Coding Basics

Introduction

As previously mentioned, this course requires a fundamental understanding of C as a prerequisite. With that being said, there are a few concepts that will be mentioned due to their importance throughout this course.

Structures

Structures or Structs are user-defined data types that allow the programmer to group related data items of different data types into a single unit. Structs can be used to store data related to a particular object. Structs help organize large amounts of related data in a way that can be easily accessed and manipulated. Each item within a struct is called a "member" or "element", these terms are used interchangeably within the course.

A common occurrence one will see when working with the Windows API is that some APIs require a populated structure as input, while others will take a declared structure and populate it. Below is an example of the `THREADENTRY32` struct, it is not necessary to understand what the members are used for at this point.

```
typedef struct tagTHREADENTRY32 {
  DWORD dwSize; // Member 1
  DWORD cntUsage; // Member 2
  DWORD th32ThreadID;
  DWORD th32OwnerProcessID;
  LONG  tpBasePri;
  LONG  tpDeltaPri;
  DWORD dwFlags;
} THREADENTRY32;
```

Declaring a Structure

Structures used in this course are generally declared with the use of `typedef` keyword to give a structure an alias. For example, the structure below is created with the name `_STRUCTURE_NAME` but `typedef` adds two other names, `STRUCTURE_NAME` and `*PSTRUCTURE_NAME`.

```
typedef struct _STRUCTURE_NAME {

  // structure elements

} STRUCTURE_NAME, *PSTRUCTURE_NAME;
```

The `STRUCTURE_NAME` alias refers to the structure name, whereas `PSTRUCTURE_NAME` represents a pointer to that structure. Microsoft generally uses the `P` prefix to indicate a pointer type.

Initializing a Structure

Initializing a structure will vary depending on whether one is initializing the actual structure type or a pointer to the structure. Continuing the previous example, initializing a structure is the same when using `_STRUCTURE_NAME` or `STRUCTURE_NAME`, as shown below.

```
STRUCTURE_NAME    struct1 = { 0 };  // The '{ 0 }' part, is used to
initialize all the elements of struct1 to zero
// OR
_STRUCTURE_NAME   struct2 = { 0 };  // The '{ 0 }' part, is used to
initialize all the elements of struct2 to zero
```

This is different when initializing the structure pointer, `PSTRUCTURE_NAME`.

```
PSTRUCTURE_NAME structpointer = NULL;
```

Initializing and Accessing Structures Members

A structure's members can be initialized either directly through the structure or indirectly through a pointer to the structure. In the example below, the structure `struct1` has two members, `ID` and `Age`, initialized directly via the dot operator (`.`).

```
typedef struct _STRUCTURE_NAME {
  int ID;
  int Age;
} STRUCTURE_NAME, *PSTRUCTURE_NAME;

STRUCTURE_NAME struct1 = { 0 }; // initialize all elements of struct1 to
zero
struct1.ID   = 1470;   // initialize the ID element
struct1.Age  = 34;     // initialize the Age element
```

Another way to initialize the members is using *designated initializer syntax* where one can specify which members of the structure to initialize.

```
typedef struct _STRUCTURE_NAME {
  int ID;
  int Age;
} STRUCTURE_NAME, *PSTRUCTURE_NAME;

STRUCTURE_NAME struct1 = { .ID   = 1470,  .Age  = 34}; // initialize both
the ID and the Age elements
```

On the other hand, accessing and initializing a structure through its pointer is done via the arrow operator (->).

```
typedef struct _STRUCTURE_NAME {
  int ID;
  int Age;
} STRUCTURE_NAME, *PSTRUCTURE_NAME;

STRUCTURE_NAME struct1 = { .ID   = 1470,  .Age  = 34};

PSTRUCTURE_NAME structpointer = &struct1; // structpointer is a pointer to
the 'struct1' structure

// Updating the ID member
structpointer->ID = 8765;
printf("The structure's ID member is now : %d \n", structpointer->ID);
```

The arrow operator can be converted into dot format. For example, `structpointer->ID` is equivalent to `(*structpointer).ID`. That is, `structurepointer` is de-referenced and then accessed directly.

Enumeration

The enum or enumeration data type is used to define a set of named constants. To create an enumeration, the `enum` keyword is used, followed by the name of the enumeration and a list of identifiers, each of which represents a named constant. The compiler automatically assigns values to the constants, starting with 0 and increasing by 1 for each subsequent constant. In this course, enums can be seen representing the state of specific data, error codes or return values.

