

A 'C' Test: The 0x10 Best Questions for Would-be Embedded Programmers

An obligatory and significant part of the recruitment process for embedded systems programmers seems to be the 'C Test'. Over the years, I have had to both take and prepare such tests and in doing so have realized that these tests can be very informative for both the interviewer and interviewee. Furthermore, when given outside the pressure of an interview situation, they can also be quite entertaining (hence this article).

From the interviewee's perspective, you can learn a lot about the person that has written or administered the test. Is the test designed to show off the writer's knowledge of the minutiae of the ANSI standard rather than to test practical know-how? Does it test ludicrous knowledge, such as the ASCII values of certain characters? Are the questions heavily slanted towards your knowledge of system calls and memory allocation strategies, indicating that the writer may spend his time programming computers instead of embedded systems? If any of these are true, then I know I would seriously doubt whether I want the job in question.

From the interviewer's perspective, a test can reveal several things about the candidate. Primarily, one can determine the level of the candidate's knowledge of C. However, it is also very interesting to see how the person responds to questions to which they do not know the answers. Do they make intelligent choices, backed up with some good intuition, or do they just guess? Are they defensive when they are stumped, or do they exhibit a real curiosity about the problem, and see it as an opportunity to learn something? I find this information as useful as their raw performance on the test.

With these ideas in mind, I have attempted to construct a test that is heavily slanted towards the requirements of embedded systems. This is a lousy test to give to someone seeking a job writing compilers! The questions are almost all drawn from situations I have encountered over the years. Some of them are very tough; however, they should all be informative.

This test may be given to a wide range of candidates. Most entry-level applicants will do poorly on this test, while seasoned veterans should do very well. Points are not assigned to each question, as this tends to arbitrarily weight certain questions. However, if you choose to adapt this test for your own uses, feel free to assign scores.

Preprocessor

1. Using the `#define` statement, how would you declare a manifest constant that returns the number of seconds in a year? Disregard leap years in your answer.

```
#define SECONDS_PER_YEAR (60UL * 60UL * 24UL * 365UL)
```

I'm looking for several things here:

- a. Basic knowledge of the `#define` syntax (i.e. no semi-colon at the end, the need to parenthesize etc.).
 - b. A good choice of name, with capitalization and underscores.
 - c. An understanding that the pre-processor will evaluate constant expressions for you. Thus, it is clearer, and penalty free to spell out how you are calculating the number of seconds in a year, rather than actually doing the calculation yourself.
 - d. A realization that the expression will overflow an integer argument on a 16 bit machine—hence the need for the `L`, telling the compiler to treat the expression as a Long.
 - e. As a bonus, if you modified the expression with a `UL` (indicating unsigned long), then you are off to a great start because you are showing that you are mindful of the perils of signed and unsigned types—and remember, first impressions count!
2. Write the 'standard' `MIN` macro. That is, a macro that takes two arguments and returns the smaller of the two arguments.

```
#define MIN(A,B) ((A) <= (B) ? (A) : (B))
```

The purpose of this question is to test the following:

- a. Basic knowledge of the `#define` directive as used in macros. This is important, because until the inline operator becomes part of standard C, macros are the only portable way of generating inline code. Inline code is often necessary in embedded systems in order to achieve the required performance level.
- b. Knowledge of the ternary conditional operator. This exists in C because it allows the compiler to potentially produce more optimal code than an if-then-else sequence. Given that performance is normally an issue in embedded systems, knowledge and use of this construct is important.
- c. Understanding of the need to very carefully parenthesize arguments to macros.
- d. I also use this question to start a discussion on the side effects of macros, e.g. what happens when you write code such as:

```
least = MIN(*p++, b);
```

3. *What is the purpose of the preprocessor directive #error?*

Either you know the answer to this, or you don't. If you don't, then see reference 1. This question is very useful for differentiating between normal folks and the nerds. It's only the nerds that actually read the appendices of C textbooks that find out about such things. Of course, if you aren't looking for a nerd, the candidate better hope she doesn't know the answer.

