Welcome to C Frequently Asked(Unanswerd) Questions

Online material converted to word and pdf format by Harinath
Mail your opinions and queries to harinathreddy@sify.com

FA(U)Q (Frequently Asked (Unanswered) Questions

Declarations and Initializations

1.1: How do you decide which integer type to use?

A: If you might need large values (above 32,767 or below -32,767), use long. Otherwise, if space is very important (i.e. if there are large arrays or many structures), use short. Otherwise, use int. If well-defined overflow characteristics are important and negative values are not, or if you want to steer clear of sign- extension problems when manipulating bits or bytes, use one of the corresponding unsigned types. (Beware when mixing signed and unsigned values in expressions, though.)

Although character types (especially unsigned char) can be used as "tiny" integers, doing so is sometimes more trouble than it's worth, due to unpredictable sign extension and increased code size. (Using unsigned char can help; see question 12.1 for a related problem.)

A similar space/time tradeoff applies when deciding between float and double. None of the above rules apply if the address of a variable is taken and must have a particular type.

If for some reason you need to declare something with an *exact* size (usually the only good reason for doing so is when attempting to conform to some externally-imposed storage layout, but see question 20.5), be sure to encapsulate the choice behind an appropriate typedef.

References: K&R1 Sec. 2.2 p. 34; K&R2 Sec. 2.2 p. 36, Sec. A4.2 pp. 195-6, Sec. B11 p. 257; ANSI Sec. 2.2.4.2.1, Sec. 3.1.2.5; ISO Sec. 5.2.4.2.1, Sec. 6.1.2.5; H&S Secs. 5.1,5.2 pp. 110-114.

1.4: What should the 64-bit type on new, 64-bit machines be?

A: Some vendors of C products for 64-bit machines support 64-bit long ints. Others fear that too much existing code is written to assume that ints and longs are the same size, or that one or the other of them is exactly 32 bits, and introduce a new, nonstandard, 64-bit long long (or __longlong) type instead.

Programmers interested in writing portable code should therefore insulate their 64-bit type needs behind appropriate typedefs. Vendors who feel compelled to introduce a new, longer integral type should advertise it as being "at least 64 bits" (which is truly new, a type traditional C does not have), and not "exactly 64 bits."

References: ANSI Sec. F.5.6; ISO Sec. G.5.6.

1.7: What's the best way to declare and define global variables?

A: First, though there can be many "declarations" (and in many translation units) of a single "global" (strictly speaking, "external") variable or function, there must be exactly one "definition". (The definition is the declaration that actually allocates space, and provides an initialization value, if any.) The best arrangement is to place each definition in some relevant .c file, with an external declaration in a header (".h") file, which is #included wherever the declaration is needed. The .c file containing the definition should also #include the same header file, so that the compiler can check that the definition matches the declarations.

This rule promotes a high degree of portability: it is consistent with the requirements of the ANSI C Standard, and is also consistent with most pre-ANSI compilers and linkers. (Unix compilers and linkers typically use a "common model" which allows multiple definitions, as long as at most one is initialized; this behavior is mentioned as a "common extension" by the ANSI Standard, no pun intended. A few very odd systems may require an explicit initializer to distinguish a definition from an external declaration.)

It is possible to use preprocessor tricks to arrange that a line like

```
DEFINE(int, i);
```

need only be entered once in one header file, and turned into a definition or a declaration depending on the setting of some macro, but it's not clear if this is worth the trouble.

It's especially important to put global declarations in header files if you want the compiler to catch inconsistent declarations for you. In particular, never place a prototype for an external function in a .c file: it wouldn't generally be checked for consistency with the definition, and an incompatible prototype is worse than useless.

See also questions 10.6 and 18.8.

References: K&R1 Sec. 4.5 pp. 76-7; K&R2 Sec. 4.4 pp. 80-1; ANSI Sec. 3.1.2.2, Sec. 3.7, Sec. 3.7.2, Sec. F.5.11; ISO Sec. 6.1.2.2, Sec. 6.7, Sec. 6.7.2, Sec. G.5.11; Rationale Sec. 3.1.2.2; H&S Sec. 4.8 pp. 101-104, Sec. 9.2.3 p. 267; CT&P Sec. 4.2 pp. 54-56.

1.11: What does extern mean in a function declaration?

A: It can be used as a stylistic hint to indicate that the function's definition is probably in another source file, but there is no formal difference between

```
extern int f();
```

and

```
int f();
```

References: ANSI Sec. 3.1.2.2, Sec. 3.5.1; ISO Sec. 6.1.2.2, Sec. 6.5.1; Rationale Sec. 3.1.2.2; H&S Secs. 4.3,4.3.1 pp. 75- 6.

1.12: What's the auto keyword good for?

A: Nothing; it's archaic. See also question 20.37.

References: K&R1 Sec. A8.1 p. 193; ANSI Sec. 3.1.2.4, Sec. 3.5.1; ISO Sec. 6.1.2.4, Sec. 6.5.1; H&S Sec. 4.3 p. 75, Sec. 4.3.1 p. 76.

1.14: I can't seem to define a linked list successfully. I tried

```
typedef struct {
        char *item;
        NODEPTR next;
} *NODEPTR;
```

but the compiler gave me error messages. Can't a structure in C contain a pointer to itself?

A: Structures in C can certainly contain pointers to themselves; the discussion and example in section 6.5 of K&R make this clear. The problem with the NODEPTR example is that the typedef has not been defined at the point where the "next" field is declared. To fix this code, first give the structure a tag ("struct node"). Then, declare the "next" field as a simple "struct node *", or disentangle the typedef declaration from the structure definition, or both. One corrected version would be

```
struct node {
        char *item;
        struct node *next;
};

typedef struct node *NODEPTR;
```

and there are at least three other equivalently correct ways of arranging it.

A similar problem, with a similar solution, can arise when attempting to declare a pair of typedef'ed mutually referential structures.

See also question 2.1.

References: K&R1 Sec. 6.5 p. 101; K&R2 Sec. 6.5 p. 139; ANSI Sec. 3.5.2, Sec. 3.5.2.3, esp. examples; ISO Sec. 6.5.2, Sec. 6.5.2.3; H&S Sec. 5.6.1 pp. 132-3.

1.21: How do I declare an array of N pointers to functions returning pointers to functions returning pointers to characters?

A: The first part of this question can be answered in at least three ways:

1. char *(*(*a[N])())();

2. Build the declaration up incrementally, using typedefs:

```
typedef char *pc;       /* pointer to char */
typedef pc fpc();       /* function returning pointer
to char */
typedef fpc *pfpc;      /* pointer to above */
typedef pfpc fpfpc();   /* function returning... */
typedef fpfpc *pfpfpc;  /* pointer to... */
pfpfpc a[N];            /* array of... */
```

3. Use the cdecl program, which turns English into C and vice versa:

```
cdecl> declare a as array of pointer to function
returning
        pointer to function returning pointer to char
char *(*(*a[])())()
```

cdecl can also explain complicated declarations, help with casts, and indicate which set of parentheses the arguments go in (for complicated function definitions, like the one above). Versions of cdecl are in volume 14 of comp.sources.unix (see question 18.16) and K&R2.

Any good book on C should explain how to read these complicated C declarations "inside out" to understand them ("declaration mimics use").

The pointer-to-function declarations in the examples above have not included parameter type information. When the parameters have complicated types, declarations can *really* get messy. (Modern versions of cdecl can help here, too.)

References: K&R2 Sec. 5.12 p. 122; ANSI Sec. 3.5ff (esp. Sec. 3.5.4); ISO Sec. 6.5ff (esp. Sec. 6.5.4); H&S Sec. 4.5 pp. 85-92, Sec. 5.10.1 pp. 149-50.

1.22: How can I declare a function that can return a pointer to a function of the same type? I'm building a state machine with one function for each state, each of which returns a pointer to the function for the next state. But I can't find a way to declare the functions.

A: You can't quite do it directly. Either have the function return a generic function pointer, with some judicious casts to adjust the types as the pointers are passed around; or have it return a structure containing only a pointer to a function returning that structure.

1.25: My compiler is complaining about an invalid redeclaration of a function, but I only define it once and call it once.

A: Functions which are called without a declaration in scope (perhaps because the first call precedes the function's definition) are assumed to be declared as returning int (and without any argument type information), leading to discrepancies if the function is later declared or defined otherwise. Non-int functions must be declared before they are called.

Another possible source of this problem is that the function has the same name as another one declared in some header file.

See also questions 11.3 and 15.1.

References: K&R1 Sec. 4.2 p. 70; K&R2 Sec. 4.2 p. 72; ANSI Sec. 3.3.2.2; ISO Sec. 6.3.2.2; H&S Sec. 4.7 p. 101.

1.30: What can I safely assume about the initial values of variables which are not explicitly initialized? If global variables start out as "zero," is that good enough for null pointers and floating-point zeroes?

A: Variables with "static" duration (that is, those declared outside of functions, and those declared with the storage class static), are guaranteed initialized (just once, at program startup) to zero, as if the programmer had typed "= 0". Therefore, such variables are initialized to the null pointer (of the correct type; see also section 5) if they are pointers, and to 0.0 if they are floating-point.

Variables with "automatic" duration (i.e. local variables without the static storage class) start out containing garbage, unless they are explicitly initialized. (Nothing useful can be predicted about the garbage.)

Dynamically-allocated memory obtained with malloc() and realloc() is also likely to contain garbage, and must be initialized by the calling program, as appropriate. Memory obtained with calloc() is all-bits-0, but this is not necessarily useful for pointer or floating-point values (see question 7.31, and section 5).

References: K&R1 Sec. 4.9 pp. 82-4; K&R2 Sec. 4.9 pp. 85-86; ANSI Sec. 3.5.7, Sec. 4.10.3.1, Sec. 4.10.5.3; ISO Sec. 6.5.7, Sec. 7.10.3.1, Sec. 7.10.5.3; H&S Sec. 4.2.8 pp. 72-3, Sec. 4.6 pp. 92-3, Sec. 4.6.2 pp. 94-5, Sec. 4.6.3 p. 96, Sec. 16.1 p. 386.

1.31: This code, straight out of a book, isn't compiling:

```
f()
{
        char a[] = "Hello, world!";
}
```

A: Perhaps you have a pre-ANSI compiler, which doesn't allow initialization of "automatic aggregates" (i.e. non-static local arrays, structures, and unions). As a workaround, you can make the array global or static (if you won't need a fresh copy during any subsequent calls), or replace it with a pointer (if the array won't be written to). (You can always initialize local char * variables to point to string literals, but see question 1.32 below.) If neither of these conditions hold, you'll have to initialize the array by hand with strcpy() when f() is called. See also question 11.29.

1.32: What is the difference between these initializations?

```
char a[] = "string literal";
char *p  = "string literal";
```

My program crashes if I try to assign a new value to p[i].

A: A string literal can be used in two slightly different ways. As an array initializer (as in the declaration of char a[]), it specifies the initial values of the characters in that array. Anywhere else, it turns into an unnamed, static array of characters, which may be stored in read-only memory, which is why you can't safely modify it. In an expression context, the array is converted at once to a pointer, as usual (see section 6), so the second declaration initializes p to point to the unnamed array's first element.

(For compiling old code, some compilers have a switch controlling whether strings are writable or not.)

See also questions 1.31, 6.1, 6.2, and 6.8.

References: K&R2 Sec. 5.5 p. 104; ANSI Sec. 3.1.4, Sec. 3.5.7; ISO Sec. 6.1.4, Sec. 6.5.7; Rationale Sec. 3.1.4; H&S Sec. 2.7.4 pp. 31-2.

1.34: I finally figured out the syntax for declaring pointers to functions, but now how do I initialize one?

A: Use something like

```
extern int func();
int (*fp)() = func;
```

When the name of a function appears in an expression like this, it "decays" into a pointer (that is, it has its address implicitly taken), much as an array name does.

An explicit declaration for the function is normally needed, since implicit external function declaration does not happen in this case (because the function name in the initialization is not part of a function call).

See also question 4.12.

Structures, Unions and Enumerations

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

```
struct x1 { ... };
typedef struct { ... } x2;
```

A: The first form declares a "structure tag"; the second declares a "typedef". The main difference is that the second declaration is of a slightly more abstract type -- its users don't necessarily know that it is a structure, and the keyword struct is not used when declaring instances of it.

2.2: Why doesn't

```
struct x { ... };
x thestruct;
```

work?

A: C is not C++. Typedef names are not automatically generated for structure tags. See also question 2.1 above.

2.3: Can a structure contain a pointer to itself?

A: Most certainly. See question 1.14.

2.4: What's the best way of implementing opaque (abstract) data types in C?

A: One good way is for clients to use structure pointers (perhaps additionally hidden behind typedefs) which point to structure types which are not publicly defined.

2.6: I came across some code that declared a structure like this:

```
struct name {
        int namelen;
        char namestr[1];
};
```

and then did some tricky allocation to make the namestr array act like it had several elements. Is this legal or portable?

A: This technique is popular, although Dennis Ritchie has called it "unwarranted chumminess with the C implementation." An official interpretation has deemed that it is not strictly conforming with the C Standard. (A thorough treatment of the arguments surrounding the legality of the technique is beyond the scope of this list.) It does seem to be portable to all known implementations. (Compilers which check array bounds carefully might issue warnings.)

Another possibility is to declare the variable-size element very large, rather than very small; in the case of the above example:

```
...
char namestr[MAXSIZE];
...
```

where MAXSIZE is larger than any name which will be stored. However, it looks like this technique is disallowed by a strict interpretation of the Standard as well.

References: Rationale Sec. 3.5.4.2.

2.7: I heard that structures could be assigned to variables and passed to and from functions, but K&R1 says not.

A: What K&R1 said was that the restrictions on structure operations would be lifted in a forthcoming version of the compiler, and in fact structure assignment and passing were fully functional in Ritchie's compiler even as K&R1 was being published. Although a few early C compilers lacked these operations, all modern compilers support them, and they are part of the ANSI C standard, so there should be no reluctance to use them.

(Note that when a structure is assigned, passed, or returned, the copying is done monolithically; anything pointed to by any pointer fields is *not* copied.)

References: K&R1 Sec. 6.2 p. 121; K&R2 Sec. 6.2 p. 129; ANSI Sec. 3.1.2.5, Sec. 3.2.2.1, Sec. 3.3.16; ISO Sec. 6.1.2.5, Sec. 6.2.2.1, Sec. 6.3.16; H&S Sec. 5.6.2 p. 133.

2.8: Why can't you compare structures?

A: There is no single, good way for a compiler to implement structure comparison which is consistent with C's low-level flavor. A simple byte-by-byte comparison could founder on random bits present in unused "holes" in the structure (such padding is used to keep the alignment of later fields correct; see question 2.12). A field-by-field comparison might require unacceptable amounts of repetitive code for large structures.

If you need to compare two structures, you'll have to write your own function to do so, field by field.

References: K&R2 Sec. 6.2 p. 129; ANSI Sec. 4.11.4.1 footnote 136; Rationale Sec. 3.3.9; H&S Sec. 5.6.2 p. 133.

2.9: How are structure passing and returning implemented?

A: When structures are passed as arguments to functions, the entire structure is typically pushed on the stack, using as many words as are required. (Programmers often choose to use pointers to structures instead, precisely to avoid this overhead.) Some compilers merely pass a pointer to the structure, though they may have to make a local copy to preserve pass-by-value semantics.

Structures are often returned from functions in a location pointed to by an extra, compiler-supplied "hidden" argument to the function. Some older compilers used a special, static location for structure returns, although this made structure- valued functions non-reentrant, which ANSI C disallows.

References: ANSI Sec. 2.2.3; ISO Sec. 5.2.3.

2.10: How can I pass constant values to functions which accept structure arguments?

A: C has no way of generating anonymous structure values. You will have to use a temporary structure variable or a little structure- building function. (gcc provides structure constants as an extension, and the mechanism will probably be added to a future revision of the C Standard.) See also question 4.10.

2.11: How can I read/write structures from/to data files?

A: It is relatively straightforward to write a structure out using fwrite():

```
fwrite(&somestruct, sizeof somestruct, 1, fp);
```

and a corresponding fread invocation can read it back in. (Under pre-ANSI C, a (char *) cast on the first argument is required. What's important is that fwrite() receive a byte pointer, not a structure pointer.) However, data files so written will *not* be portable (see questions 2.12 and 20.5). Note also that if the structure contains any pointers, only the pointer values will be written, and they are most unlikely to be valid when read back in. Finally, note that for widespread portability you must use the "b" flag when fopening the files; see question 12.38.

A more portable solution, though it's a bit more work initially, is to write a pair of functions for writing and reading a structure, field-by-field, in a portable (perhaps even human- readable) way.

References: H&S Sec. 15.13 p. 381.

2.12: My compiler is leaving holes in structures, which is wasting space and preventing "binary" I/O to external data files. Can I turn off the padding, or otherwise control the alignment of structure fields?

A: Your compiler may provide an extension to give you this control (perhaps a #pragma; see question 11.20), but there is no standard method.

See also question 20.5.

References: K&R2 Sec. 6.4 p. 138; H&S Sec. 5.6.4 p. 135.

2.13: Why does sizeof report a larger size than I expect for a structure type, as if there were padding at the end?

A: Structures may have this padding (as well as internal padding), if necessary, to ensure that alignment properties will be preserved when an array of contiguous structures is allocated. Even when the structure is not part of an array, the end padding remains, so that sizeof can always return a consistent size. See question 2.12 above.

References: H&S Sec. 5.6.7 pp. 139-40.

2.14: How can I determine the byte offset of a field within a structure?

A: ANSI C defines the offsetof() macro, which should be used if available; see . If you don't have it, one possible implementation is

```
#define offsetof(type, mem) ((size_t) \
```

```
                        ((char *)&((type *)0)->mem - (char *)(type
*)0))
```

This implementation is not 100% portable; some compilers may legitimately refuse to accept it.

See question 2.15 below for a usage hint.

References: ANSI Sec. 4.1.5; ISO Sec. 7.1.6; Rationale Sec. 3.5.4.2; H&S Sec. 11.1 pp. 292-3.

2.15: How can I access structure fields by name at run time?

A: Build a table of names and offsets, using the offsetof() macro. The offset of field b in struct a is

```
offsetb = offsetof(struct a, b)
```

If structp is a pointer to an instance of this structure, and field b is an int (with offset as computed above), b's value can be set indirectly with

```
*(int *)((char *)structp + offsetb) = value;
```

2.18: This program works correctly, but it dumps core after it finishes. Why?

```
struct list {
        char *item;
        struct list *next;
}

/* Here is the main program. */

main(argc, argv)
{ ... }
```

A: A missing semicolon causes main() to be declared as returning a structure. (The connection is hard to see because of the intervening comment.) Since structure-valued functions are usually implemented by adding a hidden return pointer (see question 2.9), the generated code for main() tries to accept three arguments, although only two are passed (in this case, by the C start-up code). See also questions 10.9 and 16.4.

References: CT&P Sec. 2.3 pp. 21-2.

2.20: Can I initialize unions?

A: ANSI Standard C allows an initializer for the first member of a union. There is no standard way of initializing any other member (nor, under a pre-ANSI compiler, is there generally any way of initializing a union at all).

References: K&R2 Sec. 6.8 pp. 148-9; ANSI Sec. 3.5.7; ISO Sec. 6.5.7; H&S Sec. 4.6.7 p. 100.

2.22: What is the difference between an enumeration and a set of preprocessor #defines?

A: At the present time, there is little difference. Although many people might have wished otherwise, the C Standard says that enumerations may be freely intermixed with other integral types, without errors. (If such intermixing were disallowed without explicit casts, judicious use of enumerations could catch certain programming errors.)

Some advantages of enumerations are that the numeric values are automatically assigned, that a debugger may be able to display the symbolic values when enumeration variables are examined, and that they obey block scope. (A compiler may also generate nonfatal warnings when enumerations and integers are indiscriminately mixed, since doing so can still be considered bad style even though it is not strictly illegal.) A disadvantage is that the programmer has little control over those nonfatal warnings; some programmers also resent not having control over the sizes of enumeration variables.

References: K&R2 Sec. 2.3 p. 39, Sec. A4.2 p. 196; ANSI Sec. 3.1.2.5, Sec. 3.5.2, Sec. 3.5.2.2, Appendix E; ISO Sec. 6.1.2.5, Sec. 6.5.2, Sec. 6.5.2.2, Annex F; H&S Sec. 5.5 pp. 127-9, Sec. 5.11.2 p. 153.

