

---

## Interview Questions

---

**Steve Summit <s...@eskimo.com>**

[posted and mailed]

In article <6imrb3\$k...@asansor.ege.edu.tr>, Ulas Ergin wrote:  
> I am looking forward to learn the answers to Steve Summit's C  
Interview  
> Questions. I think I have a lot to learn from them.  
> Would you please post (or e-mail) the answers please, Mr Summit?

I had already forgotten; thanks for the reminder.  
Here are the questions again, interspersed with my answers.  
I hope this won't touch off *\*too\** many flame wars, although I've  
got a nagging fear that someone will follow up to quibble with at  
least one of these, after stripping off the disclaimers and all  
identifying context...

(As I mention in the disclaimers just below, I prepared these  
answers before I had read a draft of the impending C9X standard,  
so a few of them are dated already.)

\* \* \*

"Killer" C Interview Questions  
(And a few Easy Ones, too)

Steve Summit  
Copyright 1997, 1998

[DISCLAIMER: This is, on balance, a very difficult test.  
DO NOT use it in an actual interview situation; you would in all  
likelihood only embarrass yourself and insult your interviewee.  
In particular, you would obviously not want to ask a question  
which you yourself did not know the answer to, but the answers  
to many of these questions are not obvious or well-known.

There are several trick questions here, as well as a number which  
are explicitly marked "poor". The poor questions are, alas, not  
uncommon in actual interviews, and are presented here only so  
that their faults and wrong answers can be presented.

This test is intended for study purposes only. Most of the

answers can be found in the comp.lang.c FAQ list.

The answers here were prepared before I read a draft of the impending C9X standard. I would word several of them differently today in anticipation of that standard, and a few of them will become wrong when that standard is adopted.

Acknowledgments: This test is an expanded version of one I prepared for a training class held at the request of Tony McNamara, at a now-defunct company called SCS/Compute (no relation).]

1.1: How can you print a literal % with printf?

A: %%

1.2: Why doesn't \% print a literal % with printf?

A: Backslash sequences are interpreted by the compiler (\n, \", \0, etc.), and \% is not one of the recognized backslash sequences. It's not clear what the compiler would do with a \% sequence -- it might delete it, or replace it with a single %, or perhaps pass it through as \ %. But it's printf's behavior we're trying to change, and printf's special character is %. So it's a %-sequence we should be looking for to print a literal %, and printf defines the one we want as %%.

1.3: Are the parentheses in a return statement mandatory?

A: No. The formal syntax of a return statement is

```
return expression ;
```

But it's legal to put parentheses around any expression, of course, whether they're needed or not.

1.4: How can %f work for type double in printf if %lf is required in scanf?

A: In variable-length argument lists such as printf's, the old "default argument promotions" apply, and type float is implicitly converted to double. So printf always receives doubles, and defines %f to be the sequence that works whether you had passed a float or a double. (Strictly speaking, %lf is *not* a valid printf format specifier, although most versions of printf quietly accept it.)

scanf, on the other hand, always accepts pointers, and

the types pointer-to-float and pointer-to-double are very different (especially when you're using them for storing values). No implicit promotions apply.

1.5: If a machine uses some nonzero internal bit pattern for null pointers, how should the NULL macro be defined?

A: As 0 (or (char \*)0), as usual. The \*compiler\* is responsible for translating null pointer constants into internal null pointer representations, not the preprocessor.

1.6: If p is a pointer, is the test

```
if(p)
```

valid? What if a machine uses some nonzero internal bit pattern for null pointers?

A: The test is always valid. Since the definition of "true" in C is "not equal to 0," the test is equivalent to

```
if(p != 0)
```

and the compiler then translates the 0 into the appropriate internal representation of a null pointer.

1.7: What is the ANSI Standard definition of a null pointer constant?

A: "An integral constant expression with the value 0, or such an expression cast to type (void \*)".

1.8: What does the auto keyword mean? When is it needed?

A: auto is a storage-class specifier, just like extern and static. But since automatic duration is the default for local variables (and meaningless, in fact illegal, for global variables), the keyword is never needed. (It's a relic from the dawn of C.)

1.9: What does \*p++ increment?

A: The pointer p. To increment what p points to, use (\*p)++ or ++\*p.

1.10: What's the value of the expression 5["abcdef"] ?

A: 'f'.

(The string literal "abcdef" is an array, and the expression is equivalent to "abcdef"[5]. Why is the inside-out expression equivalent? Because a[b] is equivalent to \*(a + b) which is equivalent to \*(b + a) which is equivalent to b[a].

1.11: [POOR QUESTION] How can you swap two integer variables without using a temporary?

