Basic Interview Questions(Embedded C Programming)

August 26, 2012 by Vijayabharathi C

1. What is Storage class? Explain with example

The storage class determines the part of memory where storage is allocated for an object (particularly variables and functions) and how long the storage allocation continues to exist. In C program, there are four storage classes: automatic, register, external and static.

1. ***Auto***

    - *They are declared at the start of a program's block such as in the curly braces ({ }). Memory is allocated automatically upon entry to a block and freed automatically upon exit from the block.*

    - *Automatic variables may be specified upon declaration to be of storage class auto. However, it is not required to use the keyword auto because by default, storage class within a block is auto.*

    Register

    Automatic variables are allocated in the main memory of the processor; accessing these memory location for computation will take long time.

    - *when we required to optimize the execution time, move the critical variable to processor register. this can be done by using the register key word.*

    - *when storage class is register, compiler is instructed to allocate a register for this variable.*

    - *scope of the register variable is same as auto variable.*

    NOTE: Allocation of register is not guaranteed always, it depends on number register available in processor and number register used for manipulation. if you define 4 variable as register storage

class and and processor has only 2 register for variable allocation, then compiler will allocate 2 variable in registers and treat the remaining 2 variable as auto variable. therefore usage of register keyword should should be justified and cross checked with disassembly weather register is allocated or not.

Extern

- *For using the external global variable from other files extern keyword is used.*
- *any file can access this global variable and lifetime over entire program run.*

Static

- *static variable have lifetime over entire program run.*
- *scope of this variable is limited based on the place of declaration.*
- *if static variable is defined in a file and not inside any function, then scope of the variable is limited within that file.*
- *if static variable is defined inside a function, then the scope of the variable is limited within that function.*
- *we can use this variable any file and any function indirectly by accessing through pointer.*

2. what is qualifiers?

*Qualifiers defines the property of the variable. Two qualifiers are **const** and **volatile**. The **const** type qualifier declares an object to be unmodifiable. The **volatile** type qualifier declares an item whose value can legitimately be changed by something beyond the control of the program in which it appears, such as a concurrently executing thread / interrupt routine.*

3. What are volatile variables? Where we should use?

A volatile variable is one that can change unexpectedly. Consequently, the compiler can make no assumptions about the value of the variable. In particular, the optimizer must be careful to reload the variable every time it is used instead of holding a copy in a register. Examples of volatile variables are:

- Hardware registers in peripherals (for example, status registers)
- Non-automatic variables referenced within an interrupt service routine
- Variables shared by multiple tasks in a multi-threaded applications

4. What does the keyword const mean? What do the following declarations mean?

```
const int a;
int const a;
const int *a;
int * const a;
int const * a const;
```

The first two mean the same thing, namely a is a const (read-only) integer.

The third means a is a pointer to a const integer (that is, the integer isn't modifiable, but the pointer is).

The fourth declares a to be a const pointer to an integer (that is, the integer pointed to by a is modifiable,but the pointer is not).

The final declaration declares a to be a const pointer to a const integer
(that is, neither the integer pointed to by a, nor the pointer itself may be modified).

5. Can a parameter be both const and volatile ? Explain.

Yes. An example is a read-only status register. It is volatile because it can change unexpectedly. It is const because the program should not attempt to modify it

6. Can a pointer be volatile ? Explain.

Yes, although this is not very common. An example is when an interrupt service routine modifies a pointer to a buffer

7. What's wrong with the following function?

```
int square(volatile int *ptr)
{
return *ptr * *ptr;

}
```

This one is wicked. The intent of the code is to return the square of the value pointed to by *ptr . However, since *ptr points to a volatile parameter, the compiler will generate code that looks something like this:

```
int square(volatile int *ptr)
{
int a,b;
a = *ptr;
b = *ptr;
return a * b;
}

}
```

Because it's possible for the value of *ptr to change unexpectedly, it is possible for a and b to be different. Consequently, this code could return a number that is not a square! The correct way to code this is:

```
long square(volatile int *ptr)
{
int a;
a = *ptr;
return a * a;
}
```