2.24: Is there an easy way to print enumeration values symbolically?

A: No. You can write a little function to map an enumeration constant to a string. (If all you're worried about is debugging, a good debugger should automatically

Expressions

3.1: Why doesn't this code:

```
a[i] = i++;
```

work?

A: The subexpression i++ causes a side effect -- it modifies i's value -- which leads to undefined behavior since i is also referenced elsewhere in the same expression. (Note that although the language in K&R suggests that the behavior of this expression is unspecified, the C Standard makes the stronger statement that it is undefined -- see question 11.33.)

References: K&R1 Sec. 2.12; K&R2 Sec. 2.12; ANSI Sec. 3.3; ISO Sec. 6.3.

3.2: Under my compiler, the code

```
int i = 7;
printf("%d\n", i++ * i++);
```

prints 49. Regardless of the order of evaluation, shouldn't it print 56?

A: Although the postincrement and postdecrement operators ++ and -- perform their operations after yielding the former value, the implication of "after" is often misunderstood. It is *not* guaranteed that an increment or decrement is performed immediately after giving up the previous value and before any other part of the expression is evaluated. It is merely guaranteed that the update will be performed sometime before the expression is considered "finished" (before the next "sequence point," in ANSI C's terminology; see question 3.8). In the example, the compiler chose to multiply the previous value by itself and to perform both increments afterwards.

The behavior of code which contains multiple, ambiguous side effects has always been undefined. (Loosely speaking, by "multiple, ambiguous side effects" we mean any combination of ++, --, =, +=, -=, etc. in a single expression which causes the same object either to be modified twice or modified and then inspected. This is a rough definition; see question 3.8 for a precise one, and question 11.33 for the meaning of "undefined.") Don't even try to find out how your compiler implements such things (contrary to the ill-advised exercises in many C textbooks); as K&R wisely point out, "if you don't know *how* they are done on various machines, that innocence may help to protect you."

References: K&R1 Sec. 2.12 p. 50; K&R2 Sec. 2.12 p. 54; ANSI Sec. 3.3; ISO Sec. 6.3; CT&P Sec. 3.7 p. 47; PCS Sec. 9.5 pp. 120-1.

3.3: I've experimented with the code

```
[CENSORED]
```

on several compilers. Some gave i the value 3, some gave 4, but one gave 7. I know the behavior is undefined, but how could it give 7?

A: [I apologize for the censorship of the question, but the expression that used to be there was indecent, and by the newly-passed Communications Decency Act of the U.S., I am prohibited from transmitting "indecent" material, whatever that is. Suffice it to say that the expression tried to modify the same variable twice between sequence points. --scs]

Undefined behavior means *anything* can happen. See questions 3.9 and 11.33. (Also, note that neither i++ nor ++i is the same as i+1. If you want to increment i, use i=i+1 or i++ or ++i, not some combination. See also question 3.12.)

3.4: Can I use explicit parentheses to force the order of evaluation I want? Even if I don't, doesn't precedence dictate it?

A: Not in general.

Operator precedence and explicit parentheses impose only a partial ordering on the evaluation of an expression. In the expression

```
f() + g() * h()
```

although we know that the multiplication will happen before the addition, there is no telling which of the three functions will be called first.

When you need to ensure the order of subexpression evaluation, you may need to use explicit temporary variables and separate statements.

References: K&R1 Sec. 2.12 p. 49, Sec. A.7 p. 185; K&R2 Sec. 2.12 pp. 52-3, Sec. A.7 p. 200.

3.5: But what about the && and || operators? I see code like "while((c = getchar()) != EOF && c != '\n')" ...

A: There is a special exception for those operators (as well as the ?: operator): left-to-right evaluation is guaranteed (as is an intermediate sequence point, see question 3.8). Any book on C should make this clear.

References: K&R1 Sec. 2.6 p. 38, Secs. A7.11-12 pp. 190-1; K&R2 Sec. 2.6 p. 41, Secs. A7.14-15 pp. 207-8; ANSI Sec. 3.3.13, Sec. 3.3.14, Sec. 3.3.15; ISO Sec. 6.3.13, Sec. 6.3.14, Sec. 6.3.15; H&S Sec. 7.7 pp. 217-8, Sec. 7.8 pp. 218-20, Sec. 7.12.1 p. 229; CT&P Sec. 3.7 pp. 46-7.

3.8: How can I understand these complex expressions? What's a "sequence point"?

A: A sequence point is the point (at the end of a full expression, or at the ||, &&, ?:, or comma operators, or just before a function call) at which the dust has settled and all side effects are guaranteed to be complete. The ANSI/ISO C Standard states that

```
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.
```

The second sentence can be difficult to understand. It says that if an object is written to within a full expression, any and all accesses to it within the same expression must be for the purposes of computing the value to be written. This rule effectively constrains legal expressions to those in which the accesses demonstrably precede the modification.

See also question 3.9 below.

References: ANSI Sec. 2.1.2.3, Sec. 3.3, Appendix B; ISO Sec. 5.1.2.3, Sec. 6.3, Annex C; Rationale Sec. 2.1.2.3; H&S Sec. 7.12.1 pp. 228-9.

3.9: So given

```
a[i] = i++;
```

we don't know which cell of a[] gets written to, but i does get incremented by one.

A: *No.* Once an expression or program becomes undefined, *all* aspects of it become undefined. See questions 3.2, 3.3, 11.33, and 11.35.

3.12: If I'm not using the value of the expression, should I use i++ or ++i to increment a variable?

A: Since the two forms differ only in the value yielded, they are entirely equivalent when only their side effect is needed.

See also question 3.3.

References: K&R1 Sec. 2.8 p. 43; K&R2 Sec. 2.8 p. 47; ANSI Sec. 3.3.2.4, Sec. 3.3.3.1; ISO Sec. 6.3.2.4, Sec. 6.3.3.1; H&S Sec. 7.4.4 pp. 192-3, Sec. 7.5.8 pp. 199-200.

3.14: Why doesn't the code

```
int a = 1000, b = 1000;
long int c = a * b;
```

work?

A: Under C's integral promotion rules, the multiplication is carried out using int arithmetic, and the result may overflow or be truncated before being promoted and assigned to the long int left-hand side. Use an explicit cast to force long arithmetic:

```
long int c = (long int)a * b;
```

Note that (long int)(a * b) would *not* have the desired effect.

A similar problem can arise when two integers are divided, with the result assigned to a floating-point variable.

References: K&R1 Sec. 2.7 p. 41; K&R2 Sec. 2.7 p. 44; ANSI Sec. 3.2.1.5; ISO Sec. 6.2.1.5; H&S Sec. 6.3.4 p. 176; CT&P Sec. 3.9 pp. 49-50.

3.16: I have a complicated expression which I have to assign to one of two variables, depending on a condition. Can I use code like this?

```
((condition) ? a : b) = complicated_expression;
```

A: No. The ?: operator, like most operators, yields a value, and you can't assign to a value. (In other words, ?: does not yield an "lvalue".) If you really want to, you can try something like

```
*((condition) ? &a : &b) = complicated_expression;
```

although this is admittedly not as pretty.

References: ANSI Sec. 3.3.15 esp. footnote 50; ISO Sec. 6.3.15; H&S Sec. 7.1 pp. 179-180.

Pointers

4.2: I'm trying to declare a pointer and allocate some space for it, but it's not working. What's wrong with this code?

```
char *p;
*p = malloc(10);
```

A: The pointer you declared is p, not *p. To make a pointer point somewhere, you just use the name of the pointer:

```
p = malloc(10);
```

It's when you're manipulating the pointed-to memory that you use * as an indirection operator:

```
*p = 'H';
```

See also questions 1.21, 7.1, and 8.3.

References: CT&P Sec. 3.1 p. 28.

4.3: Does *p++ increment p, or what it points to?

A: Unary operators like *, ++, and -- all associate (group) from right to left. Therefore, *p++ increments p (and returns the value pointed to by p before the increment). To increment the value pointed to by p, use (*p)++ (or perhaps ++*p, if the order of the side effect doesn't matter).

References: K&R1 Sec. 5.1 p. 91; K&R2 Sec. 5.1 p. 95; ANSI Sec. 3.3.2, Sec. 3.3.3; ISO Sec. 6.3.2, Sec. 6.3.3; H&S Sec. 7.4.4 pp. 192-3, Sec. 7.5 p. 193, Secs. 7.5.7,7.5.8 pp. 199- 200.

4.5: I have a char * pointer that happens to point to some ints, and I want to step it over them. Why doesn't

```
((int *)p)++;
```

work?

A: In C, a cast operator does not mean "pretend these bits have a different type, and treat them accordingly"; it is a conversion operator, and by definition it yields an rvalue, which cannot be assigned to, or incremented with ++. (It is an anomaly in pcc- derived compilers, and an extension in gcc, that expressions such as the above are ever accepted.) Say what you mean: use

```
p = (char *)((int *)p + 1);
```

or (since p is a char *) simply

```
p += sizeof(int);
```

Whenever possible, you should choose appropriate pointer types in the first place, instead of trying to treat one type as another.

References: K&R2 Sec. A7.5 p. 205; ANSI Sec. 3.3.4 (esp. footnote 14); ISO Sec. 6.3.4; Rationale Sec. 3.3.2.4; H&S Sec. 7.1 pp. 179-80.

4.8: I have a function which accepts, and is supposed to initialize, a pointer:

```
void f(ip)
int *ip;
{
        static int dummy = 5;
        ip = &dummy;
}
```

But when I call it like this:

```
int *ip;
f(ip);
```

the pointer in the caller remains unchanged.

A: Are you sure the function initialized what you thought it did? Remember that arguments in C are passed by value. The called function altered only the passed copy of the pointer. You'll either want to pass the address of the pointer (the function will end up accepting a pointer-to-a-pointer), or have the function return the pointer.

See also questions 4.9 and 4.11.

4.9: Can I use a void ** pointer to pass a generic pointer to a function by reference?

A: Not portably. There is no generic pointer-to-pointer type in C. void * acts as a generic pointer only because conversions are applied automatically when other pointer types are assigned to and from void *'s; these conversions cannot be performed (the correct underlying pointer type is not known) if an attempt is made to indirect upon a void ** value which points at something other than a void *.

4.10: I have a function

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

which accepts a pointer to an int. How can I pass a constant by reference? A call like

```
f(&5);
```

doesn't seem to work.

A: You can't do this directly. You will have to declare a temporary variable, and then pass its address to the function:

```
int five = 5;
f(&five);
```

See also questions 2.10, 4.8, and 20.1.

4.11: Does C even have "pass by reference"?

A: Not really. Strictly speaking, C always uses pass by value. You can simulate pass by reference yourself, by defining functions which accept pointers and then using the &

operator when calling, and the compiler will essentially simulate it for you when you pass an array to a function (by passing a pointer instead, see question 6.4 et al.), but C has nothing truly equivalent to formal pass by reference or C++ reference parameters. (However, function-like preprocessor macros do provide a form of "call by name".)

See also questions 4.8 and 20.1.

References: K&R1 Sec. 1.8 pp. 24-5, Sec. 5.2 pp. 91-3; K&R2 Sec. 1.8 pp. 27-8, Sec. 5.2 pp. 91-3; ANSI Sec. 3.3.2.2, esp. footnote 39; ISO Sec. 6.3.2.2; H&S Sec. 9.5 pp. 273-4.

4.12: I've seen different methods used for calling functions via pointers. What's the story?

A: Originally, a pointer to a function had to be "turned into" a "real" function, with the * operator (and an extra pair of parentheses, to keep the precedence straight), before calling:

```
int r, func(), (*fp)() = func;
r = (*fp)();
```

It can also be argued that functions are always called via pointers, and that "real" function names always decay implicitly into pointers (in expressions, as they do in initializations; see question 1.34). This reasoning, made widespread through pcc and adopted in the ANSI standard, means that

```
r = fp();
```

is legal and works correctly, whether fp is the name of a function or a pointer to one. (The usage has always been unambiguous; there is nothing you ever could have done with a function pointer followed by an argument list except call the function pointed to.) An explicit * is still allowed (and recommended, if portability to older compilers is important).

See also question 1.34.

References: K&R1 Sec. 5.12 p. 116; K&R2 Sec. 5.11 p. 120; ANSI Sec. 3.3.2.2; ISO Sec. 6.3.2.2; Rationale Sec. 3.3.2.2; H&S Sec. 5.8 p. 147, Sec. 7.4.3 p. 190.

Null Pointers

5.1: What is this infamous null pointer, anyway?

A: The language definition states that for each pointer type, there is a special value -- the "null pointer" -- which is distinguishable from all other pointer values and which is "guaranteed to compare unequal to a pointer to any object or function." That is, the address-of operator & will never yield a null pointer, nor will a successful call to malloc(). (malloc() does return a null pointer when it fails, and this is a typical use of null pointers: as a "special" pointer value with some other meaning, usually "not allocated" or "not pointing anywhere yet.")

A null pointer is conceptually different from an uninitialized pointer. A null pointer is known not to point to any object or function; an uninitialized pointer might point anywhere. See also questions 1.30, 7.1, and 7.31.

As mentioned above, there is a null pointer for each pointer type, and the internal values of null pointers for different types may be different. Although programmers need not know the internal values, the compiler must always be informed which type of null pointer is required, so that it can make the distinction if necessary (see questions 5.2, 5.5, and 5.6 below).

References: K&R1 Sec. 5.4 pp. 97-8; K&R2 Sec. 5.4 p. 102; ANSI Sec. 3.2.2.3; ISO Sec. 6.2.2.3; Rationale Sec. 3.2.2.3; H&S Sec. 5.3.2 pp. 121-3.

5.2: How do I get a null pointer in my programs?

A: According to the language definition, a constant 0 in a pointer context is converted into a null pointer at compile time. That is, in an initialization, assignment, or comparison when one side is a variable or expression of pointer type, the compiler can tell that a constant 0 on the other side requests a null pointer, and generate the correctly-typed null pointer value. Therefore, the following fragments are perfectly legal:

```
char *p = 0;
if(p != 0)
```

(See also question 5.3.)

However, an argument being passed to a function is not necessarily recognizable as a pointer context, and the compiler may not be able to tell that an unadorned 0 "means" a null pointer. To generate a null pointer in a function call context, an explicit cast may be required, to force the 0 to be recognized as a pointer. For example, the Unix system call execl takes a variable-length, null-pointer-terminated list of character pointer arguments, and is correctly called like this:

```
execl("/bin/sh", "sh", "-c", "date", (char *)0);
```

If the (char *) cast on the last argument were omitted, the compiler would not know to pass a null pointer, and would pass an integer 0 instead. (Note that many Unix manuals get this example wrong .)

When function prototypes are in scope, argument passing becomes an "assignment context," and most casts may safely be omitted, since the prototype tells the compiler that a pointer is required, and of which type, enabling it to correctly convert an unadorned 0. Function prototypes cannot provide the types for variable arguments in variable-length argument lists however, so explicit casts are still required for those arguments. (See also question 15.3.) It is safest to properly cast all null pointer constants in function calls: to guard against varargs functions or those without prototypes, to allow interim use of non-ANSI compilers, and to demonstrate that you know what you are doing. (Incidentally, it's also a simpler rule to remember.)

Summary:

```
Unadorned 0 okay:        Explicit cast required:

initialization           function call,
                         no prototype in scope
assignment
                         variable argument in
comparison               varargs function call

function call,
prototype in scope,
fixed argument
```

References: K&R1 Sec. A7.7 p. 190, Sec. A7.14 p. 192; K&R2 Sec. A7.10 p. 207, Sec. A7.17 p. 209; ANSI Sec. 3.2.2.3; ISO Sec. 6.2.2.3; H&S Sec. 4.6.3 p. 95, Sec. 6.2.7 p. 171.

5.3: Is the abbreviated pointer comparison "if(p)" to test for non- null pointers valid? What if the internal representation for null pointers is nonzero?

A: When C requires the Boolean value of an expression (in the if, while, for, and do statements, and with the &&, ||, !, and ?: operators), a false value is inferred when the expression compares equal to zero, and a true value otherwise. That is, whenever one writes

```
if(expr)
```

where "expr" is any expression at all, the compiler essentially acts as if it had been written as

```
if((expr) != 0)
```

Substituting the trivial pointer expression "p" for "expr," we have

```
if(p)   is equivalent to              if(p != 0)
```

and this is a comparison context, so the compiler can tell that the (implicit) 0 is actually a null pointer constant, and use the correct null pointer value. There is no trickery involved here; compilers do work this way, and generate identical code for both constructs. The internal representation of a null pointer does *not* matter.

The boolean negation operator, !, can be described as follows:

```
!expr   is essentially equivalent to    (expr)?0:1
        or to                           ((expr) == 0)
```

which leads to the conclusion that

```
if(!p)  is equivalent to               if(p == 0)
```

"Abbreviations" such as if(p), though perfectly legal, are considered by some to be bad style (and by others to be good style; see question 17.10).

See also question 9.2.

References: K&R2 Sec. A7.4.7 p. 204; ANSI Sec. 3.3.3.3, Sec. 3.3.9, Sec. 3.3.13, Sec. 3.3.14, Sec. 3.3.15, Sec. 3.6.4.1, Sec. 3.6.5; ISO Sec. 6.3.3.3, Sec. 6.3.9, Sec. 6.3.13, Sec. 6.3.14, Sec. 6.3.15, Sec. 6.6.4.1, Sec. 6.6.5; H&S Sec. 5.3.2 p. 122.

5.4: What is NULL and how is it #defined?

A: As a matter of style, many programmers prefer not to have unadorned 0's scattered through their programs. Therefore, the preprocessor macro NULL is #defined (by or) with the value 0, possibly cast to (void *) (see also question 5.6). A programmer who wishes to make explicit the distinction between 0 the integer and 0 the null pointer constant can then use NULL whenever a null pointer is required.

Using NULL is a stylistic convention only; the preprocessor turns NULL back into 0 which is then recognized by the compiler, in pointer contexts, as before. In particular, a cast may still be necessary before NULL (as before 0) in a function call argument. The table under question 5.2 above applies for NULL as well as 0 (an unadorned NULL is equivalent to an unadorned 0).

NULL should *only* be used for pointers; see question 5.9.

References: K&R1 Sec. 5.4 pp. 97-8; K&R2 Sec. 5.4 p. 102; ANSI Sec. 4.1.5, Sec. 3.2.2.3; ISO Sec. 7.1.6, Sec. 6.2.2.3; Rationale Sec. 4.1.5; H&S Sec. 5.3.2 p. 122, Sec. 11.1 p. 292.

5.5: How should NULL be defined on a machine which uses a nonzero bit pattern as the internal representation of a null pointer?

A: The same as on any other machine: as 0 (or ((void *)0)).

Whenever a programmer requests a null pointer, either by writing "0" or "NULL," it is the compiler's responsibility to generate whatever bit pattern the machine uses for that null pointer. Therefore, #defining NULL as 0 on a machine for which internal null pointers are nonzero is as valid as on any other: the compiler must always be able to generate the machine's correct null pointers in response to unadorned 0's seen in pointer contexts. See also questions 5.2, 5.10, and 5.17.

References: ANSI Sec. 4.1.5; ISO Sec. 7.1.6; Rationale Sec. 4.1.5.

5.6: If NULL were defined as follows:

```
#define NULL ((char *)0)
```

wouldn't that make function calls which pass an uncast NULL work?

A: Not in general. The problem is that there are machines which use different internal representations for pointers to different types of data. The suggested definition would make uncast NULL arguments to functions expecting pointers to characters work correctly, but pointer arguments of other types would still be problematical, and legal constructions such as

```
FILE *fp = NULL;
```

could fail.

Nevertheless, ANSI C allows the alternate definition

```
#define NULL ((void *)0)
```

for NULL. Besides potentially helping incorrect programs to work (but only on machines with homogeneous pointers, thus questionably valid assistance), this definition may catch programs which use NULL incorrectly (e.g. when the ASCII NUL character was really intended; see question 5.9).

References: Rationale Sec. 4.1.5.

5.9: If NULL and 0 are equivalent as null pointer constants, which should I use?