A: The reason that this question is poor is that the answer ceased to be interesting when we came down out of the trees and stopped using assembly language.

The "classic" solution, expressed in C, is

```
a ^= b;  
b ^= a;  
a ^= b;
```

Due to the marvels of the exclusive-OR operator, after these three operations, a's and b's values will be swapped.

However, it is exactly as many lines, and (if we can spare one measly word on the stack) is likely to be more efficient, to write the obvious

```
int t = a;  
a = b;  
b = t;
```

No, this doesn't meet the stipulation of not using a temporary. But the whole reason we're using C and not assembly language (well, one reason, anyway) is that we're not interested in keeping track of how many registers we have.

If the processor happens to have an EXCH instruction, the compiler is more likely to recognize the possibility of using it if we use the three-assignment idiom, rather than the three-XOR.

By the way, the even more seductively concise rendition of the "classic" trick in C, namely

```
a ^= b ^= a ^= b
```

is, strictly speaking, undefined, because it modifies a twice between sequence points. Also, if an attempt is made to use the idiom (in any form) in a function which

is supposed to swap the locations pointed to by two pointers, as in

```
swap(int *p1, *p2)
{
    *p1 ^= *p2;
    *p2 ^= *p1;
    *p1 ^= *p2;
}
```

then the function will fail if it is ever asked to swap a value with itself, as in

```
swap(&a, &a);
```

or

```
swap(&a[i], &a[j]);
```

when  $i == j$ . (The latter case is not uncommon in sorting algorithms. The effect when  $p1 == p2$  is that the pointed-to value is set to 0.)

1.12: What is `sizeof('A')` ?

A: The same as `sizeof(int)`. Character constants have type `int` in C. (This is one area in which C++ differs.)

1.13: According to the ANSI Standard, how many bits are there in an `int`? A `char`? A `short int`? A `long int`? In other words, what is `sizeof(int)` ? `sizeof(char)` ? `sizeof(short int)` ? `sizeof(long int)` ?

A: ANSI guarantees that the range of a signed `char` is at least  $\pm 127$ , of a `short int` is at least  $\pm 32767$ , of an `int` at least  $\pm 32767$ , and a `long int` at least  $\pm 2147483648$ .

So

we can deduce that a `char` must be at least 8 bits, an `int` or a `short int` must be at least 16 bits, and a `long int` must be at least 32 bits. The only guarantees about `sizeof` are that

```
1 = sizeof(char) <= sizeof(short) <= sizeof(int) <=
sizeof(long)
```

1.14: If `arr` is an array, in an ordinary expression, what's the difference between `arr` and `&arr` ?

A: If the array is of type T, the expression `arr` yields a pointer of type `pointer-to-T` pointing to the array's first element. The expression `&arr`, on the other hand, yields a pointer of type `pointer-to-array-of-T` pointing to the entire array. (The two pointers will likely have the same "value," but the types are distinct. The difference would be visible if you assigned or incremented the resulting pointer.)

1.15: What's the difference between

```
char *p = malloc(n);
```

and

```
char *p = malloc(n * sizeof(char));
```

?

A: There is little or no difference, since `sizeof(char)` is by definition exactly 1.

1.16: What's the difference between these three declarations?

```
char *a = "abc";  
char b[] = "abc";  
char c[3] = "abc";
```

A: The first declares a pointer-to-char, initialized to point to a four-character array somewhere in (possibly read-only) memory containing the four characters `a b c \0`. The second declares an array (a writable array) of 4 characters, initially containing the characters `a b c \0`. The third declares an array of 3 characters, initially containing `a b c`. (The third array is therefore not an immediately valid string.)

1.17: The first line of a source file contains the line

```
extern int f(struct x *);
```

The compiler warns about "struct x declared inside parameter list". What is the compiler worried about?

A: For two structures to be compatible, they must not only have the same tag name but be defined in the same scope. A function prototype, however, introduces a new, nested scope for its parameters. Therefore, the structure tag `x` is defined in this narrow scope, which almost immediately disappears. No other `struct x` pointer in this

translation unit can therefore be compatible with `f`'s first parameter, so it will be impossible to call `f` correctly (at least, without drawing more warnings). The warning alluded to in the question is trying to tell you that you shouldn't mention struct tags for the first time in function prototypes.

(The warning message in the question is actually produced by `gcc`, and the message runs on for two more lines, explaining that the scope of the structure declared "is only this definition or declaration, which is probably not what you want.")

1.18: List several ways for a function to safely return a string.

A: It can return a pointer to a static array, or it can return a pointer obtained from `malloc`, or it can fill in a buffer supplied by the caller.