8. Data Declarations

a) int a; // An integer
b) int *a; // A pointer to an integer
c) int **a; // A pointer to a pointer to an integer
d) int a[10]; // An array of 10 integers
e) int *a[10]; // An array of 10 pointers to integers
f) int (*a)[10]; // A pointer to an array of 10 integers
g) int (*a)(int); // A pointer to a function a that takes an integer argument and returns an integer
h) int (*a[10])(int); // An array of 10 pointers to functions that take an integer argument and return an integer

10. What are Dangling pointers and Wild Pointers

Dangling Pointer :

Dangling pointers in computer programming are pointers that do not point to a valid object of the appropriate type. Dangling pointers arise when an object is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

Examples of Dangling Pointers

```
int main()
{
    int *p;
    p = (int *) malloc (sizeof (int));

   free(p);
   *p=10;

}
```

In the above piece of code we are using *p after we free the memory to it.
Such usage is called dangling pointer usage.

```
int main()
{
int *p = NULL;
{
int a = 10;
p = &a;
}
/*address of a is out of scope and pointer p is now called the dangling pointer, we should initialize the p to NULL before coming out or initialize the pointer to some known value before using it again*/
…
}

int* fun1()
{
int a = 10;
return(&a); /*in this line we are returning the pointer of variable ‘a’ which is out scope.*/
}
```

Wild Pointers:

Wild pointers are created by omitting necessary initialization prior to first use. Thus, strictly speaking, every pointer in programming languages which do not enforce initialization begins as a wild pointer. This most often occurs due to jumping over the initialization, not by omitting it. Most compilers are able to warn about this.

```
{

int* a;

/* a is wild pointer, it is not initialized and it may have some garbage value*/

}
```

correct way is

```
{

int* a = NULL;

}
```

10. When should unions be used? Why do we need them in Embedded Systems programming?

Unions are particularly useful in Embedded programming or in situations where direct access to the hardware/memory is needed. Here is a trivial example:

```
typedef union
{
   struct {
      unsigned char byte1;
      unsigned char byte2;
      unsigned char byte3;
      unsigned char byte4;
   } bytes;
   unsigned int dword;
} HW_Register;
HW_Register reg;
```

Then you can access the reg as follows:

```
reg.dword = 0×12345678;
reg.bytes.byte3 = 4;
```

Endianism and processor architecture are of course important.

Another useful feature is the bit modifier:

```
typedef union
{
   struct {
```

```
        unsigned char b1:1;
        unsigned char b2:1;
        unsigned char b3:1;
        unsigned char b4:1;
        unsigned char reserved:4;
    } bits;
    unsigned char byte;
} HW_RegisterB;
HW_RegisterB reg;
```

With this code you can access directly a single bit in the register/memory address:

```
x = reg.bits.b2;
```

Low level system programming is a reasonable example.

unions are used to breakdown hardware registers into the component bits. So, you can access an 8-bit register into the component bits.

This structure would allow a control register to be accessed as a control_byte or via the individual bits. It would be important to ensure the bits map on to the correct register bits for a given endianness.

```
typedef union {
    unsigned char control_byte;
    struct {
        unsigned int nibble  : 4;
        unsigned int nmi     : 1;
        unsigned int enabled : 1;
        unsigned int fired   : 1;
        unsigned int control : 1;
    }
} ControlRegister;
```

11. Why is sizeof('a') not 1?

Perhaps surprisingly, character constants in C are of type int, so sizeof('a') is sizeof(int) (though it's different in C++).
Result:
In Turbo C output is: 2
In Turbo C++ output is: 1

12. why n++ executes faster than n+1?

The expression n++ requires a single machine instruction such as INR to carry out the increment operation whereas, n+1 requires more instructions to carry out this operation.

13. Volatile explanation revisited !

Another use for `volatile` is signal handlers. If you have code like this:

```
quit = 0;
while (!quit)
{
    /* very small loop which is completely visible to the compiler *
}
```

The compiler is allowed to notice the loop body does not touch the `quit` variable and convert the loop to a `while (true)` loop. Even if the `quit` variable is set on the signal handler for `SIGINT` and`SIGTERM`; the compiler has no way to know that.

