Accelerated Learning Series

(www.ALearn.net – A site dedicated to education)

Modules in Computer Science & Engineering

C Programming

A set of notes detailing fundamental concepts in higher level Programming languages.

Author: SKG.
Dec 2008.

Revision History

Version 1.0 (Dec 2008)

- First Version Created

Table of Contents.

Prerequisites

- x86 Assembly Language Programming (ALS notes in Hardware Engineering)

Preface

What must have been obvious in our study of the x86 Assembler programming language is that writing assembly instructions to accomplish even the most basic of tasks was intricate and laborious, requiring a certain level of dedication to detail that can only be expected from the most diehard enthusiasts. While the assembler abstracted us from the binary codes by providing mnemonics such as “MOV” and “ADD”, it did little else in shielding the programmer from individual instructions executed by the CPU.

Programming a computer to perform a certain task could be accomplished with substantially greater ease if we had a way to define logic operations without having to worry about how the CPU could accomplish these operations. Things as simple as assigning a value to a variable without having to worry about which register to first put the value in and then which instruction would allow us to move the register value to a memory location, can save a substantial amount of effort and consequently allow the programmer to better focus on the task that needs to be accomplished as opposed to the implementation details in hardware. This was the incentive to pursue higher level programming languages.

At the heart of a high level language is a **compiler** or an **interpreter** that translates high level code into machine level instructions. The difference between a complied language and an interpreted language is essentially the time when the translation to machine level code occurs. A compiled language translates the code ahead of execution, thus allowing it the luxury of greater optimizations. An interpreted language does the translation at execution time and hence is generally less optimized.

The sixties and the early seventies was the period when the groundbreaking work in high level languages began. Among the earliest of these languages was one called “**TMG**” by **R. M. McClure**. This was followed by a language called “**B**” by **Ken Thompson**, which in turn was followed by the “**C**” programming language by **Dennis Ritchie**. The “C” programming language would turn out to be the most popular programming language for over a quarter century because it offered the ability to define high level logic operations without substantially compromising the ability to perform bit-wise and register level operations.

This flexibility was crucial in the use of the “C” language in writing **operating systems**, a layer that abstracts hardware platforms from application programs. Interfacing with operating systems written in “C” was more natural for applications that were themselves written in “C”. Thus the “C” programming language became the de facto standard in high level languages in the latter part of the twentieth century.

As with any programming language, “C” has a number of constructs and key words used to define operations. With the “C” language there is almost a one-to-one mapping between assembler level constructs and “C” constructs. In addition “C” has a very limited set of key words, making it very easy to learn and use.

It is my hope that our earlier study of the x86 assembler will make the study of “C” more intuitive because of the similarity in the constructs.

Acknowledgements

I am grateful to S. Dziok, M. Benson and A. Walker for reviewing this set of notes and providing valuable feedback.

1.0 – Getting familiar with a Development Environment

For the purpose of this study, a C compiler and a C linker (that links with the C-Runtime libraries), are all that we need. However, Microsoft generously provides free access to the Visual C++ Express edition for non-commercial use. Since this development tool provides very rich compiling and debugging features, I have decided to use this tool throughout this set of notes.

In this section, we will download and install this tool and get familiar with using it. We will leverage this knowledge in subsequent sections as we experiment with C constructs.

1.1 – Installing Visual C++ Express Edition

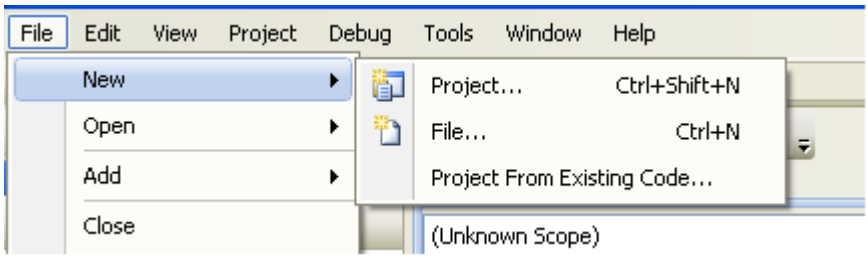
To download and install the Microsoft Visual C++ Express Edition, follow these steps.

1) Go to www.microsoft.com/express/vc/ and click on the "**Download Now**!" link.
2) This will prompt you to either save or install. Choose the save option, so that you can install later at your convenience. Provide a location on your hard drive where this tool can be saved.
3) Once the download completes, double click on the file that was saved ("vcsetup.exe") to start the install process.
4) To save on disk space and install time, do not opt to install the SQL libraries as we will not be using them in this set of notes.

1.2 – Creating an empty project

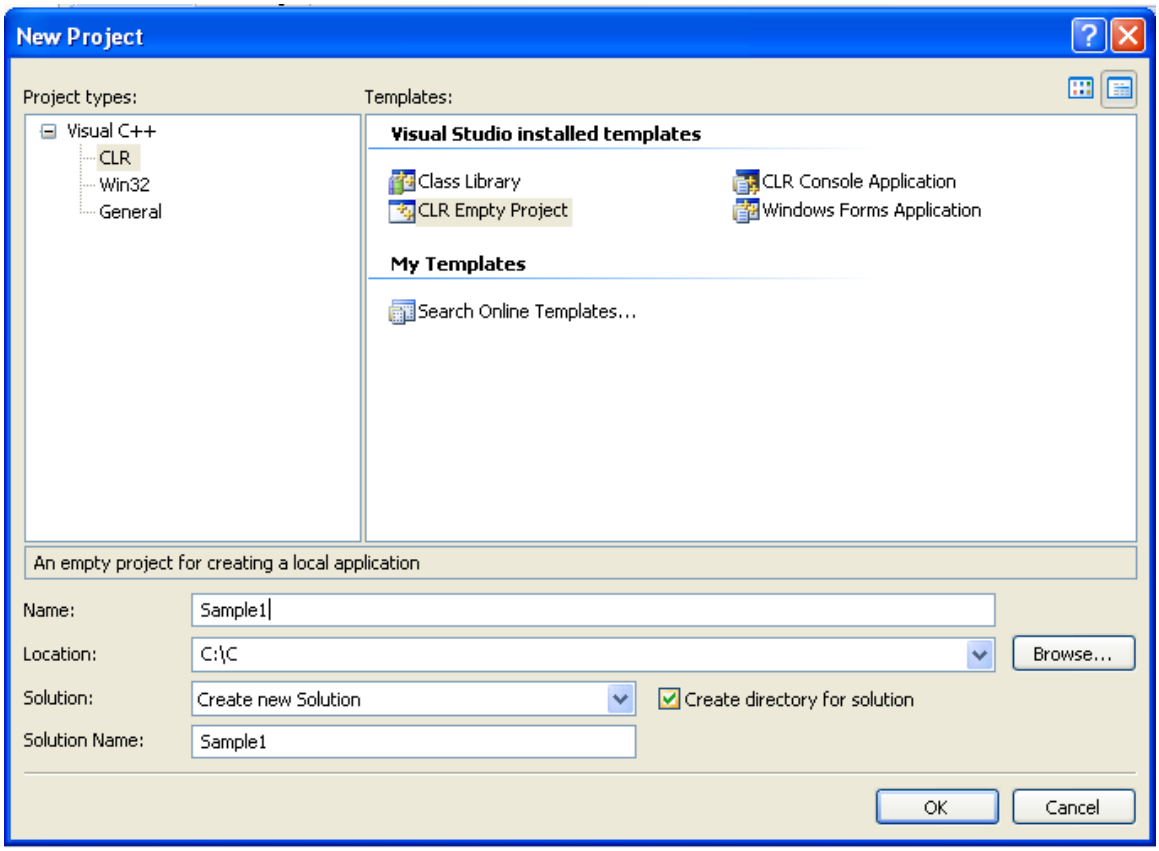
The Visual C++ development tool relies on a project environment as the framework that tracks the files used to build a program. As a first step we need to create an empty project to which we can add the files with the C code in subsequent sections.

To create an empty project, click on the “File” menu and then on the “New” menu and then on the “Project” menu, as shown below.



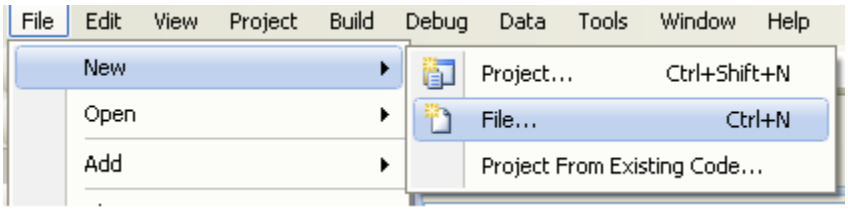
This will prompt the following dialog. Select the “CLR” option under “Visual C++” and then select the “CLR Empty Project” from the Visual Studio installed templates.

Give a name and location for the empty project as shown below and click on “OK”.

