C Primer

Compiled by Nathan Lane
nathandelane.com

Quote Sharmz:

> As I said after your first post, I am motivated. Especially when you talk about your personal experiences. As for myself. I learnt HTML when I was 12, so about two years back and I never really got that much into it but C has fascinated me. However the main problem I seem to face is learning it. I am not sure whether I am learning the right way. I have a few books but both you and I seem to agree that books aren't direct sources of knowledge. I have just started learning about pointers and they seem to confuse me. Can you briefly if possible outline the most important parts of C that I should learn because Ive been cowering over it for a year with minimal progress.

Introduction.

This document contains my opinion and my opinion only. As such it should not be construed as the end-all be-all to C primer documents. You should search out more clear sources or references where mine are not so clear. You should also consult programmers who use C more often when you have questions regarding some of the writings herein. I will try as often as possible to cite my sources. If it is not possible, then I will give a citation, and in the references I will refer you to the web search I performed to get my information.

ANSI/ISO Standard C.

C, the programming language was originally created in “Ma Bell”[1] Laboratories or Bell System, now divided into several conglomerates, most notably Qwest, Verizon, AT&T, Cincinatti Bell, Alcatel-Lucent, Avaya, LSI Corporation, and Telecordia Technologies, AT&T still holding rights to the Bell System trademark[1]. The reason for its creation was to simplify programming a computer. Before high-level, general purpose languages like C, all computers had to be programmed in their respective machine language. Even assemblers were invented as a high-level language to machine language. Because this task was tedious at best and system architectures changed and evolved so rapidly, it was not plausible that programmers should have to continue in this way. C became the first heavily typed compiled high-level programming language[2].

The C programming language was first submitted as a standard in 1989 under ISO/IEC 9899[3].

The C Libraries.

Everything in C is actually about as basic as you can get and still be high-level. C has fifteen standard libraries, assert.h, ctype.h, errno.h, float.h, limits.h, locals.h, math.h, signal.h,

stdarg.h, stddef.h, stdio.h, stdlib.h, string.h, and time.h.

Most commonly used are float.h for system-dependent floating-type values, like FLT_MAX and FLT_MIN, defining the maximum and minimum value for floating-point types, stdio.h for standard input and output functions for both screen and files, stdlib.h for standard utility functions like conversion functions, and memory handling functions, time.h for date and time functions, limits.h for language-dependent maximums and minimums (char, int, long, short, ...), and math.h for mathematical functions like trignometric functions, powers and square roots, and rounding functions.

First C Program.

The most common first C program might be a “hello world” program[4]. You have probably written one of these, but in case you haven't, here is a brief example.

```
#include <stdio.h>

int main(void)
{
    printf(“%s, %s!”, “Hello”, “World”);

    return 0;
}
```

This program provides an illustration of standard output (stdout) in the C programming language. The standard method of printing output to the console is by using the printf(char *, ...) function found in stdio.h. Printf takes two arguments, first is the required format string which may or may not contain formatting characters (i.e. %s), and following is an array of parameters containing data to be inserted into the format string. In this case “Hello” and “World” are the data parameters.

In effect, the compiler sees the format string, “%s, %s!”, like this, “<insert first string here>, <insert second string here>!”, where ',' and '!' are simply characters to be output to the console. It then sees “Hello” as the first string and inserts it into the string to print, “Hello, <insert second string here>!”. Then it inserts the next string because it sees that it needs to, “Hello, World!”, and then outputs the entire string, resulting in

```
Hello, World!
```

The above program could also have been written as

```
#include <stdio.h>

int main(void)
{
    char *hello = “Hello”;
```

```
    char *world = “World”;

    printf(“%s, %s!”, hello, world);

    return 0;
}
```

The only difference between this program and the first program is that variables were used here. In C variables must always be declared at the beginning of a block of code. A block of code always begins with a curly brace, '{' and ends with a curly brace, '}'. It would be a syntax error to declare variables otherwise in C, though some C compilers may allow you to do so. In C++ this tradition was broken and it is completely allowable to declare a variable outside of a new code block or anywhere in the middle of the code.

