Programming: Notes

Adam Szava

January 2021

Computers

Storage types are the different types of data storage hardware that exist in a computer. They include:

- CPU Registers: Only a few cells on CPU
- Main Memory (RAM): Billions of cells on circuit, not on CPU
- Secondary Storage: Hundreds and thousands of billions of cells on disks or tapes. (This type of data is non-volatile as in it will not lose data when power is off.)

CPU registers are the most *expensive* while secondary storage are the least *expensive* type, but they are the biggest and the slowest.

One **bit** is considered to be one binary digit, while a **byte** is a group of 8 bits, which represent one character. A **word** is the width of a memory cell, as in how many bits can fit in the registry. Each byte has a memory address, this is used to locate the byte of data. In a 32bit system, a string of 32 bits uses the first (leftmost) bit to indicate sign with 0 for + and 1 for -, while the other 31 bits are used for the magnitude of the number. This method is outdated.

Two's complement is another method to identify negative numbers, essentially to convert a number to its negative, you look at it's binary number, find the first 1 from the right side, and invert every digit after that. As in:

9 = 0000 0000 0000 0000 0000 0000 0000 1001
-9 = 1111 1111 1111 1111 1111 1111 1111 0111

If you take a long binary string, which is broken up into groups of 4, you can convert it to base-16. This is because a the string 1111 which is the highest number a 4-digit binary number can have is 15, which is the number of characters in base-16 called *Hexadecimal*. Therefore each 4-digit binary string corresponds to a unique digit in base-16 as in the following table:

Hexadecimal $\longleftrightarrow$ Binary

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

IEEE-754 is a method for storing floating point numbers as binary strings. There are two types of IEEE formats called **single precision** and **double precision**. The purpose of this is to take any real number, and approximate (with potentially 0% error) it in the form $\pm 1.??? \times 2^e$. This is a good way for the computer to understand the number. The former is 32bit, while the latter is 64bit. In both cases, the binary string is split into 3 sections:

- s = Sign (1^{st} bit)
- e = Exponent (Next 8 bits for single, or 11 bits for double)
- m = Mantissa (Remaining bits, length depends on single/double precision)

IEEE $\rightarrow$ decimal

In general here is the formula to turn an IEEE floating point number into decimal:

$$(-1)^s \times (1+m) \times 2^e$$

To determine s just use the first bit in the string, it is either a 1 or 0.

To determine e you take the next 8 bits and convert them to decimal. From there you subtract the bias which is **127** for single precision, or **1026** for double precision. The bias is a set number that is a part of the design of the system.

To determine m you take the remaining bits, and convert them to binary considering the first digit to be the 2^{-1}'s place, and then the following one being the 2^{-2}'s place and so on.

Decimal → IEEE

Introduction to C

Preprocessor is a system program that modifies C prior to compilation. The preprocessor directives are instructions to the preprocessor, and always begin with #.

There are two types of PPD:

- # include < library > → includes a library of functions.
- # define VAR value → defines a constant.

In C, you can comment using / ∗ ∗/

In C, every line of instructions must end in a semi-colon (;). This is the only symbol that separates lines of codes, line breaks and tabs are not interpreted by the compiler.

The following is the general skeleton of a program in C:

```
#include <stdio.h>
/* optional includes */
/* optional macros */

int
main(void)
{
    /* optional declarative statements */
    /* one or more executable statements */
    return(0);
}
```

Variables

All variables in C must be declared. There are three different variable types:

- int → integers
- double (or float) → reals
- char → characters

An *identifier* is the name of the variables used within a program. When you declare a variable you give it an identifier. The statement is in the form:

$$variable\_type \;\; identifier;$$

There are five rules to naming variables in C:

1. **Rule #1** do not use any reserved words like: double, char, int, do, float, if...
2. **Rule #2** never begin an identifier with a number.
3. **Rule #3** only use letters, digits, and underscores (no spaces or other symbols).
4. **Rule #4** never use standard identifier names used by C like: printf, scanf...
5. **Rule #5** use all caps only for macros.