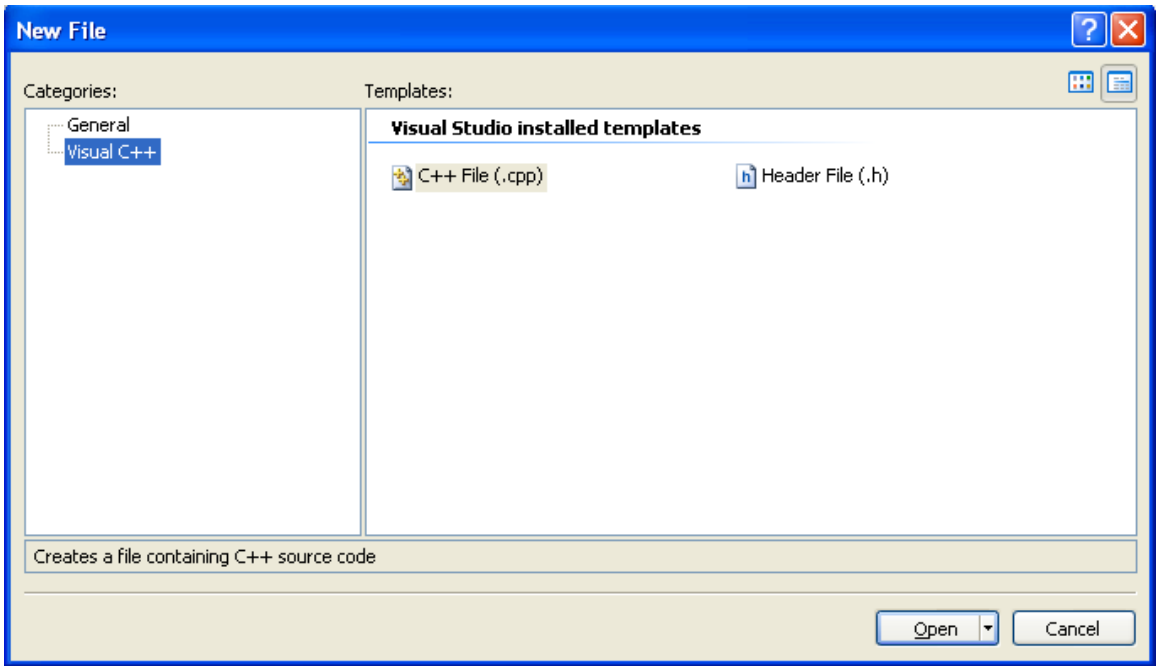


1.3 – Your First “C” Program – “Hello World!”

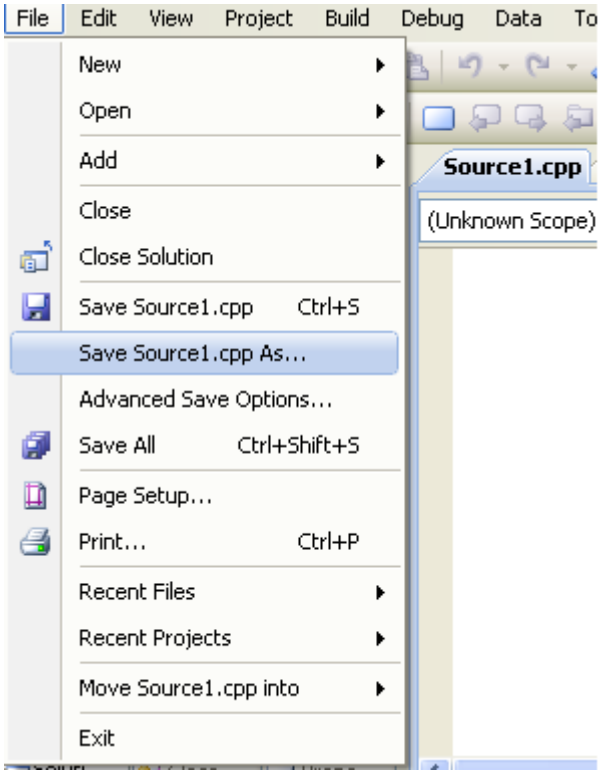
Now that you have created a blank project, you will have to create a new file to write your C code. To do this, click on the “File” menu, then “New”, then “File”, as shown below.



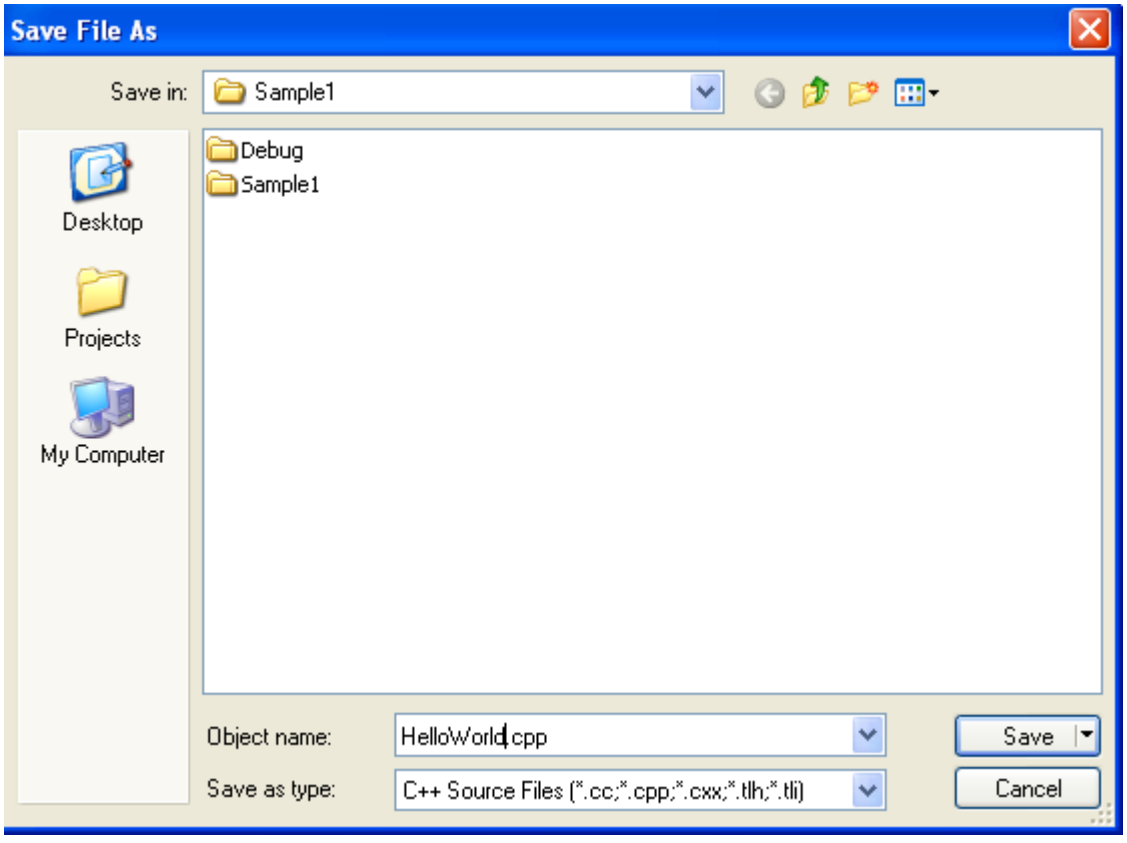
This will prompt the following dialog. Select “Visual C++” and then select “C++ File (.cpp)” and click “Open” as shown below.



This will open a file with a default name called “Source1.cpp”. To rename this file, click on “File” and then “Save Source1.cpp As…” as shown below.



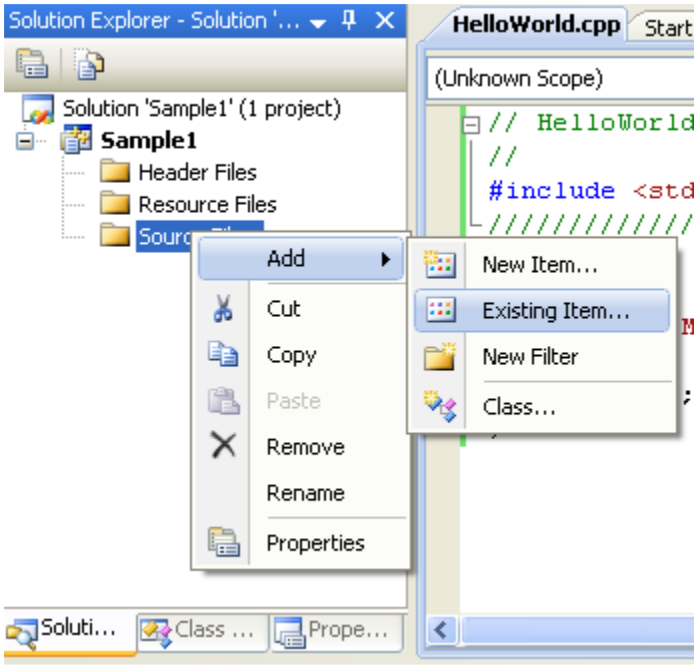
Then navigate to where you wish to save the file and give it a name. As shown below, I have called mine "HelloWorld.cpp".



Now type the following lines of code into the “HelloWorld.cpp” file in the editor (or copy and paste it) and save the file.

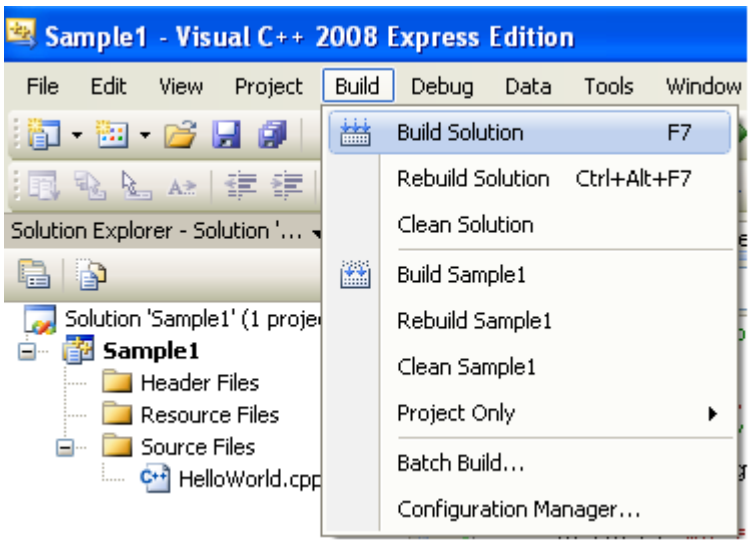
```
// HelloWorld.cpp
//
#include <stdio.h>
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
int main(int argc, char* argv[])
{
	printf("My first \"C\" program! Hello World!\n");

	return 0;
}
```

Now you are ready to add this file to your project. To do this right click on the “Source Files” folder in the Solution Explorer and click on “Add” and “Existing Item..” as shown below.