A: Many programmers believe that NULL should be used in all pointer contexts, as a reminder that the value is to be thought of as a pointer. Others feel that the confusion surrounding NULL and 0 is only compounded by hiding 0 behind a macro, and prefer to use unadorned 0 instead. There is no one right answer. (See also questions 9.2 and 17.10.) C programmers must understand that NULL and 0 are interchangeable in pointer contexts, and that an uncast 0 is perfectly acceptable. Any usage of NULL (as opposed to 0) should be considered a gentle reminder that a pointer is involved; programmers should not depend on it (either for their own understanding or the compiler's) for distinguishing pointer 0's from integer 0's.

NULL should *not* be used when another kind of 0 is required, even though it might work, because doing so sends the wrong stylistic message. (Furthermore, ANSI allows the definition of NULL to be ((void *)0), which will not work at all in non- pointer contexts.) In particular, do not use NULL when the ASCII null character (NUL) is desired. Provide your own definition

```
#define NUL '\0'
```

if you must.

References: K&R1 Sec. 5.4 pp. 97-8; K&R2 Sec. 5.4 p. 102.

5.10: But wouldn't it be better to use NULL (rather than 0), in case the value of NULL changes, perhaps on a machine with nonzero internal null pointers?

A: No. (Using NULL may be preferable, but not for this reason.) Although symbolic constants are often used in place of numbers because the numbers might change, this is *not* the reason that NULL is used in place of 0. Once again, the language guarantees that source-code 0's (in pointer contexts) generate null pointers. NULL is used only as a stylistic convention. See questions 5.5 and 9.2.

5.12: I use the preprocessor macro

```
#define Nullptr(type) (type *)0
```

to help me build null pointers of the correct type.

A: This trick, though popular and superficially attractive, does not buy much. It is not needed in assignments and comparisons; see question 5.2. It does not even save keystrokes. Its use may suggest to the reader that the program's author is shaky on the subject of null pointers, requiring that the #definition of the macro, its invocations, and *all* other pointer usages be checked. See also questions 9.1 and 10.2.

5.13: This is strange. NULL is guaranteed to be 0, but the null pointer is not?

A: When the term "null" or "NULL" is casually used, one of several things may be meant:

1. The conceptual null pointer, the abstract language concept defined in question 5.1. It is implemented with...
2. The internal (or run-time) representation of a null pointer, which may or may not be all-bits-0 and which may be different for different pointer types. The actual values should be of concern only to compiler writers. Authors of C programs never see them, since they use...

3. The null pointer constant, which is a constant integer 0 (see question 5.2). It is often hidden behind...
4. The NULL macro, which is #defined to be "0" or "((void *)0)" (see question 5.4). Finally, as red herrings, we have...
5. The ASCII null character (NUL), which does have all bits zero, but has no necessary relation to the null pointer except in name; and...
6. The "null string," which is another name for the empty string (""). Using the term "null string" can be confusing in C, because an empty string involves a null ('\0') character, but *not* a null pointer, which brings us full circle...

This article uses the phrase "null pointer" (in lower case) for sense 1, the character "0" or the phrase "null pointer constant" for sense 3, and the capitalized word "NULL" for sense 4.

5.14: Why is there so much confusion surrounding null pointers? Why do these questions come up so often?

A: C programmers traditionally like to know more than they need to about the underlying machine implementation. The fact that null pointers are represented both in source code, and internally to most machines, as zero invites unwarranted assumptions. The use of a preprocessor macro (NULL) may seem to suggest that the value could change some day, or on some weird machine. The construct "if(p == 0)" is easily misread as calling for conversion of p to an integral type, rather than 0 to a pointer type, before the comparison. Finally, the distinction between the several uses of the term "null" (listed in question 5.13 above) is often overlooked.

One good way to wade out of the confusion is to imagine that C used a keyword (perhaps "nil", like Pascal) as a null pointer constant. The compiler could either turn "nil" into the correct type of null pointer when it could determine the type from the source code, or complain when it could not. Now in fact, in C the keyword for a null pointer constant is not "nil" but "0", which works almost as well, except that an uncast "0" in a non- pointer context generates an integer zero instead of an error message, and if that uncast 0 was supposed to be a null pointer constant, the code may not work.

5.15: I'm confused. I just can't understand all this null pointer stuff.

A: Follow these two simple rules:

1. When you want a null pointer constant in source code, use "0" or "NULL".
2. If the usage of "0" or "NULL" is an argument in a function call, cast it to the pointer type expected by the function being called.

The rest of the discussion has to do with other people's misunderstandings, with the internal representation of null pointers (which you shouldn't need to know), and with ANSI C refinements. Understand questions 5.1, 5.2, and 5.4, and consider 5.3, 5.9, 5.13, and 5.14, and you'll do fine.

5.16: Given all the confusion surrounding null pointers, wouldn't it be easier simply to require them to be represented internally by zeroes?

A: If for no other reason, doing so would be ill-advised because it would unnecessarily constrain implementations which would otherwise naturally represent null pointers by special, nonzero bit patterns, particularly when those values would trigger automatic hardware traps for invalid accesses.

Besides, what would such a requirement really accomplish? Proper understanding of null pointers does not require knowledge of the internal representation, whether zero or nonzero. Assuming that null pointers are internally zero does not make any code easier to write (except for a certain ill-advised usage of calloc(); see question 7.31). Known-zero internal pointers would not obviate casts in function calls, because the *size* of the pointer might still be different from that of an int. (If "nil" were used to request null pointers, as mentioned in question 5.14 above, the urge to assume an internal zero representation would not even arise.)

5.17: Seriously, have any actual machines really used nonzero null pointers, or different representations for pointers to different types?

A: The Prime 50 series used segment 07777, offset 0 for the null pointer, at least for PL/I. Later models used segment 0, offset 0 for null pointers in C, necessitating new instructions such as TCNP (Test C Null Pointer), evidently as a sop to all the extant poorly-written C code which made incorrect assumptions. Older, word-addressed Prime machines were also notorious for requiring larger byte pointers (char *'s) than word pointers (int *'s).

The Eclipse MV series from Data General has three architecturally supported pointer formats (word, byte, and bit pointers), two of which are used by C compilers: byte pointers for char * and void *, and word pointers for everything else.

Some Honeywell-Bull mainframes use the bit pattern 06000 for (internal) null pointers.

The CDC Cyber 180 Series has 48-bit pointers consisting of a ring, segment, and offset. Most users (in ring 11) have null pointers of 0xB00000000000. It was common on old CDC ones- complement machines to use an all-one-bits word as a special flag for all kinds of data, including invalid addresses.

The old HP 3000 series uses a different addressing scheme for byte addresses than for word addresses; like several of the machines above it therefore uses different representations for char * and void * pointers than for other pointers.

The Symbolics Lisp Machine, a tagged architecture, does not even have conventional numeric pointers; it uses the pair (basically a nonexistent <object, offset> handle) as a C null pointer.

Depending on the "memory model" in use, 8086-family processors (PC compatibles) may use 16-bit data pointers and 32-bit function pointers, or vice versa.

Some 64-bit Cray machines represent int * in the lower 48 bits of a word; char * additionally uses the upper 16 bits to indicate a byte address within a word.

References: K&R1 Sec. A14.4 p. 211.

5.20: What does a run-time "null pointer assignment" error mean? How do I track it down?

A: This message, which typically occurs with MS-DOS compilers (see, therefore, section 19) means that you've written, via a null (perhaps because uninitialized) pointer, to location 0. (See also question 16.8.)

A debugger may let you set a data breakpoint or watchpoint or something on location 0. Alternatively, you could write a bit of code to stash away a copy of 20 or so bytes from location 0, and periodically check that the memory at location 0 hasn't changed.

Arrays and Pointers

6.1: I had the definition char a[6] in one source file, and in another I declared extern char *a. Why didn't it work?

A: The declaration extern char *a simply does not match the actual definition. The type pointer-to-type-T is not the same as array- of-type-T. Use extern char a[].

References: ANSI Sec. 3.5.4.2; ISO Sec. 6.5.4.2; CT&P Sec. 3.3 pp. 33-4, Sec. 4.5 pp. 64-5.

6.2: But I heard that char a[] was identical to char *a.

A: Not at all. (What you heard has to do with formal parameters to functions; see question 6.4.) Arrays are not pointers. The array declaration char a[6] requests that space for six characters be set aside, to be known by the name "a." That is, there is a location named "a" at which six characters can sit. The pointer declaration char *p, on the other hand, requests a place which holds a pointer, to be known by the name "p." This pointer can point almost anywhere: to any char, or to any contiguous array of chars, or nowhere (see also questions 5.1 and 1.30).

As usual, a picture is worth a thousand words. The declarations

```
char a[] = "hello";
char *p = "world";
```

would initialize data structures which could be represented like this:

```
+---+---+---+---+---+---+
```

```
a: | h | e | l | l | o |\0 |
   +---+---+---+---+---+---+
   +-----+      +---+---+---+---+---+---+
p: |   *======> | w | o | r | l | d |\0 |
   +-----+      +---+---+---+---+---+---+
```

It is important to realize that a reference like *x*[3] generates different code depending on whether *x* is an array or a pointer. Given the declarations above, when the compiler sees the expression a[3], it emits code to start at the location "a," move three past it, and fetch the character there. When it sees the expression p[3], it emits code to start at the location "p," fetch the pointer value there, add three to the pointer, and finally fetch the character pointed to. In other words, a[3] is three places past (the start of) the object *named* a, while p[3] is three places past the object *pointed to* by p. In the example above, both a[3] and p[3] happen to be the character 'l', but the compiler gets there differently.

References: K&R2 Sec. 5.5 p. 104; CT&P Sec. 4.5 pp. 64-5.

6.3: So what is meant by the "equivalence of pointers and arrays" in C?

A: Much of the confusion surrounding arrays and pointers in C can be traced to a misunderstanding of this statement. Saying that arrays and pointers are "equivalent" means neither that they are identical nor even interchangeable.

"Equivalence" refers to the following key definition:

```
An lvalue of type array-of-T which appears in an
expression decays (with three exceptions) into a
pointer to its first element; the type of the
resultant pointer is pointer-to-T.
```

(The exceptions are when the array is the operand of a sizeof or & operator, or is a string literal initializer for a character array.)

As a consequence of this definition, the compiler doesn't apply the array subscripting operator [] that differently to arrays and pointers, after all. In an expression of the form a[i], the array decays into a pointer, following the rule above, and is then subscripted just as would be a pointer variable in the expression p[i] (although the eventual memory accesses will be different, as explained in question 6.2). If you were to assign the array's address to the pointer:

```
p = a;
```

then p[3] and a[3] would access the same element.

See also question 6.8.

References: K&R1 Sec. 5.3 pp. 93-6; K&R2 Sec. 5.3 p. 99; ANSI Sec. 3.2.2.1, Sec. 3.3.2.1, Sec. 3.3.6; ISO Sec. 6.2.2.1, Sec. 6.3.2.1, Sec. 6.3.6; H&S Sec. 5.4.1 p. 124.

6.4: Then why are array and pointer declarations interchangeable as function formal parameters?

A: It's supposed to be a convenience.

Since arrays decay immediately into pointers, an array is never actually passed to a function. Allowing pointer parameters to be declared as arrays is a simply a way of making it look as though the array was being passed -- a programmer may wish to emphasize that a parameter is traditionally treated as if it were an array, or that an array (strictly speaking, the address) is traditionally passed. As a convenience, therefore, any parameter declarations which "look like" arrays, e.g.

```
f(a)
char a[];
{ ... }
```

are treated by the compiler as if they were pointers, since that is what the function will receive if an array is passed:

```
f(a)
char *a;
{ ... }
```

This conversion holds only within function formal parameter declarations, nowhere else. If the conversion bothers you, avoid it; many people have concluded that the confusion it causes outweighs the small advantage of having the declaration "look like" the call or the uses within the function.

See also question 6.21.

References: K&R1 Sec. 5.3 p. 95, Sec. A10.1 p. 205; K&R2 Sec. 5.3 p. 100, Sec. A8.6.3 p. 218, Sec. A10.1 p. 226; ANSI Sec. 3.5.4.3, Sec. 3.7.1, Sec. 3.9.6; ISO Sec. 6.5.4.3, Sec. 6.7.1, Sec. 6.9.6; H&S Sec. 9.3 p. 271; CT&P Sec. 3.3 pp. 33-4.

6.7: How can an array be an lvalue, if you can't assign to it?

A: The ANSI C Standard defines a "modifiable lvalue," which an array is not.

References: ANSI Sec. 3.2.2.1; ISO Sec. 6.2.2.1; Rationale Sec. 3.2.2.1; H&S Sec. 7.1 p. 179.

6.8: Practically speaking, what is the difference between arrays and pointers?

A: Arrays automatically allocate space, but can't be relocated or resized. Pointers must be explicitly assigned to point to allocated space (perhaps using malloc), but can be reassigned (i.e. pointed at different objects) at will, and have many other uses besides serving as the base of blocks of memory.

Due to the so-called equivalence of arrays and pointers (see question 6.3), arrays and pointers often seem interchangeable, and in particular a pointer to a block of memory assigned by malloc is frequently treated (and can be referenced using []) exactly as if it were a true array. See questions 6.14 and 6.16. (Be careful with sizeof, though.)

See also questions 1.32 and 20.14.

6.9: Someone explained to me that arrays were really just constant pointers.

A: This is a bit of an oversimplification. An array name is "constant" in that it cannot be assigned to, but an array is *not* a pointer, as the discussion and pictures in question 6.2 should make clear. See also questions 6.3 and 6.8.

6.11: I came across some "joke" code containing the "expression" 5["abcdef"] . How can this be legal C?

A: Yes, Virginia, array subscripting is commutative in C. This curious fact follows from the pointer definition of array subscripting, namely that a[e] is identical to *((a)+(e)), for *any* two expressions a and e, as long as one of them is a pointer expression and one is integral. This unsuspected commutativity is often mentioned in C texts as if it were something to be proud of, but it finds no useful application outside of the Obfuscated C Contest (see question 20.36).

References: Rationale Sec. 3.3.2.1; H&S Sec. 5.4.1 p. 124, Sec. 7.4.1 pp. 186-7.

6.12: Since array references decay into pointers, if arr is an array, what's the difference between arr and &arr?

A: The type.

In Standard C, &arr yields a pointer, of type pointer-to-array- of-T, to the entire array. (In pre-ANSI C, the & in &arr generally elicited a warning, and was generally ignored.) Under all C compilers, a simple reference (without an explicit &) to an array yields a pointer, of type pointer-to-T, to the array's first element. (See also questions 6.3, 6.13, and 6.18.)

References: ANSI Sec. 3.2.2.1, Sec. 3.3.3.2; ISO Sec. 6.2.2.1, Sec. 6.3.3.2; Rationale Sec. 3.3.3.2; H&S Sec. 7.5.6 p. 198.

6.13: How do I declare a pointer to an array?

A: Usually, you don't want to. When people speak casually of a pointer to an array, they usually mean a pointer to its first element.

Instead of a pointer to an array, consider using a pointer to one of the array's elements. Arrays of type T decay into pointers to type T (see question 6.3), which is convenient; subscripting or incrementing the resultant pointer will access the individual members of the array. True pointers to arrays, when subscripted or incremented, step over entire arrays, and are generally useful only when operating on arrays of arrays, if at all. (See question 6.18.)

If you really need to declare a pointer to an entire array, use something like "int (*ap)[N];" where N is the size of the array. (See also question 1.21.) If the size of the array is unknown, N can in principle be omitted, but the resulting type, "pointer to array of unknown size," is useless.

See also question 6.12 above.

References: ANSI Sec. 3.2.2.1; ISO Sec. 6.2.2.1.

6.14: How can I set an array's size at compile time? How can I avoid fixed-sized arrays?

A: The equivalence between arrays and pointers (see question 6.3) allows a pointer to malloc'ed memory to simulate an array quite effectively. After executing

```
#include
int *dynarray = (int *)malloc(10 * sizeof(int));
```

(and if the call to malloc() succeeds), you can reference dynarray[i] (for i from 0 to 9) just as if dynarray were a conventional, statically-allocated array (int a[10]). See also question 6.16.

6.15: How can I declare local arrays of a size matching a passed-in array?

A: You can't, in C. Array dimensions must be compile-time constants. (gcc provides parameterized arrays as an extension.) You'll have to use malloc(), and remember to call free() before the function returns. See also questions 6.14, 6.16, 6.19, 7.22, and maybe 7.32.

References: ANSI Sec. 3.4, Sec. 3.5.4.2; ISO Sec. 6.4, Sec. 6.5.4.2.

6.16: How can I dynamically allocate a multidimensional array?

A: It is usually best to allocate an array of pointers, and then initialize each pointer to a dynamically-allocated "row." Here is a two-dimensional example:

```
#include

int **array1 = (int **)malloc(nrows * sizeof(int *));
for(i = 0; i < nrows; i++)
```

```
                        array1[i] = (int *)malloc(ncolumns *
sizeof(int));
```

(In real code, of course, all of malloc's return values would be checked.)

You can keep the array's contents contiguous, while making later reallocation of individual rows difficult, with a bit of explicit pointer arithmetic:

```
                int **array2 = (int **)malloc(nrows * sizeof(int *));
                array2[0] = (int *)malloc(nrows * ncolumns *
sizeof(int));
                for(i = 1; i < nrows; i++)
                        array2[i] = array2[0] + i * ncolumns;
```

In either case, the elements of the dynamic array can be accessed with normal-looking array subscripts: arrayx[i][j] (for 0 <= i <= NROWS and 0 <= j <= NCOLUMNS).

If the double indirection implied by the above schemes is for some reason unacceptable, you can simulate a two-dimensional array with a single, dynamically-allocated one-dimensional array:

```
                int *array3 = (int *)malloc(nrows * ncolumns *
sizeof(int));
```

However, you must now perform subscript calculations manually, accessing the i,jth element with array3[i * ncolumns + j]. (A macro could hide the explicit calculation, but invoking it would require parentheses and commas which wouldn't look exactly like multidimensional array syntax, and the macro would need access to at least one of the dimensions, as well. See also question 6.19.)

Finally, you could use pointers to arrays:

```
                int (*array4)[NCOLUMNS] =
                        (int (*)[NCOLUMNS])malloc(nrows *
sizeof(*array4));
```

but the syntax starts getting horrific and at most one dimension may be specified at run time.

With all of these techniques, you may of course need to remember to free the arrays (which may take several steps; see question 7.23) when they are no longer needed, and you cannot necessarily intermix dynamically-allocated arrays with conventional, statically-allocated ones (see question 6.20, and also question 6.18).

All of these techniques can also be extended to three or more dimensions.

6.17: Here's a neat trick: if I write

```
                int realarray[10];
```

```
int *array = &realarray[-1];
```

I can treat "array" as if it were a 1-based array.

A: Although this technique is attractive (and was used in old editions of the book _Numerical Recipes in C_), it does not conform to the C standards. Pointer arithmetic is defined only as long as the pointer points within the same allocated block of memory, or to the imaginary "terminating" element one past it; otherwise, the behavior is undefined, *even if the pointer is not dereferenced*. The code above could fail if, while subtracting the offset, an illegal address were generated (perhaps because the address tried to "wrap around" past the beginning of some memory segment).

References: K&R2 Sec. 5.3 p. 100, Sec. 5.4 pp. 102-3, Sec. A7.7 pp. 205-6; ANSI Sec. 3.3.6; ISO Sec. 6.3.6; Rationale Sec. 3.2.2.3.

6.18: My compiler complained when I passed a two-dimensional array to a function expecting a pointer to a pointer.

A: The rule (see question 6.3) by which arrays decay into pointers is not applied recursively. An array of arrays (i.e. a two- dimensional array in C) decays into a pointer to an array, not a pointer to a pointer. Pointers to arrays can be confusing, and must be treated carefully; see also question 6.13. (The confusion is heightened by the existence of incorrect compilers, including some old versions of pcc and pcc-derived lints, which improperly accept assignments of multi-dimensional arrays to multi-level pointers.)

If you are passing a two-dimensional array to a function:

```
int array[NROWS][NCOLUMNS];
f(array);
```

the function's declaration must match:

```
f(int a[][NCOLUMNS])
{ ... }
```

or

```
f(int (*ap)[NCOLUMNS])  /* ap is a pointer to an array
*/
{ ... }
```

In the first declaration, the compiler performs the usual implicit parameter rewriting of "array of array" to "pointer to array" (see questions 6.3 and 6.4); in the second form the pointer declaration is explicit. Since the called function does not allocate space for the array, it does not need to know the overall size, so the number of rows, NROWS, can be omitted. The "shape" of the array is still important, so the column dimension NCOLUMNS (and, for three- or more dimensional arrays, the intervening ones) must be retained.