A C program has two modes of operation:

1. Interactive mode: user responds to prompts through keyboard.
2. Batch mode: program uses file prepared beforehand.

Two important functions in C are the *printf()* and *scanf()* functions.

printf()

printf("text with placeholders in the form %m.nd", variable names);

Placeholders in the text are pointers for where the variables need to go, in order of how they are in the *variable names* section. A placeholder is always in the form $\% \pm m.nd$ where m is the total width of the variable in the text (including negative sign and decimal place), n is the number of decimal places used, and d is the specific tag used to identify the data type used. The plus or minus tells whether the additional spaces should be put on the right or left of the number (this is called *justification*).

- %d → integers
- %lf → long float (double)
- %c → characters (char)

scanf()

scanf("placeholders in a line no spaces", &varnames);

This function prompts the user to type in a piece of data and assigns it to the variable stated after.

Reading from a File

There are three parts to reading from a file:

1. Declaring the file.
2. Opening the file.

3. Closing the file.

Part 1 is achieved by:

$$FILE\ *in;$$

Part 2 is achieved by:

$$in = fopen(\text{"}title.txt\text{"},\text{"}r\text{"});$$

$$fscanf(in,\text{"}\%d\text{"},\&temp);$$

Part 3 is achieved by:

$$fclose(in);$$

Writing to a File

There are three parts to writing to a file:

1. Declaring the file.
2. Opening the file.
3. Closing the file.

Part 1 is achieved by:

$$FILE\ *out;$$

Part 2 is achieved by:

$$out = fopen(\text{"}title.txt\text{"},\text{"}w\text{"});$$

$$fprintf(out,\text{"}\%d\text{"},\&temp);$$

Part 3 is achieved by:

$$fclose(out);$$

Working with Variables

A variable can only hold one value. Before you use a variable, you need to declare it using:

$$variable\_type\ \ identifier;$$

Each variable has an address in memory, which can be directly accessed using &identifier operator. A pointer variable is a special type of variable that only holds the address of another variable. You declare a pointer variable using:

$$variable\_type*\ \ pointer\_identifier;$$

And then later you assign it to be a pointer to the address of a variable of the same data type:

$$pointer\_identifier = \&identifier;$$

Inaccuracies

- **Cancellation Error**: happens when the magnitude of two operands are too different.
- **Arithmetic Underflow**: happens when some operand is too close to zero and gets interpreted as such.
- **Arithmetic Overflow**: happens when some operand is too large.

Expressions

Expressions are combinations of numbers (or variables which contain numbers) using operations. The standard operations are

$$+ \quad - \quad * \quad / \quad \%$$

% is called modulus and returns the remainder after division of the two numbers, as in:

$$5\%2 = 1$$

An **integer expression** contains only integers, and will return an integer even if the actual result should be a double. For example:

$$5/2 = 2$$

Since both 5 and 2 are integers, the result must be an integer. C always rounds down in these cases.
A **mixed expression** is an expression that contains both integers and doubles, this type of expression will return a double. For example:

$$5.0/2.0 = 2.5$$

$$5.0/2 = 2.5$$

$$5/2.0 = 2.5$$

$$\text{(double) } 5/2 = 2.5$$

The final example is called **Type Casting**, as in you define what type the output should be by writing it in brackets before the expression.

Order of Operations

In C, the order of operations are:

1. Parenthesis
2. Unary Operations (right to left)
3. Binary Operations: $*$ / % (Left to right)
4. Binary Operations: $+ -$ (Left to right)

Comparison Operators

< > <= >= == and ! = are the comparison operators. They compare the values on the left and right side are return true or false.

The logical operators are:

- && which is the and operator, returning true if both the left and right expressions are true.
- || which is the or operator, returning true if either the left or right expressions are true.