1.19: [hard] How would you implement the `va_arg()` macro in `<stdarg.h>`?

A: A straightforward implementation, assuming a conventional stack-based architecture, is

```
#define va_arg(argp, type) (((type *) (argp +=  
sizeof(type)))[-1])
```

This assumes that type `va_list` is `char *`.

1.20: Under what circumstances is the declaration

```
typedef xxx int16;
```

where `xxx` is replaced with an appropriate type for a particular machine, useful?

A: It is potentially useful if the `int16` typedef is used to declare variables or structures which will be read from or written to some external data file or stream in some fixed, "binary" format. (However, the typedef can at most ensure that the internal type is the same size as the external representation; it cannot correct for any byte order discrepancies.)

Such a typedef may also be useful for allowing precompiled object files or libraries to be used with different compilers (compilers which define basic types such as `int` differently), without recompilation.

1.21: Suppose that you declare

```
struct x *xp;
```

without any definition of struct x. Is this legal?  
Under what circumstances would it be useful?

A: It is perfectly legal to refer to a structure which has not been "fleshed out," as long as the compiler is never asked to compute the size of the structure or generate offsets to any members. Passing around pointers to otherwise undefined structures is quite acceptable, and is a good way of implementing "opaque" data types in C.

1.22: What's the difference between

```
struct x1 { ... };
```

```
typedef struct { ... } x2;
```

?

A: The first declaration declares a structure tag x1; the second declares a typedef name x2. The difference becomes clear when you declare actual variables of the two structure types:

```
struct x1 a, b;
```

but

```
x2 a, b;
```

(This distinction is insignificant in C++, where all structure and class tags automatically become full-fledged types, as if via typedef.)

1.23: What do these declarations mean?

```
int **a();  
int (*b)();  
int (*c[3])();  
int (*d)[10];
```

A: declare a as function returning pointer to pointer to int  
declare b as pointer to function returning int  
declare c as array of 3 pointers to functions returning  
int  
declare d as pointer to array of 10 ints

The way to read these is "inside out," remembering that [] and () bind more tightly than \*, unless overridden by

explicit parentheses.

1.24: State the declaration for a pointer to a function returning a pointer to char.

A: `char *(*f)();`

1.25: If `sizeof(long int)` is 4, why might `sizeof` report the size of the structure

```
    struct x {char c; long int i;};
```

as 8 instead of 5?

A: The compiler will typically allocate invisible padding between the two members of the structure, to keep `i` aligned on a longword boundary.

1.26: If `sizeof(long int)` is 4, why might `sizeof` report the size of the structure

```
    struct y {long int i; char c;};
```

as 8 instead of 5?

A: The compiler will typically allocate invisible padding at the end of structure, so that if an array of these structures is allocated, the `i`'s will all be aligned.

1.27: [POOR QUESTION] If `i` starts out as 1, what does the expression

```
    i++ + i++
```

evaluate to? What is `i`'s final value?

A: This is a poor question because it has no answer. The expression attempts to modify `i` twice between sequence points (not to mention modifying and inspecting `i`'s value, where the inspection is for purposes other than determining the value to be stored), so the expression is undefined. Different compilers can (and do) generate different results, and none of them is "wrong."

1.28: Consider these definitions:

```
#define Push(val)    (*stackp++ = (val))
#define Pop()        (*--stackp)
```

```
int stack[100];
```

```
int *stackp = stack;
```

Now consider the expression

```
Push(Pop() + Pop())
```

1. What is the expression trying to do? In what sort of program might such an expression be found?

2. What are some deficiencies of this implementation? Under what circumstances might it fail?

A: The expression is apparently intended to pop two values from a stack, add them, and push the result. This code might be found in a calculator program, or in the evaluation loop of the engine for a stack-based language.

The implementation has at least four problems, however. The Push macro does not check for stack overflow; if more than 100 values are pushed, the results will be unpredictable. Similarly, the the Pop macro does not check for stack underflow; an attempt to pop a value when the stack is empty will likewise result in undefined behavior.

On a stylistic note, the stackp variable is global as far as the Push and Pop macros are concerned. If it is certain that, in a particular program, only one stack will be used, this assumption may be a reasonable one, as it allows considerably more succinct invocations. If multiple stacks are a possibility, however, it might be preferable to pass the stack pointer as an argument to the Push and Pop macros.

Finally, the most serious problem is that the "add" operation as shown above is *\*not\** guaranteed to work! After macro expansion, it becomes

```
(*stackp++ = ((*--stackp) + (*--stackp)))
```

This expression modifies a single object more than once between sequence points; specifically, it modifies stackp three times. It is not guaranteed to work; moreover, there are popular compilers (one is gcc) under which it *\*will\** *\*not\** work as expected. (The extra parentheses do nothing to affect the evaluation order; in particular, they do not make it any more defined.)

