are square brackets) type variable, just like in the printf example, and the *name* variable must be a char[]-type variable. Now char[] is an array, and in this case we must define the size of the array. This is because C doesn't know how to handle pointers like char* directly – they must be initialized. So by creating a char-array or char[]-type variable, C doesn't have to worry about anything because the pointer was already handled. I'll show you how I declared name next, but first the scanf-function waits for line input from the console, meaning that the user may type as much as he wants until he hits the Return or Enter key on the keyboard. That signals scanf to pick up and read the line entered by the user.

Now the full program.

```
#include <stdio.h>

int main(void)
{
    char name[80];

    printf("What is your name? ");
    scanf("%s", &name);
    printf("Your name is %s.\n", name);

    return 0;
}
```

A char[]-type variable must be initialized in one of two ways, either by giving the length in between [and] or by setting the value. Setting the value also effectively sets the length of the variable. In this case I chose to specify the length.

```
...
    char name[80];
...
```

That means that I can enter up to 80 characters and they will be captured by the scanf statement successfully. In some cases you may be able to enter more and successfully capture them, but it would result in a buffer overflow, which may produce undesirable results.

To avoid a buffer overflow we add a buffer limit to the format string in this way

```
...
    scanf("%79s", &name);
...
```

But wait why 79 and not 80? We need one character for the termination character. In buffered I/O strings have a termination character. In most cases on all operating systems this character is the integer-value \0 or ASCII 0. We need to leave room for that with a single char. So we set the input limit to 79.

Scanf is very useful if you are expecting a particular type of input, like a number or a specific number of inputs like three strings. The great catchall to scanf however, is that scanf stops looking for data when it finds a white space character, such as a horizontal tab or space[1]. Enter *gets*.

Gets-function

The gets-function is similar to the scanf function in that it reads user input. It differs in that it only reads string data, meaning that it can only read input to a char*- or char[]-type variable. Gets both accepts a single argument, the char*- or char[]-variable, and returns the value that

it read unless an error occurs, in which a null pointer is returned, or in other words an uninitialized char*-variable.

Using the gets-function is very straightforward as seen in this next example.

```
#include <stdio.h>

int main(void)
{
    char name[80];

    printf("What is your name? ");
    gets(name);
    printf("Your name is %s.\n", name);

    return 0;
}
```

The extremely nice thing about this function is that gets reads until it reaches a newline character '\n' and it reads whitespace characters. Most strings will contain whitespace characters, so if you are expecting a string, then you should use gets, but if you are expecting something different, then you should use scanf unless you are going to parse the numbers out of the string, but I won't go into that here.

More Pointers and Streams.

Pointers are used a lot in computers. In computer architecture an instruction pointer points to the next line of machine code when running a program for example. Data is stored in memory by an operating system while a program is running and it uses pointers to access that data. Another use for pointers is to access files. In C there is what is called a file-pointer. It is defined as FILE*. FILE* is a type similar to char* or int*, except that FILE* refers to a stream or file on a disk or in the operating system.

All input and output is handled across streams which are accessed by pointers. This includes simple I/O through the console. In fact the console defines three streams, one for input, one for output, and one for errors, named stdin, stdout, and stderr respectively. Other common streams include your network interface (/dev/eth0 commonly on *nix systems), your printer (sometimes known as *lpt* in older systems and *prn* in newer systems), a modem may have both an input and output stream, and on Linux/Unix systems there are various other streams, usually located under the .dev directory, such as

/dev/cdrom – CD ROM stream
/dev/dvd – DVD Rom stream
/dev/fd – a general usage file duplicator stream
/dev/null – a stream of null or \0 bytes (often used to wipe disk drives or create empty files)
/dev/random – a stream of random bytes (similar to /dev/null)

and

/dev/tty0 – a terminal stream for terminal 0 (the first terminal in Linux, Linux has several terminals)

Next I'm going to talk about file input and output, in which I will use the FILE*-type. You may treat the file-pointer the same way as other pointers, but it has some special features making it useful for files, so please pay attention to those.

