A ‘C’ Test: The 0×10 Best Questions for Would-be Embedded Programmers

A step-by-step guide through one of the most popular recruitment tools for 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

(g) A pointer to a function that takes an integer as an argument and returns an integer

(h) *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 module 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 incomplete[2] 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:

(a) 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…)

(b) const has the potential for generating tighter code by giving the optimizer some additional information.

(c) 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      (0×1 << 3)

static int a;

void set_bit3(void) {

a |= BIT3;

}
```

```
void clear_bit3(void) {

a &= ~BIT3;

}
```

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 / compilers, 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:

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

15. 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

16. 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.

C language interview questions and answers

1. What is C language?

The C programming language is a standardized programming language developed in the early 1970s by Ken Thompson and Dennis Ritchie for use on the UNIX operating system. It has since spread to many other operating systems, and is one of the most widely used programming languages. C is prized for its efficiency, and is the most popular programming language for writing system software, though it is also used for writing applications.

2. What does static variable mean?

There are 3 main uses for the static.

1. If you declare within a function: It retains the value between function calls

2. If it is declared for a function name: By default function is extern..so it will be visible from other files if the function declaration is as static..it is invisible for the outer files

3. Static for global variables: By default we can use the global variables from outside files If it is static global..that variable is limited to with in the file.

```
#include <stdio.h>
int t = 10;
main(){
int x = 0;
void funct1();
funct1();
printf("After first call \n");
funct1();
printf("After second call \n");
funct1();
printf("After third call \n");
}
void funct1()
{
   static int y = 0;
   int z = 10;
   printf("value of y %d z %d",y,z);
   y=y+10;
}
```

```
value of y 0 z 10 After first call
value of y 10 z 10 After second call
value of y 20 z 10 After third call
```

3. What are the different storage classes in C?

C has three types of storage: automatic, static and allocated. Variable having block scope and without static specifier have automatic storage duration.

Variables with block scope, and with static specifier have static scope. Global variables (i.e, file scope) with or without the the static specifier also have static scope. Memory obtained from calls to malloc(), alloc() or realloc() belongs to allocated storage class.

4. What is hashing?

To hash means to grind up, and that’s essentially what hashing is all about. The heart of a hashing algorithm is a hash function that takes your nice, neat data and grinds it into some random-looking integer.

The idea behind hashing is that some data either has no inherent ordering (such as images) or is expensive to compare (such as images). If the data has no inherent ordering, you can’t perform comparison searches.

5. Can static variables be declared in a header file?

You can’t declare a static variable without defining it as well (this is because the storage class modifiers static and extern are mutually exclusive). A static variable can be defined in a header file, but this would cause each source file that included the header file to have its own private copy of the variable, which is probably not what was intended.

6. Can a variable be both constant and volatile?

Yes. The const modifier means that this code cannot change the value of the variable, but that does not mean that the value cannot be changed by means outside this code.

The function itself did not change the value of the timer, so it was declared const. However, the value was changed by hardware on the computer, so it was declared volatile. If a variable is both const and volatile, the two modifiers can appear in either order.

7. Can include files be nested?

Yes. Include files can be nested any number of times. As long as you use precautionary measures, you can avoid including the same file twice. In the past, nesting header files was seen as bad programming practice, because it complicates the dependency tracking function of the MAKE program and thus slows down compilation. Many of today’s popular compilers make up for this difficulty by implementing a concept called precompiled headers, in which all headers and associated dependencies are stored in a precompiled state.

8. What is a null pointer?

There are times when it’s necessary to have a pointer that doesn’t point to anything. The macro NULL, defined in , has a value that’s guaranteed to be different from any valid pointer. NULL is a literal zero, possibly cast to void* or char*.

Some people, notably C++ programmers, prefer to use 0 rather than NULL.
The null pointer is used in three ways:
1) To stop indirection in a recursive data structure.
2) As an error value.
3) As a sentinel value.

9. What is the output of printf("%d") ?

When we write printf("%d",x); this means compiler will print the value of x. But as here, there is nothing after %d so compiler will show in output window garbage value.

10. What is the difference between calloc() and malloc() ?

calloc(...) allocates a block of memory for an array of elements of a certain size. By default the block is initialized to 0. The total number of memory allocated will be (number_of_elements * size).

malloc(...) takes in only a single argument which is the memory required in bytes. malloc(...) allocated bytes of memory and not blocks of memory like calloc(...).

malloc(...) allocates memory blocks and returns a void pointer to the allocated space, or NULL if there is insufficient memory available.

calloc(...) allocates an array in memory with elements initialized to 0 and returns a pointer to the allocated space. calloc(...) calls malloc(...) in order to use the C++ _set_new_mode function to set the new handler mode.

11. What is the difference between printf() and sprintf() ?

sprintf() writes data to the character array whereas printf(...) writes data to the standard output device.

12. How to reduce a final size of executable?

Size of the final executable can be reduced using dynamic linking for libraries.

13. Can you tell me how to check whether a linked list is circular?

Create two pointers, and set both to the start of the list. Update each as follows:

```
while (pointer1) {
pointer1 = pointer1->next;
pointer2 = pointer2->next;
if (pointer2) pointer2=pointer2->next;
if (pointer1 == pointer2) {
print ("circular");
}
}
```

If a list is circular, at some point pointer2 will wrap around and be either at the item just before pointer1, or the item before that. Either way, its either 1 or 2 jumps until they meet.

14. Advantages of a macro over a function?

Macro gets to see the Compilation environment, so it can expand __ __TIME__ __FILE__ #defines. It is expanded by the preprocessor.

For example, you can’t do this without macros

```
#define PRINT(EXPR) printf( #EXPR “=%d\n”, EXPR)
PRINT( 5+6*7 ) // expands into printf(”5+6*7=%d”, 5+6*7 );
```

You can define your mini language with macros:

```
#define strequal(A,B) (!strcmp(A,B))
```

15. What is the difference between strings and character arrays?

A major difference is: string will have static storage duration, whereas as a character array will not, unless it is explicity specified by using the static keyword.

Actually, a string is a character array with following properties:

* the multibyte character sequence, to which we generally call string, is used to initialize an array of static storage duration. The size of this array is just sufficient to contain these characters plus the terminating NUL character.
* it not specified what happens if this array, i.e., string, is modified.
* Two strings of same value[1] may share same memory area.

16. Write down the equivalent pointer expression for referring the same element a[i][j][k][l] ?

```
a[i] == *(a+i)
a[i][j] == *(*(a+i)+j)
a[i][j][k] == *(*(*(a+i)+j)+k)
a[i][j][k][l] == *(*(*(*(a+i)+j)+k)+l)
```

17. Which bit wise operator is suitable for checking whether a particular bit is on or off?

The bitwise AND operator. Here is an example:

```
enum {
KBit0 = 1,
KBit1,
…
KBit31,
};
if ( some_int & KBit24 )
printf ( “Bit number 24 is ON\n” );
else
printf ( “Bit number 24 is OFF\n” );
```

18. Which bit wise operator is suitable for turning off a particular bit in a number?

The bitwise AND operator, again. In the following code snippet, the bit number 24 is reset to zero.

```
some_int = some_int & ~KBit24;
```

19. Which bit wise operator is suitable for putting on a particular bit in a number?

The bitwise OR operator. In the following code snippet, the bit number 24 is turned ON:

```
some_int = some_int | KBit24;
```

20. Does there exist any other function which can be used to convert an integer or a float to a string?

Some implementations provide a nonstandard function called itoa(), which converts an integer to string.