Infinite Loops

4. *Infinite loops often arise in embedded systems. How does one code an infinite loop in C?*

There are several solutions to this question. My preferred solution is:

```
while(1)
{
    ...
}
```

Another common construct is:

```
for(;;)
{
    ...
}
```

Personally, I dislike this construct because the syntax doesn't exactly spell out what is going on. Thus, if a candidate gives this as a solution, I'll use it as an opportunity to explore their rationale for doing so. If their answer is basically—"I was taught to do it this way and I have never thought about it since"—then it tells me something (bad) about them. Conversely, if they state that it's the K&R preferred method and the only way to get an infinite loop passed Lint, then they score bonus points.

A third solution is to use a goto:

```
Loop:
...
goto Loop;
```

Candidates that propose this are either assembly language programmers (which is probably good), or else they are closet BASIC/FORTRAN programmers looking to get into a new field.

Data declarations

5. *Using the variable a, write down definitions for the following:*

- a. *An integer*
- b. *A pointer to an integer*
- c. *A pointer to a pointer to an integer*

- d. *An array of ten integers*
- e. *An array of ten pointers to integers (f) A pointer to an array of ten integers*
- f. *A pointer to a function that takes an integer as an argument and returns an integer*
- g. *An array of ten pointers to functions that take an integer argument and return an integer*

The answers are:

- 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`

People often claim that a couple of these are the sorts of thing that one looks up in textbooks—and I agree. While writing this article, I consulted textbooks to ensure the syntax was correct. However, I expect to be asked this question (or something close to it) when in an interview situation. Consequently, I make sure I know the answers—at least for the few hours of the interview. Candidates that don't know the answers (or at least most of them) are simply unprepared for the interview. If they can't be prepared for the interview, what will they be prepared for?

Static

6. *What are the uses of the keyword static?*

This simple question is rarely answered completely. Static has three distinct uses in C:

- a. A variable declared static within the body of a function maintains its value between function invocations.
- b. A variable declared static within a module,¹ (but outside the body of a function) is accessible by all functions within that module. It is not accessible by functions within any other module. That is, it is a localized global.

- c. Functions declared static within a module¹ may only be called by other functions within that module. That is, the scope of the function is localized to the module within which it is declared.

Most candidates get the first part correct. A reasonable number get the second part correct, while a pitiful number understand answer (c). This is a serious weakness in a candidate, since they obviously do not understand the importance and benefits of localizing the scope of both data and code.

Const

7. *What does the keyword const mean?*

As soon as the interviewee says ‘const means constant’, I know I’m dealing with an amateur. Dan Saks has exhaustively covered const in the last year, such that every reader of ESP should be extremely familiar with what const can and cannot do for you. If you haven’t been reading that column, suffice it to say that const means “read-only”. Although this answer doesn’t really do the subject justice, I’d accept it as a correct answer. (If you want the detailed answer, then read Saks’ columns—carefully!).

If the candidate gets the answer correct, then I’ll ask him these supplemental questions:

What do the following incomplete² 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 (i.e., the integer isn’t modifiable, but the pointer is). The fourth declares a to be a const pointer to an integer (i.e., 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 (i.e., neither the integer pointed to by a, nor the pointer itself may be modified).

If the candidate correctly answers these questions, I’ll be impressed.

Incidentally, one might wonder why I put so much emphasis on const, since it is very easy to write a correctly functioning program without ever using it. There are several reasons:

1. The use of `const` conveys some very useful information to someone reading your code. In effect, declaring a parameter `const` tells the user about its intended usage. If you spend a lot of time cleaning up the mess left by other people, then you’ll quickly learn to appreciate this extra piece of information. (Of course, programmers that use `const`, rarely leave a mess for others to clean up...)

1. Translation unit for the pedagogues out there.

2. I’ve had complaints that these code fragments are incorrect syntax. This is correct. However, writing down syntactically correct code pretty much gives the game away.

2. `const` has the potential for generating tighter code by giving the optimizer some additional information.
3. Code that uses `const` liberally is inherently protected by the compiler against inadvertent coding constructs that result in parameters being changed that should not be. In short, they tend to have fewer bugs.

Volatile

8. *What does the keyword `volatile` mean? Give three different examples of its 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:

- a. Hardware registers in peripherals (e.g., status registers)
- b. Non-stack variables referenced within an interrupt service routine.
- c. Variables shared by multiple tasks in a multi-threaded application.

If a candidate does not know the answer to this question, they aren't hired. I consider this the most fundamental question that distinguishes between a 'C programmer' and an 'embedded systems programmer'. Embedded folks deal with hardware, interrupts, RTOSes, and the like. All of these require `volatile` variables. Failure to understand the concept of `volatile` will lead to disaster.