1.29: [POOR QUESTION] Write a small function to sort an array of integers

of integers.

A: This is a poor question because no one writes small functions to sort arrays of integers any more, except as pedagogical exercises. If you have an array of integers that needs sorting, the thing to do is call your library sort routine -- in C, `qsort()`. So here is my "small function":

```
static int intcmp(const void *, const void *);

sortints(int a[], int n)
{
    qsort(a, n, sizeof(int), intcmp);
}

static int intcmp(const void *p1, const void *p2)
{
    int i1 = *(const int *)p1;
    int i2 = *(const int *)p2;
    if(i1 < i2)
        return -1;
    else if(i1 > i2)
        return 1;
    else
        return 0;
}
```

(The reason for using two comparisons and three explicit return statements rather than the "more obvious" `return i1 - i2;` is that `i1 - i2` can underflow, with unpredictable results.)

1.30: State the ANSI rules for determining whether an expression is defined or undefined.

A: An expression is undefined if, between sequence points, it attempts to modify the same location twice, or if it attempts to both read from and write to the same location. It's permissible to read and write the same location only if the laws of causality (a higher authority even than X3.159) prove that the read must unfailingly precede the write, that is, if the write is of a value which was computed from the value which was read. This exception means that old standbys such as `i = i + 1` are still legal.

Sequence points occur at the ends of full expressions (expression statements, and the expressions in `if`, `while`, `for`, `do/while`, `switch`, and `return` statements, and initializers), at the `&&`, `||`, and comma operators, at the end of the first expression in a `?:` expression, and just

before the call of a function (after the arguments have all been evaluated).

(The actual language from the ANSI Standard is

```
Between the previous and next sequence point an
object shall have its stored value modified at
most once by the evaluation of an expression.
Furthermore, the prior value shall be accessed
only to determine the value to be stored.
```

)

1.30a: What's the difference between these two declarations?

```
extern char x[];
extern char *x;
```

A: The first is an external declaration for an array of char named x, defined elsewhere. The second is an external declaration for a pointer to char named x, also defined elsewhere. These declarations could not both appear in the same program, because they specify incompatible types for x.

1.31: What's the difference between these two declarations?

```
int f1();
extern int f2();
```

A: There is no difference; the extern keyword is essentially optional in external function declarations.

1.32: What's the difference between these two declarations?

```
extern int f1();
extern int f2(void);
```

A: The first is an old-style function declaration declaring f1 as a function taking an unspecified (but fixed) number of arguments; the second is a prototype declaration declaring f2 as a function taking precisely zero arguments.

1.33: What's the difference between these two definitions?

```
int f1()
{
}

int f2(void)
```

```
{  
}
```

A: There is no difference, other than that the first uses the old definition style and the second uses the prototype style. Both functions take zero arguments.

1.34: How does operator precedence influence order of evaluation?

A: Only partially. Precedence affects the binding of operators to operands, but it does *not* control (or even influence) the order in which the operands themselves are evaluated. For example, in

```
a() + b() * c()
```

we have no idea what order the three functions will be called in. (The compiler might choose to call a first, even though its result will be needed last.)

1.35: Will the expression in

```
if(n != 0 && sum / n != 0)
```

ever divide by 0?

A: No. The "short circuiting" behavior of the && operator guarantees that sum / n will not be evaluated if n is 0 (because n != 0 is false).

1.36: Will the expression

```
x = ((n == 0) ? 0 : sum / n)
```

ever divide by 0?

A: No. Only one of the pair of controlled expressions in a ?: expression is evaluated. In this example, if n is 0, the third expression will not be evaluated at all.

1.37: Explain these three fragments:

```
if((p = malloc(10)) != NULL) ...
```

```
if((fp = fopen(filename, "r")) == NULL) ...
```

```
while((c = getc(fp)) != EOF) ...
```

A: The first calls malloc, assigns the result to p, and does

something if the just-assigned result is not NULL. The second calls `fopen`, assigns the result to `fp`, and does something if the just-assigned result is NULL. The third repeatedly calls `getc`, assigns the results in turn to `c`, and does something as long as each just-assigned result is not EOF.

1.38: What's the difference between these two statements?

```
++i;
i++;
```

A: There is no difference. The only difference between the prefix and postfix forms of the autoincrement operator is the value passed on to the surrounding expression, but since the expressions in the question stand alone as expression statements, the value is discarded, and each expression merely serves to increment `i`.