A common mistake that new programmers in C using the printf-function make, is using the wrong format strings in correlation with the data parameters. Here is an example of this failure to notice detail.

```
#include <stdio.h>

int main(void)
{
    char *hello = “Hello”;
    char *world = “World”;

    printf(“This will incur undefined behavior: %d, %s!”, hello, world);

    return 0;
}
```

The problem with the above code is with the %d. The %d format string tells the C compiler that it should insert an integer-type variable's data there, however I supplied a char*-type variable as the first data parameter. So let's see what the output would be.

```
This will incur undefined behavior: 4206592, World!
```

So because char* is in fact an integer-type, we get the integer variant of “Hello” in place of the string. This is not a pointer location, which we will discuss later, rather it is the integer value of “Hello”. If you run this a hundred times, then you will always have the same number. Anyway this is known as “undefined behavior”.

Let's look at another bad example of how to program using C's printf-function.

```
#include <stdio.h>

int main(void)
{
    char *hello = “Hello”;
```

```
    char *world = “World”;

    printf(“This will incur undefined behavior: %f, %s!”, hello, world);

    return 0;
}
```

This compiled fine as well, but when I run it I get once again undefined behavior. This time I'm asking for a floating-point-type variable, and instead supplying a string. Let's look at the output.

```
This will incur a syntax error: 0.000000, Φ╞!
```

Whoa, what's that on the end? “Φ ╞” Must have come from me asking for a floating-point-type variable and supplying a string, but that didn't happen before. The moral of this is that you as the programmer must always try to know exactly what you're outputting and supply the correct data parameters to the printf-function.

Control Structures.

Control structures are a common feature of most high-level programming languages as well as low-level programming languages. The simplest control structure is a goto statement, though not in use much anymore, which directs the program to go to a specific label in the code or specific line of code. The most commonly used control structures are for-loops, if-else-statements, do-while-loops, while-loops, switch-statements, and conditional-statements (which are written differently from if-else-statements, but are very similar).

For-loops

Control structures are easy to understand. They simply *control* what the program does when they are encountered. For example, let's look at the for-loop.

```
#include <stdio.h>

int main(void)
{
    int i;

    for(i = 0; i < 10; i++)
    {
        printf(“This is loop #%s.\n”, I);
    }

    return 0;
}
```

The output of this small program is:

```
This is loop #0.
This is loop #1.
This is loop #2.
This is loop #3.
This is loop #4.
This is loop #5.
This is loop #6.
This is loop #7.
This is loop #8.
This is loop #9.
```

The for-loop depends on three pieces, the initialization, the end-condition, and the function, which is usually either an incrementor (++) or a decrementor (--), each separated by a semi-colon (;). The for loop definition looks similar to this:

```
for(initialization; end-condition; function);
```

Any type may be used in a for-loop. For example this code is valid C code.

```
#include <stdio.h>

int main(void)
{
    double d;

    for(d = 0.0; d < 1.0; d += 0.1)
    {
        printf("Current value of d is %f.\n", d);
    }

    return 0;
}
```

The resulting output is:

```
Current value of d is 0.000000.
Current value of d is 0.100000.
Current value of d is 0.200000.
Current value of d is 0.300000.
Current value of d is 0.400000.
Current value of d is 0.500000.
Current value of d is 0.600000.
Current value of d is 0.700000.
Current value of d is 0.800000.
Current value of d is 0.900000.
Current value of d is 1.000000.
```

In order to effectively use a control structure, you must simply understand that there are basically two variable type in C, floating-point- and integer-types. Integer types include int, short, long, char, and pointer variants of those types. Floating-point types include float and double and their respective pointer variants. We'll talk about pointers later on.

Here are some more examples of for-loop definitions.

For-loop with incrementor

```
for(i = 0; i < 100; i++)...
```

For-loop with another form of incrementor

```
for(h = 25; h <= 30; ++h)...
```

For-loop with multiple variables

```
for(i = 0, j = 10; i < 10, j > 0; i++, --j)...
```

Infinite for-loop

```
For(;;)...
```

Don't use infinite for-loops unless you absolutely have a reason to. Using infinite loops is something that could easily crash your system, so be careful with them.