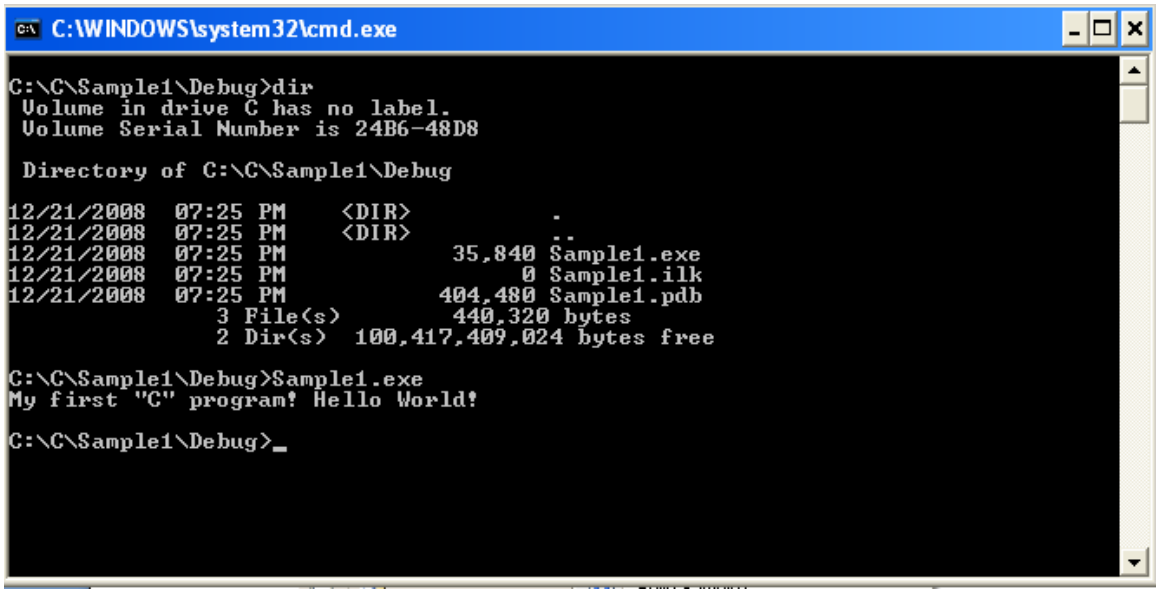
Then navigate to the location of “HelloWorld.cpp” and select it.

Now you are ready to build this program into an executable. “Building” a program is a 2 step process – first you have to compile it and then you have to link it to other libraries that this program depends on. The Visual C++ tool does both these steps when you select the “Build Solution” menu as shown below.



Assuming there were no compile time or link time errors, the build process will generate an executable file with the extension ".exe". This will be located in folder under the location where you created your empty project. This folder will be named "Release" or "Debug" depending on the build selection you have chosen.

Using the command window, navigate to this folder and find the .exe and run it. It should generate the following output.

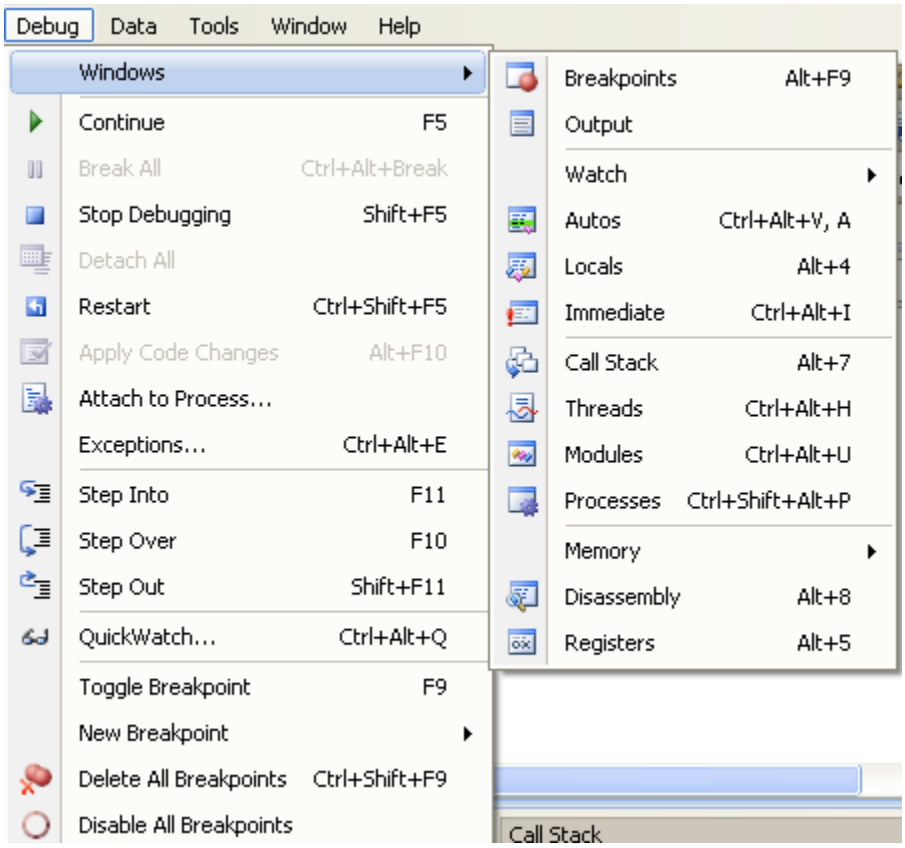


Note that Visual C++ uses a C++ compiler as opposed to a C compiler. C++ is a language that builds upon the C language. The C++ compiler recognizes all the C syntax and constructs since that are shared by the C++ language. For our purpose we will only use the C features of this compiler. Even though we are using file name with the ".cpp" extension, these files could just as well be named with the ".c" extension.

1.4 – Debugging a program

Having completed the x86 assembler module, you will certainly appreciate the debugging capabilities of the Visual C++ development tool.

The “Debug” menu item lists the common debugging features of this tool. The following screen capture was taken during a debug session. Hence you will notice all the run-time debugging options available.



The easiest way to start a debug session is to set a breakpoint at the very beginning of your program in “HelloWorld.cpp”. To do this, click on the first line in the “main” routine and press “F9” to set a breakpoint there. You will notice a solid red dot to the left of the line to indicate a breakpoint is set. Then you press F5 to run the program in the debugger. The debugger will launch the program and break when it hits your breakpoint.

```
//
#include <stdio.h>
////////////////////////////////////////////////////////
int main(int argc, char* argv[])
{
    printf("My first \"C\" program! Hello World!\n");

    return 0;
}
```

Once you hit the breakpoint, you have a very rich set of debugging options. You can view all the call stacks, Threads, Modules, Memory and Registers, in the context of your application. You can also step one line at a

time and watch your variables as you step over lines of “C” code. You also have the option to view the assembly instructions for each line of “C” code and you can even set breakpoints at the assembly instruction level. To see the assembly instructions, select the “Disassembly” window from the Debug menu.

In the course of executing the subsequent samples in this set of notes, we will exploit these features as necessary and it will prove extremely useful in increasing our productivity.

As an exercise, based on your background in assembly programming, think about how a program like the debugger is able to stop the execution of any other program at a specific location and how it accomplishes the task of stepping over single lines of C instructions.

2.0 – Data Types, Operators and Expressions

In this section we study the basic data types defined by the C language and how we can extend these types with user defined data types. Then we will study the common arithmetic, logical and bitwise operators. We will follow that with the study of a sample that uses user-defined data types.

.

2.1 – Basic Data Types

All “C” compilers recognize a few basic data types. These include the following:

char – This represents 1 byte
int – The size of an “int” is machine dependent. It is commonly 4 bytes.
float – This is single precision floating point. Its size is machine dependent.
double - This is a double precision floating point. Its size is machine dependent.

A “**floating point**” is a term used in computing to represent rational numbers (numbers that are fractions eg. ½) as a string of digits delimited by a decimal point. For example when ½ is written as 0.5, it is a floating point representation. The term “floating” (as opposed to fixed) refers to the fact that there are no assumptions made on the number of digits to the left or right of the decimal point, other than the obvious memory limitations.

Single precision refers to the storage of a value at one memory location. If the value is divided into two parts and each part is stored at a different memory location, it is referred to as **double precision**.

Although there are only 4 basic data types in “C”, there are several ways you can declare variables of each of these types.

You can use the “**short**” and “**long**” qualifiers when defining integers. These can alter the size of the integer that is used to define a variable. The actual size is implementation dependent.

You can use the “**signed**” or “**unsigned**” qualifiers with the “char” or “int” variables. Using the “unsigned” qualifier implies that all the bits in that data type can represent the data value, whereas using the “signed” qualifier implies that the most significant bit will be reserved as the sign bit.

You can also declare **constants** using the “#define” syntax as shown below:

```
#define MY_CONST_VALUE        100
```

Subsequent to the above declaration, every time you use “MY_CONST_VALUE” in your program, the preprocessor (a utility that runs prior to the compiler) will substitute it with “100”. This is also referred to as a **macro expansion**. And the constant names are referred to as **macros**.

The way you declare a variable using these basic data types ultimately dictates the size and representation of the data. For example, each of the declarations below use the type “char”, but each variable is of a different size.

```
#define MY_ARRAY_SIZE 10

char    var1;
char    var2 [ MY_ARRAY_SIZE ];
char    *var3;
```

“**var1**” is a variable of type “char”. So it is a 1 byte variable.

“**var2**” is an array of 10 characters (Note the compiler would substitute “MY_ARRAY_SIZE” with 10 because of the earlier constant declaration). **In the “C” language, array indices always start with “0”**. So “var2[0]” is a char as is “var2[9]”. Note that “var2[10]” does not exist and using it will cause undesired results.

“**var3**” is an interesting variable declaration. The leading “*****” implies it is a “**pointer**” to a variable of type “char”. A pointer variable is perhaps the most complex variable type to understand in the C language. It is also the variable type that allows the C language to be particularly powerful and efficient in data manipulation. We will be using this data type in subsequent sections. For now think of it as a data type that stores the address of a memory location. In a 32-bit machine, each address is 32-bits wide. So a pointer data type will be 32 bits, even though it is pointing to a single byte of memory (char) at that location.

Note that while C only accounts for the four data types above, most compilers offer a greater set of basic data types. The Visual C++ compiler offers a “BYTE” (1 byte), “WORD” (2 bytes) and a “DWORD” (4 bytes).