1.39: Why might a compiler warn about conversions or assignments from `char *` to `int *` ?

A: In general, compilers complain about assignments between pointers of different types (and are required by the Standard to so complain) because such assignments do not make sense. A pointer to type `T1` is supposed to point to objects of type `T1`, and presumably the only reason for assigning the pointer to a pointer of a different type, say `pointer-to-T2`, would be to try to access the pointed-to object as a value of type `T2`, but if the pointed-to object is of type `T2`, why were we pointing at it with a `pointer-to-T1` in the first place?

In the particular example cited in the question, the warning also implies the possibility of unaligned access. For example, this code:

```
int a[2] = {0, 1};
char *p = &a;           /* suspicious */
int *ip = p + 1;       /* even more suspicious */

printf("%d\n", *ip);
```

is likely to crash (perhaps with a "Bus Error") because the programmer has contrived to make `ip` point to an odd, unaligned address.

When it is desired to use pointers of the "wrong" type, explicit casts must generally be used. One class of

exceptions is exemplified by malloc: the memory it allocates, and hence the pointers it returns, are supposed to be usable as any type the programmer wishes, so malloc's return value will almost always be the "wrong" type. To avoid the need for so much explicit, dangerous casting, ANSI invented the void \* type, which quietly interconverts (i.e. without warning) between other pointer types. Pointers of type void \* are therefore used as containers to hold "generic" pointers which are known to be safely usable as pointers to other, more specific types.

1.40: When do ANSI function prototype declarations \*not\* provide argument type checking, or implicit conversions?

A: In the variable-length part of variable-length argument lists, and (perhaps obviously, perhaps not) when no prototype is in scope at all. The point is that it is not safe to assume that since prototypes have been invented, programmers don't have to be careful about matching function-call arguments any more. Care must still be exercised in variable-length argument lists, and if prototypes are to take care of the rest, care must be exercised to use prototypes correctly.

1.41: State the rule(s) underlying the "equivalence" of arrays and pointers in C.

A: Rule 1: When an array appears in an expression where its value is needed, the value generated is a pointer to the array's first element. Rule 2: Array-like subscripts (integer expressions in brackets) may be used to subscript pointers as well as arrays; the expression p[i] is by definition equivalent to \*(p+i). (Actually, by rule 1, subscripts \*always\* find themselves applied to pointers, never arrays.)

1.42: What's the difference between these two declarations?

```
extern int f2(char []);
extern int f1(char *);
```

A: There is no difference. The compiler always quietly rewrites function declarations so that any array parameters are actually declared as pointers, because (by the equivalence of arrays and pointers) a pointer is what the function will actually receive.

1.43: Rewrite the parameter declaration in

```
f(int x[5][7])
{
}
```

to explicitly show the pointer type which the compiler will assume for x.

```
A:      f(int (*x)[7])
        {
        }
```

Note that the type `int (*)[7]` is *not* the same as `int **`.

1.44: A program uses a fixed-size array, and in response to user complaints you have been asked to replace it with a dynamically-allocated "array," obtained from `malloc`. Which parts of the program will need attention? What "gotchas" must you be careful of?

A: Ideally, you will merely have to change the declaration of the array from an array to a pointer, and add one call to `malloc` (with a check for a null return, of course) to initialize the pointer to a dynamically-allocated "array." All of the code which accesses the array can remain unchanged, because expressions of the form `x[i]` are valid whether `x` is an array or a pointer.

The only thing to be careful of is that if the existing code ever used the `sizeof` operator to determine the size of the array, that determination becomes grossly invalid, because after the change, `sizeof` will return only the size of the pointer.

1.45: A program which uses a dynamically allocated array is still running into problems because the initial allocation is not always big enough. Your task is now to use `realloc` to make the "array" bigger, if need be. What must you be careful of?

A: The actual call to `realloc` is straightforward enough, to request that the base pointer now point at a larger block of memory. The problem is that the larger block of memory may be in a different place; the base pointer may move. Therefore, you must reassign not only the base pointer, but also any copies of the base pointer you may have made, and also any pointers which may have been set to point anywhere into the middle of the array. (For pointers into the array, you must in general convert them

temporarily into offsets from the base pointer, then call `realloc`, then recompute new pointers based on the offsets and the new base pointer. See also question 2.10.)

1.46: How can you use `sizeof` to determine the number of elements in an array?

A: The standard idiom is

```
sizeof(array) / sizeof(array[0])
```

(or, equivalently, `sizeof(array) / sizeof(*array)` ).

1.47: When `sizeof` doesn't work (when the array is declared extern, or is a parameter to a function), what are some strategies for determining the size of an array?