If-else-statements

If-else-statements are conditional statements used to control the flow of a program. The *else* part of the if-else-statement is optional, but often good to use in every case. If it is not used then you may end up with an unknown defect when handling an uncommon state or case. Here is an example of an if-else-statement.

```
#include <stdio.h>

int main(void)
{
    int val = 24;

    if(val == 24)
    {
        printf("There are 24 hours in a day.\n");
    }
    else
    {
        printf("There are not %d hours in a day.\n", val);
    }

    return 0;
}
```

This program prints the statement, “There are 24 hours in a day.” if val equals 24, and the statement, “There are not %d hours in a day.” if val does not equal 24, where %d is replaced by whatever value is in the variable val. Try it out with val equals 24 and then set val to a different number and see what happens.

val = 24

```
...
    int val = 24;
...
```

Output

```
There are 24 hours in a day.
```

val = 25

```
...
    int val = 25;
...
```

Output

```
There are not 25 hours in a day.
```

Now with the conditional statement I introduced a new operator. Some common operators are =, +, -, /, |, &, &&, ||, *, ==, <, >, !, !=, <=, and >=. The == operator is called the “equal to” operator. This is used to determine whether the left value is equal to the right value. On the other hand the = or assignment operator assigns the value on the right to the variable on the left[5]. It is a common mistake for new programmers to interchange these two operators, which would result in compile-time errors.

Here is a brief example of what not to do.

```
#include <stdio.h>

int main(void)
{
    int val = 24;

    if(val = 24)
    {
        printf("This is undefined behavior.\n");
    }
    else
    {
        printf("This statement will never be reached.\n");
    }

    return 0;
}
```

The output of this follows.

```
This is undefined behavior.
```

An else part of the if-else-statement may also include another if-statement as in this example.

```
#include <stdio.h>

int main(void)
{
    int val = 24;

    if(val == 24)
    {
        printf("There are 24 hours in a day.\n");
    }
    else if(val == 60)
    {
        printf("There are 60 seconds in a minute and 60 minutes in an hour.\n");
    }
    else
    {
        printf("There are not %d hours in a day, nor %d seconds in a minute or %d
minutes in an hour.\n", val, val, val);
    }

    return 0;
}
```

Changing val to 60 now, will result in the first else-if block being executed, and any number besides 24 or 60 will result in the final else block being executed.

Do-while- and while-loops

Do-while- and while-loops are closely related. The same keyword, while, is used in both constructs. The only difference is at which point the while statement occurs in the loop. In a do-while-loop the while statement comes at the end of the loop, causing the condition in the while statement to be evaluated at the end of a loop cycle. In the while-loop the while statement comes before the loop, causing the condition in the while statement to be evaluated at the start of a loop cycle. This results in a do-while-loop always being executed at least once, but a while loop may never be executed. See the following two examples.

While-loop

```
#include <stdio.h>

int main(void)
{
    int x = 1;
    int max = 1;

    while(x < max)
    {
        printf("This printf statement will never be executed.\n");
    }

    return 0;
}
```

Do-while-loop

```
#include <stdio.h>

int main(void)
{
    int x = 1;
    int max = 1;

    do
    {
        printf("This printf statement gets executed once.\n");
    }
    while(x < max);

    return 0;
}
```

Often while-loops and do-while-loops can be used in place of for-loops, but you must keep track of incrementation inside of these loops, whereas the for-loop takes care of that for you. For this reason, I believe while- and do-while-loops give the programmer more control.

Switch-statements

Switch statements are very similar to if-else-statements. They simply offer a more concise method of writing program control. A switch statement's definition is as follows:

```
switch(expression-to-be-evaluated)
{
    [case value-1:
        statements...
        [break;]
    case value-n:
        statements...
        break;
    [default:
        [statements...]
        break;]]
}
```

The rules of writing a switch statement, that is useful, are that it must have an expression to be evaluated, it must have at least one case block, and it must end with a break statement. However everything inside of the switch block is syntactically not required, which is why is is all contained in square braces ([,]) in the definition above. Also a break must occur at the end of the last case block. It is not required that a break statement follow every case block if you desire to allow multiple values to fall through to a single block of code however. The following example illustrates this. Though it is relatively long, this example is still quite simple.