2.2 – User Defined Data Types

Structures:

Often times you want different data types to represent different aspects of a single entity. For example if you wanted to store information about a student in your school, you probably want a field to store the student's name, another to store their age, yet another to store their grade and so on. The name can be defined as an array of characters, the age and grade will likely be unsigned integers. The C language allows you to define a structure containing these three fields as follows:

```
struct StudentRecord
{
        char            Name;
        unsigned int    Age;
        unsigned int    Grade;
};
```

Subsequent to the above definition of "StudentRecord", you can declare a variable of type "StudentRecord" as follow:

struct StudentRecord myRecord;

You can also declare the variable "myRecord" at the same time you defined the structure as follows:

```
struct StudentRecord
{
        char            Name;
        unsigned int    Age;
        unsigned int    Grade;
} myRecord;
```

"myRecord" is declared as a variable of type "struct StudentRecord". You can access each of the fields in this type using the "." qualifier as follows:

```
myRecord.Name
myRecord.Age
myRecord.Grade
```

If you wanted to declare "myRecord" as a pointer to a structure of type "StudentRecord", then you would declare it as follows:

```
struct StudentRecord *myRecord;
```

In this case, you access individual member fields of myRecord using the "->" qualifier as follows:

```
myRecord->Name
myRecord->Age
myRecord->Grade
```

Note that when declare "myRecord" as a pointer, you have only reserved space for an address. You have not reserved space for the actual contents of the variable. We will discuss this in more detail in the section on pointer variables.

Type Definitions:

If you wanted to avoid using the "struct" qualifier each time you declared a variable of type "struct StudentRecord", then you would define "StudentRecord" using the "typedef" "C" definition as follows:

typedef struct

```
{
        char            Name;
        unsigned int    Age;
        unsigned int    Grade;
}StudentRecord;
```

You can then declare a variable of type “StudentRecord” as follows:

StudentRecord myRecord;

Notice how “StudentRecord” is treated as if it were a basic data type.

Arrays vs. Linked lists:

Often times, C programmers maintain a list of a particular data type. One way to implement this would be to use an array as follows:

StudentRecord myRecord[10];

This can be rather inflexible, however. You need to know ahead of time the total number of elements you wish to maintain. Adding or removing a record from anywhere other than the end of the array will requiring shifting other elements.

The fact that C supports pointers allows for an alternative. If each record had a field that pointed to another record of its own type, we could construct a linked list of elements of a particular type. The following definition shows how this is done.

```
typedef struct sRecord
{
        char            Name;
        unsigned int    Age;
        unsigned int    Grade;
        struct sRecord* NextRecord;
}StudentRecord;
```

Now you can allocate memory for a “StudentRecord” whenever you need to and add this memory location to your list by simply setting the “NextRecord” to point to it. Here is an example:

```
StudentRecord   *listHead;
StudentRecord   var1, var2;

listHead = &var1;
var1.NextRecord = &var2;
var2.NextRecord = NULL;
```

We declare “listHead” as a pointer to “StudentRecord”. This means listHead can be assigned to an address where a “StudentRecord” variable is located.

Then we declare 2 variables (var1 and var2) of type “StudentRecord”. In C we determine the address where a variable is located using the “&” qualifier. So we can assign listHead to the address where “var1” is located as shown above. We can then set the “NextRecord” field in var1 to point to the address where “var2” is located. And finally point the “NextRecord” field in var2 to NULL to indicate that it is not pointing to anything. Assigning a NULL to the last pointer in the list is a common practice in C programming and is often referred to as **NULL termination**.

With those three simple assignments, we have created a linked list of two records. We can add further records at any point and we can slot them between any two records without requiring the shift of other records in memory. You can now begin to appreciate the power of pointers.

Bit-Fields:

In our examples thus far we had a need to use a basic data type such as a “char” or “int” for each of the fields in our structure. What if the data we wanted to store was binary in nature – on or off? We could still store it as a char, but it would be an utter waste of the remaining 7 bits in the “char” data type. We would only need 1 of the 8 bits for our purpose. Similarly, if our data only needed 3 bits (meaning 2^3 or 8 permutations), we would be wasting memory by using a “char” for such a variable. To remedy these situations, C allows us to assign variable names to bits within a data type as follows:

```
struct
{
        unsigned int    flag_bit_0      : 1;
        unsigned int    flag_bit_1      : 1;
        unsigned int    flag_bit_2_3    :  2;
} flags;
```

You can now access individual bits within the “unsigned int” variable called “flags” as follows:

```
flags.flag_bit_0            /* To access bit zero */
flags.flag_bit_1            /* To access bit 1 */
flags.flag_bit_2_3          /* To access bit 2 and 3 */
```

Note that anything that prefixed with “/*” and terminated by “*/” is assumed to be a comment by the C compiler. The C++ compiler also allows anything up to the end of a line, after “//” to be ignored as a comment.

Unions:

Sometimes you may find the need to have different data types to represent things that can otherwise be considered similar. For example if you were monitoring a set of sensors, where each sensor provided data that was either an integer or a floating point, you would probably want to maintain an array or linked list of sensor data items. But it would get a bit tricky if some of these nodes had to have floating point data while others had to have integer data. One way to handle this would be to allow a structure to have 2 fields – int and a float, but that would be a waste of memory.

C allows for a “union” variable type where you can define multiple fields with multiple types and the compiler will allocate memory corresponding to the largest type within the multiple types and depending on which field name you use to access the variable, an assumption will be made on the data type you wish to use. The following is an example of a union declaration.

```
union
{
        int     var_int;
        float   var_float;
} union_var;

union_var.var_int               //To use this location as an int
union_var.var_float             //To use this location as a float
```

2.3 – Arithmetic, Logical and Bitwise operators

Arithmetic Operators

"C" uses the following arithmetic operators:

+	Addition
-	Subtraction
*****	Multiplication
/	Division
%	Modulus
++	Increment by 1 eg. Var1++; // Var1= Var1 +1;
--	Decrement by 1 eg. Var1--; // Var1 = Var1 – 1;
+=	Add and assign eg. Var1+=Var2; // Var1 = Var1 + Var2;
-=	Subtract and assign eg. Var1-=Var2; // Var1 = Var1 – Var2;
***=**	Multiply and assign eg. Var1*=Var2; // Var1 = Var1 * Var2;
/=	Divide and assign eg. Var1 /= Var2 ; // Var1 = Var1 / Var2;
%=	Mod and assign eg. Var1 %=Var2; // Var1 = Var1 % Var2;

Logical Operators

"C" uses the following logical operators

>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not Equal to
&&	Logical AND
||	Logical OR

Bitwise Operators

"C" uses the following bitwise operators

&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
<<	Bit Left shift
>>	Bit Right shift
~	One's complement

Note that in general "*", "/" and "%"get executed before "+" and "-". There are similar precedence rules for all of these operators when used within the same execution line. The easiest way to avoid precedence rule errors is to make sure your wrap the operations that need to be executed first within parentheses. The following is an example of the use of parenthesis to explicitly state the precedence order.

(var1 & 0x1F) == 0;

Here, we are trying to check if the lower 5 bits of var1 is zero. Note how we make sure that the bitwise AND happens before the equality check, by wrapping it in parentheses.

2.4 – Common Syntax and Expressions

The most productive way to learn the syntax and expressions used in the C language is to write samples that exploit the knowledge we have already gained.

The following sample declares a user defined structure and then allocates two variables of that type, creates a linked list with those variables and then traverses the list.

In the process we use the assignment expression "=", and the "if" and "else" expressions. We also learn the syntax used in writing a "C" expression and how an expression is terminated. Let us study the expression below:

```
var1.Age = 10;
```

Here we are assigning a value to the "Age" field of the variable "var1". We indicate that our expression is complete with the help of a semicolon. The semicolon indicates to the compiler that the expression is terminated. A terminated expression is also known as a "statement" in "C". There is no reason (other than ease of readability) to put statements on different lines. The compiler recognizes a complete statement based on the semicolon. If you wish, you can put all your statements on the same line in the editor and the compiler will have no objections.

Another "C" syntax is the use of blocks that are defined with curly braces as shown below.

```
if( currentStudent != NULL )
{
        /* Block 1 */

}
else
{
        /* Block 2 */
}
```

This is a way to tell the complier that the code within a block should only be executed if the condition specified at the entrance to the block is satisfied. In the example above the "if" clause confirms that the variable "currentStudent" is not NULL. If that condition is met, then the code in block 1 is executed. If that condition is not met, then the code in block 2 is executed as part of the "else" statement. Note that there is no requirement to have an "if" clause prior to using the curly braces to define a block. You can have unconditional blocks. It does not serve any particular purpose, but you may find it useful while debugging.

Note that in the sample we use the "printf" command. This is actually a function call (similar to a procedure call in assembly) in C. The function is implemented as part of the C-Runtime library to which the linker will link your program after the compiler has compiled the program. The compiler, however, needs to know the signature (argument list and return values) of functions implemented in libraries to validate your call and find the best way to pass these arguments and accept return values. Header files are used for this purpose. We include header files at the top of the file (eg. #include <stdio.h>). These header files have the function declarations for the functions implemented in libraries. These function declaration are also known as function prototypes.