A: Use a sentinel value as the last element of the array; pass the size around in a separate variable or as a separate function parameter; use a preprocessor macro to define the size.

1.48: Why might explicit casts on `malloc`'s return value, as in

```
int *ip = (int *)malloc(10 * sizeof(int));
```

be a bad idea?

A: Although such casts used to be required (before the `void *` type, which converts quietly and implicitly, was invented), they can now be considered poor style, because they will probably muzzle the compiler's attempts to warn you on those occasions when you forget to `#include <stdlib.h>` or otherwise declare `malloc`, such that `malloc` will be incorrectly assumed to be a function returning `int`.

1.49: How are Boolean true/false values defined in C? What values can the `==` (and other logical and comparison operators) yield?

A: The value 0 is considered "false," and any nonzero value is considered "true." The relational and logical operators all yield 0 for false, 1 for true.

1.50: `x` is an integer, having some value. What is the value of the expression

```
0 <= !x && !!x < 2
```

?

A: 1.

1.51: In your opinion, is it acceptable for a header file to contain `#include` directives for other header files?

A: The argument in favor of "nested `#include` files" is that they allow each header to arrange to have any subsidiary definitions, upon which its own definitions depend, made automatically. (For example, a file containing a prototype for a function that accepts an argument of type `FILE *` could `#include <stdio.h>` to define `FILE`.) The alternative is to potentially require everyone who includes a particular header file to include one or several others first, or risk cryptic errors.

The argument against is that nested headers can be confusing, can make definitions difficult to find, and can in some circumstances even make it difficult to determine which file(s) is/are being included.

1.52: How can a header file be protected against being included multiple times (perhaps due to nested `#include` directives)?

A: The standard trick is to place lines like

```
#ifndef headerfilename_H
#define headerfilename_H
```

at the beginning of the file, and an extra `#endif` at the end.

1.53: A source file contains as its first two lines:

```
#include "a.h"
int i;
```

The compiler complains about an invalid declaration on line 2. What's probably happening?

A: It's likely that the last declaration in `a.h` is missing its trailing semicolon, causing that declaration to merge into `"int i"`, with meaningless results. (That is, the merged declaration is probably something along the lines of

```
extern int f() int i;
or
```

```
    struct x { int y; } int i;  
.)
```

1.54: What's the difference between a header file and a library?

A: A header file typically contains declarations and definitions, but it never contains executable code. (A header file arguably shouldn't even contain any function bodies which would compile into executable code.) A library, on the other hand, contains only compiled, executable code and data.

A third-party library is often delivered as a library and a header file. Both pieces are important. The header file is included during compilation, and the library is included during linking.

1.55: What are the acceptable declaration(s) for main()?

A: The most common declarations, all legal, are:

```
main()  
int main()  
int main(void)  
int main(int argc, char **argv)  
int main(int argc, char *argv[])  
int main(argc, argv) int argc; char *argv[];
```

(Basically: the return type must be an implicit or explicit int; the parameter list must either be empty, or void, or one int plus one array of strings; and the function may be declared using either old-style or prototyped syntax. The actual names of the two parameters are arbitrary, although of course argc and argv are traditional.)

1.56: You wish to use ANSI function prototypes to guard against errors due to accidentally calling functions with incorrect arguments. Where should you place the prototype declarations? How can you ensure that the prototypes will be maximally effective?

A: The prototype for a global function should be placed in a header file, and the header file should be included in all files where the function is called, \*and\* in the file where the function is defined. The prototype for a static function should be placed at the top of the file where the function is defined.

Since following these rules is only slightly less hard than getting all function calls right by hand (i.e. without the aid of prototypes), the compiler should be configured to warn about functions called without prototypes in scope, \*and\* about functions defined without prototypes in scope.

1.57: Why must the variable used to hold getchar's return value be declared as int?

A: Because getchar can return, besides all values of type char, the additional "out of band" value EOF, and there obviously isn't room in a variable of type char to hold one more than the number of values which can be unambiguously stored in a variable of type char.

1.58: You must write code to read and write "binary" data files. How do you proceed? How will you actually open, read, and write the files?

A: When calling fopen, the files must be opened using the b modifier (e.g. "rb", "wb"). Binary data files are generally read and written a byte at a time using getc and putc, or a data structure at a time using fread and fwrite.