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

```
int main(void)
{
    int age = 20;

    switch(age)
    {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
        case 10:
        case 11:
        case 12:
            printf("You are but a child at age %d.\n", age);
            break;
        case 13:
        case 14:
        case 15:
        case 16:
        case 17:
        case 18:
        case 19:
            printf("You are a growing teenager at age %d.\n", age);
            break;
        default:
            printf("You are a growing adult at age %d.\n", age);
            break;
    }

    return 0;
}
```

The above program checks the value of age and matches it against the correct case. If the case does not have a break statement, then it effectively falls through to the next case block, executing until it reaches a break statement, which breaks out of the case and switch-statement. The default statement at the end applies to any case that is not covered in the switch block. The only major drawback to using switch-statements in place of if-then-else statements is that you cannot provide completely conditional execution. For example I cannot ensure that age will not be negative and I cannot ensure that age will not be greater than 20. If age were negative, then the default case would still be called, however this would not make sense to a user:

Age = -19

```
...
    int age = -19;
...
```

Output

```
You are a growing adult at age -19.
```

Conditional-statements

Conditional statements are just a shorthand way of writing an if-else-statement with only one else. Some compilers allow you to embed conditional-statements in each other allowing essentially a way to copy if-else-statement structure with multiple else-if blocks, but as this becomes unreadable and is not directly supported by the ISO standard, it is not recommended. Conditional statements take this form.

```
Conditional-expression ? executed-when-true : executed-when-false;
```

Here is an example of using a conditional-statement.

```
#include <stdio.h>

int main(void)
{
    int age = 13;

    (age < 20) ? printf("You are not an adult.\n") : printf("You are an
adult.\n");

    return 0;
}
```

With age equaling 13, the first printf-statement will be executed. If you set age to 20 or higher, then the second printf-statement will be executed. The reason for this has to do with the conditional statement. (age < 20) evaluates to true when the value of age is in fact less than 20, resulting in the first printf-statement being executed, but if the value of age is equal to or greater than 20, then (age < 20) evaluates to false, which results in the second printf-statement being executed.

You should also take notice that in this case there is only one semi-colon at the end of the second printf-statement. Semi-colons are used to divide syntactic lines of code, however in a conditional, since ? And : are also line delimiters, a semicolon after each printf-statement is not required. Multiple statements in conditional groups are also not allowed in C. For example, this would result in a syntax error at compile time.

```
#include <stdio.h>

int main(void)
{
    int age = 13;

    (age < 20) ? printf("You are not an adult.\n"); printf("Age was set to
%d.\n", age) : printf("You are an adult.\n"); printf("Age was set to %d.\n",
```

```
age);

    return 0;
}
```

Instead you could write the multiple printf-statements into a single printf-statement for each grouping like this and get the same effect.

```
#include <stdio.h>

int main(void)
{
      int age = 13;

      (age < 20) ? printf("You are not an adult.\nAge was set to %d.\n", age) :
printf("You are an adult.\nAge was set to %d.\n", age);

      return 0;
}
```

This program would have the following output.

```
You are not an adult.
Age was set to 13.
```

Functions.

Functions are the simplest module in a C program. Functions break up the statements of a C program into smaller more manageable components. Up until now you will have noticed the usage of one simple function in every program that we have written, in fact, a function named *main*. The main function is required in every C program that you write. This function is acceptably the entry point of your C program, or where the program starts. The basic definition of a function looks like this in C.

```
[access-modifier(s)] return-type function_name([arguments[, ...]|void]);
```

Access-modifiers are keywords used to set the access of a function. Most often they will not be used. We will not discuss access modifiers here.

Return-type is required. This is the type of the value that will be returned to a user once the function has completed. If no value is to be returned then *void* should be used as the return-type.

Function_name is the name of the function, for example the printf-function's name is *printf*.

The arguments to a function are optional. If the function does not require any data then you may type *void* in the parentheses, however if data is required by the function, then you must

declare the arguments needed by the function.

The main function that I have used throughout this document so far has had the following definition.

```
int main(void)
```

This means that I will return a value of int-type to the operating system using the *return*-statement when the program finishes and that I am accepting no arguments to the program. The main function is a special function in that it can only be defined in a couple of ways and it can only be defined once per program, but your own functions do not have this limitation.