Copy and build the sample below and step over each line in the debugger as you study what it is trying to accomplish.

```
/* Sample2.cpp */
#include <stdio.h>

typedef struct sRecord
{
        char                    Name[100];
        unsigned int    Age;
```

```
        unsigned int   Grade;
        struct sRecord *NextRecord;
}StudentRecord;

int main(int argc, char* argv[])
{
        StudentRecord  *headList, *currentStudent;
        StudentRecord  var1, var2;
        unsigned int   count;

        /* Print the size of the Basic data types.*/
        printf("Size of Basic Data types are... char=%d, int=%d, float=%d double=%d long
int=%d short int=%d\r\n",
                        sizeof(char), sizeof(int), sizeof(float), sizeof(double),
sizeof(long int), sizeof(short int) );

        /* Print the size of our structure. */
        printf("Size of the Student Record Structure is %d\r\n", sizeof(StudentRecord));

        /* Set the age of the 2 students we are tracking */
        var1.Age = 10;
        var2.Age = 12;

        /* Create a linked list of the 2 student records.*/
        headList = &var1;
        var1.NextRecord = &var2;
        var2.NextRecord = NULL;

        /* Parse the linked list and print the student ages */
        count = 0;
        currentStudent = headList;
        if( currentStudent != NULL )
        {
                count++;        /*Increase student count */
                printf("Student %d is %d years old\r\n", count, currentStudent->Age );
                currentStudent = currentStudent->NextRecord;
                if( currentStudent != NULL )
                {
                        count++;        /*Increase student count */
                        printf("Student %d is %d years old\r\n", count, currentStudent->Age
);
                        currentStudent = currentStudent->NextRecord;
                        if( currentStudent != NULL )
                        {
                                count++;        /*Increase student count */
                                printf("Student %d is %d years old\r\n", count,
currentStudent->Age );
                                currentStudent = currentStudent->NextRecord;
                        }
                        else
                        {
                                printf("There are only %d students in the list\r\n", count
);
                        }
                }
                else
                {
                        printf("There are only %d students in the list\r\n", count );
                }
        }
        else
        {
                printf("There are only %d students in the list\r\n", count );
        }

        return 0;
}
```

3.0 – Execution Flow Control

In our study of the x86 assembler programming we discovered that “jump” and “loop” instruction were essential in our ability to control the flow of our execution. The “C” programming language provides similar constructs for flow control. We will study these constructs in this section.

3.1 – if - else if - else

We have already seen the use of an “if” statement in our previous sample. The “if” statement is the most common method to decide on an execution path in “C”. The following is the syntax for using an “if” statement.

```
if( expression 1 )
{
        /* Block 1 */

}
else if ( expression 2 )
{
        /* Block 2 */
}
else
{
        /* Block 3 */
}
```

Note that the expression within the bracket after the “if” and “else if” are not terminated with a semicolon. Hence they are not statements by themselves. They are checks that result in a TRUE or FALSE evaluation. For example, in the “if” statement below we are checking if the variable “currentStudent” is not a NULL. This check does not change the value of “currentStudent” nor does it change anything else. It simply determines if we should execute the code within the subsequent block.

```
if( currentStudent != NULL )
{
        /* Block 1 */

}
```

Note that if you only have a single statement within a block, you don’t need the curly braces to define the block, although it is highly recommended for ease of readability and subsequent maintenance of the code.

You can have any number of “else if” checks as required.

The final “else” fields the case where none of the expressions in previous “if” and “else if” statements were evaluated to a TRUE.

3.2 – While, For and Do-While loops

While loop:

The “while” loop allows the repetitive execution of a block of code until the “while” expression returns FALSE. The expression is specified much like it was done with the “if” statement.

```
while( expression )
{
        /* Block of code */

}
```

Unlike the “if” statement, we don’t get out of the block when we get to the last statement in the “while” block. Instead we go back and reevaluate the expression in the “while” loop and if it is still TRUE, we will again execute all the code within the while block.

Note how the “while” construct would have been helpful to us in our previous sample. Instead of having nested “if” statements, checking if the “currentStudent” is NULL, we could have used a single “while” statement as follows:

```
/* Parse the linked list and print the student ages */
count = 0;
currentStudent = headList;
while( currentStudent != NULL )
{
        count++;        /*Increase student count */
        printf("Student %d is %d years old\r\n", count, currentStudent->Age );
        currentStudent = currentStudent->NextRecord;
}
```

For loop:

The “for” loop is an extension of the “while” loop.

Notice how with a “while” loop we had to do some initialization before we evaluated the while expression (eg. Assigning “currentStudent” to “headList”).

We also had do some more initialization at end of the while block (eg. Reassigning “currentStudent” to the next record).

The “for” loop allows us a way to account for these initializations as part of the statement construct itself.

The following is the syntax for a “for” loop. Notice how it allows for 2 statements and 1 expression. Statement 1 is executed the first time we enter the for loop. Expression1 is evaluated each time we enter the for loop and only if the expression evaluates to a TRUE will we execute the “for” block of code. Statement 3 is executed at the end of each iteration of the loop.

```
for( statement1; expression1; statement2 )
{
        /* Block of code */

}
```

Our while sample above could be rewritten into a for loop as follows:

```
/* Parse the linked list and print the student ages */
```

```
count = 0;
for(            currentStudent = headList;
                currentStudent != NULL;
                currentStudent = currentStudent->NextRecord;  )
{
        count++;        /*Increase student count */
        printf("Student %d is %d years old\r\n", count, currentStudent->Age );

}
```

Note that you can have multiple statements as part of statement1 and statement 2 in a “for” loop. In this case, you separate them with a comma.

Do-While loop:

With both the “while” and “for” loops, the expression that determined if we were going to execute the code in the loop block was executed at the start of the loop. The “do-while” allows for the expression to be evaluated at the end of the loop instead, thus ensuring that the loop code is executed at least once.

The syntax for the do-while is as follows:

```
do
{
        /* Block of code */

}
while( expression )
```

3.3 – Break and Continue

Within any loop block, it is sometimes convenient to either exit out of the loop unconditionally prior to getting to the expression gate in the loop block or re-check the expression and start at the beginning of the loop block without going all the way to the end of the loop block. C allows for both these options.

To exit from a loop block at anytime, use the “break” statement as follows:

```
for( statement1; expression2; statement3 )
{
        /* First block of code  */

        break;

        /* Second block of code */

}
```

The “break” statement within this loop block will prevent the second block of code from ever being executed. Often you will have an “if” clause to determine if you wish to call a “break”.

To continue at the top of a loop block without going through the rest of a loop, you can use the “continue” statement as follows:

```
for( statement1; expression2; statement3 )
{
        /* First block of code  */

        continue;

        /* Second block of code */

}
```

The “continue” statement within this loop block will prevent the second block of code from ever being executed. Often you will have an “if” clause to determine if you wish to call “continue”.

3.4 – Switch Statement

All the execution control statements that we have studied so far allowed for the execution of a block of code based on the evaluation of an expression to be TRUE. If the expression was evaluated to be FALSE, we would not execute the associated code block. **Note that in C, TRUE refers to all values that are non-zero. FALSE refer to a zero value.**

The "switch" statement allows the possibility of executing different blocks of code for different integer expression evaluations, as opposed to the binary, TRUE and FALSE evaluations. So if an expression evaluated to a "2", you can specify a code block which is different from a code block that will be executed if the expression evaluated to a "3", for example.

The following is the syntax for a "switch" statement:

```
switch( expression )
{
        case 0:
        {
           /* Code Block for 0 */
        }
        break;

        case 1:
        {
           /* Code Block for 1 */
        }
        break;

        case 2:
        {
           /* Code Block for 2 */
        }
        break;

        default:
        {
           /* Code Block for default */
        }
        break;
}
```

Note that the expression evaluation determines the starting case block. To ensure that you don't execute the case block subsequent to the expression evaluation case, you need to use the "break" statement at the end of each case block. In some cases, you may want to fall through to the subsequent block. If so, you can avoid the call to "break".

The "default" case fields all cases for which you have not declared an explicit case block.

3.5 – Goto Statements

Although frowned upon by advocates of structured programming, C provides a way to abruptly change the execution path at anytime by calling the “goto” statement with a label identifying where to move to.

A label is a name, much like a variable name, that must be followed by a colon. It serves to identify a location within your program. A label name is scoped at a function level. In other words, you can have the same label name multiple times within your program, as long as they are in different functions within the program.

Though a “goto” statement has the possibility of making the task of studying code paths particularly difficult, there are situations where its use is legitimate. Take the case where you have nested loops. If you wanted to get out of all the loops because you have encountered a catastrophe, a “goto” might come to the rescue. A “break” will be insufficient for this purpose, because it will only get you out of the innermost loop.

The following is an example of the syntax used with a goto statement:

```
int MyFunction( void )
{
        while ( expression )
        {
                while ( expression )
                {
                        If( Major_Error ) goto Error;

                }
        }

        return(0);

Error:
        /* Execute error recovery */


        return(-1);
}
```

Notice how this function will return “-1” if an error is encountered. Otherwise it will return “0”.

4.0 – Program Structure

In the previous sections we discussed data types and execution flow constructs. In this section, we will discuss how we generally structure the code in C programs. In the process we will study function calls and the significance of various variable declaration options.

4.1 – Function Calls

Every C program has a “main” entry point. This is where the execution will begin once your program is loaded. Theoretically, you could write your entire code within the “main” routine, but in reality this will be difficult to maintain and you will deny yourself the benefit of code reuse. A more elegant alternative would be to isolate generic functionality into procedures, much as we did in the assembly language. These procedures are referred to as functions in the C language.

Every function in C is uniquely identified by a function signature. In C the signature of a function is essentially the name of the function. A function declaration will state its name, the arguments (if any) that the function takes, and its return type.

In the declaration below we have a function by the name “MyFunction1” that takes two arguments, an integer and a character and it returns an integer.

```
int MyFunction1( int arg1, char arg2 );
```

If “MyFunction1” was not going to return any value, it would be declared as follows:

```
void MyFunction1( int arg1, char arg2 );
```

Similarly if “MyFunction1” was not going to take any arguments, it would declared as follows:

```
int MyFunction1( void );
```

Most computer languages subsequent to C use more than the function name as the function signature. C++ for example uses the argument and return types as part of a function signature. This means that functions with the same name can be responsible for different actions. This is also known as function overloading. Languages that use argument and return types as part of the function signature, are also referred to as “**strongly typed**” languages. Consequently C is sometimes referred to as a “**weakly typed**” language.

The original implementation of C was even more weakly typed than the current ANSI standard implementation. The ANSI standard allows the function declarations to state all the data types for both arguments and return values, thus allowing compilers to check that declarations match function definitions and calls.