If a function is already declared as accepting a pointer to a pointer, it is probably meaningless to pass a two-dimensional array directly to it.

See also questions 6.12 and 6.15.

References: K&R1 Sec. 5.10 p. 110; K&R2 Sec. 5.9 p. 113; H&S Sec. 5.4.3 p. 126.

6.19: How do I write functions which accept two-dimensional arrays when the "width" is not known at compile time?

A: It's not easy. One way is to pass in a pointer to the [0][0] element, along with the two dimensions, and simulate array subscripting "by hand:"

```
                f2(aryp, nrows, ncolumns)
                int *aryp;
                int nrows, ncolumns;
                { ... array[i][j] is accessed as aryp[i * ncolumns + j]
... }
```

This function could be called with the array from question 6.18 as

```
                f2(&array[0][0], NROWS, NCOLUMNS);
```

It must be noted, however, that a program which performs multidimensional array subscripting "by hand" in this way is not in strict conformance with the ANSI C Standard; according to an official interpretation, the behavior of accessing (&array[0][0])[x] is not defined for x >= NCOLUMNS.

gcc allows local arrays to be declared having sizes which are specified by a function's arguments, but this is a nonstandard extension.

When you want to be able to use a function on multidimensional arrays of various sizes, one solution is to simulate all the arrays dynamically, as in question 6.16.

See also questions 6.18, 6.20, and 6.15.

References: ANSI Sec. 3.3.6; ISO Sec. 6.3.6.

6.20: How can I use statically- and dynamically-allocated multidimensional arrays interchangeably when passing them to functions?

A: There is no single perfect method. Given the declarations

```
                int array[NROWS][NCOLUMNS];
                int **array1;                   /* ragged */
                int **array2;                   /* contiguous */
                int *array3;                    /* "flattened" */
                int (*array4)[NCOLUMNS];
```

with the pointers initialized as in the code fragments in question 6.16, and functions declared as

```
f1(int a[][NCOLUMNS], int nrows, int ncolumns);
f2(int *aryp, int nrows, int ncolumns);
f3(int **pp, int nrows, int ncolumns);
```

where f1() accepts a conventional two-dimensional array, f2() accepts a "flattened" two-dimensional array, and f3() accepts a pointer-to-pointer, simulated array (see also questions 6.18 and 6.19), the following calls should work as expected:

```
f1(array, NROWS, NCOLUMNS);
f1(array4, nrows, NCOLUMNS);
f2(&array[0][0], NROWS, NCOLUMNS);
f2(*array, NROWS, NCOLUMNS);
f2(*array2, nrows, ncolumns);
f2(array3, nrows, ncolumns);
f2(*array4, nrows, NCOLUMNS);
f3(array1, nrows, ncolumns);
f3(array2, nrows, ncolumns);
```

The following two calls would probably work on most systems, but involve questionable casts, and work only if the dynamic ncolumns matches the static NCOLUMNS:

```
f1((int (*)[NCOLUMNS])(*array2), nrows, ncolumns);
f1((int (*)[NCOLUMNS])array3, nrows, ncolumns);
```

It must again be noted that passing &array[0][0] (or, equivalently, *array) to f2() is not strictly conforming; see question 6.19.

If you can understand why all of the above calls work and are written as they are, and if you understand why the combinations that are not listed would not work, then you have a *very* good understanding of arrays and pointers in C.

Rather than worrying about all of this, one approach to using multidimensional arrays of various sizes is to make them *all* dynamic, as in question 6.16. If there are no static multidimensional arrays -- if all arrays are allocated like array1 or array2 in question 6.16 -- then all functions can be written like f3().

6.21: Why doesn't sizeof properly report the size of an array when the array is a parameter to a function?

A: The compiler pretends that the array parameter was declared as a pointer (see question 6.4), and sizeof reports the size of the pointer.

References: H&S Sec. 7.5.2 p. 195.

Memory Allocation

7.1: Why doesn't this fragment work?

```
char *answer;
printf("Type something:\n");
gets(answer);
printf("You typed \"%s\"\n", answer);
```

A: The pointer variable answer(), which is handed to gets() as the location into which the response should be stored, has not been set to point to any valid storage. That is, we cannot say where the pointer answer() points. (Since local variables are not initialized, and typically contain garbage, it is not even guaranteed that answer() starts out as a null pointer. See questions 1.30 and 5.1.)

The simplest way to correct the question-asking program is to use a local array, instead of a pointer, and let the compiler worry about allocation:

```
#include
#include

char answer[100], *p;
printf("Type something:\n");
fgets(answer, sizeof answer, stdin);
if((p = strchr(answer, '\n')) != NULL)
        *p = '\0';
printf("You typed \"%s\"\n", answer);
```

This example also uses fgets() instead of gets(), so that the end of the array cannot be overwritten. (See question 12.23. Unfortunately for this example, fgets() does not automatically delete the trailing \n, as gets() would.) It would also be possible to use malloc() to allocate the answer buffer.

7.2: I can't get strcat() to work. I tried

```
char *s1 = "Hello, ";
char *s2 = "world!";
char *s3 = strcat(s1, s2);
```

but I got strange results.

A: As in question 7.1 above, the main problem here is that space for the concatenated result is not properly allocated. C does not provide an automatically-managed string type. C compilers only allocate memory for objects explicitly mentioned in the source code (in the case of "strings," this includes character arrays and string literals). The programmer must arrange for sufficient space for the results of run-time operations such as string concatenation, typically by declaring arrays, or by calling malloc().

strcat() performs no allocation; the second string is appended to the first one, in place. Therefore, one fix would be to declare the first string as an array:

```
char s1[20] = "Hello, ";
```

Since strcat() returns the value of its first argument (s1, in this case), the variable s3 is superfluous.

The original call to strcat() in the question actually has two problems: the string literal pointed to by s1, besides not being big enough for any concatenated text, is not necessarily writable at all. See question 1.32.

References: CT&P Sec. 3.2 p. 32.

7.3: But the man page for strcat() says that it takes two char *'s as arguments. How am I supposed to know to allocate things?

A: In general, when using pointers you *always* have to consider memory allocation, if only to make sure that the compiler is doing it for you. If a library function's documentation does not explicitly mention allocation, it is usually the caller's problem.

The Synopsis section at the top of a Unix-style man page or in the ANSI C standard can be misleading. The code fragments presented there are closer to the function definitions used by an implementor than the invocations used by the caller. In particular, many functions which accept pointers (e.g. to structures or strings) are usually called with the address of some object (a structure, or an array -- see questions 6.3 and 6.4). Other common examples are time() (see question 13.12) and stat().

7.5: I have a function that is supposed to return a string, but when it returns to its caller, the returned string is garbage.

A: Make sure that the pointed-to memory is properly allocated. The returned pointer should be to a statically-allocated buffer, or to a buffer passed in by the caller, or to memory obtained with malloc(), but *not* to a local (automatic) array. In other words, never do something like

```
char *itoa(int n)
{
        char retbuf[20];                /* WRONG */
        sprintf(retbuf, "%d", n);
        return retbuf;                  /* WRONG */
}
```

One fix (which is imperfect, especially if the function in question is called recursively, or if several of its return values are needed simultaneously) would be to declare the return buffer as

```
        static char retbuf[20];
```

See also questions 12.21 and 20.1.

References: ANSI Sec. 3.1.2.4; ISO Sec. 6.1.2.4.

7.6: Why am I getting "warning: assignment of pointer from integer lacks a cast" for calls to malloc()?

A: Have you #included , or otherwise arranged for malloc() to be declared properly?

References: H&S Sec. 4.7 p. 101.

7.7: Why does some code carefully cast the values returned by malloc to the pointer type being allocated?

A: Before ANSI/ISO Standard C introduced the void * generic pointer type, these casts were typically required to silence warnings (and perhaps induce conversions) when assigning between incompatible pointer types. (Under ANSI/ISO Standard C, these casts are no longer necessary.)

References: H&S Sec. 16.1 pp. 386-7.

7.8: I see code like

```
char *p = malloc(strlen(s) + 1);
strcpy(p, s);
```

Shouldn't that be malloc((strlen(s) + 1) * sizeof(char))?

A: It's never necessary to multiply by sizeof(char), since sizeof(char) is, by definition, exactly 1. (On the other hand, multiplying by sizeof(char) doesn't hurt, and may help by introducing a size_t into the expression.) See also question 8.9.

References: ANSI Sec. 3.3.3.4; ISO Sec. 6.3.3.4; H&S Sec. 7.5.2 p. 195.

7.14: I've heard that some operating systems don't actually allocate malloc'ed memory until the program tries to use it. Is this legal?

A: It's hard to say. The Standard doesn't say that systems can act this way, but it doesn't explicitly say that they can't, either.

References: ANSI Sec. 4.10.3; ISO Sec. 7.10.3.

7.16: I'm allocating a large array for some numeric work, using the line

```
double *array = malloc(256 * 256 * sizeof(double));
```

malloc() isn't returning null, but the program is acting strangely, as if it's overwriting memory, or malloc() isn't allocating as much as I asked for, or something.

A: Notice that 256 x 256 is 65,536, which will not fit in a 16-bit int, even before you multiply it by sizeof(double). If you need to allocate this much memory, you'll have to be careful. If size_t (the type accepted by malloc()) is a 32-bit type on your machine, but int is 16 bits, you might be able to get away with writing 256 * (256 * sizeof(double)) (see question 3.14). Otherwise, you'll have to break your data structure up into smaller chunks, or use a 32-bit machine, or use some nonstandard memory allocation routines. See also question 19.23.

7.17: I've got 8 meg of memory in my PC. Why can I only seem to malloc() 640K or so?

A: Under the segmented architecture of PC compatibles, it can be difficult to use more than 640K with any degree of transparency. See also question 19.23.

7.19: My program is crashing, apparently somewhere down inside malloc, but I can't see anything wrong with it.

A: It is unfortunately very easy to corrupt malloc's internal data structures, and the resulting problems can be stubborn. The most common source of problems is writing more to a malloc'ed region than it was allocated to hold; a particularly common bug is to malloc(strlen(s)) instead of strlen(s) + 1. Other problems may involve using pointers to freed storage, freeing pointers twice, freeing pointers not obtained from malloc, or trying to realloc a null pointer (see question 7.30).

See also questions 7.26, 16.8, and 18.2.

7.20: You can't use dynamically-allocated memory after you free it, can you?

A: No. Some early documentation for malloc() stated that the contents of freed memory were "left undisturbed," but this ill- advised guarantee was never universal and is not required by the C Standard.

Few programmers would use the contents of freed memory deliberately, but it is easy to do so accidentally. Consider the following (correct) code for freeing a singly-linked list:

```
struct list *listp, *nextp;
for(listp = base; listp != NULL; listp = nextp) {
        nextp = listp->next;
        free((void *)listp);
}
```

and notice what would happen if the more-obvious loop iteration expression listp = listp->next were used, without the temporary nextp pointer.

References: K&R2 Sec. 7.8.5 p. 167; ANSI Sec. 4.10.3; ISO Sec. 7.10.3; Rationale Sec. 4.10.3.2; H&S Sec. 16.2 p. 387; CT&P Sec. 7.10 p. 95.

7.21: Why isn't a pointer null after calling free()? How unsafe is it to use (assign, compare) a pointer value after it's been freed?

A: When you call free(), the memory pointed to by the passed pointer is freed, but the value of the pointer in the caller remains unchanged, because C's pass-by-value semantics mean that called functions never permanently change the values of their arguments. (See also question 4.8.)

A pointer value which has been freed is, strictly speaking, invalid, and *any* use of it, even if is not dereferenced can theoretically lead to trouble, though as a quality of implementation issue, most implementations will probably not go out of their way to generate exceptions for innocuous uses of invalid pointers.

References: ANSI Sec. 4.10.3; ISO Sec. 7.10.3; Rationale Sec. 3.2.2.3.

7.22: When I call malloc() to allocate memory for a local pointer, do I have to explicitly free() it?

A: Yes. Remember that a pointer is different from what it points to. Local variables are deallocated when the function returns, but in the case of a pointer variable, this means that the pointer is deallocated, *not* what it points to. Memory allocated with malloc() always persists until you explicitly free it. In general, for every call to malloc(), there should be a corresponding call to free().

7.23: I'm allocating structures which contain pointers to other dynamically-allocated objects. When I free a structure, do I have to free each subsidiary pointer first?

A: Yes. In general, you must arrange that each pointer returned from malloc() be individually passed to free(), exactly once (if it is freed at all).

A good rule of thumb is that for each call to malloc() in a program, you should be able to point at the call to free() which frees the memory allocated by that malloc() call.

See also question 7.24.

7.24: Must I free allocated memory before the program exits?

A: You shouldn't have to. A real operating system definitively reclaims all memory when a program exits. Nevertheless, some personal computers are said not to reliably recover memory, and all that can be inferred from the ANSI/ISO C Standard is that this is a "quality of implementation issue."

References: ANSI Sec. 4.10.3.2; ISO Sec. 7.10.3.2.

7.25: I have a program which mallocs and later frees a lot of memory, but memory usage (as reported by ps) doesn't seem to go back down.

A: Most implementations of malloc/free do not return freed memory to the operating system (if there is one), but merely make it available for future malloc() calls within the same program.

7.26: How does free() know how many bytes to free?

A: The malloc/free implementation remembers the size of each block allocated and returned, so it is not necessary to remind it of the size when freeing.

7.27: So can I query the malloc package to find out how big an allocated block is?

A: Not portably.

7.30: Is it legal to pass a null pointer as the first argument to realloc()? Why would you want to?

A: ANSI C sanctions this usage (and the related realloc(..., 0), which frees), although several earlier implementations do not support it, so it may not be fully portable. Passing an initially-null pointer to realloc() can make it easier to write a self-starting incremental allocation algorithm.

References: ANSI Sec. 4.10.3.4; ISO Sec. 7.10.3.4; H&S Sec. 16.3 p. 388.

7.31: What's the difference between calloc() and malloc()? Is it safe to take advantage of calloc's zero-filling? Does free() work on memory allocated with calloc(), or do you need a cfree()?

A: calloc(m, n) is essentially equivalent to

```
p = malloc(m * n);
memset(p, 0, m * n);
```

The zero fill is all-bits-zero, and does *not* therefore guarantee useful null pointer values (see section 5 of this list) or floating-point zero values. free() is properly used to free the memory allocated by calloc().

References: ANSI Sec. 4.10.3 to 4.10.3.2; ISO Sec. 7.10.3 to 7.10.3.2; H&S Sec. 16.1 p. 386, Sec. 16.2 p. 386; PCS Sec. 11 pp. 141,142.

7.32: What is alloca() and why is its use discouraged?

A: alloca() allocates memory which is automatically freed when the function which called alloca() returns. That is, memory allocated with alloca is local to a particular function's "stack frame" or context.

alloca() cannot be written portably, and is difficult to implement on machines without a conventional stack. Its use is problematical (and the obvious implementation on a stack-based machine fails) when its return value is passed directly to another function, as in fgets(alloca(100), 100, stdin).

For these reasons, alloca() is not Standard and cannot be used in programs which must be widely portable, no matter how useful it might be.

See also question 7.22.

References: Rationale Sec. 4.10.3.

Characters and Strings

8.1: Why doesn't

```
strcat(string, '!');
```

work?

A: There is a very real difference between characters and strings, and strcat() concatenates *strings*.

Characters in C are represented by small integers corresponding to their character set values (see also question 8.6 below). Strings are represented by arrays of characters; you usually manipulate a pointer to the first character of the array. It is never correct to use one when the other is expected. To append a ! to a string, use

```
strcat(string, "!");
```

See also questions 1.32, 7.2, and 16.6.

References: CT&P Sec. 1.5 pp. 9-10.

8.2: I'm checking a string to see if it matches a particular value. Why isn't this code working?

```
char *string;
...
if(string == "value") {
        /* string matches "value" */
        ...
}
```

A: Strings in C are represented as arrays of characters, and C never manipulates (assigns, compares, etc.) arrays as a whole. The == operator in the code fragment above compares two pointers -- the value of the pointer variable string and a pointer to the string literal "value" -- to see if they are equal, that is, if they point to the same place. They probably don't, so the comparison never succeeds.

To compare two strings, you generally use the library function strcmp():

```
if(strcmp(string, "value") == 0) {
        /* string matches "value" */
        ...
}
```

8.3: If I can say

```
char a[] = "Hello, world!";
```

why can't I say

```
char a[14];
a = "Hello, world!";
```

A: Strings are arrays, and you can't assign arrays directly. Use strcpy() instead:

```
strcpy(a, "Hello, world!");
```

See also questions 1.32, 4.2, and 7.2.

8.6: How can I get the numeric (character set) value corresponding to a character, or vice versa?

A: In C, characters are represented by small integers corresponding to their values (in the machine's character set), so you don't need a conversion routine: if you have the character, you have its value.

8.9: I think something's wrong with my compiler: I just noticed that sizeof('a') is 2, not 1 (i.e. not sizeof(char)).

A: Perhaps surprisingly, character constants in C are of type int, so sizeof('a') is sizeof(int) (though it's different in C++). See also question 7.8.

References: ANSI Sec. 3.1.3.4; ISO Sec. 6.1.3.4; H&S Sec. 2.7.3 p. 29.

Boolean Expressions

9.1: What is the right type to use for Boolean values in C? Why isn't it a standard type? Should I use #defines or enums for the true and false values?

A: C does not provide a standard Boolean type, in part because picking one involves a space/time tradeoff which can best be decided by the programmer. (Using an int may be faster, while using char may save data space. Smaller types may make the generated code bigger or slower, though, if they require lots of conversions to and from int.)

The choice between #defines and enumeration constants for the true/false values is arbitrary and not terribly interesting (see also questions 2.22 and 17.10). Use any of

```
#define TRUE  1                 #define YES 1
#define FALSE 0                 #define NO  0

enum bool {false, true};        enum bool {no, yes};
```

or use raw 1 and 0, as long as you are consistent within one program or project. (An enumeration may be preferable if your debugger shows the names of enumeration constants when examining variables.)

Some people prefer variants like

```
#define TRUE (1==1)
#define FALSE (!TRUE)
```

or define "helper" macros such as

```
#define Istrue(e) ((e) != 0)
```

These don't buy anything (see question 9.2 below; see also questions 5.12 and 10.2).

9.2: Isn't #defining TRUE to be 1 dangerous, since any nonzero value is considered "true" in C? What if a built-in logical or relational operator "returns" something other than 1?

A: It is true (sic) that any nonzero value is considered true in C, but this applies only "on input", i.e. where a Boolean value is expected. When a Boolean value is generated by a built-in operator, it is guaranteed to be 1 or 0. Therefore, the test

```
if((a == b) == TRUE)
```

would work as expected (as long as TRUE is 1), but it is obviously silly. In general, explicit tests against TRUE and FALSE are inappropriate, because some library functions (notably isupper(), isalpha(), etc.) return, on success, a nonzero value which is *not* necessarily 1. (Besides, if you believe that "if((a == b) == TRUE)" is an improvement over "if(a == b)", why stop there? Why not use "if(((a == b) == TRUE) == TRUE)"?) A good rule of thumb is to use TRUE and FALSE (or the like) only for assignment to a Boolean variable or function parameter, or as the return value from a Boolean function, but never in a comparison.

The preprocessor macros TRUE and FALSE (and, of course, NULL) are used for code readability, not because the underlying values might ever change. (See also questions 5.3 and 5.10.)

On the other hand, Boolean values and definitions can evidently be confusing, and some programmers feel that TRUE and FALSE macros only compound the confusion. (See also question 5.9.)

References: K&R1 Sec. 2.6 p. 39, Sec. 2.7 p. 41; K&R2 Sec. 2.6 p. 42, Sec. 2.7 p. 44, Sec. A7.4.7 p. 204, Sec. A7.9 p. 206; ANSI Sec. 3.3.3.3, Sec. 3.3.8, Sec. 3.3.9, Sec. 3.3.13, Sec. 3.3.14, Sec. 3.3.15, Sec. 3.6.4.1, Sec. 3.6.5; ISO Sec. 6.3.3.3, Sec. 6.3.8, Sec. 6.3.9, Sec. 6.3.13, Sec. 6.3.14, Sec. 6.3.15, Sec. 6.6.4.1, Sec. 6.6.5; H&S Sec. 7.5.4 pp. 196-7, Sec. 7.6.4 pp. 207-8, Sec. 7.6.5 pp. 208-9, Sec. 7.7 pp. 217-8, Sec. 7.8 pp. 218-9, Sec. 8.5 pp. 238-9, Sec. 8.6 pp. 241-4; "What the Tortoise Said to Achilles".

