Windows Programming

Lecture 07

Calling convention

Calling convention

cdecl and __stdcall just tells the compiler whether the called function or the calling function cleans up the stack (I think there is also pascal which tells the order of parameters that are put onto the stack). I believe that __stdcall says that the called function cleans up the stack when it is about to return. So if it is called in a bunch of different places, all of those calls do not need to extra code to clean up the stack after the function call.

__cdecl

This is the default calling convention for C programs. Because the stack is cleaned up by the caller. The **__cdecl** calling convention creates larger executables than __stdcall, because it requires each function call to include stack cleanup code.

__cdecl

The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing	orderRight to left
Stack-maintenance responsibility	Calling function pops the arguments from the stack
Name-decoration convention	Underscore character (_) is prefixed to names
Case-translation convention	No case translation performed

__cdecl

Place the **__cdecl** modifier before a variable or a function name.

In the following example, the compiler is instructed to use C naming and calling conventions for the system function:

```
int __cdecl system(const char *);
```

// Example of the __cdecl keyword _CRTIMP

__stdcall

The __stdcall calling convention is used to call Win32 API functions. The callee cleans the stack.

__stdcall

The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing order	Right to left
Argument-passing convention	By value, unless a pointer or reference type is passed.
Stack-maintenance responsibility	Called function pops its own arguments from the stack

__stdcall

Element	Implementation
Name-decoration convention	An underscore (_) is prefixed to the name. The name is followed by the at sign (@) followed by the number of bytes (in decimal) in the argument list. Therefore, the function declared as int func(int a, double b) is decorated as follows: _func@12 .
Case-translation convention	None

__stdcall

- Functions declared using the **__stdcall** modifier return values the same way as functions declared using __cdecl.
- In the following example, use of __**stdcall** results in all WINAPI function types being handled as a standard call:

```
#define WINAPI __stdcall
```

// Example of the __stdcall keyword

__pascal

The callee cleans the stack.

Argument-passing order:	Right to left

Storage Class Modifiers

C has a concept of '*Storage classes*' which are used to define the scope (visability) and life time of variables and/or functions.

auto - storage class

The **auto** storage class specifier lets you define a variable with automatic storage; its use and storage is restricted to the current block. The storage class keyword **auto** is optional in a data declaration. It is not permitted in a parameter declaration. A variable having the **auto** storage class specifier must be declared within a block. It cannot be used for file scope declarations .

auto - storage class

Because automatic variables require storage only while they are actually being used, defining variables with the **auto** storage class can decrease the amount of memory required to run a program. However, having many large automatic objects may cause you to run out of stack space.

auto - storage class

- Declaring variables with the **auto** storage class can also make code easier to maintain, because a change to an **auto** variable in one function never affects another function (unless it is passed as an argument).
- The following example lines declare variables having the **auto** storage class specifier:

```
auto int counter;
auto char letter = 'k';
                          //Initialization
```

auto - storage class

Initialization

You can initialize any **auto** variable except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

register - Storage Class

- The register storage class specifier indicates to the compiler that a heavily used variable (such as a loop control variable) within a block scope data definition or a parameter declaration should be allocated a register to minimize access time.

- It is equivalent to the auto storage class except that the compiler places the object, if possible, into a machine register for faster access.

register - Storage Class

Most heavily-used entities are generated by the compiler itself; therefore, register variables are given no special priority for placement in machine registers. The **register** storage class keyword is required in a data definition and in a parameter declaration that describes an object having the **register** storage class.

register - Storage Class

- An object having the **register** storage class specifier must be defined within a block or declared as a parameter to a function.
- The following example lines define automatic storage duration objects using the **register** storage class specifier:

```
register int score1 = 0, score2 = 0;
register unsigned char code = 'A';
register int *element = &order[0];
```

register - Storage Class

Initialization
You can initialize any **register** object except parameters. If you do not initialize an automatic object, its value is indeterminate. If you provide an initial value, the expression representing the initial value can be any valid C expression. For structure and union members, the initial value must be a valid constant expression if an initializer list is used. The object is then set to that initial value each time the program block that contains the object's definition is entered.

register - Storage Class

- **Storage**
  Objects with the **register** storage class specifier have automatic storage duration. Each time a block is entered, storage for **register** objects defined in that block are made available. When the block is exited, the objects are no longer available for use.
- If a **register** object is defined within a function that is recursively invoked, the memory is allocated for the variable at each invocation of the block.

static - Storage Class

static is the default storage class for global variables. The two variables below (**count** and **road**) both have a static storage class

```
static int Count;
int Road;
main()
{
      printf("%d\n", Count);
      printf("%d\n", Road);
}
```

static - Storage Class

- static variables can be 'seen' within all functions in this source file.
- static variables are created at the strat of the programme on a separate memory location.
- Local static variables are not destroyed when the function returns, since they are nor on the stack.

```
char * func(void);
main()
{
      char *Text1;
      Text1 = func();
}
char * func(void)
{
      char Text2[10]="martin";
      return(Text2);
}
```

Now, 'func' returns a pointer to the memory location where 'text2' starts BUT text2 has a storage class of 'auto' and will disappear when we exit the function and could be overwritten but something else. The answer is to specify static char Text[10]="martin"; The storage assigned to 'text2' will remain reserved for the duration if the program.

extern - Storage Class

If a variable is declared (with global scope) in one file but referenced in another, the extern keyword is used to inform the compiler of the variable's existence:

extern - Storage Class

In *declare.c*:

```
int farvar;
```

In *use.c*:

```
{
      extern int farvar;
      int a; a = farvar * 2;
}
```

- Note that the extern keyword is for declarations, not definitions
- An extern declaration does not create any storage; that must be done with a global definition

Difference

- In call by value, variable is passed as parameter so its copy is generated over the stack. So if changes are made over that copy then there will be no change on the original variable.