```
#include
char *itoa(int value, char *string, int radix);
```

DESCRIPTION
The itoa() function constructs a string representation of an integer.
PARAMETERS
value: Is the integer to be converted to string representation.
string: Points to the buffer that is to hold resulting string.
The resulting string may be as long as seventeen bytes.
radix: Is the base of the number; must be in the range 2 - 36.
A portable solution exists. One can use sprintf():

```
char s[SOME_CONST];
int i = 10;
float f = 10.20;
sprintf ( s, “%d %f\n”, i, f );
```

21. Why does malloc(0) return valid memory address ? What's the use?

malloc(0) does not return a non-NULL under every implementation. An implementation is free to behave in a manner it finds suitable, if the allocation size requested is zero. The implmentation may choose any of the following actions:

* A null pointer is returned.
* The behavior is same as if a space of non-zero size was requested. In this case, the usage of return value yields to undefined-behavior.

Notice, however, that if the implementation returns a non-NULL value for a request of a zero-length space, a pointer to object of ZERO length is returned! Think, how an object of zero size should be represented

For implementations that return non-NULL values, a typical usage is as follows:

```
void
func ( void )
{
int *p; /* p is a one-dimensional array, whose size will vary during the the lifetime of the program */
size_t c;
p = malloc(0); /* initial allocation */
if (!p)
{
perror (”FAILURE” );
return;
}
/* … */
while (1)
{
c = (size_t) … ; /* Calculate allocation size */
p = realloc ( p, c * sizeof *p );
/* use p, or break from the loop */
/* … */
}
return;
}
```

Notice that this program is not portable, since an implementation is free to return NULL for a malloc(0) request, as the C Standard does not support zero-sized objects.

22. Difference between const char* p and char const* p

In const char* p, the character pointed by ‘p’ is constant, so u cant change the value of character pointed by p but u can make ‘p’ refer to some other location.

In char const* p, the ptr ‘p’ is constant not the character referenced by it, so u cant make ‘p’ to reference to any other location but u can change the value of the char pointed by ‘p’.

23. What is the result of using Option Explicit?

When writing your C program, you can include files in two ways. The first way is to surround the file you want to include with the angled brackets < and >. This method of inclusion tells the preprocessor to look for the file in the predefined default location. This predefined default location is often an INCLUDE environment variable that denotes the path to your include files.

For instance, given the INCLUDE variable INCLUDE=C:\COMPILER\INCLUDE;S:\SOURCE\HEADERS; using the #include version of file inclusion, the compiler first checks the C:\COMPILER\INCLUDE directory for the specified file. If the file is not found there, the compiler then checks the S:\SOURCE\HEADERS directory. If the file is still not found, the preprocessor checks the current directory.

The second way to include files is to surround the file you want to include with double quotation marks. This method of inclusion tells the preprocessor to look for the file in the current directory first, then look for it in the predefined locations you have set up. Using the #include file version of file inclusion and applying it to the preceding example, the preprocessor first checks the current directory for the specified file. If the file is not found in the current directory, the C:COMPILERINCLUDE directory is searched. If the file is still not found, the preprocessor checks the S:SOURCEHEADERS directory.

The #include method of file inclusion is often used to include standard headers such as stdio.h or stdlib.h.

The #include file include nonstandard header files that you have created for use in your program. This is because these headers are often modified in the current directory, and you will want the preprocessor to use your newly modified version of the header rather than the older, unmodified version.

24. What is the benefit of using an enum rather than a #define constant?

The use of an enumeration constant (enum) has many advantages over using the traditional symbolic constant style of #define. These advantages include a lower maintenance requirement, improved program readability, and better debugging capability.
1) The first advantage is that enumerated constants are generated automatically by the compiler. Conversely, symbolic constants must be manually assigned values by the programmer.
2) Another advantage of using the enumeration constant method is that your programs are more readable and thus can be understood better by others who might have to update your program later.

3) A third advantage to using enumeration constants is that some symbolic debuggers can print the value of an enumeration constant. Conversely, most symbolic debuggers cannot print the value of a symbolic constant. This can be an enormous help in debugging your program, because if your program is stopped at a line that uses an enum, you can simply inspect that constant and instantly know its value. On the other hand, because most debuggers cannot print #define values, you would most likely have to search for that value by manually looking it up in a header file.

25. What is the quickest sorting method to use?

The answer depends on what you mean by quickest. For most sorting problems, it just doesn’t matter how quick the sort is because it is done infrequently or other operations take significantly more time anyway. There are three sorting methods in this author’s toolbox that are all very fast and that are useful in different situations. Those methods are quick sort, merge sort, and radix sort.

26. When should the volatile modifier be used?

The volatile modifier is a directive to the compiler’s optimizer that operations involving this variable should not be optimized in certain ways. There are two special cases in which use of the volatile modifier is desirable. The first case involves memory-mapped hardware (a device such as a graphics adaptor that appears to the computer’s hardware as if it were part of the computer’s memory), and the second involves shared memory (memory used by two or more programs running simultaneously).

27. When should the register modifier be used?

The register modifier hints to the compiler that the variable will be heavily used and should be kept in the CPU’s registers, if possible, so that it can be accessed faster.

28. How can you determine the size of an allocated portion of memory?

You can’t, really. free() can , but there’s no way for your program to know the trick free() uses. Even if you disassemble the library and discover the trick, there’s no guarantee the trick won’t change with the next release of the compiler.

29. When does the compiler not implicitly generate the address of the first element of an array?

Whenever an array name appears in an expression such as
· array as an operand of the size of operator
· array as an operand of & operator
· array as a string literal initializer for a character array
Then the compiler does not implicitly generate the address of the address of the first element of an array.

30. 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.

31. Why doesn't the following statement work?

```
char str[ ] = "Hello" ;
strcat ( str, '!' ) ;
```

Answer: The string function strcat( ) concatenates strings and not a character. The basic difference between a string and a character is that a string is a collection of characters, represented by an array of characters whereas a character is a single character. To make the above statement work writes the statement as shown below:

strcat (str, "!") ;

32. What is the benefit of using #define to declare a constant?

Using the #define method of declaring a constant enables you to declare a constant in one place and use it throughout your program. This helps make your programs more maintainable, because you need to maintain only the #define statement and not several instances of individual constants throughout your program.

For instance, if your program used the value of pi (approximately 3.14159) several times, you might want to declare a constant for pi as follows: #define PI 3.14159

Using the #define method of declaring a constant is probably the most familiar way of declaring constants to traditional C programmers. Besides being the most common method of declaring constants, it also takes up the least memory.

Constants defined in this manner are simply placed directly into your source code, with no variable space allocated in memory. Unfortunately, this is one reason why most debuggers cannot inspect constants created using the #define method.

33. What is the purpose of main( ) function ?

The function main( ) invokes other functions within it.It is the first function to be called when the program starts execution.

· It is the starting function
· It returns an int value to the environment that called the program
· Recursive call is allowed for main( ) also.
· It is a user-defined function
· Program execution ends when the closing brace of the function main( ) is reached.
· It has two arguments 1)argument count and 2) argument vector (represents strings passed).
· Any user-defined name can also be used as parameters for main( ) instead of argc and argv

34. How can I search for data in a linked list?

Unfortunately, the only way to search a linked list is with a linear search, because the only way a linked list's members can be accessed is sequentially.
Sometimes it is quicker to take the data from a linked list and store it in a different data structure so that searches can be more efficient.

35. Why should we assign NULL to the elements (pointer) after freeing them?

This is paranoia based on long experience. After a pointer has been freed, you can no longer use the pointed-to data. The pointer is said to dangle; it doesn't point at anything useful.

If you NULL out or zero out a pointer immediately after freeing it, your program can no longer get in trouble by using that pointer. True, you might go indirect on the null pointer instead, but that's something your debugger might be able to help you with immediately.

Also, there still might be copies of the pointer that refer to the memory that has been deallocated; that's the nature of C. Zeroing out pointers after freeing them won't solve all problems.

36. What is a null pointer assignment error? What are bus errors, memory faults, and core dumps?

These are all serious errors, symptoms of a wild pointer or subscript. Null pointer assignment is a message you might get when an MS-DOS program finishes executing. Some such programs can arrange for a small amount of memory to be available "where the NULL pointer points to (so to speak). If the program tries to write to that area, it will overwrite the data put there by the compiler.

When the program is done, code generated by the compiler examines that area. If that data has been changed, the compiler-generated code complains with null pointer assignment. This message carries only enough information to get you worried. There's no way to tell, just from a null pointer assignment message, what part of your program is responsible for the error. Some debuggers, and some compilers, can give you more help in finding the problem.

Bus error: core dumped and Memory fault: core dumped are messages you might see from a program running under UNIX. They're more programmer friendly. Both mean that a pointer or an array subscript was wildly out of bounds. You can get these messages on a read or on a write. They aren't restricted to null pointer problems. The core dumped part of the message is telling you about a file, called core, that has just been written in your current directory. This is a dump of everything on the stack and in the heap at the time the program was running. With the help of a debugger, you can use the core dump to find where the bad pointer was used. That might not tell you why the pointer was bad, but it's a step in the right direction. If you don't have write permission in the current directory, you won't get a core file, or the core dumped message

37. Predict the output or error(s) for the following programmes:

```
void main()
{
int const * p=5;
printf("%d",++(*p));
}
```

Answer: Compiler error: Cannot modify a constant value.
Explanation: p is a pointer to a "constant integer". But we tried to change the value of the "constant integer".

38. main()

```
{
char s[ ]="man";
int i;
for(i=0;s[ i ];i++)
printf("\n%c%c%c%c",s[ i ],*(s+i),*(i+s),i[s]);
}
```

Answer:

```
mmm
aaaa
nnnn
```

Explanation: s[i], *(i+s), *(s+i), i[s] are all different ways of expressing the same idea. Generally array name is the base address for that array. Here s is the base address. i is the index number/ displacement from the base address. So, indirecting it with * is same as s[i]. i[s] may be surprising. But in the case of C it is same as s[i].

39. main()

```
{
float me = 1.1;
double you = 1.1;
if(me==you)
printf("I love U");
else
printf("I hate U");
}
```

Answer: I hate U
Explanation: For floating point numbers (float, double, long double) the values cannot be predicted exactly. Depending on the number of bytes, the precession with of the value represented varies. Float takes 4 bytes and long double takes 10 bytes. So float stores 0.9 with less precision than long double.

Rule of Thumb: Never compare or at-least be cautious when using floating point numbers with relational operators (== , >, <, <=, >=,!=) .

40. main()

```
{
static int var = 5;
printf("%d ",var--);

if(var)
main();
}
```

Answer: 5 4 3 2 1
Explanation: When static storage class is given, it is initialized once. The change in the value of a static variable is retained even between the function calls. Main is also treated like any other ordinary function, which can be called recursively.

41. main()

```
{
int c[ ]={2.8,3.4,4,6.7,5};
int j,*p=c,*q=c;
for(j=0;j<5;j++) {
printf(" %d ",*c);
++q; }
for(j=0;j<5;j++){
printf(" %d ",*p);
++p; }
}
```

Answer: 2 2 2 2 2 2 3 4 6 5

Explanation: Initially pointer c is assigned to both p and q. In the first loop, since only q is incremented and not c , the value 2 will be printed 5 times. In second loop p itself is incremented. So the values 2 3 4 6 5 will be printed.

42. main()

```
{
extern int i;
i=20;
printf("%d",i);
}
```

Answer: Linker Error : Undefined symbol '_i'

Explanation: extern storage class in the following declaration,
extern int i;
specifies to the compiler that the memory for i is allocated in some other program and that address will be given to the current program at the time of linking. But linker finds that no other variable of name i is available in any other program with memory space allocated for it. Hence a linker error has occurred .

43. main()

```
{
int i=-1,j=-1,k=0,l=2,m;
m=i++&&j++&&k++||l++;
printf("%d %d %d %d %d",i,j,k,l,m);
}
```

Answer: 0 0 1 3 1

Explanation: Logical operations always give a result of 1 or 0. And also the logical AND (&&) operator has higher priority over the logical OR (||) operator. So the expression ‘i++&& j++ && k++’ is executed first. The result of this expression is 0 (-1 && -1 && 0 = 0). Now the expression is 0 || 2 which evaluates to 1 (because OR operator always gives 1 except for ‘0 || 0’ combination- for which it gives 0). So the value of m is 1. The values of other variables are also incremented by 1.

44. main()

```
{
char *p;
printf("%d %d ",sizeof(*p),sizeof(p));
}
```

Answer: 1 2

Explanation: The sizeof() operator gives the number of bytes taken by its operand. P is a character pointer, which needs one byte for storing its value (a character). Hence sizeof(*p) gives a value of 1. Since it needs two bytes to store the address of the character pointer sizeof(p) gives 2.

45. main()

```
{
int i=3;
switch(i)
{
default:printf("zero");
case 1: printf("one");
break;
case 2:printf("two");
break;
case 3: printf("three");
break;
}
}
```

Answer : Three

Explanation: The default case can be placed anywhere inside the loop. It is executed only when all other cases doesn't match.

46. main()

```
{
printf("%x",-1<<4);

}
```

Answer: fff0

Explanation: -1 is internally represented as all 1's. When left shifted four times the least significant 4 bits are filled with 0's.The %x format specifier specifies that the integer value be printed as a hexadecimal value.

47. main()

```
{
```

```
char string[]="Hello World";
display(string);
}
void display(char *string)
{
printf("%s",string);
}
```

Answer: Compiler Error: Type mismatch in redeclaration of function display

Explanation: In third line, when the function display is encountered, the compiler doesn't know anything about the function display. It assumes the arguments and return types to be integers, (which is the default type). When it sees the actual function display, the arguments and type contradicts with what it has assumed previously. Hence a compile time error occurs.

48. main()

```
{
int c=- -2;
printf("c=%d",c);
}
```

Answer: c=2;

Explanation: Here unary minus (or negation) operator is used twice. Same maths rules applies, ie. minus * minus= plus.

Note: However you cannot give like --2. Because -- operator can only be applied to variables as a decrement operator (eg., i--). 2 is a constant and not a variable.

49. #define int char

```
main()
{
int i=65;
printf("sizeof(i)=%d",sizeof(i));
}
```

Answer: sizeof(i)=1

Explanation: Since the #define replaces the string int by the macro char

50. main()

```
{
int i=10;
i=!i>14;
Printf ("i=%d",i);
}
```

Answer: i=0

Explanation: In the expression !i>14 , NOT (!) operator has more precedence than ‘ >’ symbol. ! is a unary logical operator. !i (!10) is 0 (not of true is false). 0>14 is false (zero).

51. #include<stdio.h>

```
main()
{
char s[]={'a','b','c','\n','c','\0'};
char *p,*str,*str1;
p=&s[3];
str=p;
str1=s;
printf("%d",++*p + ++*str1-32);
}
```

Answer: 77

Explanation: p is pointing to character '\n'. str1 is pointing to character 'a' ++*p. "p is pointing to '\n' and that is incremented by one." the ASCII value of '\n' is 10, which is then incremented to 11. The value of ++*p is 11. ++*str1, str1 is pointing to 'a' that is incremented by 1 and it becomes 'b'. ASCII value of 'b' is 98.

Now performing (11 + 98 – 32), we get 77("M"); So we get the output 77 :: "M" (Ascii is 77).

52. #include<stdio.h>