You can compare symbols and characters because they each have an ASCII number related to them, and so the operator compares that number. It is for this reason that 9 is not the same as "9".

In all cases, a return of 0 is false, and any other number is true. The **compliment** of an expression is the same expression but the output is the opposite. You can simply achieve this by putting the expression in brackets and then putting an ! in front of the brackets, essentially "not-ing" the whole thing.

If Statements

Syntax:

```
if (condition) {
        executable code;
    } else if (condition) {
        executable code;
    } else {
        executable code;
    }
```

Only certain parts of this code will run depending on if the condition is true at the time of it running. The code will run from top to bottom, and the first time a condition is true the code will run. The code will then entirely skip all of the remaining *else if / else* sections, and continue after the if statement. If you do not want the code to skip the rest of the conditions, make them *if statements* and not *else if statements*. Note: the first condition cannot be an *else if / else* statement.

If statements can be nested within other if statements.

Switch Statements

The switch statement is useful if you want to make an if statement but you know that the condition can only have a set amount of options. Those options are called *control values*, and can only be of type int or char.

The main difference is that an if statement's condition can be a range of values, while a switch statement's condition can only be a fixed discrete value. Here is the syntax:

```
switch(control variable) {
    case value1:
        executable code;
        break;
    case value2:
        executable code;
        break;
    case value3:
        executable code;
        break;
    default:
        executable code;
}
```

Each *value* is one of the options that the variable *control variable* can be. If that is the case, then the executable code will run. The line (break;) needs to be written at the end of each case. This tells the program to leave the switch statement. Default is the code that executes if *control variable* matches none of the cases.

Loops

A **loop** is a control structure that repeats a segment of code in a program. The **loop body** is the segment of code being repeated.

There are 5 main types of loops, each with many different ways to actually implement into code:

1. **General Conditional Loop** is a loop that checks a certain condition and repeats the statements in the loop body so long as the condition is true at the beginning of each new loop, otherwise it skips the loop body. This type of loop encompasses all others.

2. **Counting Loop** is a loop with a fixed number of repetitions.

3. **Sentinel-Controlled Loop** is a loop that reads values either from a file or the used input and stops looping when a certain value is entered.
4. **End-of-File Controlled Loop** is a loop that reads values from a file and stops when there are no more values to read.
5. **Input Validation Loop** is a loop that keeps asking for a value from the user until the input meets some criteria.

The *while* Loop

The while loop repeats the loop body as long as the condition is true at the beginning of each loop. The syntax is:

```
while (condition) {
    executable code;
}
```

The idea is that the condition is somehow updated within the while loop, as in at some point the loop will run enough times to make the condition false, exiting the loop. The while loop is typically used when we don't know the number of iterations we want of the loop body before running the code.

The *for* Loop

The for loop repeats the loop body a set amount of times, and uses an index variable which definitely gets updated in some way each time the loop runs, typically counting the number of times the loop has run. Here is the syntax:

```
for (initialization; condition; update) {
    executable code;
}
```

In the *initialization* section, the index variable is initialized, as in something like:

$$\text{int n} = 0;$$

Then the *condition* is the expression that has to be true at the beginning of each loop for the loop body to run, this condition typically involves the index variable. The *update* is another line of code that runs every time the loop body ends, this update typically also involves the index variable with something like:

$$i++; \equiv i+=1; \equiv i = i+1;$$

The for loop is typically used when we do know the number of iterations we want of the loop body before we run the code.

The *do-while* Loop

The do-while loop is essentially a while loop that runs the loop body once regardless of any condition *and then* checks the condition to decide whether the loop body should run again. The syntax is:

---

do {
    executable code;
} while (condition);

Status

Status is a way to get an integer value regarding the number of available data. For example:

$$status = scanf(\text{"}\%d\%d\text{"}, \&a, \&b);$$

... would return status = 2. If you were reading from a file, and reached the end of the file, the status would return -1.