- In call by reference, variable is not passed rather reference/starting address/pointer of that variable is passed. So if changes are made then the original variable will also be changed.

- Call by value is slower that call by reference

Stack

- A stack is a LIFO (last in, first out) collection: objects may be pushed onto the stack, and popped off it in reverse order of pushing

- Stack allocation means run-time allocation and deallocation of storage in last-in/first-out orde0r.

Iteration

Iteration means to repeat a process over and over again. In mathematics this process is most often the application of a mathematical function

Three iteration mechanisms in c language are

- For
- While
- Do-while

Recursion

- Recursion is a fundamental concept in mathemetics and computer science. Simply, a recursive function (program or algoritm) is one which calls itself as part of the function body.

Recursion

An essential ingredient of recursion is there must be a "termination condition"; i.e. the call to oneself must be conditional to some test or predicate condition which will cease the recursive calling. A recursive program must cease the recursion o n some condition or be in a circular state, i.e. an endless loop which will "crash" the system

Recursion

In a computer implementation of a recursive algorithm, a function will conditionally call itself. When any function call is performed, a new complete copy of the function's "information" (parameters, return addresses, etc..) is placed into general dat a and/or stack memory. When a function returns or exits, this information is returned to the free memory pool and the function ceases to actively exist. In recursive algorithms, many levels of function calls can be initiated resulting in many copies of th e function being currently active and copies of the different functions information residing in the memory spaces. Thus recursion provides no savings in storage nor will it be faster than a good non-recursive implementation. However, recursive code will o ften be more compact, easier to design, develop, implement, integrate, test, and debug.

Const–Access Modifier

- Value of a variable defined with const keyword can never change
- A C constant is usually just the written version of a number.

`Const`–Access Modifier

```
Const int i = 10;
```

Keyword

i cannot be changed in the programme.

Constant Variables

```
Const char * ptr = buff.
```

Variable pointer to constant data

```
*ptr = ‘a’;
ptr = buf2;
```

Error

Constant Variables

```
char * const ptr = buff.
```

Contant pointer to variable data

```
ptr = buff2;
* ptr = ‘a’;
```

Error

Constant Variables

```
char ch[50]=“This is a string”;
```

ch is a pointer constant

```
char *ch=“This is a string”;
```

ch is a pointer variable

Command Line Arguments

C provides a fairly simple mechanism for retrieving command line parameters entered by the user. It passes an **argv** parameter to the main function in the program. **argv** structures appear in a fair number of the more advanced library calls, so understanding them is useful to any C programmer.

Command Line Arguments

```
C:\>program a b c
```

Command tail

OR

Command lie arguments

Command Line Arguments

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

In this code, the main program accepts two parameters, argv and argc. The argv parameter is an array of pointers to string that contains the parameters entered when the program was invoked at the UNIX command line. The argc integer contains a count of the number of parameters. This particular piece of code types out the command line parameters

Command Line Arguments

- The **`char *argv[]`** line is an array of pointers to string. In other words, each element of the array is a pointer, and each pointer points to a string (technically, to the first character of the string).

- **`argc`** is the count of the nymber of parameters

Sign-magnitude Representation

The Sign-Magnetude Method is quite easy to understand. In fact, it is simply an ordinary binary number with one extra digit placed in front to represent the sign. If this extra digit is a '1', it means that the rest of the digits represent a negative number. However if the same set of digits are used but the extra digit is a '0', it means that the number is a positive one.

Sign-magnitude Representation

Let us assume that we have an 8-bit register. This means that we have 7 bits which represent a number and the other bit to represent the sign of the number

(the **Sign Bit**).
This is how numbers are represented:

Sign-magnitude Representation

0	0	1	0	0	1	0	1

In sign-magnitude representation, means +37 The red digit means that the number is positive. The rest of the digits represent 37.

And this is how -37 is represented:

1	0	1	0	0	1	0	1

Tow’s Complement

- Two's complement representation allows the use of binary arithmetic operations on signed integers, yielding the correct 2's complement results.

- **Positive Numbers**

  Positive 2's complement numbers are represented as the simple binary.

- **Negative Numbers**

  Negative 2's complement numbers are represented as the binary number that when added to a positive number of the same magnitude equals zero.

Tow’s Complement

- To calculate the 2's complement of an integer, invert the binary equivalent of the number by changing all of the ones to zeroes and all of the zeroes to ones (also called **1's complement)**, and then add one.

  For example,

  0001 0001(binary 17)   1110 1111(two's complement -17)

  NOT(0001 0001) = 1110 1110 (Invert bits)

  1110 1110 + 0000 0001 = 1110 1111 (Add 1)

Questions

```
#define HIBYTE(w)
```