```
main()
{
int a[2][2][2] = { {10,2,3,4}, {5,6,7,8} };
int *p,*q;
p=&a[2][2][2];
*q=***a;
printf("%d----%d",*p,*q);
}
```

Answer: SomeGarbageValue---1

Explanation: p=&a[2][2][2] you declare only two 2D arrays, but you are trying to access the third 2D(which you are not declared) it will print garbage values.

*q=***a starting address of a is assigned integer pointer. Now q is pointing to starting address of a. If you print *q, it will print first element of 3D array.

53. #include<stdio.h>

```
main()
{
struct xx
{
int x=3;
char name[]="hello";
};
struct xx *s;
printf("%d",s->x);
printf("%s",s->name);
}
```

Answer: Compiler Error

Explanation: You should not initialize variables in declaration

54. #include<stdio.h>

```
main()

{
struct xx
{
int x;
struct yy
{
char s;
struct xx *p;
};
struct yy *q;
};
}
```

Answer: Compiler Error

Explanation: The structure yy is nested within structure xx. Hence, the elements are of yy are to be accessed through the instance of structure xx, which needs an instance of yy to be known. If the instance is created after defining the structure the compiler will not know about the instance relative to xx. Hence for nested structure yy you have to declare member.

55. main()

```
{
printf("\nab");
printf("\bsi");
printf("\rha");
}
```

Answer: hai

Explanation:
\n - newline
\b - backspace
\r - linefeed

56. main()

```
{
int i=5;
printf("%d%d%d%d%d%d",i++,i--,++i,--i,i);
}
```

Answer: 45545

Explanation: The arguments in a function call are pushed into the stack from left to right. The evaluation is by popping out from the stack. And the evaluation is from right to left, hence the result.

57. #define square(x) x*x

```
main()
```

```
{
int i;
i = 64/square(4);
printf("%d",i);
}
```

Answer: 64

Explanation: the macro call square(4) will substituted by 4*4 so the expression becomes i = 64/4*4 . Since / and * has equal priority the expression will be evaluated as (64/4)*4 i.e. 16*4 = 64

58. main()

```
{
char *p="hai friends",*p1;
p1=p;
while(*p!='\0') ++*p++;
printf("%s %s",p,p1);
}
```

Answer: ibj!gsjfoet

Explanation: ++*p++ will be parse in the given order

_ *p that is value at the location currently pointed by p will be taken
_ ++*p the retrieved value will be incremented
_ when; is encountered the location will be incremented that is p++ will be executed Hence, in the while loop initial value pointed by p is ‘h’, which is changed to ‘i’ by executing ++*p and pointer moves to point, ‘a’ which is similarly changed to ‘b’ and so on. Similarly blank space is converted to ‘!’. Thus, we obtain value in p becomes “ibj!gsjfoet” and since p reaches ‘\0’ and p1 points to p thus p1doesnot print anything.

59. #include <stdio.h>

```
#define a 10
main()
{
#define a 50
printf("%d",a);
}
```

Answer: 50

Explanation: The preprocessor directives can be redefined anywhere in the program. So the most recently assigned value will be taken.

60. #define clrscr() 100

```
main()
{
clrscr();
printf("%d\n",clrscr());
}
```

Answer: 100

Explanation: Preprocessor executes as a seperate pass before the execution of the compiler. So textual replacement of clrscr() to 100 occurs. The input program to compiler looks like this :

```
main()
{
100;
printf("%d\n",100);
}
```

Note: 100; is an executable statement but with no action. So it doesn't give any problem

61. main()

```
{
41printf("%p",main);
}8
```

Answer: Some address will be printed.

Explanation: Function names are just addresses (just like array names are addresses). main() is also a function. So the address of function main will be printed. %p in printf specifies that the argument is an address. They are printed as hexadecimal numbers.

62. main()

```
{
clrscr();
}
clrscr();
```

Answer: No output/error

Explanation: The first clrscr() occurs inside a function. So it becomes a function call. In the second clrscr(); is a function declaration (because it is not inside any function).

63. enum colors {BLACK,BLUE,GREEN}

```
main()
{
printf("%d..%d..%d",BLACK,BLUE,GREEN);
return(1);
}
```

Answer: 0..1..2

Explanation: enum assigns numbers starting from 0, if not explicitly defined.

64. void main()

```
{
char far *farther,*farthest;
printf("%d..%d",sizeof(farther),sizeof(farthest));
}
```

Answer: 4..2

Explanation: The second pointer is of char type and not a far pointer

65. main()

```
{
int i=400,j=300;
printf("%d..%d");
}
```

Answer: 400..300

Explanation: printf takes the values of the first two assignments of the program. Any number of printf's may be given. All of them take only the first two values. If more number of assignments given in the program,then printf will take garbage values.

66. main()

```
{
char *p;
p="Hello";
printf("%c\n",*&*p);
}
```

Answer: H

Explanation: * is a dereference operator & is a reference operator. They can be applied any number of times provided it is meaningful. Here p points to the first character in the string "Hello". *p dereferences it and so its value is H. Again & references it to an address and * dereferences it to the value H.

67. main()

```
{
int i=1;
while (i<=5)
{
printf("%d",i);
if (i>2)
goto here;
i++;
}
}
```

```
fun()
{
here:
printf("PP");
}
```

Answer: Compiler error: Undefined label 'here' in function main

Explanation: Labels have functions scope, in other words the scope of the labels is limited to functions. The label 'here' is available in function fun() Hence it is not visible in function main.

68. main()

```
{
static char names[5][20]={"pascal","ada","cobol","fortran","perl"};
int i;
char *t;
t=names[3];
names[3]=names[4];
names[4]=t;
for (i=0;i<=4;i++)
printf("%s",names[i]);
}
```

Answer: Compiler error: Lvalue required in function main

Explanation: Array names are pointer constants. So it cannot be modified.

69. void main()

```
{
int i=5;
printf("%d",i++ + ++i);
}
```

Answer: Output Cannot be predicted exactly.

Explanation: Side effects are involved in the evaluation of i

70. void main()

```
{
int i=5;
printf("%d",i+++++i);
}
```

Answer: Compiler Error

Explanation: The expression i+++++i is parsed as i ++ ++ + i which is an illegal combination of operators.

71. #include<stdio.h>

```
main()
{
int i=1,j=2;
switch(i)
{
case 1: printf("GOOD");
break;
case j: printf("BAD");

break;

}

}
```

Answer: Compiler Error: Constant expression required in function main.

Explanation: The case statement can have only constant expressions (this implies that we cannot use variable names directly so an error).

Note: Enumerated types can be used in case statements.

72. main()

```
{
int i;
printf("%d",scanf("%d",&i)); // value 10 is given as input here

}
```

Answer: 1

Explanation: Scanf returns number of items successfully read and not 1/0. Here 10 is given as input which should have been scanned successfully. So number of items read is 1.

73. #define f(g,g2) g##g2

```
main()
{
int var12=100;
printf("%d",f(var,12));
}
```

Answer: 100

74. main()

```
{
int i=0;
for(;i++;printf("%d",i)) ;
printf("%d",i);
}
```

Answer: 1

Explanation: before entering into the for loop the checking condition is "evaluated". Here it evaluates to 0 (false) and comes out of the loop, and i is incremented (note the semicolon after the for loop).

75. #include<stdio.h>