Modular Programming and Functions

Functions separate programs into modules, corresponding to the individual steps in a problem solution. As you can recall from math, a function is something that can take any number of inputs and gives an output, and only returns one output for every input. We call the values sent to a function the *arguments*.

Essentially, a function is another program that is defined before the *main* program, which can be called upon at any point after it is defined. The functions definition is done after the prepossesser directives, and before the *main* program. Here is the syntax for defining a function:

Defining a function:
functiontype
functionname (declarations of each argument separated by commas){
    executable code;
    return(result);
}

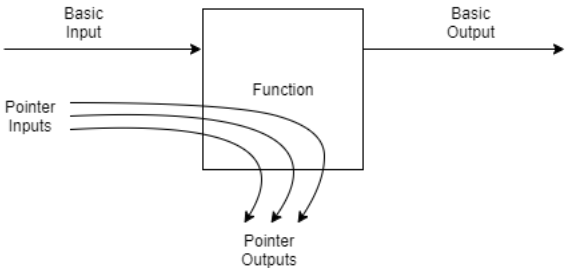
Calling a function:
functionname (inputs separated by commas);

Function Type is the type of variable that the function will return. Note that a function of type *void* is possible, and this function returns nothing. *Function Name* is the identifier used to call the function. The *result* variable inside the return() statement is what the function will output, and the variable type of *return* needs to match the *function type*. A return() statement always ends the function execution and returns the control back to the main program.

A notable part of functions is that **variables declared inside a function are unknown to the main program, and vice versa.** The only way to communicate between programs is through the arguments of the function. The **N**umber, **O**rder and **T**ype of variables used in the function call all matter.

Functions with Multiple Outputs

Unlike in math, in C programming functions can result in multiple outputs in a non-traditional way. To begin, observe the following diagram and note the different types of inputs and outputs:



The basic inputs and output are the same as described before, and can be used at the same time as *pointer* inputs and outputs. For a pointer input, in the definition of a function, define one or more of the input variables to be a pointer variable, as in a variable that will take in and hold an address. For example:

Defining a function:

```
functiontype
functionname (int y, int *addr1, double *addr2){
        executable code;
        return(result);
    }
```

The previous function would take in 3 arguments in it's function call, one integer, one address of an integer, and one address of a double. At this point, the function is capable of changing the value stored in those variables we've pointed to by using a line of code like the following:

```
*addr1 = 5;
```

It is at this point, that $^*addr1$ and the change to the variable its pointing to is essentially an output of the function, also called a *pointer output*.

Recursion

Recursion is calling a function within itself. Certain processes are easily implemented by a recursive function definition. Usually, there is some condition that needs to be met for the recursion to stop.

Recursive Behavior can be defined by two properties:

1. A simple base case.
2. A set of rules which reduce all other cases to the base case.

For example. the factorial function can be defined as $0! = 1$ (base case) and for all $n > 0, n! = n(n-1)!$ (rule which reduces all inputs n to the base case).

Arrays

An array is a list of variables all with the same type. Like a list of the ages of everyone in Toronto, or a list of all their first initials. One array can hold any number of entries. You declare an array using the following syntax:

```
type arrName[length];

type arrName[length] = entries;

type arrName[] = entries;
```

When you declare an array to have a length of n, then to access the last entry in the array you must say $aarName[n-1]$. This is because the index (which is used to keep track of what entry you are referencing in the array) begins counting at 0, so the first entry in the array can be accessed by: $arrName[0]$, while $arrName[1]$ would give you the second entry. Note that $arrName[n]$ does not exist. The reason this happens is that $arrName$ results in the address of the 0^{th} entry in the array (just as $\&arrName[0]$ would). The address of the 1^{st} entry in the array (index of 1) is the address of the 0^{th} entry plus 1. So the address of the n^{th} entry in the array is the address of the 0^{th} entry plus n.