Some other examples of a valid main function follow.

Allow for command-line arguments to your program

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

No return value to the operating system and no command-line arguments allowed

```
Void main(void)
```

No return value to the operating system and command-line arguments allowed

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

Now let's focus on creating our own function. A very simple function to work on is a power math function. In algebra, the power function takes a base number *n* and multiplies it by itself *r* times, where the function looks like n^r or n^r.

We will start off by writing our program so that it will compile properly and then we can focus all of our work on the function itself.

Skeleton program that should be used whenever writing a program.

```
int main(void)
{
    return 0;
}
```

Now let's add a definition of our power function. The definition tells the compiler to look for our function. In C functions must be defined before they are used. A good rule is to always define them before the main function.

```
int power(void);

int main(void)
{
    return 0;
```

```
}
```

We know that our power function will take two arguments of int type, one for *n* and one for *r*.

```
int power(int n, int r);

int main(void)
{
    return 0;
}
```

Now that we have defined our function with a function definition, we can implement or write the function. We can do that either before or after the main function. Let's do it after the main function.

```
int power(int n, int r);

int main(void)
{
    return 0;
}

int power(int n, int r)
{
    return 0;
}
```

We will shortly replace that *return 0;* statement in the power function with the actual code to return the result of our power function. Remember that in C we need to define our variables at the beginning of a block of code, so let's define a return variable at the top of our function. A function is a valid block of code. At the same time we'll initialize the variable to a default value, and make our return statement return that value.

```
int power(int n, int r);

int main(void)
{
    return 0;
}

int power(int n, int r)
{
    int result = -1;

    return result;
}
```

Now that we have that completed, we have a skeleton of our function. We now need to write the code that actually causes the function to work. I'll write it quickly here, but please look carefully at the code to make sure you understand it.

```
int power(int n, int r);

int main(void)
{
    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;
}
```

Now that our function is complete, let's test it out. We'll write a couple of lines of code in our main function to test the power function.

```
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;
}
```

The output of this small program follows.

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

Do you see how I used a function like a variable in the printf-statement? Take a look at the following line of code.

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

If you remember, the power function we made returns an int-type value, and I expect an int-type value in my format string as the third value. This is called inlining. I chose to inline the result of the power function instead of creating a new variable to hold the result of the power function. This is a very common practice, but it is not always valuable. Inlining though is more of a standard than an exception[6].

Pointers.

Pointers are used to point to a specific part of your computer's memory. This may be where a value, function, or other object is stored. When we talk about memory here, we are talking about RAM, instead of your hard drive. Pointers are pretty much only found in languages like C and C++. Assembly, C#, Java, and other languages call them references, and they are not as dynamic in those languages as they are in C and C++.

It is always important to remember that pointers never contain values themselves, rather they point to the location of where values can be found. For example, let's write a small program.

```
#include <stdio.h>

int main(void)
{
    int *p;

    printf("The address of int *p, a pointer, is %d.\n", p);

    return 0;
}
```

The resulting output would look something like this.

```
The address of int *p, a pointer, is 2147311616.
```

Run it again, and you might see something like this.

```
The address of int *p, a pointer, is 2147323904.
```

Did you notice that the number at the end changed? That is because when the operating system runs the program it allocates some space, based on the type of the pointer, somewhere in random-access-memory, or RAM. It can never be considered the same, but this is what makes pointers valuable. As long as there is memory you can use a pointer to point to it.