```
main()
{
char s[]={'a','b','c','\n','c','\0'};
char *p,*str,*str1;
p=&s[3];
str=p;
str1=s;
printf("%d",++*p + ++*str1-32);
}
```

Answer: M

Explanation: p is pointing to character '\n'.str1 is pointing to character 'a' ++*p "p is pointing to '\n' and that is incremented by one." the ASCII value of '\n' is 10. then it is incremented to 11. the value of ++*p is 11. ++*str1 "str1 is pointing to 'a' that is incremented by 1 and it becomes 'b'. ASCII value of 'b' is 98. Both 11 and 98 is added and result is subtracted from 32. i.e. (11+98-32)=77("M");

76. #include<stdio.h>

```
main()
{
struct xx
{
int x=3;
char name[]="hello";
};
struct xx *s=malloc(sizeof(struct xx));
printf("%d",s->x);
printf("%s",s->name);
}
```

Answer: Compiler Error

Explanation: Initialization should not be done for structure members inside the structure declaration

77. #include<stdio.h>

```
main()
{
struct xx
{
int x;
struct yy
{
char s;
struct xx *p;
};
struct yy *q;
};
}
```

Answer: Compiler Error

Explanation: in the end of nested structure yy a member have to be declared.

78. main()

```
{
extern int i;
i=20;
printf("%d",sizeof(i));
}
```

Answer: Linker error: undefined symbol '_i'.

Explanation: extern declaration specifies that the variable i is defined somewhere else. The compiler passes the external variable to be resolved by the linker. So compiler doesn't find an error. During linking the linker searches for the definition of i. Since it is not found the linker flags an error.

79. main()

```
{
printf("%d", out);
}
int out=100;
```

Answer: Compiler error: undefined symbol out in function main.

Explanation: The rule is that a variable is available for use from the point of declaration. Even though a is a global variable, it is not available for main. Hence an error.

80. main()

```
{
extern out;
printf("%d", out);
}
int out=100;
```

Answer: 100

Explanation: This is the correct way of writing the previous program.

81. main()

```
{
show();
}
void show()
{
printf("I'm the greatest");
}
```

Answer: Compier error: Type mismatch in redeclaration of show.

Explanation: When the compiler sees the function show it doesn't know anything about it. So the default return type (ie, int) is assumed. But when compiler sees the actual definition of show mismatch occurs since it is declared as void. Hence the error.

The solutions are as follows:
1. declare void show() in main() .
2. define show() before main().
3. declare extern void show() before the use of show().

82. main( )

```
{
int a[2][3][2] = {{{2,4},{7,8},{3,4}},{{2,2},{2,3},{3,4}}};
printf(“%u %u %u %d \n”,a,*a,**a,***a);
printf(“%u %u %u %d \n”,a+1,*a+1,**a+1,***a+1);
}
```

Answer:

```
100, 100, 100, 2
114, 104, 102, 3
```

Explanation: The given array is a 3-D one. It can also be viewed as a 1-D array.

2	4	7	8	3	4	2	2	2	3	3	4

100 102 104 106 108
110 112 114 116 118 120 122

thus, for the first printf statement a, *a, **a give address of first element. since the indirection ***a gives the value. Hence, the first line of the output.

for the second printf a+1 increases in the third dimension thus points to value at 114, *a+1 increments in second dimension thus points to 104, **a +1 increments the first dimension thus points to 102 and ***a+1 first gets the value at first location and then increments it by 1. Hence, the output.

83. main( )

```
{
int a[ ] = {10,20,30,40,50},j,*p;
for(j=0; j<5; j++)
{
printf(“%d” ,*a);
a++;
}
p = a;
for(j=0; j<5; j++)
{
printf("%d " ,*p);
p++;
}
}
```

Answer: Compiler error: lvalue required.

Explanation: Error is in line with statement a++. The operand must be an lvalue and may be of any of scalar type for the any operator, array name only when subscripted is an lvalue. Simply array name is a non modifiable lvalue.

84. main( )

```
{
static int a[ ] = {0,1,2,3,4};
int *p[ ] = {a,a+1,a+2,a+3,a+4};
int **ptr = p;
ptr++;
printf("\n %d %d %d", ptr-p, *ptr-a, **ptr);
*ptr++;
printf("\n %d %d %d", ptr-p, *ptr-a, **ptr);
*++ptr;
printf("\n %d %d %d", ptr-p, *ptr-a, **ptr);
++*ptr;
printf("\n %d %d %d", ptr-p, *ptr-a, **ptr);
}
```

Answer:

```
111
222
333
344
```

Explanation: Let us consider the array and the two pointers with some address

a

0	1	2	3	4
100	102	104	106	108

p

100	102	104	106	108
1000	1002	1004	1006	1008

ptr

1000
2000

After execution of the instruction ptr++ value in ptr becomes 1002, if scaling factor for integer is 2 bytes. Now ptr – p is value in ptr –starting location of array p, (1002 – 1000) / (scaling factor) = 1,

*ptr – a = value at address pointed by ptr – starting value of array a, 1002 has a value 102 so the value is (102 – 100)/(scaling factor) = 1, **ptr is the value stored in the location pointed by the pointer of ptr = value pointed by value pointed by 1002 = value pointed by 102 = 1. Hence the output of the firs printf is 1, 1, 1.

After execution of *ptr++ increments value of the value in ptr by scaling factor, so it becomes1004. Hence, the outputs for the second printf are ptr – p = 2, *ptr – a = 2, **ptr = 2.

After execution of *++ptr increments value of the value in ptr by scaling factor, so it becomes1004. Hence, the outputs for the third printf are ptr – p = 3, *ptr – a = 3, **ptr = 3.

After execution of ++*ptr value in ptr remains the same, the value pointed by the value is incremented by the scaling factor. So the value in array p at location 1006 changes from 106 10 108,. Hence, the outputs for the fourth printf are ptr – p = 1006 – 1000 = 3, *ptr – a = 108 – 100 = 4, **ptr = 4.

85. What is dangling pointer in c?

If any pointer is pointing the memory address of any variable but after some variable has deleted from that memory location while pointer is still pointing such memory location. Such pointer is known as dangling pointer and this problem is known as dangling pointer problem.

86. What are merits and demerits of array in c?

Merits:
(a) We can easily access each element of array.
(b) Not necessity to declare too many variables.
(c) Array elements are stored in continuous memory location.

Demerits:
(a) Wastage of memory space. We cannot change size of array at the run time.
(b) It can store only similar type of data

87. Where are the auto variables stored?

Auto variables are stored in main memory and their default value is a garbage value.

88. Why Preincrement operator is faster than Postincrement?

Evaluation of any expression is from left to right. Preincrement is faster because it doesn't need to save the current value for next instruction whereas Postincrement needs to saves current value to be incremented after execution of current instruction.

89. Difference between arrays and linked list?

Major differences between arrays and linked lists are: (i) In array consecutive elements are stored in consecutive memory locations whereas in linked list it not so. (ii) In array address of next element is consecutive and whereas in linked list it is specified in the address part of each node.(iii) Linked List makes better use of memory than arrays.(iv) Insertion or deletion of an element in array is difficult than insertion or deletion in linked list

90. What is the use of typedef?

(i)It increases the portability.
(ii) It simplify the complex declaration and improve readability of the program.

91. What are library Functions?

Library Functions are predefined functions and stored in .lib files.

92. What is a structure?

Structure is a collection of heterogeneous (i.e. related data items which can be of different types) held together to a single unit. The data items enclosed within a structure are called its members which may be of data type int, float, char, array etc.

93. What is a pointer?

Pointer is a variable that contains address of another variable in the memory. Pointers are quite useful in creation of linked data structures (such as linked lst, trees graphs), managing object allocated memory dynamically, optimize the program to execute faster and use less memory.

94. What are the techniques you use for debugging?

(i)Using compiler's features
(ii)Read The Fine Module
(iii)printf( ) debugging
(iv)Code grinding
(v)Assertion

95. What are macros? What are its advantages and disadvantages?

Macro is a Pre-processor.Major advantage of using the macro is to increase the speed of the execution of the program.
Major disadvantage of the macros are:
(i) No type checking is performed in macro. This may cause error.
(ii) A macro call may cause unexpected results.

96. What is difference between Structure and Unions?

(i) In structure every member has its own memory whereas in union its members share the same member space.
(ii) In structure, it is possible to initialize all the members at the same time which is not possible in case of union.
(iii) A structure requires more space than union(for the same type of members).

(iv) In union different interpretations of the same memory space are possible which is not so in case of structures.

97. What are the advantages of using Unions?

(i) Efficient use of memory as it it does not demand memory space for its all members rather it require memory space for its largest member only.
(ii) Same memory space can be interpreted differently for different members of the union.

98. What is the difference between ordinary variable and pointer in C?

An ordinary variable is like a container it can hold any value and we can change the value of ordinary variable at a time throughout the program .A pointer is a variable that stores the address of another Variable.

99. What are segment and offset addresses?
When paging technique is performed, the page will breaks into segments and its sequence is said to be segments and its width can be said as offset. In short,segment is a physical address and offset is logical address.

100. When should a type cast be used?

There are two situations in which to use a type cast. The first use is to change the type of an operand to an arithmetic operation so that the operation will be performed properly.
The second case is to cast pointer types to and from void * in order to interface with functions that expect or return void pointers. For example, the following line type casts the return value of the call to malloc() to be a pointer to a foo structure.
struct foo *p = (struct foo *) malloc(sizeof(struct foo));

101. What is the difference between %d and %*d in c language?

%d give the original value of the variable and %*d give the address of the variable.

eg:-int a=10,b=20;
printf("%d%d",a,b);