1.59: Write the function

```
void error(const char *message, ...);
```

which accepts a message string, possibly containing % sequences, along with optional extra arguments corresponding to the % sequences, and prints the string "error: ", followed by the message as printf would print it, followed by a newline, all to stderr.

```
A: #include <stdio.h>
#include <stdarg.h>

void error(char *fmt, ...)
{
    va_list argp;
    fprintf(stderr, "error: ");
    va_start(argp, fmt);
    vfprintf(stderr, fmt, argp);
    va_end(argp);
    fprintf(stderr, "\n");
}
```

1.60: Write the function

```
char *vstrcat(char *, ...);
```

which accepts a variable number of strings and concatenates them all together into a block of malloc'ed memory just big enough for the result. The end of the list of strings will be indicated with a null pointer. For example, the call

```
char *p = vstrcat("Hello, ", "world!", (char *)NULL);
```

should return the string "Hello, world!".

```
A:      #include <stdlib.h>          /* for malloc, NULL,
size_t */
        #include <stdarg.h>        /* for va_ stuff */
        #include <string.h>        /* for strcat et al. */

char *vstrcat(char *first, ...)
{
    size_t len;
    char *retbuf;
    va_list argp;
    char *p;

    if(first == NULL)
        return NULL;

    len = strlen(first);

    va_start(argp, first);

    while((p = va_arg(argp, char *)) != NULL)
        len += strlen(p);

    va_end(argp);

    retbuf = malloc(len + 1); /* +1 for
trailing \0 */

    if(retbuf == NULL)
        return NULL; /* error */

    (void)strcpy(retbuf, first);

    va_start(argp, first); /* restart; 2nd
scan */

    while((p = va_arg(argp, char *)) != NULL)
        (void)strcat(retbuf, p);
```

```

        va_end(argp);

        return retbuf;
    }

```

1.61: Write a stripped-down version of printf which accepts only the %c, %d, %o, %s, %x, and %% format specifiers. (Do not worry about width, precision, flags, or length modifiers.)

A: [Although this question is obviously supposed to test one's familiarity with the va\_ macros, a significant nuisance in composing a working answer is performing the sub-task of converting integers to digit strings. For some reason, back when I composed this test, I felt it appropriate to defer that task to an "itoa" function; perhaps I had just presented an implementation of itoa to the same class for whom I first prepared this test.]

```

#include <stdio.h>
#include <stdarg.h>

extern char *itoa(int, char *, int);

myprintf(const char *fmt, ...)
{
    const char *p;
    va_list argp;
    int i;
    char *s;
    char fmtbuf[256];

    va_start(argp, fmt);

    for(p = fmt; *p != '\0'; p++)
    {
        if(*p != '%')
        {
            putchar(*p);
            continue;
        }

        switch(*++p)
        {
            case 'c':
                i = va_arg(argp, int);
                putchar(i);

```

```

        break;

    case 'd':
        i = va_arg(argp, int);
        s = itoa(i, fmtbuf, 10);
        fputs(s, stdout);
        break;

    case 's':
        s = va_arg(argp, char *);
        fputs(s, stdout);
        break;

    case 'x':
        i = va_arg(argp, int);
        s = itoa(i, fmtbuf, 16);
        fputs(s, stdout);
        break;

    case '%':
        putchar('%');
        break;
    }
}

va_end(argp);
}

```

1.62: You are to write a program which accepts single keystrokes from the user, without waiting for the RETURN key. You are to restrict yourself only to features guaranteed by the ANSI/ISO C Standard. How do you proceed?

A: You proceed by pondering the sorrow of your fate, and perhaps by complaining to your boss/professor/psychologist that you've been given an impossible task. There is no ANSI Standard function for reading one keystroke from the user without waiting for the RETURN key. You'll have to use facilities specific to your operating system; you won't be able to write the code strictly portably.

1.63: [POOR QUESTION] How do you convert an integer to binary or hexadecimal?

A: The question is poor because an integer is a *\*number\**; it doesn't make much sense to ask what base it's in. If I'm holding eleven apples, what base is that in?

holding eleven apples, what base is that in.

(Of course, internal to the computer, an integer is almost certainly represented in binary, although it's not at all unreasonable to think of it as being hexadecimal, or decimal for that matter.)

The only time the base of a number matters is when it's being read from or written to the outside world as a string of digits. In those cases, and depending on just what you're doing, you can specify the base by picking the correct printf or scanf format specifier (%d, %o, or %x), or by picking the third argument to strtol. (There isn't a Standard function to convert an integer to a string using an arbitrary base. For that task, it's a straightforward exercise to write a function to do the conversion. Some versions of the nonstandard itoa function also accept a base or radix argument.)

1.64: You're trying to discover the sizes of the basic types under a certain compiler. You write the code

```
printf("sizeof(char) = %d\n", sizeof(char));
printf("sizeof(short) = %d\n", sizeof(short));
printf("sizeof(int) = %d\n", sizeof(int));
printf("sizeof(long) = %d\n", sizeof(long));
```

However, all four values are printed as 0. What have you learned?