Now let's say you wanted to set the value at the address of the pointer.

```
#include <stdio.h>

int main(void)
{
    int *p;

    *p = 24;

    printf("The value of int *p, a pointer, is %d.\n", *p);

    return 0;
```

Notice that when I assigned the value 24 to *p, I used *p. This means, “the value at [p]”. So in effect my line that looked like this:

```
*p = 24;
```

Reads, “the value at [p] is 24”. And in order to get the value at p, you must again use the * operator, like this:

```
int regularVariable = *p;
```

Or as in my printf-statement:

```
printf("The value of int *p, a pointer, is %d.\n", *p);
```

On the other hand what if you wanted to get the address of another variable? This is possible because, in fact, all variables are stored in RAM while a program using them is running. Let's look at another example program.

```
#include <stdio.h>

int main(void)
{
      int r;

      r = 36;

      printf("The location of int r in memory is %d.\n", &r);

      return 0;
}
```

This program's output looks like:

```
The location of int r in memory is 2293620.
```

The & operator as it applies to variables is called a reference operator. & gets the reference location of, or dereferences, a variable. So in the first pointer program, we could have done this and the results would be the same as in the following code.

```
#include <stdio.h>

int main(void)
{
    int *p;

    printf("The address of int *p, a pointer, is %d.\n", &p);

    return 0;
}
```

The program's output would be similar to the first pointer program.

```
The address of int *p, a pointer, is 2293620.
```

One useful purpose of pointers is to create references to other variables. Using them in this way you can pass variables to a function that doesn't return anything, perform operations on the variables passed, and return those values in the same variables. Let's look at a program that uses references to variables.

```
#include <stdio.h>

int main(void)
{
    int base = 2;
    int *refBase;

    refBase = &base;

    printf("int base = %d, *refBase = %d.\n", base, *refBase);

    *refBase = base * 2;

    printf("After *refBase = base * 2, base = %d", base);

    return 0;
}
```

Output

```
int base = 2, *refBase = 2.
After *refBase = base * 2, base = 4
```

Let's look more closely at what happened here. First I assigned the value 2 to the int-type variable *base*.

```
int base = 2;
```

Then I referenced base in the int-type pointer *refBase*.

```
refBase = &base;
```

After that I showed you that both base and refBase contained the same value even though they are different variables, and I never explicitly assigned 2 to refBase.

```
printf("int base = %d, *refBase = %d.\n", base, *refBase);
```

Next I performed an operation on the value in refBase, making it equal to base * 2, or 4.

```
*refBase = base * 2;
```

Then I showed you the value of base after the operation performed on refBase,

```
printf("After *refBase = base * 2, base = %d", base);
```

Notice that in that last statement I used *base* and NOT *refBase* to show you what the value was. Now let's try it in a function. Once again let's use the power math function to demonstrate. We'll start out with our skeleton program from before.

```
int power(int n, int r);

int main(void)
{
    return 0;
}

int power(int n, int r)
{
    int result = -1;

    return result;
}
```

Now because we're using pointers, we're going to change it a little bit.

```
void power(int *result, int n, int r);

int main(void)
{
    return 0;
}

void power(int *result, int n, int r)
{
}
```

We are going to return the result of all of the operations in the first variable. But we are still going to accept n and r separately this time so that it isn't too confusing. So instead of passing two variables to power, we now need to pass three variables, and one needs to be a pointer.

Next let's take the original guts of the power function and redefine them.

Originally

```
...
    int base = n;
    int result = base;
    int i;

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

    return result;
...
```

Now with the pointers

```
...
    int base = n;
    int i;

    *result = base;

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

We'll also need to change our main function, so here's the new program.

```
void power(int *result, int n, int r);

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

    power(&result, n, r);

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

    return 0;
}

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

    *result = base;

    for(i = 1; i < r; i++)
    {
```

```
        *result = *result * base;
    }
}
```

So now instead of returning the value we are taking the reference of the result value and changing it. As you will see, upon running this example, the resulting output is exactly the same.

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

The End?

This is nowhere near the end of C programming as we know it. Massive programs have been programmed in C, including the first C++ compiler. Think about it. It had to come from somewhere. This should get you started though, and if you run into any problems then please, feel free to contact me. My contact information is on the next page.

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. Wikipedia, wikipedia.org 2008, http://en.wikipedia.org/wiki/Bell_System
2. Bell Labs, bell-labs.com 2008, http://cm.bell-labs.com/who/dmr/chist.html
3. ISO/IEC, open-std.org 2008, http://www.open-std.org/jtc1/sc22/wg14/www/standards
4. Wikipedia, wikipedia.org 2008, http://en.wikipedia.org/wiki/Hello_world_program
5. Wikipedia, wikipedia.org 2008, http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B
6. No reference