9.3: Is if(p), where p is a pointer, a valid conditional?

A: Yes. See question 5.3.

C Preprocessor

10.2: Here are some cute preprocessor macros:

```
#define begin   {
#define end     }
```

What do y'all think?

A: Bleah. See also section 17.

10.3: How can I write a generic macro to swap two values?

A: There is no good answer to this question. If the values are integers, a well-known trick using exclusive-OR could perhaps be used, but it will not work for floating-point values or pointers, or if the two values are the same variable (and the "obvious" supercompressed implementation for integral types a^=b^=a^=b is illegal due to multiple side-effects; see question 3.2). If the macro is intended to be used on values of arbitrary type (the usual goal), it cannot use a temporary, since it does not know what type of temporary it needs (and would have a hard time naming it if it did), and standard C does not provide a typeof operator.

The best all-around solution is probably to forget about using a macro, unless you're willing to pass in the type as a third argument.

10.4: What's the best way to write a multi-statement macro?

A: The usual goal is to write a macro that can be invoked as if it were a statement consisting of a single function call. This means that the "caller" will be supplying the final semicolon, so the macro body should not. The macro body cannot therefore be a simple brace-enclosed compound statement, because syntax errors would result if it were invoked (apparently as a single statement, but with a resultant extra semicolon) as the if branch of an if/else statement with an explicit else clause.

The traditional solution, therefore, is to use

```
#define MACRO(arg1, arg2) do {   \
        /* declarations */       \
        stmt1;                   \
        stmt2;                   \
        /* ... */                \
        } while(0)       /* (no trailing ; ) */
```

When the caller appends a semicolon, this expansion becomes a single statement regardless of context. (An optimizing compiler will remove any "dead" tests or branches on the constant condition 0, although lint may complain.)

If all of the statements in the intended macro are simple expressions, with no declarations or loops, another technique is to write a single, parenthesized expression using one or more comma operators. (For an example, see the first DEBUG() macro in question 10.26.) This technique also allows a value to be "returned."

References: H&S Sec. 3.3.2 p. 45; CT&P Sec. 6.3 pp. 82-3.

10.6: I'm splitting up a program into multiple source files for the first time, and I'm wondering what to put in .c files and what to put in .h files. (What does ".h" mean, anyway?)

A: As a general rule, you should put these things in header (.h) files:

```
macro definitions (preprocessor #defines)
```

```
structure, union, and enumeration declarations
typedef declarations
external function declarations (see also question 1.11)
global variable declarations
```

It's especially important to put a declaration or definition in a header file when it will be shared between several other files. (In particular, never put external function prototypes in .c files. See also question 1.7.)

On the other hand, when a definition or declaration should remain private to one source file, it's fine to leave it there.

See also questions 1.7 and 10.7.

References: K&R2 Sec. 4.5 pp. 81-2; H&S Sec. 9.2.3 p. 267; CT&P Sec. 4.6 pp. 66-7.

10.7: Is it acceptable for one header file to #include another?

A: It's a question of style, and thus receives considerable debate. Many people believe that "nested #include files" are to be avoided: the prestigious Indian Hill Style Guide (see question 17.9) disparages them; they can make it harder to find relevant definitions; they can lead to multiple-definition errors if a file is #included twice; and they make manual Makefile maintenance very difficult. On the other hand, they make it possible to use header files in a modular way (a header file can #include what it needs itself, rather than requiring each #includer to do so); a tool like grep (or a tags file) makes it easy to find definitions no matter where they are; a popular trick along the lines of:

```
#ifndef HFILENAME_USED
#define HFILENAME_USED
...header file contents...
#endif
```

(where a different bracketing macro name is used for each header file) makes a header file "idempotent" so that it can safely be #included multiple times; and automated Makefile maintenance tools (which are a virtual necessity in large projects anyway; see question 18.1) handle dependency generation in the face of nested #include files easily. See also question 17.10.

References: Rationale Sec. 4.1.2.

10.8: Where are header ("#include") files searched for?

A: The exact behavior is implementation-defined (which means that it is supposed to be documented; see question 11.33). Typically, headers named with <> syntax are searched for in one or more standard places. Header files named with "" syntax are first searched for in the "current directory," then (if not found) in the same standard places.

Traditionally (especially under Unix compilers), the current directory is taken to be the directory containing the file containing the #include directive. Under other compilers, however, the current directory (if any) is the directory in which the compiler was initially invoked. Check your compiler documentation.

References: K&R2 Sec. A12.4 p. 231; ANSI Sec. 3.8.2; ISO Sec. 6.8.2; H&S Sec. 3.4 p. 55.

10.9: I'm getting strange syntax errors on the very first declaration in a file, but it looks fine.

A: Perhaps there's a missing semicolon at the end of the last declaration in the last header file you're #including. See also questions 2.18 and 11.29.

10.11: I seem to be missing the system header file . Can someone send me a copy?

A: Standard headers exist in part so that definitions appropriate to your compiler, operating system, and processor can be supplied. You cannot just pick up a copy of someone else's header file and expect it to work, unless that person is using exactly the same environment. Ask your compiler vendor why the file was not provided (or to send a replacement copy).

10.12: How can I construct preprocessor #if expressions which compare strings?

A: You can't do it directly; preprocessor #if arithmetic uses only integers. You can #define several manifest constants, however, and implement conditionals on those.

See also question 20.17.

References: K&R2 Sec. 4.11.3 p. 91; ANSI Sec. 3.8.1; ISO Sec. 6.8.1; H&S Sec. 7.11.1 p. 225.

10.13: Does the sizeof operator work in preprocessor #if directives?

A: No. Preprocessing happens during an earlier phase of compilation, before type names have been parsed. Instead of sizeof, consider using the predefined constants in ANSI's , if applicable, or perhaps a "configure" script. (Better yet, try to write code which is inherently insensitive to type sizes.)

References: ANSI Sec. 2.1.1.2, Sec. 3.8.1 footnote 83; ISO Sec. 5.1.1.2, Sec. 6.8.1; H&S Sec. 7.11.1 p. 225.

10.14: Can I use an #ifdef in a #define line, to define something two different ways?

A: No. You can't "run the preprocessor on itself," so to speak. What you can do is use one of two completely separate #define lines, depending on the #ifdef setting.

References: ANSI Sec. 3.8.3, Sec. 3.8.3.4; ISO Sec. 6.8.3, Sec. 6.8.3.4; H&S Sec. 3.2 pp. 40-1.

10.15: Is there anything like an #ifdef for typedefs?

A: Unfortunately, no. (See also question 10.13.)

References: ANSI Sec. 2.1.1.2, Sec. 3.8.1 footnote 83; ISO Sec. 5.1.1.2, Sec. 6.8.1; H&S Sec. 7.11.1 p. 225.

10.16: How can I use a preprocessor #if expression to tell if a machine is big-endian or little-endian?

A: You probably can't. (Preprocessor arithmetic uses only long integers, and there is no concept of addressing.) Are you sure you need to know the machine's endianness explicitly? Usually it's better to write code which doesn't care). See also question 20.9.

References: ANSI Sec. 3.8.1; ISO Sec. 6.8.1; H&S Sec. 7.11.1 p. 225.

10.18: I inherited some code which contains far too many #ifdef's for my taste. How can I preprocess the code to leave only one conditional compilation set, without running it through the preprocessor and expanding all of the #include's and #define's as well?

A: There are programs floating around called unifdef, rmifdef, and scpp ("selective C preprocessor") which do exactly this. See question 18.16.

10.19: How can I list all of the pre#defined identifiers?

A: There's no standard way, although it is a common need. If the compiler documentation is unhelpful, the most expedient way is probably to extract printable strings from the compiler or preprocessor executable with something like the Unix strings utility. Beware that many traditional system-specific pre#defined identifiers (e.g. "unix") are non-Standard (because they clash with the user's namespace) and are being removed or renamed.

10.20: I have some old code that tries to construct identifiers with a macro like

```
#define Paste(a, b) a/**/b
```

but it doesn't work any more.

A: It was an undocumented feature of some early preprocessor implementations (notably John Reiser's) that comments disappeared entirely and could therefore be used for token pasting. ANSI affirms (as did K&R1) that comments are replaced with white space. However, since the need for pasting tokens was demonstrated and real, ANSI introduced a well-defined token- pasting operator, ##, which can be used like this:

```
#define Paste(a, b) a##b
```

See also question 11.17.

References: ANSI Sec. 3.8.3.3; ISO Sec. 6.8.3.3; Rationale Sec. 3.8.3.3; H&S Sec. 3.3.9 p. 52.

10.22: Why is the macro

```
#define TRACE(n) printf("TRACE: %d\n", n)
```

giving me the warning "macro replacement within a string literal"? It seems to be expanding

```
        TRACE(count);
as
        printf("TRACE: %d\count", count);
```

A: See question 11.18.

10.23: How can I use a macro argument inside a string literal in the macro expansion?

A: See question 11.18.

10.25: I've got this tricky preprocessing I want to do and I can't figure out a way to do it.

A: C's preprocessor is not intended as a general-purpose tool. (Note also that it is not guaranteed to be available as a separate program.) Rather than forcing it to do something inappropriate, consider writing your own little special-purpose preprocessing tool, instead. You can easily get a utility like make(1) to run it for you automatically.

If you are trying to preprocess something other than C, consider using a general-purpose preprocessor. (One older one available on most Unix systems is m4.)

10.26: How can I write a macro which takes a variable number of arguments?

A: One popular trick is to define and invoke the macro with a single, parenthesized "argument" which in the macro expansion becomes the entire argument list, parentheses and all, for a function such as printf():

```
#define DEBUG(args) (printf("DEBUG: "), printf args)

if(n != 0) DEBUG(("n is %d\n", n));
```

The obvious disadvantage is that the caller must always remember to use the extra parentheses.

gcc has an extension which allows a function-like macro to accept a variable number of arguments, but it's not standard. Other possible solutions are to use different macros (DEBUG1, DEBUG2, etc.) depending on the number of arguments, to play games with commas:

```
#define DEBUG(args) (printf("DEBUG: "), printf(args))
#define _ ,

DEBUG("i = %d" _ i)
```

It is often better to use a bona-fide function, which can take a variable number of arguments in a well-defined way. See questions 15.4 and 15.5.

ANSI/ISO Standard C

11.1: What is the "ANSI C Standard?"

A: In 1983, the American National Standards Institute (ANSI) commissioned a committee, X3J11, to standardize the C language. After a long, arduous process, including several widespread public reviews, the committee's work was finally ratified as ANS X3.159-1989 on December 14, 1989, and published in the spring of 1990. For the most part, ANSI C standardizes existing practice, with a few additions from C++ (most notably function prototypes) and support for multinational character sets (including the controversial trigraph sequences). The ANSI C standard also formalizes the C run-time library support routines.

More recently, the Standard has been adopted as an international standard, ISO/IEC 9899:1990, and this ISO Standard replaces the earlier X3.159 even within the United

States. Its sections are numbered differently (briefly, ISO sections 5 through 7 correspond roughly to the old ANSI sections 2 through 4). As an ISO Standard, it is subject to ongoing revision through the release of Technical Corrigenda and Normative Addenda.

In 1994, Technical Corrigendum 1 amended the Standard in about 40 places, most of them minor corrections or clarifications. More recently, Normative Addendum 1 added about 50 pages of new material, mostly specifying new library functions for internationalization. The production of Technical Corrigenda is an ongoing process, and a second one is expected in late 1995. In addition, both ANSI and ISO require periodic review of their standards. This process is beginning in 1995, and will likely result in a completely revised standard (nicknamed "C9X," on the assumption of completion by 1999).

The original ANSI Standard included a "Rationale," explaining many of its decisions, and discussing a number of subtle points, including several of those covered here. (The Rationale was "not part of ANSI Standard X3.159-1989, but... included for information only," and is not included with the ISO Standard.)

11.2: How can I get a copy of the Standard?

A: Copies are available in the United States from

```
                American National Standards Institute
                11 W. 42nd St., 13th floor
                New York, NY  10036  USA
                (+1) 212 642 4900

        and

                Global Engineering Documents
                15 Inverness Way E
                Englewood, CO  80112  USA
                (+1) 303 397 2715
                (800) 854 7179  (U.S. & Canada)

        In other countries, contact the appropriate national standards
        body, or ISO in Geneva at:

                ISO Sales
                Case Postale 56
                CH-1211 Geneve 20
                Switzerland

        (or see URL http://www.iso.ch or check the comp.std.internat
FAQ
        list, Standards.Faq).

        At the time of this writing, the cost is $130.00 from ANSI or
        $410.00 from Global.  Copies of the original X3.159 (including
        the Rationale) may still be available at $205.00 from ANSI or
        $162.50 from Global.  Note that ANSI derives revenues to
support
```

```
        its operations from the sale of printed standards, so
electronic
        copies are *not* available.

        In the U.S., it may be possible to get a copy of the original
        ANSI X3.159 (including the Rationale) as "FIPS PUB 160" from

                National Technical Information Service (NTIS)
                U.S. Department of Commerce
                Springfield, VA  22161
                703 487 4650
```

The mistitled _Annotated ANSI C Standard_, with annotations by Herbert Schildt, contains most of the text of ISO 9899; it is published by Osborne/McGraw-Hill, ISBN 0-07-881952-0, and sells in the U.S. for approximately $40. It has been suggested that the price differential between this work and the official standard reflects the value of the annotations: they are plagued by numerous errors and omissions, and a few pages of the Standard itself are missing. Many people on the net recommend ignoring the annotations entirely. A review of the annotations ("annotated annotations") by Clive Feather can be found on the web at http://www.lysator.liu.se/c/schildt.html .

The text of the Rationale (not the full Standard) can be obtained by anonymous ftp from ftp.uu.net (see question 18.16) in directory doc/standards/ansi/X3.159-1989, and is also available on the web at http://www.lysator.liu.se/c/rat/title.html . The Rationale has also been printed by Silicon Press, ISBN 0-929306-07-4.

11.3: My ANSI compiler complains about a mismatch when it sees

```
extern int func(float);

int func(x)
float x;
{ ...
```

A: You have mixed the new-style prototype declaration "extern int func(float);" with the old-style definition "int func(x) float x;". It is usually safe to mix the two styles (see question 11.4), but not in this case.

Old C (and ANSI C, in the absence of prototypes, and in variable- length argument lists; see question 15.2) "widens" certain arguments when they are passed to functions. floats are promoted to double, and characters and short integers are promoted to int. (For old-style function definitions, the values are automatically converted back to the corresponding narrower types within the body of the called function, if they are declared that way there.)

This problem can be fixed either by using new-style syntax consistently in the definition:

```
int func(float x) { ... }
```

or by changing the new-style prototype declaration to match the old-style definition:

```
extern int func(double);
```

(In this case, it would be clearest to change the old-style definition to use double as well, as long as the address of that parameter is not taken.)

It may also be safer to avoid "narrow" (char, short int, and float) function arguments and return types altogether.

See also question 1.25.

References: K&R1 Sec. A7.1 p. 186; K&R2 Sec. A7.3.2 p. 202; ANSI Sec. 3.3.2.2, Sec. 3.5.4.3; ISO Sec. 6.3.2.2, Sec. 6.5.4.3; Rationale Sec. 3.3.2.2, Sec. 3.5.4.3; H&S Sec. 9.2 pp. 265-7, Sec. 9.4 pp. 272-3.

11.4: Can you mix old-style and new-style function syntax?

A: Doing so is perfectly legal, as long as you're careful (see especially question 11.3). Note however that old-style syntax is marked as obsolescent, so official support for it may be removed some day.

References: ANSI Sec. 3.7.1, Sec. 3.9.5; ISO Sec. 6.7.1, Sec. 6.9.5; H&S Sec. 9.2.2 pp. 265-7, Sec. 9.2.5 pp. 269-70.

11.5: Why does the declaration

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

give me an obscure warning message about "struct x introduced in prototype scope"?

A: In a quirk of C's normal block scoping rules, a structure declared (or even mentioned) for the first time within a prototype cannot be compatible with other structures declared in the same source file (it goes out of scope at the end of the prototype).

To resolve the problem, precede the prototype with the vacuous- looking declaration

```
struct x;
```

which places an (incomplete) declaration of struct x at file scope, so that all following declarations involving struct x can at least be sure they're referring to the same struct x.

References: ANSI Sec. 3.1.2.1, Sec. 3.1.2.6, Sec. 3.5.2.3; ISO Sec. 6.1.2.1, Sec. 6.1.2.6, Sec. 6.5.2.3.

11.8: I don't understand why I can't use const values in initializers and array dimensions, as in

```
const int n = 5;
int a[n];
```

A: The const qualifier really means "read-only;" an object so qualified is a run-time object which cannot (normally) be assigned to. The value of a const-qualified object is therefore *not* a constant expression in the full sense of the term. (C is unlike C++ in this regard.) When you need a true compile- time constant, use a preprocessor #define.

References: ANSI Sec. 3.4; ISO Sec. 6.4; H&S Secs. 7.11.2,7.11.3 pp. 226-7.

11.9: What's the difference between "const char *p" and "char * const p"?

A: "char const *p" declares a pointer to a constant character (you can't change the character); "char * const p" declares a constant pointer to a (variable) character (i.e. you can't change the pointer).

Read these "inside out" to understand them; see also question 1.21.

References: ANSI Sec. 3.5.4.1 examples; ISO Sec. 6.5.4.1; Rationale Sec. 3.5.4.1; H&S Sec. 4.4.4 p. 81.

11.10: Why can't I pass a char ** to a function which expects a const char **?

A: You can use a pointer-to-T (for any type T) where a pointer-to- const-T is expected. However, the rule (an explicit exception) which permits slight mismatches in qualified pointer types is not applied recursively, but only at the top level.

You must use explicit casts (e.g. (const char **) in this case) when assigning (or passing) pointers which have qualifier mismatches at other than the first level of indirection.

References: ANSI Sec. 3.1.2.6, Sec. 3.3.16.1, Sec. 3.5.3; ISO Sec. 6.1.2.6, Sec. 6.3.16.1, Sec. 6.5.3; H&S Sec. 7.9.1 pp. 221- 2.

11.12: Can I declare main() as void, to shut off these annoying "main returns no value" messages?

A: No. main() must be declared as returning an int, and as taking either zero or two arguments, of the appropriate types. If you're calling exit() but still getting warnings, you may have to insert a redundant return statement (or use some kind of "not reached" directive, if available).

Declaring a function as void does not merely shut off or rearrange warnings: it may also result in a different function call/return sequence, incompatible with what the caller (in main's case, the C run-time startup code) expects.

(Note that this discussion of main() pertains only to "hosted" implementations; none of it applies to "freestanding" implementations, which may not even have main(). However, freestanding implementations are comparatively rare, and if you're using one, you probably know it. If you've never heard of the distinction, you're probably using a hosted implementation, and the above rules apply.)

References: ANSI Sec. 2.1.2.2.1, Sec. F.5.1; ISO Sec. 5.1.2.2.1, Sec. G.5.1; H&S Sec. 20.1 p. 416; CT&P Sec. 3.10 pp. 50-51.

11.13: But what about main's third argument, envp?

A: It's a non-standard (though common) extension. If you really need to access the environment in ways beyind what the standard getenv() function provides, though, the global variable environ is probably a better avenue (though it's equally non-standard).

References: ANSI Sec. F.5.1; ISO Sec. G.5.1; H&S Sec. 20.1 pp. 416-7.

11.14: I believe that declaring void main() can't fail, since I'm calling exit() instead of returning, and anyway my operating system ignores a program's exit/return status.

A: It doesn't matter whether main() returns or not, or whether anyone looks at the status; the problem is that when main() is misdeclared, its caller (the runtime startup code) may not even be able to *call* it correctly (due to the potential clash of calling conventions; see question 11.12). Your operating system may ignore the exit status, and void main() may work for you, but it is not portable and not correct.

11.15: The book I've been using, _C Programing for the Compleat Idiot_, always uses void main().