File Input and Output.

Files are the most common software component of a computer system. You can store nearly everything in files from images to videos, and text to binary data. Files are one of the only persistent forms of data. Being able to access and manipulate a file gives the programmer great power.

In order to access a file we must use a special C type called a file-pointer. The type is defined as FILE*. This next code snippets shows how to declare a FILE*-type variable.

```
...
    FILE *myFile;
...
```

This creates a new FILE*-variable. Just like any other variable it must be declared at the top of a code block. To open a file you use the *fopen*-function. Fopen takes two parameters, first a string containing the path to the file, including the file's name, that you want to open, and second a string containing access characters. Access characters tell C how to open the file and what you want to do with it. Valid access characters include “r” for reading, “w” for writing, and “a” for appending or writing to the end of a file. Multiple access characters may be used at the same time, such as “rw” for reading and writing the file[2]. The fopen-function also returns the file-pointer, or FILE*-type variable. To gain read access to a file named “temp.txt” in the current directory for example, see the following code.

```
#include <stdio.h>

int main(void)
{
    FILE *myFile;

    myFile = fopen(“temp.txt”, “r”);

    return 0;
}
```

This program simply opens a file and sets the FILE*-variable named myFile to point to the now open temp.txt file.

The next important thing to know is how to close a file. Leaving a file open is considered bad and may cause memory leaks or undefined behavior. The fclose-function is used for closing a file. It accepts a single parameter, a FILE*-variable, and returns zero if the file was successfully closed and the special constant EOF (end of file) if an error occurred. Let's create a more useful file handling program now.

Reading a File

```
#include <stdio.h>

int main(void)
{
    FILE *myFile;
    char lineFromFile[80];

    myFile = fopen("C:\\WINDOWS\\system32\\drivers\\etc\\hosts", "r");
    if(myFile != NULL)
    {
        while(fgets(lineFromFile, 80, myFile) != NULL)
        {
            printf("%s", lineFromFile);
        }

        fclose(myFile);
    }
    else
    {
        printf("There was an error opening the file at
C:\\WINDOWS\\system32\\drivers\\etc\\hosts.\n");
    }

    return 0;
}
```

Here you will see some new things going on. Let's start at the top. First we need to include stdio.h in order to use both the file I/O functions and the printf-function.

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

Next, inside the main function we define two variables, one FILE*-type and one char[]-type with 80 characters for the limit.

```
FILE *myFile;
char lineFromFile[80];
```

Next we try to open the file.

```
myFile = fopen("C:\\WINDOWS\\system32\\drivers\\etc\\hosts", "r");
```

After that we ensure that no errors occurred in trying to open the file, and don't do anything if

the file could not be opened except print out a message to the user on the console stating that there was an error.

```
if(myFile != NULL)
{
    ...
}
else
{
    printf("There was an error opening the file at
C:\\WINDOWS\\system32\\drivers\\etc\\hosts.\n");
}
```

Remember an error occurs while trying to open a file when fopen returns null (NULL is the C constant). So if myFile equals null, then we know there was a problem and we should avaoid trying to do any more work on the file. If the fopen was successful, then we attempt to read the contents of the file, line by line, and output them to the console in this while-loop,

```
while(fgets(lineFromFile, 80, myFile) != NULL)
{
    printf("%s", lineFromFile);
}
```

Here we do a couple of things in one line of code. Take a look at the expression in the while() function.

```
fgets(lineFromFile, sizeof(lineFromFile), myFile) != NULL
```

The first thing we do is try to read a complete line of text from the file using the fgets-function, which is similar to the gets function. The only differences are the number of arguments they take and that fgets include newline '\n' characters in the string whereas gets does not include newline '\n' characters in the string that it reads. The fgets-function is defined like this:

```
Char* fgets(string-to-copy-data-to, length-of-data-to-be-read, file-variable)
```