On the (dubious) assumption that the interviewee gets this question correct, I like to probe a little deeper, to see if they really understand the full significance of `volatile`. In particular, I'll ask them the following:

- a. *Can a parameter be both `const` and `volatile`? Explain your answer.*
- b. *Can a pointer be `volatile`? Explain your answer.*
- c. *What is wrong with the following function?*

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

The answers are as follows:

- a. 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.
- b. Yes. Although this is not very common. An example is when an interrupt service routine modifies a pointer to a buffer.

- c. 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;
}
```

Since it is 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;
}
```

Bit Manipulation

9. *Embedded systems always require the user to manipulate bits in registers or variables. Given an integer variable `a`, write two code fragments. The first should set bit 3 of `a`. The second should clear bit 3 of `a`. In both cases, the remaining bits should be unmodified.*

These are the three basic responses to this question:

- a. No idea. The interviewee cannot have done any embedded systems work.
- b. Use bit fields. Bit fields are right up there with trigraphs as the most brain-dead portion of C. Bit fields are inherently non-portable across compilers, and as such guarantee that your code is not reusable. I recently had the misfortune to look at a driver written by Infineon for one of their more complex communications chip. It used bit fields, and was completely useless because my compiler implemented the bit fields the other way around. The moral—never let a non-embedded person anywhere near a real piece of hardware!3
- c. Use `#defines` and bit masks. This is a highly portable method, and is the one that should be used. My optimal solution to this problem would be:

```
#define BIT3 (0x1 << 3) static int a;
void set_bit3(void) { a |= BIT3;
}
void clear_bit3(void) { a &= ~BIT3;
}
```

3. I've recently softened my stance on bit fields. At least one compiler vendor (IAR) now offers a compiler switch for specifying the bit field ordering. Furthermore the compiler generates optimal code with bit field defined registers – and as such I now do use bit fields in IAR applications.

Some people prefer to define a mask, together with manifest constants for the set & clear values. This is also acceptable. The important elements that I'm looking for are the use of manifest constants, together with the |= and &= ~ constructs.

Accessing fixed memory locations

10. *Embedded systems are often characterized by requiring the programmer to access a specific memory location. On a certain project it is required to set an integer variable at the absolute address 0x67a9 to the value 0xaa55. The compiler is a pure ANSI compiler. Write code to accomplish this task.*

This problem tests whether you know that it is legal to typecast an integer to a pointer in order to access an absolute location. The exact syntax varies depending upon one's style. However, I would typically be looking for something like this:

```
int *ptr;
ptr = (int *)0x67a9;
*ptr = 0xaa55;
```

A more obfuscated approach is:

```
*(int * const)(0x67a9) = 0xaa55;
```

Even if your taste runs more to the second solution, I suggest the first solution when you are in an interview situation.

Interrupts

11. *Interrupts are an important part of embedded systems. Consequently, many compiler vendors offer an extension to standard C to support interrupts. Typically, this new key word is interrupt. The following code uses __interrupt to define an __interrupt service routine. Comment on the code.*

```
__interrupt double compute_area(double radius) {
{
    double area = PI * radius * radius;
    printf("\nArea = %f", area);
    return area;
}
```

This function has so much wrong with it, it's almost tough to know where to start.

- a. Interrupt service routines cannot return a value. If you don't understand this, then you aren't hired.
- b. ISR's cannot be passed parameters. See item (a) for your employment prospects if you missed this.

- c. On many processors/compiler, floating point operations are not necessarily re-entrant. In some cases one needs to stack additional registers, in other cases, one simply cannot do floating point in an ISR. Furthermore, given that a general rule of thumb is that ISRs should be short and sweet, one wonders about the wisdom of doing floating point math here.
- d. In a similar vein to point (c), printf() often has problems with reentrancy and performance. If you missed points (c) & (d) then I wouldn't be too hard on you. Needless to say, if you got these two points, then your employment prospects are looking better and better.