A function declaration should be encountered by an ANSI C compiler prior to a function call, else it is unable to cross check the parameters being passed to a function and the return values being accepted from functions. To meet this requirement, C programs will declare functions (also known as **function prototypes** or **forward declarations**) at the top of a C file as shown below:

```
int MyFunction1( int arg1, char arg2 );

int main(int argc, char* argv[])
{
        int retVal;

        retVal = MyFunction1( 1, 'c' );

        return( retVal );
}
```

Here the “main” routine is making a call to “MyFunction1”. The compiler needs to confirm that the arguments being passed to “MyFunction1” are of the types that it expects and that return value of “MyFunction1” matches the type of variable that is being assigned to it. The “MyFunction1” declaration at the top of file serves this purpose.

Note that the actual definition of “MyFuntion1” is not needed by the compiler. The actual definition is only needed at link time, when all the compiled code (also known as object files), are linked to form an

executable. Thus the definition of “MyFunction1” could be in a completely different C file or even supplied by a 3rd party as a library to which you can link. The use of functions thus lends itself to the reuse of code.

Sometimes it is convenient to place all the function prototypes in a file and include that file in all the C files that use those functions. This is called a header file and by convention, header files have a “.h” filename extension and C files have a “.c” filename extension.

It is worth noting that the “main” routine is itself a function that returns an integer and takes an integer and a pointer to a character array as arguments. The loader is responsible for passing these arguments to the “main” routine after it loads a program. The integer argument represents the number of command line parameters being passed in and the array of char pointers represents each of the command line parameters.

4.2 – Variable Types and Declarations

The C language allows for many variations in declaring and accessing variables. While these variations can be a source of flexibility, a lack of understanding on how these variations impact implementation can lead to undesired results. In this section we will cover the bulk of the common declaration and access techniques.

Global variables:

A variable that can be accessed from any function in a program without any restriction is referred to as a global variable. Syntactically a global variable is declared outside of all functions. In the example below the variable "MyGlobalVariable" is declared outside the "main" and "MyFunction1" routines, but can be accessed by both those routines. This sort of declaration also referred to as **unlimited scope**.

```
int MyFunction1( int arg1, char arg2 );

int MyGlobalVariable;

int main(int argc, char* argv[])
{
        int retVal;

        MyGlobalVariable++;

        retVal = MyFunction1( 1, 'c' );

        return( retVal );
}

int MyFunction1( int arg1, char arg2 )
{
        MyGlobalVariable++;

        return(0);
}
```

While C does not define where global variables have to be placed in memory, most compilers will store **global variables as part of the data segment**. This is because the compiler is aware that this memory location is valid for the entirety of execution.

While global variables allow the convenience of access from any function, this convenience can also be the cause of unexpected results because of the increased opportunities for data corruption. For this reason the use of global variables is often discouraged.

Local variables:

Also known as **Automatic variables**, local variables exist only when the code within the scope of the variable is being executed. Any variable declared within a block of code (within a set of curly braces) is a local variable by default. In the example below, notice that "myLocalVariable" is declared inside the code block for "MyFunction1". Hence this variable only exists when the execution is within "MyFunction1".

Most compilers would allocate **memory for local variables on the stack** just before starting the execution of the function. This implies that if a function calls itself (also known as recursion) each call will get its own instance of "myLocalVariable". When the stack unwinds and the execution returns from a function, the local variables are automatically discarded as part of the stack unwinding.

```
int MyFunction1( int arg1, char arg2 );

int MyGlobalVariable;

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

```
        MyGlobalVariable++;

        retVal = MyFunction1( 1, 'c' );

        return( retVal );
}

int MyFunction1( int arg1, char arg2 )
{
        int myLocalVariable;

        myLocalVariable++;

        MyGlobalVariable++;

        return(0);
}
```

Dynamic variables:

Sometimes it is necessary to allocate memory based on certain conditions that are only known at runtime. In such situations C allows for dynamic allocations of memory. These allocations are carved out of a chunk of memory that is reserved at the start of execution of a program and is referred to as the **heap**.

Dynamic (or heap) variables are similar to global variables in that, once allocated they are available for use by all functions until they are freed. The C runtime library provides a few functions that can be used to manage dynamic allocations. The most common functions used for this purpose are the "**malloc**" and "**free**" functions.

In the example below, "MyFunction1" takes the dynamic allocation size as an argument. This size is used in the call to "malloc". Note that the "malloc" function takes that size in bytes. If malloc is successful, it will return a pointer to the heap allocation. If it is not successful (eg. ran out of heap memory), it will return NULL. Whenever dealing with pointers, it is important to check for a NULL pointer before using it, otherwise, you are likely to run into undesired results.

```
char* MyFunction1( int size );

int main(int argc, char* argv[])
{
        char* myHeapVariable;

        myHeapVariable = MyFunction1( 100 );

        /* Use the 100 bytes pointed to by myHeapVariable */

        if( myHeapVariable )
        {
                free( myHeapVariable );
        }

        return( 0 );
}

char* MyFunction1( int size )
{
        char* ptr;

        ptr = malloc( size );

        if( ptr )
        {
                return(ptr);
        }
        else
        {
                return(NULL);
```

```
        }
}
```

Static qualifier:

Sometimes you want the advantages of a global variable without some of the disadvantages. For example you may wish to persist the value of a variable within a function between function calls. If you declared this variable as a local variable, the value would be lost once we exit the function. Making the variable global would remedy the problem, but then leave the possibility that others could access it.

The “static” qualifier provides a way to create a global variable with limited scope. In the example below, we have declared “ptr” as a static variable in the “AllocateFree” function with an initial value of NULL. The 2nd argument to the “AllocateFree” function dictates if an allocation or free is being requested. If an allocation is being requested, the first argument reflects the size of the allocation. If a free is being requested, the first argument is not relevant. The “AllocateFree” function will only allocate memory if it has not previously allocated the memory. If it has previously allocated the memory, it will return the same pointer that it saved in its static variable (ptr). Note that “ptr” cannot be directly accessed by the main routine.

Note the use of “if(!ptr)”. This is equivalent to “if(ptr != NULL)”. It is a common shorthand in the C language.

```
char* AllocateFree( int size,  );

int main(int argc, char* argv[])
{
        char* myHeapVariable;

        myHeapVariable = AllocateFree( 100, 1 );

        /* Use the 100 bytes pointed to by myHeapVariable */

        if( myHeapVariable )
        {
                AllocateFree ( 100, 0 );
        }

        return( 0 );
}

char* AllocateFree ( int size, char allocate )
{
        static char* ptr = NULL;

        if( allocate != 0)
        {
                if( !ptr )
                {
                        ptr = malloc( size );
                }

        }
        else
        {
                if( ptr )
                {
                        free( ptr );
                        prt = NULL;
                }
        }

        return( ptr );
}
```

A static qualifier can also be used with a global variable to limit its scope to the C file within which the global variable is declared.

Register qualifier:

As you will recall from our study of hardware, accessing memory is a lot slower than accessing registers inside the CPU. If a variable is going to be used very frequently, it would be a lot more efficient to use a register to save the contents of the variable than to use a memory location. You can do this in C using the register qualifier. The compiler will try and accommodate this request, based on the hardware constraints that it is working with.

The syntax for the use of the register qualifier with an integer variable is as follows:

```
register int variable;
```

4.3 – The Preprocessor

The C language provides for an initial step prior to the start of compilation known as preprocessing. The preprocessor provides facilities to implement certain features before the compilation begins. We have already used some of these preprocessor features in our examples thus far. Preprocessor directives start with the "#" symbol.

File Inclusion:

"#include" is a directive to include a file (usually a header file) prior to the start of compilation. Below are a couple of examples of its use. The use of the "<" and ">" brackets asks the preprocessor to use the include path to locate the file. If the quotes are used instead, it means the file is located in the current directory.

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

Macro Substitution:

"#define" is a directive to perform macro substitution. The macro name that appears immediately after the "#define" is substituted with a replacement that is stated subsequent to the macro name. We used macros to define constants previously. The examples below define a constant and another macro that determines the greater of 2 values. Whereever "MAX(..)" is used in the file, the preprocessor will replace it with the check that follows. Most coding standards will make macros all uppercase, so the reader knows easily it is a macro definition.

```
#define  MY_CONSTANT_1  1000
#define  MAX(x,y)       ((x) > (y) ? (x) : (y))
```

Conditional Inclusion:

Sometimes you want certain parts of the code to be included or omitted during compilation. Editing the file each time can be time consuming. The preprocessor directive to conditionally include parts of the file can be useful in this situation.

In the example below, unless "TRACE_LEVEL_1" has been previously defined using the "#define" macro, the "printf" code will not be included as part of the compilation.

```
#if defined (TRACE_LEVEL_1)
      printf("This is more detailed trace\r\n");
#endif
```

The syntax also allows for multiple conditions as follows:

```
#if defined (TRACE_LEVEL_1)
      printf("This is more detailed trace at trace level 1\r\n");
#elif defined (TRACE_LEVEL_2)
      printf("This is more detailed trace at trace level 2\r\n");
#else
      printf("This is the default trace level.\r\n");
#endif
```

5.0 – Pointers in C

The use of pointers is an integral part of programming in the C language. They contribute substantially to the efficiency of code written in C. Unfortunately pointers also lend themselves to bad coding practices that can lead to undesired results. Understanding the inner workings of pointers in C is essential to fully exploiting the power of the C language, while avoiding its pitfalls.

In this section we will study some of the common applications of pointers in C programming.

5.1 – A Pointer Variable

A variable in the C language is essentially a label that identifies a memory location. The type of the variable will dictate the size of the memory location. A “char” variable for example, will be 1 byte in memory, while an integer variable may be 2 bytes or 4 bytes.

A memory location is always identified at a byte level of granularity. The number of uniquely addressable bytes will be determined by the number of address lines available on the system. Most personal computers use 32 bits for addressing. This means that they can access 2^32 bytes (4 Gigabytes) of memory.

The label we use to identify a variable represents an address. This will be the address of the byte at one end (depending on whether it is big-endian or little-endian) of the memory location used to store the entire variable. The compiler will recognize the type of variable we are dealing with and make sure that when we assign values to this variable the full size of the variable is taken into account.

A pointer variable is essentially a variable that allows you to store the addresses of other variables, so that you don’t have to explicitly know the label for those variables to access them. Since a pointer variable stores an address, its size is always the same (32 bits in a 32 bit system).

As an example, let us look at some code in the disassembly window of our debugger.

Here we have defined two global variables – “intVar” and “intPtr”. “intVar” is an integer variable, while “intPtr” is an integer pointer.