An example of an enum is the list of "Weekdays" which contains 7 constants. In the example below, Monday has a value of 0, Tuesday has a value of 1, and so on. It's important to note that enum lists cannot be modified or accessed using the dot (.) operator. Instead, each element is accessed directly using its named constant value.

```
enum Weekdays {
  Monday,          // 0
  Tuesday,         // 1
  Wednesday,       // 2
  Thursday,        // 3
  Friday,          // 4
  Saturday,        // 5
  Sunday           // 6
};

// Defining a "Weekdays" enum variable
enum Weekdays EnumName = Friday;        // 4
```

```
// Check the value of "EnumName"
switch (EnumName){
    case Monday:
      printf("Today Is Monday !\n");
      break;
    case Tuesday:
      printf("Today Is Tuesday !\n");
      break;
    case Wednesday:
      printf("Today Is Wednesday !\n");
      break;
    case Thursday:
      printf("Today Is Thursday !\n");
      break;
    case Friday:
      printf("Today Is Friday !\n");
      break;
    case Saturday:
      printf("Today Is Saturday !\n");
      break;
    case Sunday:
      printf("Today Is Sunday !\n");
      break;
    default:
      break;
}
```

Union

In the C programming language, a Union is a data type that permits the storage of various data types in the same memory location. Unions provide an efficient way to use a single memory location for multiple purposes. Unions are not commonly used but can be seen in Windows-defined structures. The code below illustrates how to define a union in C:

```
union ExampleUnion {
   int    IntegerVar;
   char   CharVar;
   float  FloatVar;
};
```

`ExampleUnion` can store `char`, `int` and `float` data types in the same memory location. To access the members of a union in C, one can use the dot operator, similar to that used for structures.

It's important to note that in a union, assigning a new value to any member will change the value of all other members as well because they share the same memory location to store their data. Additionally, the memory allocated for a union is equal to the size of its largest member.

Bitwise Operators

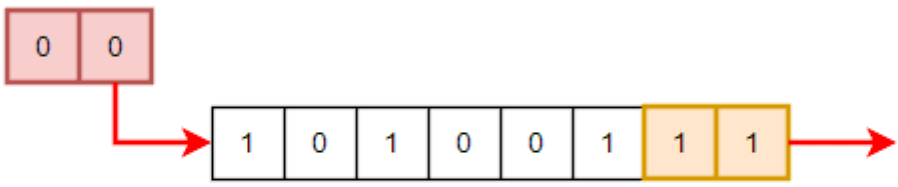
Bitwise operators are operators that manipulate the individual bits of a binary value, performing operations on each corresponding bit position. The bitwise operators are shown below:

- Right shift (>>)
- Left shift (<<)
- Bitwise OR (|)
- Bitwise AND (&)
- Bitwise XOR (^)
- Bitwise NOT (~)

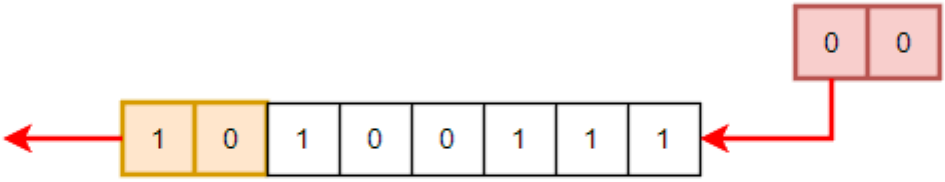
Right and Left Shift

The right shift (>>) and left shift (<<) operators are used to shift the bits of a binary number to the right and left by a specified number of positions, respectively.

Shifting right discards the rightmost number of bits by the specified value and zero bits of the same amount are inserted into the left. For example, the image below shows `10100111` shifted right by `2`, to become `00101001`.



On the other hand, shifting left discards the leftmost bits and the same number of zero bits are inserted from the right handside. For example, the image below shows `10100111` shifted left by `2`, to become `10011100`.



Bitwise OR

The bitwise OR operation is a logical operation that involves two binary values at the bit level. It evaluates each bit of the first operand against the corresponding bit of the second operand, generating a new binary value. The new binary value contains a 1 in any bit position where either one or both of the corresponding bits in the original values are 1.

The following table represents the bitwise OR output with all the possible input bits.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise AND

The bitwise AND operation is a logical operation that involves two binary values at the bit level. This operation sets the bits of the new binary value to 1 only in the case where the corresponding bits of both input operands are 1.

The following table represents the bitwise AND output with all the possible input bits.

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise XOR

The bitwise XOR operation (also known as exclusive OR) is a logical operation that involves two binary values at the bit level. If only one of the bits is 1, the result in each position is 1. Conversely, if both bits are 0 or 1, the output is 0.

The following table represents the bitwise XOR output with all the possible input bits.

Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise NOT

The bitwise NOT operation takes one binary number and flips all its bits. In other words, it changes all 0s to 1s and all 1s to 0s. The following table represents the bitwise NOT output with all the possible input bits.

Input	Output
0	1
1	0

Passing By Value

Passing by value is a method of passing arguments to a function where the argument is a copy of the object's value. This means that when an argument is passed by value, the value of the object is copied and the function can only modify its local copy of the object's value, not the original object itself.

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

int main()
{
   int x = 5;
   int y = 10;
   int sum = add(x, y); // x and y are passed by value

   return 0;
}
```

Passing By Reference

Passing by reference is a method of passing arguments to a function where the argument is a pointer to the object, rather than a copy of the object's value. This means that when an argument is passed by reference, the memory address of the object is passed instead of the value of the object. The function can then access and modify the object directly, without creating a local copy of the object.

```
void add(int *a, int *b, int *result)
{

  int A = *a; // A is now the same value of a passed in from the main
function
  int B = *b; // B is now the same value of b passed in from the main
function

  *result = B + A;
}

int main()
{
   int x = 5;
   int y = 10;
   int sum = 0;

   add(&x, &y, &sum);

   // 'sum' now is 15

   return 0;
}
```