A: Perhaps its author counts himself among the target audience. Many books unaccountably use void main() in examples. They're wrong.

11.16: Is exit(status) truly equivalent to returning the same status from main()?

A: Yes and no. The Standard says that they are equivalent. However, a few older, nonconforming systems may have problems with one or the other form. Also, a return from main() cannot be expected to work if data local to main() might be needed during cleanup; see also question 16.4. (Finally, the two forms are obviously not equivalent in a recursive call to main().)

References: K&R2 Sec. 7.6 pp. 163-4; ANSI Sec. 2.1.2.2.3; ISO Sec. 5.1.2.2.3.

11.17: I'm trying to use the ANSI "stringizing" preprocessing operator `#' to insert the value of a symbolic constant into a message, but it keeps stringizing the macro's name rather than its value.

A: You can use something like the following two-step procedure to force a macro to be expanded as well as stringized:

```
#define Str(x) #x
#define Xstr(x) Str(x)
#define OP plus
char *opname = Xstr(OP);
```

This code sets opname to "plus" rather than "OP".

An equivalent circumlocution is necessary with the token-pasting operator ## when the values (rather than the names) of two macros are to be concatenated.

References: ANSI Sec. 3.8.3.2, Sec. 3.8.3.5 example; ISO Sec. 6.8.3.2, Sec. 6.8.3.5.

11.18: What does the message "warning: macro replacement within a string literal" mean?

A: Some pre-ANSI compilers/preprocessors interpreted macro definitions like

```
#define TRACE(var, fmt) printf("TRACE: var = fmt\n",
var)
```

such that invocations like

```
TRACE(i, %d);
```

were expanded as

```
printf("TRACE: i = %d\n", i);
```

In other words, macro parameters were expanded even inside string literals and character constants.

Macro expansion is *not* defined in this way by K&R or by Standard C. When you do want to turn macro arguments into strings, you can use the new # preprocessing operator, along with string literal concatenation (another new ANSI feature):

```
#define TRACE(var, fmt) \
        printf("TRACE: " #var " = " #fmt "\n", var)
```

See also question 11.17 above.

References: H&S Sec. 3.3.8 p. 51.

11.19: I'm getting strange syntax errors inside lines I've #ifdeffed out.

A: Under ANSI C, the text inside a "turned off" #if, #ifdef, or #ifndef must still consist of "valid preprocessing tokens." This means that there must be no newlines inside quotes, and no unterminated comments or quotes (note particularly that an apostrophe within a contracted word looks like the beginning of a character constant). Therefore, natural-language comments and pseudocode should always be written between the "official" comment delimiters /* and */. (But see question 20.20, and also 10.25.)

References: ANSI Sec. 2.1.1.2, Sec. 3.1; ISO Sec. 5.1.1.2, Sec. 6.1; H&S Sec. 3.2 p. 40.

11.20: What are #pragmas and what are they good for?

A: The #pragma directive provides a single, well-defined "escape hatch" which can be used for all sorts of implementation- specific controls and extensions: source listing control, structure packing, warning suppression (like lint's old /* NOTREACHED */ comments), etc.

References: ANSI Sec. 3.8.6; ISO Sec. 6.8.6; H&S Sec. 3.7 p. 61.

11.21: What does "#pragma once" mean? I found it in some header files.

A: It is an extension implemented by some preprocessors to help make header files idempotent; it is essentially equivalent to the #ifndef trick mentioned in question 10.7.

11.22: Is char a[3] = "abc"; legal? What does it mean?

A: It is legal in ANSI C (and perhaps in a few pre-ANSI systems), though useful only in rare circumstances. It declares an array of size three, initialized with the three characters 'a', 'b', and 'c', *without* the usual terminating '\0' character. The array is therefore not a true C string and cannot be used with strcpy, printf %s, etc.

Most of the time, you should let the compiler count the initializers when initializing arrays (in the case of the initializer "abc", of course, the computed size will be 4).

References: ANSI Sec. 3.5.7; ISO Sec. 6.5.7; H&S Sec. 4.6.4 p. 98.

11.24: Why can't I perform arithmetic on a void * pointer?

A: The compiler doesn't know the size of the pointed-to objects. Before performing arithmetic, convert the pointer either to char * or to the pointer type you're trying to manipulate (but see also question 4.5).

References: ANSI Sec. 3.1.2.5, Sec. 3.3.6; ISO Sec. 6.1.2.5, Sec. 6.3.6; H&S Sec. 7.6.2 p. 204.

11.25: What's the difference between memcpy() and memmove()?

A: memmove() offers guaranteed behavior if the source and destination arguments overlap. memcpy() makes no such guarantee, and may therefore be more efficiently implementable. When in doubt, it's safer to use memmove().

References: K&R2 Sec. B3 p. 250; ANSI Sec. 4.11.2.1, Sec. 4.11.2.2; ISO Sec. 7.11.2.1, Sec. 7.11.2.2; Rationale Sec. 4.11.2; H&S Sec. 14.3 pp. 341-2; PCS Sec. 11 pp. 165-6.

11.26: What should malloc(0) do? Return a null pointer or a pointer to 0 bytes?

A: The ANSI/ISO Standard says that it may do either; the behavior is implementation-defined (see question 11.33).

References: ANSI Sec. 4.10.3; ISO Sec. 7.10.3; PCS Sec. 16.1 p. 386.

11.27: Why does the ANSI Standard not guarantee more than six case-insensitive characters of external identifier significance?

A: The problem is older linkers which are under the control of neither the ANSI/ISO Standard nor the C compiler developers on the systems which have them. The limitation is only that identifiers be *significant* in the first six characters, not that they be restricted to six characters in length. This limitation is annoying, but certainly not unbearable, and is marked in the Standard as "obsolescent," i.e. a future revision will likely relax it.

This concession to current, restrictive linkers really had to be made, no matter how vehemently some people oppose it. (The Rationale notes that its retention was "most painful.") If you disagree, or have thought of a trick by which a compiler burdened with a restrictive linker could present the C programmer with the appearance of more significance in external identifiers, read the excellently-worded section 3.1.2 in the X3.159 Rationale (see question 11.1), which discusses several such schemes and explains why they could not be mandated.

References: ANSI Sec. 3.1.2, Sec. 3.9.1; ISO Sec. 6.1.2, Sec. 6.9.1; Rationale Sec. 3.1.2; H&S Sec. 2.5 pp. 22-3.

11.29: My compiler is rejecting the simplest possible test programs, with all kinds of syntax errors.

A: Perhaps it is a pre-ANSI compiler, unable to accept function prototypes and the like.

See also questions 1.31, 10.9, and 11.30.

11.30: Why are some ANSI/ISO Standard library routines showing up as undefined, even though I've got an ANSI compiler?

A: It's possible to have a compiler available which accepts ANSI syntax, but not to have ANSI-compatible header files or run-time libraries installed. (In fact, this situation is rather common when using a non-vendor-supplied compiler such as gcc.) See also questions 11.29, 13.25, and 13.26.

11.31: Does anyone have a tool for converting old-style C programs to ANSI C, or vice versa, or for automatically generating prototypes?

A: Two programs, protoize and unprotoize, convert back and forth between prototyped and "old style" function definitions and declarations. (These programs do *not* handle full-blown translation between "Classic" C and ANSI C.) These programs are part of the FSF's GNU C compiler distribution; see question 18.3.

The unproto program (/pub/unix/unproto5.shar.Z on ftp.win.tue.nl) is a filter which sits between the preprocessor and the next compiler pass, converting most of ANSI C to traditional C on-the-fly.

The GNU GhostScript package comes with a little program called ansi2knr.

Before converting ANSI C back to old-style, beware that such a conversion cannot always be made both safely and automatically. ANSI C introduces new features and complexities not found in K&R C. You'll especially need to be careful of prototyped function calls; you'll probably need to insert explicit casts. See also questions 11.3 and 11.29.

Several prototype generators exist, many as modifications to lint. A program called CPROTO was posted to comp.sources.misc in March, 1992. There is another program called "cextract." Many vendors supply simple utilities like these with their compilers. See also question 18.16. (But be careful when generating prototypes for old functions with "narrow" parameters; see question 11.3.)

Finally, are you sure you really need to convert lots of old code to ANSI C? The old-style function syntax is still acceptable, and a hasty conversion can easily introduce bugs. (See question 11.3.)

11.32: Why won't the Frobozz Magic C Compiler, which claims to be ANSI compliant, accept this code? I know that the code is ANSI, because gcc accepts it.

A: Many compilers support a few non-Standard extensions, gcc more so than most. Are you sure that the code being rejected doesn't rely on such an extension? It is usually a bad idea to perform experiments with a particular compiler to determine properties of a language; the applicable standard may permit variations, or the compiler may be wrong. See also question 11.35.

11.33: People seem to make a point of distinguishing between implementation-defined, unspecified, and undefined behavior. What's the difference?

A: Briefly: implementation-defined means that an implementation must choose some behavior and document it. Unspecified means that an implementation should choose some behavior, but need not document it. Undefined means that absolutely anything might happen. In no case does the Standard impose requirements; in the first two cases it occasionally suggests (and may require a choice from among) a small set of likely behaviors.

Note that since the Standard imposes *no* requirements on the behavior of a compiler faced with an instance of undefined behavior, the compiler can do absolutely anything. In particular, there is no guarantee that the rest of the program will perform normally. It's perilous to think that you can tolerate undefined behavior in a program; see question 3.2 for a relatively simple example.

If you're interested in writing portable code, you can ignore the distinctions, as you'll want to avoid code that depends on any of the three behaviors.

See also questions 3.9, and 11.34.

References: ANSI Sec. 1.6; ISO Sec. 3.10, Sec. 3.16, Sec. 3.17; Rationale Sec. 1.6.

11.34: I'm appalled that the ANSI Standard leaves so many issues undefined. Isn't a Standard's whole job to standardize these things?

A: It has always been a characteristic of C that certain constructs behaved in whatever way a particular compiler or a particular piece of hardware chose to implement them. This deliberate imprecision often allows compilers to generate more efficient code for common cases, without having to burden all programs with extra code to assure well-defined behavior of cases deemed to be less reasonable. Therefore, the Standard is simply codifying existing practice.

A programming language standard can be thought of as a treaty between the language user and the compiler implementor. Parts of that treaty consist of features which the compiler implementor agrees to provide, and which the user may assume will be available. Other parts, however, consist of rules which the user agrees to follow and which the implementor may assume will be followed. As long as both sides uphold their guarantees, programs have a fighting chance of working correctly. If *either* side reneges on any of its commitments, nothing is guaranteed to work.

See also question 11.35.

References: Rationale Sec. 1.1.

11.35: People keep saying that the behavior of i = i++ is undefined, but I just tried it on an ANSI-conforming compiler, and got the results I expected.

A: A compiler may do anything it likes when faced with undefined behavior (and, within limits, with implementation-defined and unspecified behavior), including doing what you expect. It's unwise to depend on it, though. See also questions 11.32, 11.33, and 11.34.

Stdio

12.1: What's wrong with this code?

```
char c;
while((c = getchar()) != EOF) ...
```

A: For one thing, the variable to hold getchar's return value must be an int. getchar() can return all possible character values, as well as EOF. By passing getchar's return value through a char, either a normal character might be misinterpreted as EOF, or the EOF might be altered (particularly if type char is unsigned) and so never seen.

References: K&R1 Sec. 1.5 p. 14; K&R2 Sec. 1.5.1 p. 16; ANSI Sec. 3.1.2.5, Sec. 4.9.1, Sec. 4.9.7.5; ISO Sec. 6.1.2.5, Sec. 7.9.1, Sec. 7.9.7.5; H&S Sec. 5.1.3 p. 116, Sec. 15.1, Sec. 15.6; CT&P Sec. 5.1 p. 70; PCS Sec. 11 p. 157.

12.2: Why does the code

```
while(!feof(infp)) {
        fgets(buf, MAXLINE, infp);
        fputs(buf, outfp);
}
```

copy the last line twice?

A: In C, EOF is only indicated *after* an input routine has tried to read, and has reached end-of-file. (In other words, C's I/O is not like Pascal's.) Usually, you should just check the return value of the input routine (fgets() in this case); often, you don't need to use feof() at all.

References: K&R2 Sec. 7.6 p. 164; ANSI Sec. 4.9.3, Sec. 4.9.7.1, Sec. 4.9.10.2; ISO Sec. 7.9.3, Sec. 7.9.7.1, Sec. 7.9.10.2; H&S Sec. 15.14 p. 382.

12.4: My program's prompts and intermediate output don't always show up on the screen, especially when I pipe the output through another program.

A: It's best to use an explicit fflush(stdout) whenever output should definitely be visible. Several mechanisms attempt to perform the fflush() for you, at the "right time," but they tend to apply only when stdout is an interactive terminal. (See also question 12.24.)

References: ANSI Sec. 4.9.5.2; ISO Sec. 7.9.5.2.

12.5: How can I read one character at a time, without waiting for the RETURN key?

A: See question 19.1.

12.6: How can I print a '%' character in a printf format string? I tried \%, but it didn't work.

A: Simply double the percent sign: %% .

\% can't work, because the backslash \ is the *compiler's* escape character, while here our problem is that the % is printf's escape character.

See also question 19.17.

References: K&R1 Sec. 7.3 p. 147; K&R2 Sec. 7.2 p. 154; ANSI Sec. 4.9.6.1; ISO Sec. 7.9.6.1.

12.9: Someone told me it was wrong to use %lf with printf(). How can printf() use %f for type double, if scanf() requires %lf?

A: It's true that printf's %f specifier works with both float and double arguments. Due to the "default argument promotions" (which apply in variable-length argument lists such as printf's, whether or not prototypes are in scope), values of type float are promoted to double, and printf() therefore sees only doubles. See also questions 12.13 and 15.2.

References: K&R1 Sec. 7.3 pp. 145-47, Sec. 7.4 pp. 147-50; K&R2 Sec. 7.2 pp. 153-44, Sec. 7.4 pp. 157-59; ANSI Sec. 4.9.6.1, Sec. 4.9.6.2; ISO Sec. 7.9.6.1, Sec. 7.9.6.2; H&S Sec. 15.8 pp. 357-64, Sec. 15.11 pp. 366-78; CT&P Sec. A.1 pp. 121-33.

12.10: How can I implement a variable field width with printf? That is, instead of %8d, I want the width to be specified at run time.

A: printf("%*d", width, n) will do just what you want. See also question 12.15.

References: K&R1 Sec. 7.3; K&R2 Sec. 7.2; ANSI Sec. 4.9.6.1; ISO Sec. 7.9.6.1; H&S Sec. 15.11.6; CT&P Sec. A.1.

12.11: How can I print numbers with commas separating the thousands? What about currency formatted numbers?

A: The routines in begin to provide some support for these operations, but there is no standard routine for doing either task. (The only thing printf() does in response to a custom locale setting is to change its decimal-point character.)

References: ANSI Sec. 4.4; ISO Sec. 7.4; H&S Sec. 11.6 pp. 301-4.

12.12: Why doesn't the call scanf("%d", i) work?

A: The arguments you pass to scanf() must always be pointers. To fix the fragment above, change it to scanf("%d", &i) .

12.13: Why doesn't this code:

```
double d;
scanf("%f", &d);
```

work?

A: Unlike printf(), scanf() uses %lf for values of type double, and %f for float. See also question 12.9.

12.15: How can I specify a variable width in a scanf() format string?

A: You can't; an asterisk in a scanf() format string means to suppress assignment. You may be able to use ANSI stringizing and string concatenation to accomplish about the same thing, or to construct a scanf format string on-the-fly.

12.17: When I read numbers from the keyboard with scanf "%d\n", it seems to hang until I type one extra line of input.

A: Perhaps surprisingly, \n in a scanf format string does *not* mean to expect a newline, but rather to read and discard characters as long as each is a whitespace character. See also question 12.20.

References: K&R2 Sec. B1.3 pp. 245-6; ANSI Sec. 4.9.6.2; ISO Sec. 7.9.6.2; H&S Sec. 15.8 pp. 357-64.

12.18: I'm reading a number with scanf %d and then a string with gets(), but the compiler seems to be skipping the call to gets()!

A: scanf %d won't consume a trailing newline. If the input number is immediately followed by a newline, that newline will immediately satisfy the gets().

As a general rule, you shouldn't try to interlace calls to scanf() with calls to gets() (or any other input routines); scanf's peculiar treatment of newlines almost always leads to trouble. Either use scanf() to read everything or nothing.

See also questions 12.20 and 12.23.

References: ANSI Sec. 4.9.6.2; ISO Sec. 7.9.6.2; H&S Sec. 15.8 pp. 357-64.

12.19: I figured I could use scanf() more safely if I checked its return value to make sure that the user typed the numeric values I expect, but sometimes it seems to go into an infinite loop.

A: When scanf() is attempting to convert numbers, any non-numeric characters it encounters terminate the conversion *and are left on the input stream*. Therefore, unless some other steps are taken, unexpected non-numeric input "jams" scanf() again and again: scanf() never gets past the bad character(s) to encounter later, valid data. If the user types a character like `x' in response to a numeric scanf format such as %d or %f, code that simply re-prompts and retries the same scanf() call will immediately reencounter the same `x'.

See also question 12.20.

References: ANSI Sec. 4.9.6.2; ISO Sec. 7.9.6.2; H&S Sec. 15.8 pp. 357-64.

12.20: Why does everyone say not to use scanf()? What should I use instead?

A: scanf() has a number of problems -- see questions 12.17, 12.18, and 12.19. Also, its %s format has the same problem that gets() has (see question 12.23) -- it's hard to guarantee that the receiving buffer won't overflow.

More generally, scanf() is designed for relatively structured, formatted input (its name is in fact derived from "scan formatted"). If you pay attention, it will tell you whether it succeeded or failed, but it can tell you only approximately where it failed, and not at all how or why. It's nearly impossible to do decent error recovery with scanf(); usually it's far easier to read entire lines (with fgets() or the like), then interpret them, either using sscanf() or some other techniques. (Routines like strtol(), strtok(), and atoi() are often useful; see also question 13.6.) If you do use sscanf(), don't forget to check the return value to make sure that the expected number of items were found.

References: K&R2 Sec. 7.4 p. 159.

12.21: How can I tell how much destination buffer space I'll need for an arbitrary sprintf call? How can I avoid overflowing the destination buffer with sprintf()?

A: There are not (yet) any good answers to either of these excellent questions, and this represents perhaps the biggest deficiency in the traditional stdio library.

When the format string being used with sprintf() is known and relatively simple, you can usually predict a buffer size in an ad-hoc way. If the format consists of one or two %s's, you can count the fixed characters in the format string yourself (or let sizeof count them for you) and add in the result of calling strlen() on the string(s) to be inserted. You can conservatively estimate the size that %d will expand to with code like:

```
#include
char buf[(sizeof(int) * CHAR_BIT + 2) / 3 + 1 + 1];
sprintf(buf, "%d", n);
```

(This code computes the number of characters required for a base- 8 representation of a number; a base-10 expansion is guaranteed to take as much room or less.)

When the format string is more complicated, or is not even known until run time, predicting the buffer size becomes as difficult as reimplementing sprintf(), and correspondingly error-prone (and inadvisable). A last-ditch technique which is sometimes suggested is to use fprintf() to print the same text to a bit bucket or temporary file, and then to look at fprintf's return value or the size of the file (but see question 19.12).

If there's any chance that the buffer might not be big enough, you won't want to call sprintf() without some guarantee that the buffer will not overflow and overwrite some other part of memory. Several stdio's (including GNU and 4.4bsd) provide the obvious snprintf() function, which can be used like this:

```
snprintf(buf, bufsize, "You typed \"%s\"", answer);
```

and we can hope that a future revision of the ANSI/ISO C Standard will include this function.

12.23: Why does everyone say not to use gets()?

A: Unlike fgets(), gets() cannot be told the size of the buffer it's to read into, so it cannot be prevented from overflowing that buffer. As a general rule, always use fgets(). See question 7.1 for a code fragment illustrating the replacement of gets() with fgets().

References: Rationale Sec. 4.9.7.2; H&S Sec. 15.7 p. 356.

12.24: Why does errno contain ENOTTY after a call to printf()?

A: Many implementations of the stdio package adjust their behavior slightly if stdout is a terminal. To make the determination, these implementations perform some operation which happens to fail (with ENOTTY) if stdout is not a terminal. Although the output operation goes on to complete successfully, errno still contains ENOTTY. (Note that it is only meaningful for a program to inspect the contents of errno after an error has been reported.)

