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.

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;
```

Previous | Modules | Undo | Next