To assign values to an array you can do it as you would a regular variable, you just need to specify the index of the entry in the array you want to write to.

sizeof()

The **sizeof()** function does not return the size of an array as you would expect. First of all, if you were to do $sizeof(int)$, depending on your system specifications, the output would most likely be 4. This is because an integer variable takes up 4 bytes of space on the memory. The input to the function can have an array as an input, however it will return the length **in bytes** of the array. For example an integer array of length 6 would have an output of 24, because each of the 6 entries takes up 4 bytes ($6 \times 4 = 24$).

Arrays into Functions

To pass an array into a function, the best thing is to just send the address of the 0^{th} element, and then work from there, this lets the function actually change the array as seen from the main program. You **must** also send the size of the array as a separate argument to the function as otherwise the function will have noway of knowing the length of the array. In the function call it would look like:

---

functionName(arrName, size);

... and in the function definition it would look like:

---

functionType functionName (type arrName[], int size) { }

Dynamic Allocation

Dynamic allocation is the method which creates array of variable sizes, not set specifically in the code. Dynamic allocation is very useful and is the method used in most actual applications of arrays. There is a very specific syntax to use for dynamic allocation of arrays and it is as follows:

---

int size;

type *arrName;

// through any method set the variable size to be the size of the array you want

arrName = (type*)calloc(size, sizeof(type));

```
// use the array normally

// at the end of the program

free(arrName);
```

Let's break down each line. The first declaration is just giving us a variable *size* which will hold the size of the array we want. The second line declares a pointer variable of the type of array we want, this variable will point to the 0^{th} entry in the array. Once you have the size of array you want (possibly by prompting the user), then the next line assigns the following to our pointer variable: *size* entries in the memory, each with enough bytes to hold (sizeof()) one *type* variable. Finally at the end of the program, you must free the dynamically allocated memory.

Strings

To approximate strings, we need to use an array of characters. To transform an array of characters into a string, the last entry must contain the **NULL** character which is: \0. There are two ways to declare a string into an array:

```
char strName[] = 'character','character','character', ···, 'character', '\0' ;

char strName[] = "string";
```

... both of those lines above do the same thing essentially, and the second one is notably simpler. Notice the use of singe quotes and double quotes, single quotes represent one character, while double quotes represent a string. Each character in a string can be accessed just like accessing entries in an array.

It is possible to have an array of strings, which is essentially a two dimensional array (something we will discuss right after strings), for example:

```
char listName[100][15];
```

... is an array of strings which can hold up to 100 city names, each with a maximum of 14 character (not 15 since the final character has to be the NULL character).

Filling a String

To fill a string with user input, you can use a scanf statement as follows (notice the $\%s$ placeholder):

```
scanf("%s", srtName);
```

... notice the lack of & since the name of the array **is** the address of the 0^{th} entry. The problem with scanf is that it considers the space to be the end of the string. To solve this we use a new function called **fgets()**.

In **fgets()** only a new line is the end of the string and otherwise it can be used the same as $scanf()$. The following is the syntax for $fgets()$ for user input and then input from a file:

```
fgets(strName, size, stdin);

fgets(strName, size, inputFile);
```

... here the maximum size of the string must be specified, but strings of smaller length will not cause an error.

For the following string related functions you must include the following in the preprocessor directives: $\#include < string.h >$

The function **strcpy(str1, str2)** will copy the contents of $str2$ into $str1$ (including then NULL character), given that there is enough allocated space in $str1$.

The function **strlen(str1)** will return the number of characters in $str1$ not including the NULL character.