References: ANSI Sec. 4.1.3, Sec. 4.9.10.3; ISO Sec. 7.1.4, Sec. 7.9.10.3; CT&P Sec. 5.4 p. 73; PCS Sec. 14 p. 254.

12.25: What's the difference between fgetpos/fsetpos and ftell/fseek? What are fgetpos() and fsetpos() good for?

A: fgetpos() and fsetpos() use a special typedef, fpos_t, for representing offsets (positions) in a file. The type behind this typedef, if chosen appropriately, can represent arbitrarily large offsets, allowing fgetpos() and fsetpos() to be used with arbitrarily huge files. ftell() and fseek(), on the other hand, use long int, and are therefore limited to offsets which can be represented in a long int. See also question 1.4.

References: K&R2 Sec. B1.6 p. 248; ANSI Sec. 4.9.1, Secs. 4.9.9.1,4.9.9.3; ISO Sec. 7.9.1, Secs. 7.9.9.1,7.9.9.3; H&S Sec. 15.5 p. 252.

12.26: How can I flush pending input so that a user's typeahead isn't read at the next prompt? Will fflush(stdin) work?

A: fflush() is defined only for output streams. Since its definition of "flush" is to complete the writing of buffered characters (not to discard them), discarding unread input would not be an analogous meaning for fflush on input streams.

There is no standard way to discard unread characters from a stdio input stream, nor would such a way be sufficient unread characters can also accumulate in other, OS-level input buffers.

References: ANSI Sec. 4.9.5.2; ISO Sec. 7.9.5.2; H&S Sec. 15.2.

12.30: I'm trying to update a file in place, by using fopen mode "r+", reading a certain string, and writing back a modified string, but it's not working.

A: Be sure to call fseek before you write, both to seek back to the beginning of the string you're trying to overwrite, and because an fseek or fflush is always required between reading and writing in the read/write "+" modes. Also, remember that you can only overwrite characters with the same number of replacement characters; see also question 19.14.

References: ANSI Sec. 4.9.5.3; ISO Sec. 7.9.5.3.

12.33: How can I redirect stdin or stdout to a file from within a program?

A: Use freopen() (but see question 12.34 below).

References: ANSI Sec. 4.9.5.4; ISO Sec. 7.9.5.4; H&S Sec. 15.2.

12.34: Once I've used freopen(), how can I get the original stdout (or stdin) back?

A: There isn't a good way. If you need to switch back, the best solution is not to have used freopen() in the first place. Try using your own explicit output (or input) stream variable, which you can reassign at will, while leaving the original stdout (or stdin) undisturbed.

12.38: How can I read a binary data file properly? I'm occasionally seeing 0x0a and 0x0d values getting garbled, and it seems to hit EOF prematurely if the data contains the value 0x1a.

A: When you're reading a binary data file, you should specify "rb" mode when calling fopen(), to make sure that text file translations do not occur. Similarly, when writing binary data files, use "wb".

Note that the text/binary distinction is made when you open the file: once a file is open, it doesn't matter which I/O calls you use on it. See also question 20.5.

References: ANSI Sec. 4.9.5.3; ISO Sec. 7.9.5.3; H&S Sec. 15.2.1 p. 348.

Library Functions

13.1: How can I convert numbers to strings (the opposite of atoi)? Is there an itoa function?

A: Just use sprintf(). (Don't worry that sprintf() may be overkill, potentially wasting run time or code space; it works well in practice.) See the examples in the answer to question 7.5; see also question 12.21.

You can obviously use sprintf() to convert long or floating- point numbers to strings as well (using %ld or %f).

References: K&R1 Sec. 3.6 p. 60; K&R2 Sec. 3.6 p. 64.

13.2: Why does strncpy() not always place a '\0' terminator in the destination string?

A: strncpy() was first designed to handle a now-obsolete data structure, the fixed-length, not-necessarily-\0-terminated "string." (A related quirk of strncpy's is that it pads short strings with multiple \0's, out to the specified length.) strncpy() is admittedly a bit cumbersome to use in other contexts, since you must often append a '\0' to the destination string by hand. You can get around the problem by using strncat() instead of strncpy(): if the destination string starts out empty, strncat() does what you probably wanted strncpy() to do. Another possibility is sprintf(dest, "%.*s", n, source) .

When arbitrary bytes (as opposed to strings) are being copied, memcpy() is usually a more appropriate routine to use than strncpy().

13.5: Why do some versions of toupper() act strangely if given an upper-case letter? Why does some code call islower() before toupper()?

A: Older versions of toupper() and tolower() did not always work correctly on arguments which did not need converting (i.e. on digits or punctuation or letters already of the desired case). In ANSI/ISO Standard C, these functions are guaranteed to work appropriately on all character arguments.

References: ANSI Sec. 4.3.2; ISO Sec. 7.3.2; H&S Sec. 12.9 pp. 320-1; PCS p. 182.

13.6: How can I split up a string into whitespace-separated fields? How can I duplicate the process by which main() is handed argc and argv?

A: The only Standard routine available for this kind of "tokenizing" is strtok, although it can be tricky to use and it may not do everything you want it to. (For instance, it does not handle quoting.)

References: K&R2 Sec. B3 p. 250; ANSI Sec. 4.11.5.8; ISO Sec. 7.11.5.8; H&S Sec. 13.7 pp. 333-4; PCS p. 178.

13.7: I need some code to do regular expression and wildcard matching.

A: Make sure you recognize the difference between classic regular expressions (variants of which are used in such Unix utilities as ed and grep), and filename wildcards (variants of which are used by most operating systems).

There are a number of packages available for matching regular expressions. Most packages use a pair of functions, one for "compiling" the regular expression, and one for "executing" it (i.e. matching strings against it). Look for header files named or , and functions called regcmp()/regex(), regcomp()/regexec(), or re_comp()/re_exec(). (These functions may exist in a separate regexp library.) A popular, freely- redistributable regexp package by Henry Spencer is available from ftp.cs.toronto.edu in pub/regexp.shar.Z or in several other archives. The GNU project has a package called rx. See also question 18.16.

Filename wildcard matching (sometimes called "globbing") is done in a variety of ways on different systems. On Unix, wildcards are automatically expanded by the shell before a process is invoked, so programs rarely have to worry about them explicitly. Under MS-DOS compilers, there is often a special object file which can be linked in to a program to expand wildcards while argv is being built. Several systems (including MS-DOS and

VMS) provide system services for listing or opening files specified by wildcards. Check your compiler/library documentation.

13.8: I'm trying to sort an array of strings with qsort(), using strcmp() as the comparison function, but it's not working.

A: By "array of strings" you probably mean "array of pointers to char." The arguments to qsort's comparison function are pointers to the objects being sorted, in this case, pointers to pointers to char. strcmp(), however, accepts simple pointers to char. Therefore, strcmp() can't be used directly. Write an intermediate comparison function like this:

```
                /* compare strings via pointers */
                int pstrcmp(const void *p1, const void *p2)
                {
                        return strcmp(*(char * const *)p1, *(char *
const *)p2);
                }
```

The comparison function's arguments are expressed as "generic pointers," const void *. They are converted back to what they "really are" (char **) and dereferenced, yielding char *'s which can be passed to strcmp(). (Under a pre-ANSI compiler, declare the pointer parameters as char * instead of void *, and drop the consts.)

(Don't be misled by the discussion in K&R2 Sec. 5.11 pp. 119-20, which is not discussing the Standard library's qsort).

References: ANSI Sec. 4.10.5.2; ISO Sec. 7.10.5.2; H&S Sec. 20.5 p. 419.

13.9: Now I'm trying to sort an array of structures with qsort(). My comparison function takes pointers to structures, but the compiler complains that the function is of the wrong type for qsort(). How can I cast the function pointer to shut off the warning?

A: The conversions must be in the comparison function, which must be declared as accepting "generic pointers" (const void *) as discussed in question 13.8 above. The comparison function might look like

```
                int mystructcmp(const void *p1, const void *p2)
                {
                        const struct mystruct *sp1 = p1;
                        const struct mystruct *sp2 = p2;
                        /* now compare sp1->whatever and sp2-> ... */
```

(The conversions from generic pointers to struct mystruct pointers happen in the initializations sp1 = p1 and sp2 = p2; the compiler performs the conversions implicitly since p1 and p2 are void pointers. Explicit casts, and char * pointers, would be required under a pre-ANSI compiler. See also question 7.7.)

If, on the other hand, you're sorting pointers to structures, you'll need indirection, as in question 13.8:

```
sp1 = *(struct mystruct **)p1 .
```

In general, it is a bad idea to insert casts just to "shut the compiler up." Compiler warnings are usually trying to tell you something, and unless you really know what you're doing, you ignore or muzzle them at your peril. See also question 4.9.

References: ANSI Sec. 4.10.5.2; ISO Sec. 7.10.5.2; H&S Sec. 20.5 p. 419.

13.10: How can I sort a linked list?

A: Sometimes it's easier to keep the list in order as you build it (or perhaps to use a tree instead). Algorithms like insertion sort and merge sort lend themselves ideally to use with linked lists. If you want to use a standard library function, you can allocate a temporary array of pointers, fill it in with pointers to all your list nodes, call qsort(), and finally rebuild the list pointers based on the sorted array.

References: Knuth Sec. 5.2.1 pp. 80-102, Sec. 5.2.4 pp. 159-168; Sedgewick Sec. 8 pp. 98-100, Sec. 12 pp. 163-175.

13.11: How can I sort more data than will fit in memory?

A: You want an "external sort," which you can read about in Knuth, Volume 3. The basic idea is to sort the data in chunks (as much as will fit in memory at one time), write each sorted chunk to a temporary file, and then merge the files. Your operating system may provide a general-purpose sort utility, and if so, you can try invoking it from within your program: see questions 19.27 and 19.30.

References: Knuth Sec. 5.4 pp. 247-378; Sedgewick Sec. 13 pp. 177-187.

13.12: How can I get the current date or time of day in a C program?

A: Just use the time, ctime, and/or localtime functions. (These routines have been around for years, and are in the ANSI standard.) Here is a simple example:

```
#include
#include

main()
{
        time_t now;
        time(&now);
        printf("It's %.24s.\n", ctime(&now));
        return 0;
}
```

References: K&R2 Sec. B10 pp. 255-7; ANSI Sec. 4.12; ISO Sec. 7.12; H&S Sec. 18.

13.13: I know that the library routine localtime() will convert a time_t into a broken-down struct tm, and that ctime() will convert a time_t to a printable string. How can I perform the inverse operations of converting a struct tm or a string into a time_t?

A: ANSI C specifies a library routine, mktime(), which converts a struct tm to a time_t.

Converting a string to a time_t is harder, because of the wide variety of date and time formats which might be encountered. Some systems provide a strptime() function, which is basically the inverse of strftime(). Other popular routines are partime() (widely distributed with the RCS package) and getdate() (and a few others, from the C news distribution). See question 18.16.

References: K&R2 Sec. B10 p. 256; ANSI Sec. 4.12.2.3; ISO Sec. 7.12.2.3; H&S Sec. 18.4 pp. 401-2.

13.14: How can I add N days to a date? How can I find the difference between two dates?

A: The ANSI/ISO Standard C mktime() and difftime() functions provide some support for both problems. mktime() accepts non- normalized dates, so it is straightforward to take a filled-in struct tm, add or subtract from the tm_mday field, and call mktime() to normalize the year, month, and day fields (and incidentally convert to a time_t value). difftime() computes the difference, in seconds, between two time_t values; mktime() can be used to compute time_t values for two dates to be subtracted.

These solutions are only guaranteed to work correctly for dates in the range which can be represented as time_t's. The tm_mday field is an int, so day offsets of more than 32,736 or so may cause overflow. Note also that at daylight saving time changeovers, local days are not 24 hours long.

Another approach to both problems is to use "Julian day" numbers. Implementations of Julian day routines can be found in the file JULCAL10.ZIP from the Simtel/Oakland archives (see question 18.16) and the "Date conversions" article mentioned in the References.

See also questions 13.13, 20.31, and 20.32.

References: K&R2 Sec. B10 p. 256; ANSI Secs. 4.12.2.2,4.12.2.3; ISO Secs. 7.12.2.2,7.12.2.3; H&S Secs. 18.4,18.5 pp. 401-2; David Burki, "Date Conversions".

13.15: I need a random number generator.

A: The Standard C library has one: rand(). The implementation on your system may not be perfect, but writing a better one isn't necessarily easy, either.

If you do find yourself needing to implement your own random number generator, there is plenty of literature out there; see the References. There are also any number of packages on the net: look for r250, RANLIB, and FSULTRA (see question 18.16).

References: K&R2 Sec. 2.7 p. 46, Sec. 7.8.7 p. 168; ANSI Sec. 4.10.2.1; ISO Sec. 7.10.2.1; H&S Sec. 17.7 p. 393; PCS Sec. 11 p. 172; Knuth Vol. 2 Chap. 3 pp. 1-177; Park and Miller, "Random Number Generators: Good Ones are hard to Find".

13.16: How can I get random integers in a certain range?

A: The obvious way,

```
rand() % N              /* POOR */
```

(which tries to return numbers from 0 to N-1) is poor, because the low-order bits of many random number generators are distressingly *non*-random. (See question 13.18.) A better method is something like

```
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
```

If you're worried about using floating point, you could use

```
rand() / (RAND_MAX / N + 1)
```

Both methods obviously require knowing RAND_MAX (which ANSI #defines in), and assume that N is much less than

```
RAND_MAX.
```

(Note, by the way, that RAND_MAX is a *constant* telling you what the fixed range of the C library rand() function is. You cannot set RAND_MAX to some other value, and there is no way of requesting that rand() return numbers in some other range.)

If you're starting with a random number generator which returns floating-point values between 0 and 1, all you have to do to get integers from 0 to N-1 is multiply the output of that generator by N.

References: K&R2 Sec. 7.8.7 p. 168; PCS Sec. 11 p. 172.

13.17: Each time I run my program, I get the same sequence of numbers back from rand().

A: You can call srand() to seed the pseudo-random number generator with a truly random initial value. Popular seed values are the time of day, or the elapsed time before the user presses a key (although keypress times are hard to determine portably; see question 19.37). (Note also that it's rarely useful to call srand() more than once during a run of a program; in particular, don't try calling srand() before each call to rand(), in an attempt to get "really random" numbers.)

References: K&R2 Sec. 7.8.7 p. 168; ANSI Sec. 4.10.2.2; ISO Sec. 7.10.2.2; H&S Sec. 17.7 p. 393.

13.18: I need a random true/false value, so I'm just taking rand() % 2, but it's alternating 0, 1, 0, 1, 0...

A: Poor pseudorandom number generators (such as the ones unfortunately supplied with some systems) are not very random in the low-order bits. Try using the higher-order bits: see question 13.16.

References: Knuth Sec. 3.2.1.1 pp. 12-14.

13.20: How can I generate random numbers with a normal or Gaussian distribution?

A: Here is one method, by Box and Muller, and recommended by Knuth:

```
                #include
                #include

                double gaussrand()
                {
                        static double V1, V2, S;
                        static int phase = 0;
                        double X;

                        if(phase == 0) {
                                do {
                                        double U1 = (double)rand() /
RAND_MAX;
                                        double U2 = (double)rand() /
RAND_MAX;

                                        V1 = 2 * U1 - 1;
                                        V2 = 2 * U2 - 1;
                                        S = V1 * V1 + V2 * V2;
                                        } while(S >= 1 || S == 0);

                                X = V1 * sqrt(-2 * log(S) / S);
                        } else
                                X = V2 * sqrt(-2 * log(S) / S);

                        phase = 1 - phase;

                        return X;
                }
```

See the extended versions of this list (see question 20.40) for other ideas.

References: Knuth Sec. 3.4.1 p. 117; Box and Muller, "A Note on the Generation of Random Normal Deviates"; Press et al., _Numerical Recipes in C_ Sec. 7.2 pp. 288-290.

13.24: I'm trying to port this old program. Why do I get "undefined external" errors for:	A: Those routines are variously obsolete; you should instead:
index?	use strchr.
rindex?	use strrchr.
bcopy?	use memmove, after interchanging the first and second arguments (see also question 11.25).
bcmp?	use memcmp.
bzero?	use memset, with a second argument of 0.

Contrariwise, if you're using an older system which is missing the functions in the second column, you may be able to implement them in terms of, or substitute, the functions in the first.

References: PCS Sec. 11.

13.25: I keep getting errors due to library functions being undefined, but I'm #including all the right header files.

A: In some cases (especially if the functions are nonstandard) you may have to explicitly ask for the correct libraries to be searched when you link the program. See also questions 11.30, 13.26, and 14.3.

13.26: I'm still getting errors due to library functions being undefined, even though I'm explicitly requesting the right libraries while linking.

A: Many linkers make one pass over the list of object files and libraries you specify, and extract from libraries only those modules which satisfy references which have so far come up as undefined. Therefore, the order in which libraries are listed with respect to object files (and each other) is significant; usually, you want to search the libraries last. (For example, under Unix, put any -l options towards the end of the command line.) See also question 13.28.

13.28: What does it mean when the linker says that _end is undefined?

A: That message is a quirk of the old Unix linkers. You only get an error about _end being undefined when other things are undefined, too -- fix the others, and the error about _end will disappear. (See also questions 13.25 and 13.26.)

Floating Point

14.1: When I set a float variable to, say, 3.1, why is printf() printing it as 3.0999999?

A: Most computers use base 2 for floating-point numbers as well as for integers. In base 2, 1/1010 (that is, 1/10 decimal) is an infinitely-repeating fraction: its binary representation is 0.0001100110011... . Depending on how carefully your compiler's binary/decimal conversion routines (such as those used by printf) have been written, you may see discrepancies when numbers (especially low-precision floats) not exactly representable in base 2 are assigned or read in and then printed (i.e. converted from base 10 to base 2 and back again). See also question 14.6.

14.2: I'm trying to take some square roots, but I'm getting crazy numbers.

A: Make sure that you have #included , and correctly declared other functions returning double. (Another library routine to be careful with is atof(), which is declared in .) See also question 14.3 below.

References: CT&P Sec. 4.5 pp. 65-6.

14.3: I'm trying to do some simple trig, and I am #including , but I keep getting "undefined: sin" compilation errors.

A: Make sure you're actually linking with the math library. For instance, under Unix, you usually need to use the -lm option, at the *end* of the command line, when compiling/linking. See also questions 13.25 and 13.26.

14.4: My floating-point calculations are acting strangely and giving me different answers on different machines.

A: First, see question 14.2 above.

If the problem isn't that simple, recall that digital computers usually use floating-point formats which provide a close but by no means exact simulation of real number arithmetic. Underflow, cumulative precision loss, and other anomalies are often troublesome.

Don't assume that floating-point results will be exact, and especially don't assume that floating-point values can be compared for equality. (Don't throw haphazard "fuzz factors" in, either; see question 14.5.)

These problems are no worse for C than they are for any other computer language. Certain aspects of floating-point are usually defined as "however the processor does them" (see also question 11.34), otherwise a compiler for a machine without the "right" model would have to do prohibitively expensive emulations.

This article cannot begin to list the pitfalls associated with, and workarounds appropriate for, floating-point work. A good numerical programming text should cover the basics; see also the references below.

References: Kernighan and Plauger, _The Elements of Programming Style_ Sec. 6 pp. 115-8; Knuth, Volume 2 chapter 4; David Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic".

14.5: What's a good way to check for "close enough" floating-point equality?

A: Since the absolute accuracy of floating point values varies, by definition, with their magnitude, the best way of comparing two floating point values is to use an accuracy threshold which is relative to the magnitude of the numbers being compared. Rather than

```
double a, b;
...
if(a == b)      /* WRONG */
```

```
use something like
```

```
#include

if(fabs(a - b) <= epsilon * a)
```

for some suitably-chosen epsilon.

References: Knuth Sec. 4.2.2 pp. 217-8.

14.6: How do I round numbers?

A: The simplest and most straightforward way is with code like

```
(int)(x + 0.5)
```

This technique won't work properly for negative numbers, though.

14.7: Why doesn't C have an exponentiation operator?

A: Because few processors have an exponentiation instruction. C has a pow() function, declared in , although explicit multiplication is often better for small positive integral exponents.