```
printf("%*d%*d",a,b);
```

Result is 10 20 1775 1775 .Here 1775 is the starting address of the memory allocation for the integer.a and b having same address because of contagious memory allocation.

102. How does a C program come to know about command line arguments?

When we execute our C program, operating system loads the program into memory. In case of DOS, it first loads 256 bytes into memory, called program segment prefix. This contains file tables,environment segment, and command line information. When we compile the C program the compiler inserts additional code that parses the command, assigning it to the argv array, making the arguments easily accessible within our C program.

103. How are pointer variables initialized?

Pointer variable are initialized by one of the following two ways

- Static memory allocation
- Dynamic memory allocation

104. What is modular programming?

If a program is large, it is subdivided into a number of smaller programs that are called modules or subprograms. If a complex problem is solved using more modules, this approach is known as modular programming

105. Where does global, static, local, register variables and C Program instructions get stored?

Global , static, local : In main memory
Register variable: In registers
C program : In main memory.

106. Where are the auto variables stored?

Auto variables are stored in main memory and their default value is a garbage value.

107. What is an lvalue?

An lvalue is an expression to which a value can be assigned. The lvalue expression is located on the left side of an assignment statement, whereas an rvalue is located on the right side of an assignment statement. Each assignment statement must have an lvalue and an rvalue. The lvalue expression must reference a storable variable in memory. It cannot be a constant

108. What is an argument? Differentiate between formal arguments and actual arguments?

An argument is an entity used to pass the data from calling function to the called function. Formal arguments are the arguments available in the function definition. They are preceded by their own data types. Actual arguments are available in the function call.

109. When is a switch statement better than multiple if statements?

A switch statement is generally best to use when you have more than two conditional expressions based on a single variable of numeric type.

110. Differentiate between a linker and linkage?

A linker converts an object code into an executable code by linking together the necessary build in functions. The form and place of declaration where the variable is declared in a program determine the linkage of variable.

111. Define Operator, Operand, and Expression in 'C'?

Operators are symbols which take one or more operands or expressions and perform arithmetic or logical computations.
Operands are variables or expressions which are used in operators to evaluate the expression.
Combination of operands and operators form an expression.

112. What will be the result of the following code?

```
#define TRUE 0 // some code
while(TRUE)
{
   // some code
}
```

Answer: This will not go into the loop as TRUE is defined as 0.

113. What will be printed as the result of the operation below:

```
main()
{
   int a=0;
   if(a==0)
      printf(“Cisco Systemsn”);
      printf(“Cisco Systemsn”);
}
```

Answer: Two lines with “Cisco Systems” will be printed.

114. Do you know pragma directives in c?

Pragma is implementation specific directive i.e each pragma directive has different implementation rule and use. If compiler does not recognize particular pragma it simply ignore that pragma statement without showing any error or warning message and execute the whole program assuming this pragma statement is not present.

115. Predict the output or error

```
main()
{
clrscr();
}
clrscr();
```

Ans:No output/error
Explanation:The first clrscr() occurs inside a function. So it becomes a function call. In the second clrscr(); is a function declaration (because it is not inside any function).

116. Predict the output or error

```
enum colors {BLACK,BLUE,GREEN}
 main()
{

 printf("%d..%d..%d",BLACK,BLUE,GREEN);

 return(1);
}
```

Answer: 0..1..2
Explanation: enum assigns numbers starting from 0, if not explicitly defined.

117. Predict the output or error

```
 main()
{
int i;
printf("%d",scanf("%d",&i));  // value 10 is given as input here
}
```

Answer:1
Explanation: Scanf returns number of items successfully read and not 1/0. Here 10 is given as input which should have been scanned successfully. So number of items read is 1.

118. what will be the position of the file marker?

```
        a: fseek(ptr,0,SEEK_SET);
        b: fseek(ptr,0,SEEK_CUR);
```

Ans: a: The SEEK_SET sets the file position marker to the starting of the file.
b: The SEEK_CUR sets the file position marker to the current position of the file.

119. Predict the output or error

```
main()
        {
```

```
        main();
        }
```

Ans: Runtime error : Stack overflow.
Explanation: main function calls itself again and again. Each time the function is called its return address is stored in the call stack. Since there is no condition to terminate the function call, the call stack overflows at runtime. So it terminates the program and results in an error.

120. Predict the output or error

```
main()
{
         int i=5,j=6,z;
         printf("%d",i+++j);
        }
```

Answer:11

Explanation:the expression i+++j is treated as (i++ + j)

121. Predict the output or error

```
main()
        {
        int k=1;
        printf("%d==1 is ""%s",k,k==1?"TRUE":"FALSE");
        }
```

Ans: 1==1 is TRUE
Explanation: When two strings are placed together (or separated by white-space) they are concatenated (this is called as "stringization" operation). So the string is as if it is given as "%d==1 is %s". The conditional operator(?:) evaluates to "TRUE".

122. What is use of void data type?

Void is an empty data type normally used as a return type in C/C++, C#, Java functions/methods to declare that no value will be return by the function.
The another used of void is to declare the pointer in C/C++ where It is not sure what data type is addressed by the pointer.

123. four type of scope in c:

Block scope.
Function scope.
File scope.
Program scope.

124. Tell any five properties of auto variables?

auto variables are defined inside a function. A variable declared inside the function without storage class name is, by default, an auto variable. These functions are declared on the stack. The stack provides temporary storage.

125. What is automatic type promotion in c?

In c if two operands are of different data type in a binary operation then before performing any operation compiler will automatically convert the operand of lower data type to higher data type .This phenomenon is known as automatic type conversion. For example:

```
int a=10,c;
float b=5.5f;
c=a+b;
```

Here a int variable while b is float variable. So before performing addition operation value of the variable a (Lower data type) will automatically convert into float constant (higher data type) then it will perform addition operation.

126. What are differences between sizeof operator and strlen function?

sizeof is keyword of c which can find size of a string constant including null character but strlen is function which has been defined string.h and can find number of characters in a string excluding null character.

127. What is command line argument?

Getting the arguments from command prompt in c is known as command line arguments. In c main function has three arguments.

They are:
Argument counter
Argument vector
Environment vector

128. void main(){

```
   int x=5,y=10,z=15,val;
   val=sum(x,(y=0,z=0,y),z);
   clrscr();
   printf("%d",val);
   getch();
}
sum(int x,int y,int z){
   return x+y+z;
}
```

Answer:20
Explanation: In the above program comma after Y=0 &Z=0 are behaving as operator.

129. what is nested structure?

A structure is a collection of one or more variables, possibly of different data types, grouped together under a single name for convenient handling. Structures can contain other structures as members; in other words, structures can nest.

130. What is slack byte in structure?

To store any type of data in structure there is minimum fixed byte which must be reserved by memory. This minimum byte is known as word boundary. Word boundary depends upon machine. TURBO C is based on 8086 microprocessor which has two byte word boundary. So any data type reserves at least two byte space.

131.What is prototype of printf function?

Prototype of printf function is:
int printf(const char *format ,…)

132.What is difference between declaration and definition?

During declaration we just specify the type and no memory is allocated to the variable. But during the definition an initial value is assigned and memory is allocated to the variable.

133. What is function recursion?

When a function of body calls the same function then it is called as 'recursive function.'
Example:

```
Recursion()
{
   printf("Recursion !");
   Recursion();
}
```

134. What is self referential structure ?

A self-referential structure is one of the data structures which refer to the pointer to (points) to another structure of the same type.

135. What is far pointer?

The pointer which can point or access whole the residence memory of RAM i.e. which can access all 16 segments is known as far pointer.

136. What is pascal and cdecl keyword in c language?

There are two types of parameters passing conventions in c:
1. pascal: In this style function name should (not necessary) in the uppercase .First parameter of function call is passed to the first parameter of function definition and so on.
2. cdecl: In this style function name can be both in the upper case or lower case. First parameter of function call is passed to the last parameter of function definition. It is default parameter passing convention.

137. What is use of #pragma inline directive in c language?

#pragma inline only tells the compiler that source code of program contain inline assembly language code .In c we can write assembly language program with help of asm keyword.

138. What is the meaning of multilevel pointers in c?

A pointer is pointer to another pointer which can be pointer to others pointers and so on is known as multilevel pointers. We can have any level of pointers.

139. What is huge pointer in c?

The pointer which can point or access whole the residence memory of RAM i.e. which can access all the 16 segments is known as huge pointer.

140. Is it possible to rename any function in c?
Yes, we can rename any function using typedef keyword. It is useful when function declaration is too complex and we have to give any simple name or if we have to create more numbers of function of the same type.

141. Do you know, what is the meaning and use of static keyword in c?

Keyword static is used for declaring static variables in c. This modifier is used with all data types like int, float, double, array, pointer, structure, function etc.

142. What is difference between .com program and .exe program?

Both .com and .exe program are executable program but .com program execute faster than .exe program. All drivers are .com program. .com file has higher preference than .exe For example:

143. Difference between TSR and TSO program

TSO means terminate but stay outside. It is that program, which release the main memory after the execution of the program. Example ms paint, notepad, turbo c compilers etc.

TSR means terminate but stay residence .It is those program, which after the execution of the program does not release the RAM (main memory).e.g. antivirus.

144. Describe turbo c compiler?

Turbo c is an IDE of c programming language created by Borland. Turbo C 3.0 is based on MS DOS operation system. It is one of the most popular c compilers. It uses 8086 microprocessor which is 16 bit microprocessor. It has 20 address buses and 16 data bus. Its word length is two byte.

145. Out of fgets() and gets() which function is safe to use and why?

fgets() is safer than gets(), because we can specify a maximum input length. Neither one is completely safe, because the compiler can’t prove that programmer won’t overflow the buffer he pass to fgets ().

146. Difference between strdup and strcpy?

Both copy a string. strcpy wants a buffer to copy into. strdup allocates a buffer using malloc().
Unlike strcpy(), strdup() is not specified by ANSI .

147. Differentiate between a for loop and a while loop? What are it uses?

For executing a set of statements fixed number of times we use for loop while when the number of iterations to be performed is not known in advance we use while loop.

148. What is storage class? What are the different storage classes in C?
Storage class is an attribute that changes the behavior of a variable. It controls the lifetime, scope and linkage. The storage classes in c are auto, register, and extern, static, typedef.

149. What are the uses of a pointer?

(i)It is used to access array elements
(ii)It is used for dynamic memory allocation.

(iii)It is used in Call by reference
(iv)It is used in data structures like trees, graph, linked list etc.

150.In header files whether functions are declared or defined?

Functions are declared within header file. That is function prototypes exist in a header file,not function bodies. They are defined in library (lib).

151. Difference between pass by reference and pass by value?

Pass by reference passes a pointer to the value. This allows the callee to modify the variable directly.Pass by value gives a copy of the value to the callee. This allows the callee to modify the value without modifying the variable. (In other words, the callee simply cannot modify the variable, since it lacks a reference to it.)

152. What are enumerations?

They are a list of named integer-valued constants. Example:enum color { black , orange=4,yellow, green, blue, violet };This declaration defines the symbols “black”, “orange”, “yellow”, etc. to have the values “1,” “4,” “5,” … etc. The difference between an enumeration and a macro is that the enum actually declares a type, and therefore can be type checked.

153. Are pointers integer?

No, pointers are not integers. A pointer is an address. It is a positive number.

154. What is static memory allocation?

Compiler allocates memory space for a declared variable. By using the address of operator, the reserved address is obtained and this address is assigned to a pointer variable. This way of assigning pointer value to a pointer variable at compilation time is known as static memory allocation.

155. What is dynamic memory allocation?

A dynamic memory allocation uses functions such as malloc() or calloc() to get memory dynamically. If these functions are used to get memory dynamically and the values returned by these function are assigned to pointer variables, such a way of allocating memory at run time is known as dynamic memory allocation.

156. What modular programming?

If a program is large, it is subdivided into a number of smaller programs that are called modules or subprograms. If a complex problem is solved using more modules, this approach is known as modular programming

157. What is a function?

A large program is subdivided into a number of smaller programs or subprograms. Each subprogram specifies one or more actions to be performed for the larger program. Such sub programs are called functions.

158. Difference between formal argument and actual argument?

Formal arguments are the arguments available in the function definition. They are preceded by their own data type. Actual arguments are available in the function call. These arguments are given as constants or variables or expressions to pass the values to the function.

159. what are C tokens?

There are six classes of tokens: identifier, keywords, constants, string literals, operators and other separators.

160. What are C identifiers?

These are names given to various programming element such as variables, function, arrays.It is a combination of letter, digit and underscore.It should begin with letter. Backspace is not allowed.

161. Difference between syntax vs logical error?

Syntax Error
These involves validation of syntax of language.
compiler prints diagnostic message.

Logical Error
logical error are caused by an incorrect algorithm or by a statement mistyped in such a way that it doesn't violet syntax of language.
difficult to find.

162. What are the facilities provided by preprocessor?
file inclusion
substitution facility
conditional compilation

163.What do the functions atoi(), itoa() and gcvt() do?

atoi() is a macro that converts integer to character.
itoa() It converts an integer to string
gcvt() It converts a floating point number to string

164. What is FILE?

FILE is a predefined data type. It is defined in stdio.h file.

165. What is a file?
A file is a region of storage in hard disks or in auxiliary storage devices.It contains bytes of information .It is not a data type.

166. What is a linker, and what are dynamic and static linking?

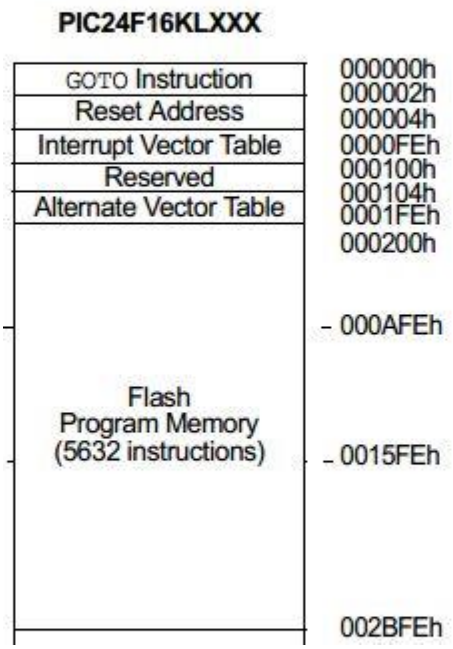
Link editors are commonly known as linkers. The compiler automatically invokes the linker as the last step in compiling a program. The linker inserts code (or maps in shared libraries) to resolve program library references, and/or combines object modules into an executable image suitable for loading into memory. On Unix-like systems, the linker is typically invoked with the `ld` command.

Static linking is the result of the linker copying all library routines used in the program into the executable image. This may require more disk space and memory than dynamic linking, but is both faster and more portable, since it does not require the presence of the library on the system where it is run.

Dynamic linking is accomplished by placing the name of a sharable library in the executable image. Actual linking with the library routines does not occur until the image is run, when both the executable and the library are placed in memory. An advantage of dynamic linking is that multiple programs can share a single copy of the library.

BOOT PROCESS IN MICROCONTROLLER

The microcontroller boot process starts by simply applying power to the system. Once the voltage rails stabilize, the microcontroller looks to the reset vector for the location in flash where the start-up instruction can be found. The reset vector is a special location within the flash memory map. For example, taking a look at Figure 1, it can be seen that at address 0x0002 is where the reset vector is located within the memory map.



ResetVector

Figure 1 – Reset Vector Location

The address that is stored at the reset vector is loaded by the microcontroller and the instructions that are contained there are then loaded and executed by the CPU. Now these first instructions aren't the start of main that the developer created. Instead, these are instructions on how to start-up the microcontroller.

The first thing that usually occurs is that the vector tables that are stored in flash are copied to RAM. They are copied from and to the location that is specified in the linker file at the time the executable program is created. One reason for copying the vector tables to RAM is that it is faster to execute from RAM than flash. This helps to decrease the latency of any interrupt calls within the system. Depending on the particular architecture of the microcontroller there may then be an instruction to update a vector table register so that the microcontroller knows where the start of the RAM table is.

Next the initialized data sections are copied into RAM. This is usually variables that are stored in the .data section of the linker. Examples of initialized data would be static, global and static local variables that have been provided with an initialization value during compile time. These are explicit definitions such as int Var = 0x32;.

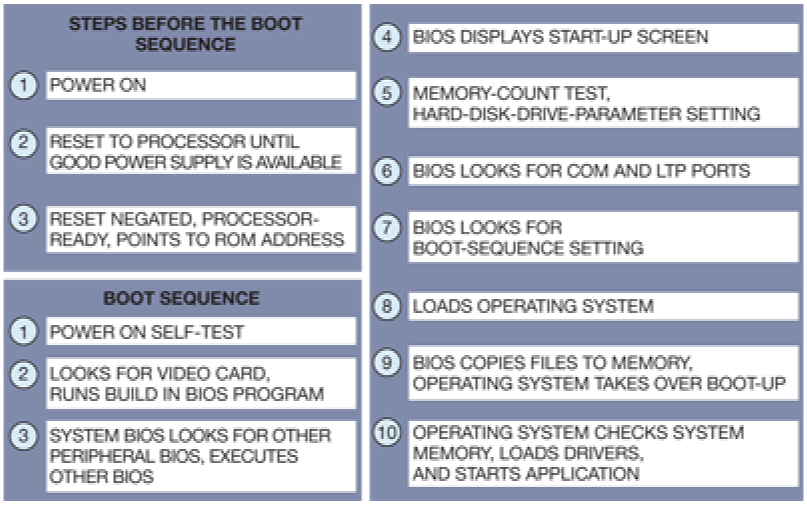
Following the copy of the data section, the .bss section is also copied. The .bss section contains variables that are not initialized explicitly or that have been initialized to a value of zero. A simple example is that the variable static int Var; would be contained within this section.

Finally, the microcontroller will copy any RAM functions from flash to RAM. Once again it is sometimes worthwhile to execute certain functions out of RAM rather than flash due to the execution speed being slightly faster. These are functions usually decided upon by the developer and purposely placed there in the linker file prior to compiling the program.

This entire process is often known as the “C Copy Down”. Without performing this copy down the C environment would not be properly setup in order to execute the program. Usually once the copy down has been completed the microcontroller then jumps to the start of main where the developers application then begins.

The microcontroller boot process is actually relatively straight forward. Often times though it is written in assembly language or some other obscure and sparsely documented way so that it is difficult to get a clear understanding of how the microcontroller actually gets to main. Instead it looks like a very complex and nearly unknowable process which then makes custom development of boot code a potentially painful process.

Jacob Beningo is a Certified Software Development Professional (CSDP) whose expertise is in embedded software. He works with companies to decrease costs and time to market while maintaining a quality and robust product. He is an avid tweeter, a tip and trick guru, a homebrew connoisseur and a fan of pineapple! Feel free to contact him at jacob@beningo.com at his website www.beningo.com, and sign-up for his monthly embedded newsletter here



STEPS BEFORE THE BOOT SEQUENCE

1 POWER ON
2 RESET TO PROCESSOR UNTIL GOOD POWER SUPPLY IS AVAILABLE
3 RESET NEGATED, PROCESSOR-READY, POINTS TO ROM ADDRESS

BOOT SEQUENCE

1 POWER ON SELF-TEST
2 LOOKS FOR VIDEO CARD, RUNS BUILD IN BIOS PROGRAM
3 SYSTEM BIOS LOOKS FOR OTHER PERIPHERAL BIOS, EXECUTES OTHER BIOS
4 BIOS DISPLAYS START-UP SCREEN
5 MEMORY-COUNT TEST, HARD-DISK-DRIVE-PARAMETER SETTING
6 BIOS LOOKS FOR COM AND LTP PORTS
7 BIOS LOOKS FOR BOOT-SEQUENCE SETTING
8 LOADS OPERATING SYSTEM
9 BIOS COPIES FILES TO MEMORY, OPERATING SYSTEM TAKES OVER BOOT-UP
10 OPERATING SYSTEM CHECKS SYSTEM MEMORY, LOADS DRIVERS, AND STARTS APPLICATION

Figure 1 Microsoft’s Windows XP moves through a suite of steps as it boots a system.