The next thing we do in the line of code above is check to make sure that we didn't return null when we read the file. If we did return null, then it is likely that we've reached the end of the file. Unfortunately null is also the return value if an error occurs. But in this case we'll assume that it means the end of the file was reached, hence this code snippet:

```
... != NULL
```

And finally we close the file, but only if we were able to open it in the first place. That is why it is inside of the if block.

```
fclose(myFile);
```

This program's output will vary from computer to computer as the host file, which is what it reads, may contain different data. Also the path for the host file in the code above is the path for it on a Windows machine. On Linux the path to the host file is “/etc/hosts”. You may have notice the double back-slashes in my path:

```
“C:\\WINDOWS\\system32\\drivers\\etc\\hosts”
```

These are there to *escape* the backslash character, because normally the back-slash character is used to create escape characters, like the newline character “\n”. Escaping the back-slash character allows me to use it individually in my string. Only one back-slash will appear in the string for each double back-slash.

Writing a File

We have discussed how to read a file, now let's discuss how to write information to a file. This may be useful if you are writing a program that keeps track of data like an address book. Let's write a simple program to accept user input and append it to a file.

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

int main(void)
{
    char userInput[80];
    FILE *namesFile;

    namesFile = fopen("names.txt", "a");
    if(namesFile != NULL)
    {
        while(strcmp(userInput, "quit") != 0)
        {
            printf("Please enter a name to be added to the names file ('quit' to
exit): ");
            if(gets(userInput) != NULL && strcmp(userInput, "quit") != 0)
            {
                if(fprintf(namesFile, "%0.79s\n", userInput) > 0)
                {
                    printf("Success\n");
                }
                else
                {
                    printf("There was an error writing to the file. Please try
again.\n");
                }
            }
            else if(strcmp(userInput, "quit") == 0)
            {
                printf("Thank you. Have a good day.\n");
            }
            else
            {
                printf("An error occurred while reading your input. Please try
```

```
again.\n");
            }
        }

        fclose(namesFile);
    }
    else
    {
        printf("There was an error opening the file names.txt for appending.\n");
    }

    return 0;
}
```

This program may seem a little bit long, but it contains necessary error handling and user feedback in case an error occurs, which is something that you'll always want to make sure you take care of, otherwise there may be bugs in your code that you don't know about and they'll be difficult to find if you're not handling all of the possible errors.

There are only a couple of new things here, so I will briefly go over those.

The first new thing is the new header file that I included, string.h. String.h contains string handling functions. The only string handling function I am using in this program is the strcmp-function, which compares two strings and tell whether they match or not.

```
#include <string.h>
```

We want the user to be able to enter the word *quit* to exit out of the program so we have this line which compares the user's input to that word:

```
while(strcmp(userInput, "quit") != 0)
```

That line basically says, “while string-compare userInput and 'quit' is not equal to zero”. The strcmp-function returns zero when the first and second string matches, or non-zero when they don't match. So as long as those strings don't match, this while loop will continue.

Next we ask the user for input in the printf statement and validate that input against our 'quit' string again.

```
if(gets(userInput) != NULL && strcmp(userInput, "quit") != 0)
```

This line uses the gets-function to get user input, then it makes sure that an error didn't occur by comparing the return value of gets to null, and then we see a new function, *&&*. This is the logical *and* operator. You use it in if-else-statements if you want to have multiple conditions for providing true in the statement before going on. We want to make sure that gets didn't return null here AND make sure the, once again, userInput does not equal 'quit'.

The next thing we do is attempt to write the userInput string out to our file, and ensure that it doesn't cause an error.