```
        int     intVar;
        int*    intPtr;
```

When we assign “2” to “intVar”, notice what the compiler is actually doing. The compiler happens to know that “intVar” is located at address “0x4096FC”. So it translates this to a “mov” instruction to move a constant “2” to that location. Notice the term “dword ptr” in the move instruction. That is what determines the size of the move. The compiler happens to know that the “intVar” is an integer variable and that is a “dword” in the Microsoft compiler. A “dword” (double word) is 4 bytes in Microsoft compiler. So even though, the move instruction is provided with the address of the byte at one end of the 4 bytes representing the variable, the move actually moves 4 bytes of data.

```
        intVar = 2;
00000052  mov         dword ptr ds:[004096FCh],2
```

When we assign the address of “intVar” to our pointer variable, notice the compiler again translates this to a “dword” move instruction. This is because an address is also 4 bytes (32 bits) on this machine. The address it is moving is the same address (0x4096FC) we used previously, because we are now assigning the address of “intVar” to “intPtr”. Note “intPtr” is located at 0x409780.

```
        intPtr = &intVar;
0000005c  mov         dword ptr ds:[00409780h],4096FCh
```

If we now want to use the pointer variable to access “intVar”, we can do so by using the “*” prefix before the “intPtr” variable. This is the C syntax to indicate that you are now talking about what is pointed to by the pointer variable as opposed to the pointer variable itself. Notice the compiler translates this to a register indirect move instruction, but first it moves the address saved in the pointer variable to the EAX register. The compiler happens to know that the pointer variable is located at address 0x409780.

```
        *intPtr = 3;
00000066  mov         eax,dword ptr ds:[00409780h]
0000006b  mov         dword ptr [eax],3
```

After the first instruction above is executed, the EAX register will have the value “0x4096FC”. Then the “3” will get moved to that address.

So in summary, from here onwards, always think of a pointer variable as a means to access a regular variable without using the variable’s name, but rather its address.

5.2 – Pointer Arithmetic

In the previous section, we used an example of an integer pointer to save the address of an integer variable. What would you expect to happen if we added "1" to the pointer variable after we assigned it to the address of the integer variable?

Let us find out what the compiler would do in this case with our code. Here is what the disassembly window tells us…

```
        intPtr = &intVar;
0000005c  mov         dword ptr ds:[00409780h],4096FCh

        intPtr++;
00000066  add         dword ptr ds:[00409780h],4
```

Notice how incrementing the "intPtr" by 1 translates to the addition of "4" to our saved address.

This is because the compiler recognizes that "intPtr" is a pointer to an integer and if we add "1" to it, we intend to move to the next integer in memory, as opposed to the next byte in memory.

Notice how your knowledge of the assembly programming is proving useful in understanding the true inner workings of the compiler. Reverse engineering of this sort can prove invaluable in solving otherwise difficult problems. It also happens to be a great learning aid.

Let us try the same with a char variable.

```
        charPtr = &charVar;
00000059  mov         dword ptr ds:[004096F0h],4096F6h

        charPtr++;
00000063  inc         dword ptr ds:[004096F0h]
```

Notice how the compiler now uses the "inc" instruction instead of the "add" instruction. This is because the compiler knows that the char variable is only 1 byte and moving to the next char only requires the address to be incremented by "1".

What about a pointer to a user defined structure? Let us define variable "srVar" and "srPtr" to refer to the "StudentRecord" structures that we use earlier and try the same experiment.

```
        srPtr = &srVar;
00000052  mov         dword ptr ds:[00409788h],409790h

        srPtr++;
0000005c  add         dword ptr ds:[00409788h],70h
```

Notice this time we add 0x70 (decimal 112). That happens to be the size of the structure.

Addition and subtraction of pointer variables will translate to changes in pointer values that reflect the size of the variable the pointer is pointing to.

Sometimes it is convenient to have a pointer variable that is not associated with any data type. Situations where this happens include functions that allow varied data types as arguments. The C language allows for a "**void**" pointer for this purpose. One limitation of a void pointer however, is that you can't perform arithmetic operations on them. However you can assign them to pointers of a different type. This is known as **type casting**. For example you can assign a void pointer to a StudentRecord pointer as follows (assuming of course that you are confident that the "voidPtr" passed to you is in fact of type "StudentPointer").

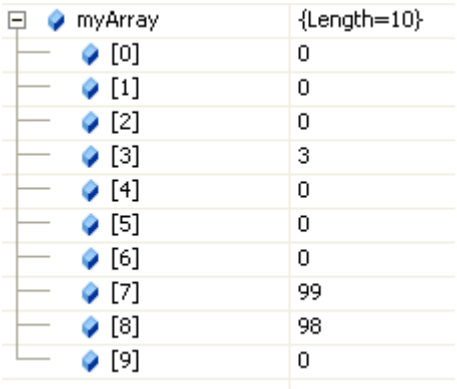
srPtr = (StudentRecord*) voidPtr;

5.3 – Pointers and Arrays

Pointers and Array are interrelated in the C language. Any operation that involves array subscripting to access a memory location can also be achieved using pointers. As it turns out, the label used to refer to an array in C is effectively a pointer.

```
int   myArray[10];
int   *intPtr;

int main(int argc, char* argv[])
{
      /* Initialize all 10 array elements to 0 */
      for(int i =0; i<10; i++ )
      {
            myArray[i] = 0;
      }

      /* Assign the value 99 to the 7th element of the array */
      *(myArray+7)= 99;

      /* Assign the value 3 to the 3rd element of the array */
      myArray[3] = 3;

      /* Initialize the "intPtr" to be "myArray" */
      intPtr = myArray;

      /* Assign the value 98 to the 8th element of the array */
      *(intPtr+8) = 98;

      return(0);

}
```

Watch "myArray" in the watch window as you step over these lines of code. You should end up with the following:



Notice that when we assign "intPtr" to "myArray", we did not use the "&" qualifier on "myArray" to get its address. This is because an array variable is effectively a pointer in C. Hence it is the equivalent of assigning one pointer variable to another.

5.4 – Argument Passing by Reference

C allows for two ways to pass arguments to functions. The first is called "**passing by value**" and the second is called "**passing by reference**".

In the examples thus far we passed arguments by value. This involves passing a variable or a constant as the argument to a function. When we do this the compiler pushes our values on the stack or into registers before making the call to the function and the function then accesses these values from the registers or the stack. Note that what is passed to the function is the current value of a variable and not the address of the variable. The significance of this is that the function cannot change the variable in the calling function. It can only use the value being passed in.

The other alternative is to pass by reference. Here we pass the address of the variable to a function. The function can now change the contents of the variable in the calling function.

Here is an example of two functions – "MyFunction1" takes an argument by value, while "MyFunction2" takes an argument by reference.

Watch "myArg" in the debugger watch window as you execute both these functions. Notice that after "MyFunction2" executes, "myArg" changes from "10" to "20".

```
int MyFunction1( int  myArgVal );
int MyFunction2( int* myArgRef );

int main(int argc, char* argv[])
{
      int   myArg = 10;

      MyFunction1( myArg );
      MyFunction2( &myArg );

      return( 0 );
}

int MyFunction1( int  myArgVal )
{
      int localVar;

      localVar = myArgVal;
      localVar++;

      return( localVar );
}

int MyFunction2( int* myArgRef )
{
      int* localVarPtr;

      localVarPtr = myArgRef;
      *localVarPtr = 20;

      return( *localVarPtr );
}
```

5.5 – Function Pointers

Functions, like variables, also reside in memory. We identify functions by function names. The compiler identifies functions by the address at which it is located. This means that we can access functions without calling them by name, if we had access to their address location.

Function pointers are a type of variable used to store function addresses. Calling a function requires knowledge of the full function signature, such as its arguments and return values. Hence a function pointer is always a user defined type of data that specifies the argument types and the return values. Like any user defined type of data, you can use the "**typedef**" operator to define a function type. You can also declare a function pointer directly without the use of the "typedef" operator.

In the example below, we define two function pointers – "MyFunk1" and "func". "MyFunk1" uses the "FUNK" type that was defined earlier using the typedef operator. "func" is declared directly without the typedef operator.

Note that in both cases, we give enough information to the compiler to know the argument types and the return values.

Once declared, function pointers can be assigned to any function that matches the signature for which it was declared. In our example we assign it to "MyFunction".

```
typedef int (*FUNK)(int,char);

int MyFunction(int var1, char char1)
{
      printf("MyFunction called args: %d and '%c'\n", var1, char1);
      return 0;
}

int main(int argc, char* argv[])
{
      int (*func)(int,char);
      FUNK  MyFunk1;

      func = MyFunction;
      MyFunk1 = MyFunction;