However, if the `quit` variable is declared `volatile`, the compiler is forced to load it every time, because it can be modified elsewhere. This is exactly what you want in this situation.

Courtesy: http://stackoverflow.com/questions/246127/why-is-volatile-needed-in-c (http://stackoverflow.com/questions/246127/why-is-volatile-needed-in-c)

14. Data Alignment & Structure Padding

Data Alignment: *Data alignment* means putting the data at a memory offset equal to some multiple of the word size, which increases the system's performance due to the way the CPU handles memory

Data Structure Padding: To align the data, it may be necessary to insert some meaningless bytes between the end of the last data structure and the start of the next, which is *data structure padding*

Here is a structure with members of various types, totaling **8 bytes** before compilation:

```
struct MixedData
{
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

After compilation the data structure will be supplemented with padding bytes to ensure a proper alignment for each of its members:

```
struct MixedData  /* After compilation in 32-bit x86 machine */
{
    char Data1; /* 1 byte */
    char Padding1[1]; /* 1 byte for the following 'short' to be alig
assuming that the address where structure begins is an even number *
    short Data2; /* 2 bytes */
    int Data3;  /* 4 bytes - largest structure member */
    char Data4; /* 1 byte */
    char Padding2[3]; /* 3 bytes to make total size of the structure
};
```

The compiled size of the structure is now 12 bytes. It is important to note that the last member is padded with the number of bytes required so that the total size of the structure should be a multiple of the largest alignment of any structure member (alignment(int) in this case, which = 4 on linux-32bit/gcc)

In this case 3 bytes are added to the last member to pad the structure to the size of a 12 bytes (alignment(int) × 3).

```
struct FinalPad {
  float x;
  char n[1];
};
```

In this example the total size of the structure sizeof(FinalPad) = 8, not 5 (so that the size is a multiple of 4 (alignment of float)).

```
struct FinalPadShort {
  short s;
  char n[3];
};
```

In this example the total size of the structure sizeof(FinalPadShort) = 6, not 5 (not 8 either) (so that the size is a multiple of 2 (alignment(short) = 2 on linux-32bit/gcc)).

It is possible to change the alignment of structures to reduce the memory they require (or to conform to an existing format) by reordering structure members or changing the compiler's alignment (or "packing") of structure members.

```
struct MixedData  /* after reordering */
{
    char Data1;
    char Data4;   /* reordered */
    short Data2;
    int Data3;
};
```

The compiled size of the structure now matches the pre-compiled size of **8 bytes**. Note that *Padding1[1]* has been replaced (and thus eliminated) by *Data4* and *Padding2[3]* is no longer necessary as the structure is already aligned to the size of a long word.

The alternative method of enforcing the *MixedData* structure to be aligned to a one byte boundary will cause the pre-processor to discard the pre-determined alignment of the structure members and thus no padding bytes would be inserted.

While there is no standard way of defining the alignment of structure members, some compilers use *#pragma* directives to specify packing inside source files. Here is an example:

```
#pragma pack(push)  /* push current alignment to stack */
#pragma pack(1)     /* set alignment to 1 byte boundary */

struct MyPackedData
{
    char Data1;
    long Data2;
    char Data3;
};

#pragma pack(pop)   /* restore original alignment from stack */
```

This structure would have a compiled size of **6 bytes** on a 32-bit system. The above directives are available in compilers from Microsoft, Borland, GNU and many others.

Another example:

```
struct MyPackedData
{
    char Data1;
    long Data2 __attribute__((packed));
    char Data3;
};

Courtesy: http://en.wikipedia.org/wiki/Data_structure_alignment

http://www.geeksforgeeks.org/structure-member-alignment-padding-and-


http://stackoverflow.com/questions/381244/purpose-of-memory-alignmer


http://stackoverflow.com/questions/6968468/padding-in-structures-in-
```

Posted in Interview Questions | Leave a Comment

Follow “ucvbtechideas”