Multiple Dimensional Arrays

A multiple dimensional array is effectively a matrix, where each element of the matrix can be of a specified data type (intgeres, doubles, char, etc.). You can declare a multidimensional array with a fixed size $m \times n$ using the following syntax:

```
int arrName[m][n];
```

To traverse through a 2D array and eithe read its contents or place new contents in it, you need two variables. One variable (for loop) will traverse the rows, and the other (another for loop) will traverse the columns. Essentially in the form:

```
for (int i = 0; i < m; ++i)
    for (int j = 0; j < n; ++i)
```

To send a 2D array into a functions, you send the address of the 0^{th} element, as well as its two dimensions. The function declaration would look something like this:

```
type fName (arrType arrName[m][n], int m, int n)
```

Dynamic Allocation of 2D Arrays

Recall that dynamic allocation is a method used to create arrays of variable sizes, not set before the program is run. Dynamic allocation of 2D arrays is done in two different ways.

Software Engineer's Method

The software engineer's method of dynamic allocation is actually to use a single dimensional array, and just treat it like a two dimensional array. Take the following matrix:

The position of the value 9 can be described by index $(1, 1)$ of the array. Alternatively, you could describe it as the 9^{th} entry of the single variable list, if you imagine that each line of the array goes to the end of the last one. Looking the 9^{th} entry in this particular example, it can be described by index $1 \times 8 + 1$, or as $i = [1 \times n + 1]$.

This means that in general to access the $(i, j)^{th}$ entry of a dynamically allocated 2D array using the software engineer's method, you would use:

$$arrName[i][j] = arrName[i * n + j]$$

... where n is the number of columns. The exact code to do this is below:

```
int nrows = m; int ncols = n;
arrType* arrName;

arrName = (arrType*) calloc (nrows*ncols, sizeof(arrType))
```

After this code you can treat the array like a 2D array and access it's elements using the notation mentioned above.

Computer Scientist's Method

To make a 2D dynamically allocated array using this method, you make an array of pointers which each in turn point to a single variable array. Those single variable arrays contain the values. You must allocate first the rows to be pointers to 1D arrays, and then allocate each of those 1D arrays to be the number of columns. The benefit of this is that you can access values in the array using standard notation *arrName[i][j]*. The exact code to achieve this is:

```
int nrows = m; int ncols = n;
arrType** arrName;

arrName = (arrType**) calloc (nrows, sizeof(arrType*));
for (int i = 0; i<nrows; ++i) {
    arrName[i] = (int*) calloc (ncols, sizeof(int));
}
```

Structures

A structure (or struct) is a way for you to define your own data type along with int, char, double, and so on. This new data type is a combination of the data types we know. Each struct has any number of properties which are each of a specific data type. This is useful when you are working with many versions of the same thing in your code, such as lists of people. Each person has a first name, last name, age, height and so on. Each person can be considered an **instance** of the **struct**. The struct is the general idea which contains the

properties, the instance is the specific version of a struct which has actual values (like name, age, etc.).

Declaring a struct happens before the main function and after the pre-processor directives, and looks generally like the following:

```
struct structName {
    // declarations of the variables that will be held within each instance
    varType1 varName1;
    varType2 varName2;
}
```

... and then within the main program, instances of the struct can be initiated by:

```
struct structName instName;
```

... just like declaring a variable. Now to either set or obtain values to/from each of the variables within an instance of a struct you use *dot notation* as follows:

```
instName.varName = value;
```

Typedef

Typedef is a way to create a shorthand for "struct structName instName;" turning it into something like "sAlias instName". This is achievable by:

```
typedef struct structName {
    // declarations of the variables that will be held within each instance
    varType1 varName1;
    varType2 varName2;
} sAlias;
```

For the remainder of the code "sAlias" is identical to saying "struct structName".

Instances can be declared with values immediately such as: "sAlias instName = {value1, value2}" as long as the values correctly correspond to their respective variables in the correct order.

Structs into Functions

To pass a struct into a function, simply include it in the function's formal parameters, all of the data from the instance of the struct that is passed in the function call will be passed into the function. Instances can also be returned by a function, and assigned to an instance in the main program.