Code Examples

12. *What does the following code output and why?*

```
void foo(void)
{
    unsigned int a = 6; int b = -20;
    (a+b > 6) ? puts("> 6") : puts("<= 6");
}
```

This question tests whether you understand the integer promotion rules in C – an area that I find is very poorly understood by many developers. Anyway, the answer is that this outputs "> 6". The reason for this is that expressions involving signed and unsigned types have all operands promoted to unsigned types. Thus -20 becomes a very large positive integer and the expression evaluates to greater than 6. This is a very important point in embedded systems where unsigned data types should be used frequently (see reference 2). If you get this one wrong, then you are perilously close to not being hired.

13. *Comment on the following code fragment?*

```
unsigned int zero = 0;
unsigned int compzero = 0xFFFF; /*1's complement of zero */
```

On machines where an int is not 16 bits, this will be incorrect. It should be coded:

```
unsigned int compzero = ~0;
```

This question really gets to whether the candidate understands the importance of word length on a computer. In my experience, good embedded programmers are critically aware of the underlying hardware and its limitations, whereas computer programmers tend to dismiss the hardware as a necessary annoyance.

By this stage, candidates are either completely demoralized—or they are on a roll and having a good time. If it is obvious that the candidate isn't very good, then the test is terminated at this point. However, if the candidate is doing well, then I throw in these supplemental questions. These questions are hard, and I expect that only the very best candidates will do well on them. In posing these questions, I'm looking more at the way the candidate tackles the problems, rather than the answers. Anyway, have fun...

Dynamic memory allocation.

14. *Although not as common as in non-embedded computers, embedded systems still do dynamically allocate memory from the heap. What are the problems with dynamic memory allocation in embedded systems?*

Here, I expect the user to mention memory fragmentation, problems with garbage collection, variable execution time, etc. This topic has been covered extensively in ESP, mainly by Plauger. His explanations are far more insightful than anything I could offer here, so go and read those back issues! Having lulled the candidate into a sense of false security, I then offer up this tidbit:

15. *What does the following code fragment output and why?*

```
char *ptr;
if ((ptr = (char *)malloc(0)) == NULL) { puts("Got a null pointer");
}
    else {
        puts("Got a valid pointer");
    }
```

This is a fun question. I stumbled across this only recently, when a colleague of mine inadvertently passed a value of 0 to malloc, and got back a valid pointer! After doing some digging, I discovered that the result of malloc(0) is implementation defined, so that the correct answer is 'it depends.' I use this to start a discussion on what the interviewee thinks is the correct thing for malloc to do. Getting the right answer here is nowhere near as important as the way you approach the problem and the rationale for your decision.

Typedef

16. *Typedef is frequently used in C to declare synonyms for pre-existing data types. It is also possible to use the preprocessor to do something similar. For instance, consider the following code fragment:*

```
#define dPS struct s *
typedef struct s * tPS;
```

The intent in both cases is to define dPS and tPS to be pointers to structure s. Which method (if any) is preferred and why?

This is a very subtle question, and anyone that gets it right (for the right reason) is to be congratulated or condemned ("get a life" springs to mind). The answer is the typedef is preferred. Consider the declarations:

```
dPS p1, p2;
tPS p3, p4;
```

The first expands to

```
struct s * p1, p2;
```

which defines p1 to be a pointer to the structure and p2 to be an actual structure, which is probably not what you wanted. The second example correctly defines p3 & p4 to be pointers.

Obfuscated syntax

17. C allows some appalling constructs. Is this construct legal, and if so what does this code do?

```
int a = 5, b = 7, c;  
c = a+++b;
```

This question is intended to be a lighthearted end to the quiz, as, believe it or not, this is perfectly legal syntax. The question is how does the compiler treat it? Those poor compiler writers actually debated this issue, and came up with the “maximum munch” rule, which stipulates that the compiler should bite off as big a (legal) chunk as it can. Hence, this code is treated as:

```
c = a++ + b;
```

Thus, after this code is executed, a = 6, b = 7 & c = 12;

If you knew the answer, or guessed correctly—then well done. If you didn’t know the answer then I would not consider this to be a problem. I find the biggest benefit of this question is that it is very good for stimulating questions on coding styles, the value of code reviews and the benefits of using lint.

Well folks, there you have it. That was my version of the C test. I hope you had as much fun doing it as I had writing it. If you think the test is a good test, then by all means use it in your recruitment. Who knows, I may get lucky in a year or two and end up being on the receiving end of my own work.

References:

1. In Praise of the #error directive. ESP September 1999.
2. Efficient C Code for Eight-Bit MCUs. ESP November 1988.