A: You've learned that this compiler defines `size_t`, the type returned by `sizeof`, as an unsigned long int, and that

the compiler also defines long integers as having a larger size than plain int. (Furthermore, you've learned that the machine probably uses big-endian byte order.)

Finally, you may have learned that the code you should have written is along the lines of either

```
printf("sizeof(int) = %u\n", (unsigned)sizeof(int));
or
printf("sizeof(int) = %lu\n", (unsigned
long)sizeof(int));
```

Section 2. What's wrong with...?

```
2.1: main(int argc, char *argv[])
{
    ...
    if(argv[1] == "-v")
```

```
++(argv[i])
```

...

A: Applied to pointers, the == operator compares only whether the pointers are equal. To compare whether the strings are equal, you'll have to call strcmp.

2.2:       a ^= b ^= a ^= b

(What is the expression trying to do?)

A: The expression is undefined because it modifies the variable a twice between sequence points. What it's trying to do is swap the variables a and b using a hoary old assembly programmer's trick.

2.3:       char\* p1, p2;

A: p2 will be declared as type char, \*not\* pointer-to-char.

2.4:       char c;  
          while((c = getchar()) != EOF)

...

A: The variable used to contain getchar's return value must be declared as int if EOF is to be reliably detected.

2.5:       while(c = getchar() != EOF)

...

A: Parentheses are missing; the code will call getchar, compare the result to EOF, assign the result \*of the comparison\* to c, and take another trip around the loop if the condition was true (i.e. if the character read was not EOF). (The net result will be that the input will be read as if it were a string of nothing but the character '\001'. The loop would still halt properly on EOF, however.)

2.6:       int i, a[10];  
          for(i = 0; i <= 10; i++)  
              a[i] = 0;

A: The loop assigns to the nonexistent eleventh value of the array, a[10], because it uses a loop continuation condition of <= 10 instead of < 10 or <= 9.

2.7:       #include <ctype.h>  
          ...  
          #define TRUE 1

```

#define FALSE 0
...
if(isalpha(c) == TRUE)
    ...

```

A: Since *any* nonzero value is considered "true" in C; it's rarely if ever a good idea to compare explicitly against a single TRUE value (or FALSE, for that matter). In particular, the <ctype.h> macros, including `isalpha()`, tend to return nonzero values other than 1, so the test as written is likely to fail even for alphabetic characters. The correct test is simply

```

if(isalpha(c))

```

2.8: `printf("%d\n", sizeof(int));`

A: The `sizeof` operator returns type `size_t`, which is an unsigned integral type *not* necessarily the same size as an `int`. The correct code is either

```

printf("sizeof(int) = %u\n", (unsigned
int)sizeof(int));
or
printf("sizeof(int) = %lu\n", (unsigned long
int)sizeof(int));

```

2.9: `p = realloc(p, newsize);`  
`if(p == NULL)`  
`{`  
`fprintf(stderr, "out of memory\n");`  
`return;`  
`}`

A: If `realloc` returns null, and assuming `p` used to point to some `malloc`'ed memory, the memory remains allocated, although having overwritten `p`, there may well be no way to use or free the memory. The code is a potential memory leak.

```

2.10: /* p points to a block of memory obtained from
malloc; */
/* p2 points somewhere within that block */

newp = realloc(p, newsize);
if(newp != NULL && newp != p)
{
  int offset = newp - p;
  p2 += offset;
}

```

,

A: Pointer subtraction is well-defined only for pointers into the same block of memory. If `realloc` moves the block of memory while changing its size, which is the very case the code is trying to test for, then the subtraction `newp - p` is invalid, because `p` and `newp` do not point into the same block of memory. (The subtraction could overflow or otherwise produce nonsensical results, especially on segmented architectures. Strictly speaking, *any* use of the old value of `p` after `realloc` moves the block is invalid, even comparing it to `newp`.)

The correct way to relocate `p2` within the possibly-moved block, correcting *all* the problems in the original (including a subtle one not mentioned), is:

```
ptrdiff_t offset = p2 - p;
newp = realloc(p, newsize);
if(newp != NULL)
{
    p = newp;
    p2 = p + offset;
}
```

```
2.11:    int a[10], b[10];
        ...
        a = b;
```

A: You can't assign arrays.

```
2.12:    int i = 0;
        char *p = i;
        if(p == 0)
        ...
```

A: Assigning an integer 0 to a pointer does not reliably result in a null pointer. (You must use a *constant* 0.)