```
if(fprintf(namesFile, "%0.79s\n", userInput) > 0)
```

Here is another new function, the fprintf-function. Fprintf is the file version of the printf-function. It behaves in much the same way. The only difference is that it requires a FILE*-type variable as the first argument, then the format string, and then any variables. Also it returns -1 if an error occurs and the number of characters written to the file if it was successful. That's why we have the *> 0*. Everything else in this program is simply user feedback.

There are many other file I/O functions that we have not discussed here, but for simple file input and output these functions are sufficient.

Encapsulation.

In programming encapsulation refers to information hiding. This involves creating objects that contain multiple pieces of data. In C a function encapsulates several statements, functions, and expressions – it effectively *hides* that functionality from the programmer using the function so that all he has to know is how to use it, not what's inside or how it works. Likewise a program encapsulates both functionality and data, so that a user of the program must not need to know exactly how it works to use it. Also in C there is another way to encapsulate data called a *struct*.

Struct is short for structure. C structs may contain variables, constants, arrays, and pointers much the same way a program does. These components of structs are called struct members or just members.

A struct is defined as follows.

```
Struct struct-name
{
    [struct-member-type1 struct-member-name1[ = value];]
    [struct-member-typen struct-member-namen[ = value];]
} [TypeName[, TypeNamen]];
```

Once a struct has been defined, it may be used in a similar way to a type, like char, int, or float. Here's an example struct definition and an example of its usage.

```
struct user
{
    char userName[80];
    char userPassword[80];
};

int main(void)
```

```
{
    struct user user1;

    user1.userName = “Nathanl”;
    user1.userPassword = “12345678”;

    return 0;
}
```

Here I have introduced you to a new operator, namely the dot-operator, or '.'. The dot-operator enables you to access struct members inside of a struct. In the above program, user1 becomes an *instance* of a struct of type user. That means that it has its own copies of both of the members that the struct *user* has, namely char userName[80] and char userPassword[80]. User1 is just another variable however.

Another way to declare a struct, and this time as a type, is as follows.

```
struct user
{
    char userName[80];
    char userPassword[80];
} User;

int main(void)
{
    User user1;

    user1.userName = “Nathanl”;
    user1.userPassword = “12345678”;

    return 0;
}
```

In this example I declared a new type at the end of the struct definition named *User*. Now I can use this type just like any other type and declare new variables with it, as in the line:

```
User user1;
```

Structs may also be passed to your own home-brewed functions. You can either use the struct notation or the type notation if you have defined that in your program. A simple definition of both a struct and a function accepting the struct might look as follows. I will not implement the function here.

```
struct math_power_parts
{
    int n;
    int r;
} PowerParts;

int power(PowerParts powerParts);
```

Here I defined a function that accepts one variable of the type PowerParts with the name *powerParts* (remember that case matters when programming in C or C++ so PowerParts and powerParts are different).

The most common struct that is part of the standard libraries of C is the *tm* structure in the time.h header file. The struct tm contains the components for the time and date on a computer[3].

Well that's it for this C primer. If you have any questions or problems then please feel free to contact me. Thanks.

Nathan Lane
Nathandelane

Appendix A

Contact Information

Nathan Lane
Nathandelane

Email: **nathandelane@nathandelane.com**
Home Page: **http://nathandelane.com**

Compiling Examples

Throughout this tutorial I utilized two programs mainly, **vim** (VI Improved) and **gcc** (GNU C Compiler) on Windows as part of **MinGW**. These examples should all also however compile on other compilers, since they use ISO standards.

In order to compile these from the command-line using gcc, follow this syntax.

```
> gcc -o programOutputFileName programSourceFileName
```

This will produce an executable program named programOutputFileName. The -o compiler directive allows you to name the output file of the compiler.

References Cited

1. CplusPlus.com scanf, cplusplus.com 2008, http://www.cplusplus.com/reference/clibrary/cstdio/scanf.html
2. CplusPlus.com fopen, cplusplus.com 2008, http://www.cplusplus.com/reference/clibrary/cstdio/fopen.html
3. CplusPlus, struct tm, cplusplus.com 2008, http://www.cplusplus.com/reference/clibrary/ctime/tm.html