      (*func)(1,'A');
      MyFunk1(2,'B');

      return( 0 );
}
```

6.0 – Sample Functions in C

The fastest way to gain proficiency in any language is to use it. In that spirit, we will devote this last section to writing samples that address common programming tasks. We will start with a problem statement and follow that up with a sample. You are encouraged to start by writing your own code to address the problem prior to looking at the sample.

.

6.1 – Singly Linked List Implementation

Create a singly linked list with 10 elements and remove element 5.

<Sample 3>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <malloc.h>

/* Declare a structure representing the linked list elements */
typedef struct LL{
      int                     var1;
      struct LL*        nextPtr;
}LLElement;

/* Declare a constant for the number of list elements */
#define MAX_LIST_LENGTH       10

/* Define function prototypes */
LLElement*  CreateSinglyLinkedList(int count);
void        RemoveElement(LLElement* headPtr, int elementId );


/*************************************************************************
* Main Routine
*************************************************************************/
int main(int argc, char* argv[])
{
      LLElement* headPointer;

      headPointer = CreateSinglyLinkedList(MAX_LIST_LENGTH);
      RemoveElement( headPointer, 5 );

      return(0);
}

/*************************************************************************
* CreateSinglyLinkedList function
*************************************************************************/
LLElement* CreateSinglyLinkedList(int count)
{

      /* Define variables of the structure type you just declared */
      LLElement   *headPtr=NULL, *ptr, *prevPtr;

      /*Create a Linked List with MAX_LIST_LENGTH elements */
      printf("\nCreating Linked List of length %d\r\n\n", count);
      for(int i=0; i< count; i++)
      {
            ptr = (LLElement*)malloc(sizeof(LLElement));
            if( ptr )
            {
                  ptr->var1 = i;
                  ptr->nextPtr = headPtr;
```

```
                headPtr = ptr;
            }
        }


        /* Walk Linked list */
        printf("\nWalking Linked List\r\n");
        for( ptr=headPtr; ptr ; ptr=ptr->nextPtr )
        {
            printf("Found linked list element %d \r\n",ptr->var1);
        }

        return( headPtr );
}

/*********************************************************************
* RemoveElement function
*********************************************************************/
void RemoveElement(LLElement* headPtr, int elementId )
{
        /* Define variables of the structure type you just declared */
        LLElement   *ptr, *prevPtr;

        /* Delete the tagged item */
        printf("\nRemoving Linked List item %d\r\n\n", elementId );
        for( prevPtr=NULL, ptr=headPtr ; ptr ;
                                        prevPtr=ptr, ptr=ptr->nextPtr )
        {
            if( ptr->var1 == elementId )
            {
                if (prevPtr)
                {
                    prevPtr->nextPtr = ptr->nextPtr;
                }
                else
                {
                    headPtr = ptr->nextPtr;
                }
                free(ptr);

                /* We can now exit the for loop */
                break;
            }
        }

        /* Walk Linked list */
        printf("\nWalking Linked List\r\n");
        for( ptr=headPtr; ptr ; ptr=ptr->nextPtr )
        {
            printf("Found linked list element %d \r\n",ptr->var1);
        }

        return;
}
```

6.2 – Bit operations in C

"Little-endian" refers to the data format where the least significant byte (little end) is represented at the first (lowest) address and each more significant byte is represented at the next higher address. This representation requires a reversal of bytes when read by the human eye because addresses are often displayed in increasing sequence and yet we expect more significant bytes to be displayed first.

"Big-endian" is the opposite of little endian. Here the most significant byte (big end) is represented at the first (lowest) address. Intel uses the "little endian" representation. Most other processors use the "big endian" representation. Most TCPIP communications are based on the Big Endian layout. In other words, any 16- or 32-bit value within the various layer headers (for example, an IP address, a packet length, or a checksum) must be sent and received with its most significant byte first.

As an exercise in using C bit operations, covert a double word initialised in the little-endian format to a big-endian format.

<Sample 4>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <wtypes.h>

/* Define function prototypes */
DWORD LittleToBigEndian(DWORD var1);

/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	DWORD lEndian, bEndian;

	lEndian=0x12345678;

	bEndian = LittleToBigEndian(lEndian);

	return(0);
}

/*****************************************************************
* LittleToBigEndian
*****************************************************************/
DWORD LittleToBigEndian(DWORD var1)
{
	DWORD var2;
	BYTE* bytePtr;

	bytePtr = (BYTE*)&var1;

	printf("Little Endian - Byte0=0x%x Byte1=0x%x Byte2=0x%x
Byte3=0x%x \n",
		bytePtr[0], bytePtr[1], bytePtr[2], bytePtr[3] );

	//Now we convert to Big Endian
	var2 = ((var1&0x000000FF)<<24) | ((var1&0x0000FF00)<<8) |
((var1&0x00FF0000)>>8) | ((var1&0xFF000000)>>24);
```

```
      bytePtr = (BYTE*)&var2;

      printf("Big Endian    - Byte0=0x%x Byte1=0x%x Byte2=0x%x
Byte3=0x%x \n",
            bytePtr[0], bytePtr[1], bytePtr[2], bytePtr[3] );

      return( var2 );
}
```

6.3 – Variable Argument Functions

There are times where a function may need to take a variable number of arguments. We have used one such function extensively in our samples – “printf”. We can pass a variable number of parameters to this function depending on how many variables we wish to print.

You may have other reasons to use a variable argument function. Sometimes you may want to accommodate varying argument types. In this case, your first parameter could give you info on what the second parameter type is going to be.

The difficulty with writing a variable length function is that you are responsible for walking the stack and picking the parameters passed in. Fortunately C provides some convenient macros to do this for us.

In the sample below we write a simple variable length function that allows us to pass a DWORD or a string as our second parameter. Note the use of the “va_start”, “va_arg” and “va_end” macros to help us get data that was passed on the stack.

<Sample 5>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <wtypes.h>

/* Define function prototypes */
void VariableLengthArg(DWORD type, ...);

/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	VariableLengthArg(0, 0x12345678);
	VariableLengthArg(1, "This is a string");
	return(0);
}

/*****************************************************************
* VariableLengthArg
*****************************************************************/
void VariableLengthArg(DWORD type, ...)
{
	va_list	argList;

	va_start( argList, type );

	switch(type)
	{
		case 0x00:
			DWORD dword;

			dword = va_arg(argList, DWORD);
			printf("DWORD=0x%x\n", dword);
			break;

		case 0x01:
```

```
                char* strPtr;

                strPtr = va_arg(argList, char*);
                printf("String=%s\n", strPtr);
                break;
        }

        va_end(argList);
}
```

6.4 – String Operations

Write a program that will ask the user for their name and age. Then compare their name against yours and print their and age on the screen.

Note that all the string functions used here are part of the standard C libraries. You can find the function signatures for any of these functions using the on-line help in Visual Studio. Highlight the function and press "F1".

In the debugger look at the "output" memory location during each of the string operations. Observer that "strcpy" and "strcat" are appending a NULL at the end of the string. This is how C determines the end of a string. The memory buffer allocated for a string must always account for a NULL.

Notice the use of the "itoa" function to convert an integer to a string.

<Sample 6>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <string.h>
#include <wtypes.h>


/*****************************************************************
* Main Routine
*****************************************************************/
int main(int argc, char* argv[])
{
	char		*yourName="Akiko";
	char		name[100];
	char		output[200];
	unsigned int	age;
	char		ageAsString[5];

	/* Get the user's name */
	printf("\nPlease enter your first name: ");
	scanf("%s",name );

	/* Get the user's age */
	printf("\nPlease enter your age: ");
	scanf("%d",&age );

	/* Check if you have the same name as the user */
	if(!strcmp( name, yourName ) )
	{
		printf("\nDid you know that we share the same name?\n");
	}
	/* May the user forgot to uppercase the first letter of a proper
noun*/
	else if( !stricmp( name, yourName ) )
	{
		printf("\nDid you know that we share the same name? I like
to uppercase the first letter of my name though!\n");
	}

	/* Format your response and print it*/
```

```
        strcpy( output, name );
        strcat( output, " is " );
        itoa( age, ageAsString, 10 );
        strcat( output, ageAsString );
        strcat( output, " years old\n");
        printf("%s", output );

        /* Format your response the easy way and print it*/
        sprintf( output, "\n%s is %d years old\n", name, age );
        printf("%s", output );

        return(0);
}
```

6.5 – #pragma pack

In section 2 we learned how to declare a structure with multiple fields. The compiler will generally try to place each field within a structure at its "**natural boundary**". What this means is that each member of a structure will be placed at an address that is divisible by the size of the member.

For example, if you have an integer member in a structure, the compiler will ensure that the integer variable is at an address that is divisible by the size of an integer on the platform (usually 4). If you had a character member variable, it can be placed at the next available location since its size is 1 byte.

The reason the compiler does this is because it allows the compiler to use mnemonics that are optimized to deal with a particular data type if the data is located at its natural boundary. These optimized mnemonics expect variables to be located at their natural boundaries, otherwise they generate an exception that is commonly referred to as a "**data-alignment fault**".

The **maximum** gap that the compiler will leave between member variables in its attempt to place members at their natural boundaries is dictated by the **"#pragma pack**" compiler directive. You can redefine this pack directive as required within your file.

In the sample below, we define a structure with a character and an integer. Since the integer is 4 bytes on a PC, the total theoretical size of our structure must be 5 bytes. Let us find that actual size when using the pack(1) and pack(4) directives.

<Sample 7>

```
/* Include all the headers needed for the functions used */
#include <stdio.h>
#include <string.h>
#include <wtypes.h>

/* Save the compiler's default pragma pack */
#pragma pack(push)

/* Ask the compiler's to use pragma pack(1) */
#pragma pack(1)
typedef struct
{
      char  charVar;
      int   intVar;
}PACK_1_STRUCT;

/* Ask the compiler's to use pragma pack(4) */
#pragma pack(4)
typedef struct
{
      char  charVar;
      int   intVar;
}PACK_4_STRUCT;

/* Restore the compiler's pragma pack */
#pragma pack(pop);


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

```
{
	printf("Size of PACK_1_STRUCT = %d\n", sizeof(PACK_1_STRUCT) );
	printf("Size of PACK_4_STRUCT = %d\n", sizeof(PACK_4_STRUCT) );

	return(0);
}
```

7.0 – Conclusion

The C programming language offers the flexibility and ease of a high level language without substantially compromising on the power of lower level programming languages. We can still manipulate bits in C, without the overhead of looking at the status register after each instruction to see if there are conditions we need to be aware of. This allows us to focus on the solutions rather than the inner working of the CPU. This is in fact the impetus for high level languages. We allow the compiler to abstract the inner working of the CPU for us, when we use high level languages. This has a tremendously positive impact on programmer productivity.

C does not go all the way in abstracting low level details. There are some areas like the management of dynamic memory where C counts on the programmer to be conscious and efficient about allocating and freeing memory. Languages that abstract even more than C, tend to rely on built-in system modules that keep an account of a program's allocation of resources and automatically cleans up without expecting this diligence on the part of the program. These sorts of languages have gained popularity in the last decade. It is conceivable that as processors become faster and memory footprints become less of an issue, the overhead of these languages will become less of a concern. Until then, C will remain the language of choice for most applications that need to interact with hardware peripherals and perform other low level operations as efficiently as possible without having to resort to assembly level programming.