Pointers to Structs

Pointers to structs are very powerful and will allow us to make *Linked Lists* which is the following topic. Before we do that we need to understand notation using with pointers to structs.

If I declare a pointer to a struct "sAlias* sPtr", then to access a variable which is being help inside of the instance that the pointer is pointing to I need to use one of the two following notation options:

(*sPtr)varName
sPtr->varName

The latter of which is preferable.

Linked Lists

Linked lists are very powerful tools used to deal with many instances of a struct. Linked lists are chains of instances of a struct in a specified order. An instance in the linked list is called a **node** of the linked list. Each node contains two values, an instance of a struct, and a pointer to the next struct in the list. The final element has $NULL$ in it's pointer value. To pass a linked list to a function, you just need to send the function the first node in the linked list since it contains the address of the next node.

I will describe in detail the following section regarding Dynamic Allocation of linked lists. Non-dynamic allocation of linked lists can be done with understanding of the previous sections.

Dynamic Allocation of Linked Lists

Say we have the following struct which we want to use to create a linked list:

```
typedef struct structName {
    // declarations of the variables that will be held within each instance
    varType1 varName1;
    varType2 varName2;
    struct structName* next;
} sAlias;
```

Now say we do not know in advance the number of nodes that we want in the linked list, and the user will somehow input this information through I/O (keyboard) or through a file.

First we must declare two pointers to an instance of the struct, one which points to the first node, and one which will act as a temporary pointer, pointing to whichever node we want to add/change/inspect at any given moment.

```
tree *p, *start;
p = start;
```

If the user then wants to input a node into the linked list, you must allocate the memory space for it under the pointer $*p$, typically this is done in a while loop which ends when the user inputs in some way they do not want to input anymore data. First however the start node must be defined outside of the loop.

```
start = (sAlias *) calloc (1, sizeof(sAlias));
p = start;
//The user should now input all the data into this start node using -> notation.
//Prompt the user for another response.

while(user still wants to input data) {
    p->next = (sAlias*) calloc (1, sizeof(sAlias));
    p = p->next;
    //The user should now input all the data into this node using -> notation.
    //Prompt the user for another response.
}
p->next = NULL;
```

Now let's understand what's happening here. Recall that we assigned a variable called next in the definition of the struct, which is a pointer to an instance of the struct. The first line allocates enough space for the starting node. The next line sets the current pointer to look at the starting node. Next we enter the while loop and the first line allocates enough space for the "next" node in the list after the current one, and then assigns the new current pointer to look at that newly created next node. This process repeats until the user in some way indicated they do not want to add any more nodes, and then the loops is ended. One final line at the end assigns the next pointer for the last element in the list to be NULL since the last node in the linked list doesn't point to anything.

Command-Line Arguments

The main program is really a function which is being called upon by the command line to run the code. This is why we call it an int, and then return(0) at the end. So far we have only used void as the arguments to the main function.

There are two possible input types to a main file. **argc** which is the number of arguments that are being input into the function (always at least 1). And **argv** which is an array of strings containing the actual values of the arguments. Note that argument zero of *argv* is the name of the program. Notice that all arguments coming from the operating system are strings (we will soon deal with how to convert these strings to integers and doubles). So our main program header can now become:

---

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

... following this in the main file, *argc* and *argv* can be used as variables as you would expect. These functions can be prompted directly from the command line which you can access on windows by going to start->run->cmd.

Numerical Arguments

To get numerical arguments from *argv* you first need a new library:

$$\#include < stdlib.h >$$

This gives us two new functions, atoi() and atof(). **atoi()** takes in a string as an input and return an integer, and stops reading the moment a character in the string is not a numerical value (not including $\pm$ or a decimal point). It will always round down. **atof()** takes in a string as an input and returns a double, and stops reading the moment a character in the string is not a numerical value (not including $\pm$, a decimal point, e or E). *Note:* atof("4.53e3") would return 45300.0.