References: ANSI Sec. 4.5.5.1; ISO Sec. 7.5.5.1; H&S Sec. 17.6 p. 393.

14.8: The pre-#defined constant M_PI seems to be missing from my machine's copy of .

A: That constant (which is apparently supposed to be the value of pi, accurate to the machine's precision), is not standard. If you need pi, you'll have to #define it yourself.

References: PCS Sec. 13 p. 237.

14.9: How do I test for IEEE NaN and other special values?

A: Many systems with high-quality IEEE floating-point implementations provide facilities (e.g. predefined constants, and functions like isnan(), either as nonstandard extensions in or perhaps in or) to deal with these values cleanly, and work is being done to formally standardize such facilities. A crude but usually effective test for NaN is exemplified by

```
#define isnan(x) ((x) != (x))
```

although non-IEEE-aware compilers may optimize the test away.

Another possibility is to to format the value in question using sprintf(): on many systems it generates strings like "NaN" and "Inf" which you could compare for in a pinch.

See also question 19.39.

14.11: What's a good way to implement complex numbers in C?

A: It is straightforward to define a simple structure and some arithmetic functions to manipulate them. See also questions 2.7, 2.10, and 14.12.

14.12: I'm looking for some code to do:
Fast Fourier Transforms (FFT's)
matrix arithmetic (multiplication, inversion, etc.)
complex arithmetic

A: Ajay Shah maintains an index of free numerical software; it is posted periodically, and available where this FAQ list is archived (see question 20.40). See also question 18.16.

14.13: I'm having trouble with a Turbo C program which crashes and says something like "floating point formats not linked."

A: The message in the question has to do with an indecent problem in Borland's compilers, which for some unfathomable reason has still not been fixed. However, by the newly-passed Communications Decency Act of the U.S., I am prohibited from transmitting or discussing "indecent" material. (If the fact that users of Borland's compilers are still having this problem isn't indecent, I don't know what is.) If you send me e-mail certifying that you are over 18 years of age, I may be able to help you. (You may also be able to see the comp.os.msdos.programmer FAQ list for more information.)

Variable-Lenght Argument Lists

15.1: I heard that you have to #include before calling printf(). Why?

A: So that a proper prototype for printf() will be in scope.

A compiler may use a different calling sequence for functions which accept variable-length argument lists. (It might do so if calls using variable-length argument lists were less efficient than those using fixed-length.) Therefore, a prototype (indicating, using the ellipsis notation "...", that the argument list is of variable length) must be in scope whenever a varargs function is called, so that the compiler knows to use the varargs calling mechanism.

References: ANSI Sec. 3.3.2.2, Sec. 4.1.6; ISO Sec. 6.3.2.2, Sec. 7.1.7; Rationale Sec. 3.3.2.2, Sec. 4.1.6; H&S Sec. 9.2.4 pp. 268-9, Sec. 9.6 pp. 275-6.

15.2: How can %f be used for both float and double arguments in printf()? Aren't they different types?

A: In the variable-length part of a variable-length argument list, the "default argument promotions" apply: types char and short int are promoted to int, and float is promoted to double. (These are the same promotions that apply to function calls without a prototype in scope, also known as "old style" function calls; see question 11.3.) Therefore, printf's %f format always sees a double. (Similarly, %c always sees an int, as does %hd.) See also questions 12.9 and 12.13.

References: ANSI Sec. 3.3.2.2; ISO Sec. 6.3.2.2; H&S Sec. 6.3.5 p. 177, Sec. 9.4 pp. 272-3.

15.3: I had a frustrating problem which turned out to be caused by the line

```
printf("%d", n);
```

where n was actually a long int. I thought that ANSI function prototypes were supposed to guard against argument type mismatches like this.

A: When a function accepts a variable number of arguments, its prototype does not (and cannot) provide any information about the number and types of those variable arguments. Therefore, the usual protections do *not* apply in the variable-length part of variable-length argument lists: the compiler cannot perform implicit conversions or (in general) warn about mismatches.

See also questions 5.2, 11.3, 12.9, and 15.2.

15.4: How can I write a function that takes a variable number of arguments?

A: Use the facilities of the header.

Here is a function which concatenates an arbitrary number of strings into malloc'ed memory:

```
#include                /* for malloc, NULL, size_t */
#include                /* for va_ stuff */
#include                /* 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 for
second scan */

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

        va_end(argp);

        return retbuf;
}
```

Usage is something like

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

Note the cast on the last argument; see questions 5.2 and 15.3. (Also note that the caller must free the returned, malloc'ed storage.)

Under a pre-ANSI compiler, rewrite the function definition without a prototype ("char *vstrcat(first) char *first; {"), include rather than , add "extern char *malloc();", and use int instead of size_t. You may also have to delete the (void) casts, and use the older varargs package instead of stdarg. See also question 15.7.

References: K&R2 Sec. 7.3 p. 155, Sec. B7 p. 254; ANSI Sec. 4.8; ISO Sec. 7.8; Rationale Sec. 4.8; H&S Sec. 11.4 pp. 296-9; CT&P Sec. A.3 pp. 139-141; PCS Sec. 11 pp. 184-5, Sec. 13 p. 242.

15.5: How can I write a function that takes a format string and a variable number of arguments, like printf(), and passes them to printf() to do most of the work?

A: Use vprintf(), vfprintf(), or vsprintf().

Here is an error() routine which prints an error message, preceded by the string "error: " and terminated with a newline:

```
#include
#include

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

See also question 15.7.

References: K&R2 Sec. 8.3 p. 174, Sec. B1.2 p. 245; ANSI Secs. 4.9.6.7,4.9.6.8,4.9.6.9; ISO Secs. 7.9.6.7,7.9.6.8,7.9.6.9; H&S Sec. 15.12 pp. 379-80; PCS Sec. 11 pp. 186-7.

15.6: How can I write a function analogous to scanf(), that calls scanf() to do most of the work?

A: Unfortunately, vscanf and the like are not standard. You're on your own.

15.7: I have a pre-ANSI compiler, without . What can I do?

A: There's an older header, , which offers about the same functionality.

To rewrite the error() function from question 15.5 to use , change the function header to:

```
void error(va_alist)
va_dcl
{
        char *fmt;
```

change the va_start line to

```
va_start(argp);
```

and add the line

```
fmt = va_arg(argp, char *);
```

between the calls to va_start and vfprintf. (Note that there is no semicolon after va_dcl.)

References: H&S Sec. 11.4 pp. 296-9; CT&P Sec. A.2 pp. 134-139; PCS Sec. 11 pp. 184-5, Sec. 13 p. 250.

15.8: How can I discover how many arguments a function was actually called with?

A: This information is not available to a portable program. Some old systems provided a nonstandard nargs() function, but its use was always questionable, since it typically returned the number of words passed, not the number of arguments. (Structures, long ints, and floating point values are usually passed as several words.)

Any function which takes a variable number of arguments must be able to determine *from the arguments themselves* how many of them there are. printf-like functions do this by looking for formatting specifiers (%d and the like) in the format string (which is why these functions fail badly if the format string does not match the argument list). Another common technique, applicable when the arguments are all of the same type, is to use a sentinel value (often 0, -1, or an appropriately-cast null pointer) at the end of the list (see the execl() and vstrcat() examples in questions 5.2 and 15.4). Finally, if their types are predictable, you can pass an explicit count of the number of variable arguments (although it's usually a nuisance for the caller to generate).

References: PCS Sec. 11 pp. 167-8.

15.9: My compiler isn't letting me declare a function

```
int f(...)
{
}
```

i.e. with no fixed arguments.

A: Standard C requires at least one fixed argument, in part so that you can hand it to va_start().

References: ANSI Sec. 3.5.4, Sec. 3.5.4.3, Sec. 4.8.1.1; ISO Sec. 6.5.4, Sec. 6.5.4.3, Sec. 7.8.1.1; H&S Sec. 9.2 p. 263.

15.10: I have a varargs function which accepts a float parameter. Why isn't

```
va_arg(argp, float)
```

working?

A: In the variable-length part of variable-length argument lists, the old "default argument promotions" apply: arguments of type float are always promoted (widened) to type double, and types char and short int are promoted to int. Therefore, it is never correct to invoke va_arg(argp, float); instead you should always use va_arg(argp, double). Similarly, use va_arg(argp, int) to retrieve arguments which were originally char, short, or int. See also questions 11.3 and 15.2.

References: ANSI Sec. 3.3.2.2; ISO Sec. 6.3.2.2; Rationale Sec. 4.8.1.2; H&S Sec. 11.4 p. 297.

15.11: I can't get va_arg() to pull in an argument of type pointer-to- function.

A: The type-rewriting games which the va_arg() macro typically plays are stymied by overly-complicated types such as pointer-to- function. If you use a typedef for the function pointer type, however, all will be well. See also question 1.21.

References: ANSI Sec. 4.8.1.2; ISO Sec. 7.8.1.2; Rationale Sec. 4.8.1.2.

15.12: How can I write a function which takes a variable number of arguments and passes them to some other function (which takes a variable number of arguments)?

A: In general, you cannot. Ideally, you should provide a version of that other function which accepts a va_list pointer (analogous to vfprintf(); see question 15.5 above). If the arguments must be passed directly as actual arguments, or if you do not have the option of rewriting the second function to accept a va_list (in other words, if the second, called function must accept a variable number of arguments, not a va_list), no portable solution is possible. (The problem could perhaps be solved by resorting to machine-specific assembly language; see also question 15.13 below.)

15.13: How can I call a function with an argument list built up at run time?

A: There is no guaranteed or portable way to do this. If you're curious, ask this list's editor, who has a few wacky ideas you could try...

Instead of an actual argument list, you might consider passing an array of generic (void *) pointers. The called function can then step through the array, much like main() might step through argv. (Obviously this works only if you have control over all the called functions.)

(See also question 19.36.)

Strange Problems

16.3: This program crashes before it even runs! (When single-stepping with a debugger, it dies before the first statement in main().)

A: You probably have one or more very large (kilobyte or more) local arrays. Many systems have fixed-size stacks, and those which perform dynamic stack allocation automatically (e.g. Unix) can be confused when the stack tries to grow by a huge chunk all at once. It is often better to declare large arrays with static duration (unless of course you need a fresh set with each recursive call, in which case you could dynamically allocate them with malloc(); see also question 1.31).

(See also questions 11.12, 16.4, 16.5, and 18.4.)

16.4: I have a program that seems to run correctly, but it crashes as it's exiting, *after* the last statement in main(). What could be causing this?

A: Look for a misdeclared main() (see questions 2.18 and 10.9), or local buffers passed to setbuf() or setvbuf(), or problems in cleanup functions registered by atexit(). See also questions 7.5 and 11.16.

References: CT&P Sec. 5.3 pp. 72-3.

16.5: This program runs perfectly on one machine, but I get weird results on another. Stranger still, adding or removing debugging printouts changes the symptoms...

A: Lots of things could be going wrong; here are a few of the more common things to check:

```
                uninitialized local variables (see also question 7.1)

                integer overflow, especially on 16-bit machines,
                especially of an intermediate result when doing things
                like a * b / c (see also question 3.14)

                undefined evaluation order (see questions 3.1 through
3.4)

                omitted declaration of external functions, especially
                those which return something other than int
                (see questions 1.25 and 14.2)

                dereferenced null pointers (see section 5)

                improper malloc/free use: assuming malloc'ed memory
                contains 0, assuming freed storage persists, freeing
                something twice (see also questions 7.20 and 7.19)

                pointer problems in general (see also question 16.8)
```

```
                   mismatch between printf() format and arguments,
especially
                   trying to print long ints using %d (see question 12.9)

                   trying to malloc(256 * 256 * sizeof(double)),
especially
                   on machines with limited memory (see also questions
7.16
                   and 19.23)

                   array bounds problems, especially of small, temporary
                   buffers, perhaps used for constructing strings with
                   sprintf() (see also questions 7.1 and 12.21)

                   invalid assumptions about the mapping of typedefs,
                   especially size_t

                   floating point problems (see questions 14.1 and 14.4)

                   anything you thought was a clever exploitation of the
way
                   you believe code is generated for your specific system
```

Proper use of function prototypes can catch several of these problems; lint would catch several more. See also questions 16.3, 16.4, and 18.4.

16.6: Why does this code:

```
char *p = "hello, world!";
p[0] = 'H';
```

crash?

A: String literals are not necessarily modifiable, except (in effect) when they are used as array initializers. Try

```
char a[] = "hello, world!";
```

See also question 1.32.

References: ANSI Sec. 3.1.4; ISO Sec. 6.1.4; H&S Sec. 2.7.4 pp. 31-2.

16.8: What do "Segmentation violation" and "Bus error" mean?

A: These generally mean that your program tried to access memory it shouldn't have, invariably as a result of improper pointer use. Likely causes are inadvertent use of null pointers (see also questions 5.2 and 5.20) or uninitialized, misaligned, or otherwise improperly allocated pointers (see questions 7.1 and 7.2); corruption of the malloc arena (see question 7.19); and mismatched function arguments, especially involving pointers;

two possibilities are scanf() (see question 12.12) and fprintf() (make sure it receives its first FILE * argument).

See also questions 16.3 and 16.4.

Style

17.1: What's the best style for code layout in C?

A: K&R, while providing the example most often copied, also supply a good excuse for disregarding it:

```
The position of braces is less important,
although people hold passionate beliefs.
We have chosen one of several popular styles.
Pick a style that suits you, then use it
consistently.
```

It is more important that the layout chosen be consistent (with itself, and with nearby or common code) than that it be "perfect." If your coding environment (i.e. local custom or company policy) does not suggest a style, and you don't feel like inventing your own, just copy K&R. (The tradeoffs between various indenting and brace placement options can be exhaustively and minutely examined, but don't warrant repetition here. See also the Indian Hill Style Guide.)

The elusive quality of "good style" involves much more than mere code layout details; don't spend time on formatting to the exclusion of more substantive code quality issues.

See also question 10.6.

References: K&R1 Sec. 1.2 p. 10; K&R2 Sec. 1.2 p. 10.

17.3: Here's a neat trick for checking whether two strings are equal:

```
if(!strcmp(s1, s2))
```

Is this good style?

A: It is not particularly good style, although it is a popular idiom. The test succeeds if the two strings are equal, but the use of ! ("not") suggests that it tests for inequality.

A better option is to use a macro:

```
#define Streq(s1, s2) (strcmp((s1), (s2)) == 0)
```

Opinions on code style, like those on religion, can be debated endlessly. Though good style is a worthy goal, and can usually be recognized, it cannot be rigorously codified. See also question 17.10.

17.4: Why do some people write if(0 == x) instead of if(x == 0)?

A: It's a trick to guard against the common error of writing

```
if(x = 0)
```

If you're in the habit of writing the constant before the ==, the compiler will complain if you accidentally type

```
if(0 = x)
```

Evidently it can be easier to remember to reverse the test than it is to remember to type the doubled = sign.

References: H&S Sec. 7.6.5 pp. 209-10.

17.5: I came across some code that puts a (void) cast before each call to printf(). Why?

A: printf() does return a value, though few programs bother to check the return values from each call. Since some compilers (and lint) will warn about discarded return values, an explicit cast to (void) is a way of saying "Yes, I've decided to ignore the return value from this call, but please continue to warn me about other (perhaps inadvertently) ignored return values." It's also common to use void casts on calls to strcpy() and strcat(), since the return value is never surprising.

References: K&R2 Sec. A6.7 p. 199; Rationale Sec. 3.3.4; H&S Sec. 6.2.9 p. 172, Sec. 7.13 pp. 229-30.

17.8: What is "Hungarian Notation"? Is it worthwhile?

A: Hungarian Notation is a naming convention, invented by Charles Simonyi, which encodes things about a variable's type (and perhaps its intended use) in its name. It is well-loved in some circles and roundly castigated in others. Its chief advantage is that it makes a variable's type or intended use obvious from its name; its chief disadvantage is that type information is not necessarily a worthwhile thing to carry around in the name of a variable.

References: Simonyi and Heller, "The Hungarian Revolution" .

17.9: Where can I get the "Indian Hill Style Guide" and other coding standards?

A: Various documents are available for anonymous ftp from:

```
Site:                     File or directory:

cs.washington.edu         pub/cstyle.tar.Z
                          (the updated Indian Hill guide)

ftp.cs.toronto.edu        doc/programming
                          (including Henry Spencer's
                          "10 Commandments for C
Programmers")

ftp.cs.umd.edu            pub/style-guide
```

You may also be interested in the books _The Elements of Programming Style_, _Plum Hall Programming Guidelines_, and _C Style: Standards and Guidelines_; see the Bibliography. (The _Standards and Guidelines_ book is not in fact a style guide, but a set of guidelines on selecting and creating style guides.)

See also question 18.9.

17.10: Some people say that goto's are evil and that I should never use them. Isn't that a bit extreme?

A: Programming style, like writing style, is somewhat of an art and cannot be codified by inflexible rules, although discussions about style often seem to center exclusively around such rules.

In the case of the goto statement, it has long been observed that unfettered use of goto's quickly leads to unmaintainable spaghetti code. However, a simple, unthinking ban on the goto statement does not necessarily lead immediately to beautiful programming: an unstructured programmer is just as capable of constructing a Byzantine tangle without using any goto's (perhaps substituting oddly-nested loops and Boolean control variables, instead).

Most observations or "rules" about programming style usually work better as guidelines than rules, and work much better if programmers understand what the guidelines are trying to accomplish. Blindly avoiding certain constructs or following rules without understanding them can lead to just as many problems as the rules were supposed to avert.

Furthermore, many opinions on programming style are just that: opinions. It's usually futile to get dragged into "style wars," because on certain issues (such as those referred to in questions 9.2, 5.3, 5.9, and 10.7), opponents can never seem to agree, or agree to disagree, or stop arguing.

Tools and Resources

18.1: I need:

```
A: Look for programs (see also
   question 18.16) named:
```

a C cross-reference generator	cflow, cxref, calls, cscope, xscope, or ixfw
a C beautifier/pretty-printer	cb, indent, GNU indent, or vgrind
a revision control or configuration management tool	RCS or SCCS
a C source obfuscator (shrouder)	obfus, shroud, or opqcp
a "make" dependency generator	makedepend, or try cc -M or cpp -M
tools to compute code metrics	ccount, Metre, lcount, or csize, or see URL http://www.qucis.queensu.ca:1999/Software-Engineering/Cmetrics.html ; there is also a package sold by McCabe and Associates
a C lines-of-source counter	this can be done very crudely with the standard Unix utility wc, and considerably better with grep -c ";"
a prototype generator	see question 11.31
a tool to track down malloc problems	see question 18.2
a "selective" C preprocessor	see question 10.18
language translation tools	see questions 11.31 and 20.26
C verifiers (lint)	see question 18.7
a C compiler!	see question 18.3

(This list of tools is by no means complete; if you know of tools not mentioned, you're welcome to contact this list's maintainer.)

Other lists of tools, and discussion about them, can be found in the Usenet newsgroups comp.compilers and comp.software-eng .

See also questions 18.16 and 18.3.

18.2: How can I track down these pesky malloc problems?

A: A number of debugging packages exist to help track down malloc problems; one popular one is Conor P. Cahill's "dbmalloc," posted to comp.sources.misc in 1992, volume 32. Others are "leak," available in volume 27 of the comp.sources.unix archives; JMalloc.c and JMalloc.h in the "Snippets" collection; and MEMDEBUG from ftp.crpht.lu in pub/sources/memdebug . See also question 18.16.

A number of commercial debugging tools exist, and can be invaluable in tracking down malloc-related and other stubborn problems:

```
                Bounds-Checker for DOS, from Nu-Mega Technologies,
                P.O. Box 7780, Nashua, NH 03060-7780, USA, 603-889-
2386.

                CodeCenter (formerly Saber-C) from Centerline Software
                (formerly Saber), 10 Fawcett Street, Cambridge, MA
                02138-1110, USA, 617-498-3000.

                Insight, from ParaSoft Corporation, 2500 E. Foothill
                Blvd., Pasadena, CA 91107, USA, 818-792-9941,
                insight@parasoft.com .

                Purify, from Pure Software, 1309 S. Mary Ave.,
Sunnyvale,
                CA 94087, USA, 800-224-7873, info-home@pure.com .

                Sentinel, from AIB Software, 1145 Herndon Pkwy #200,
Herndon,
                VA 22070, USA, 703-787-7700, 800-296-3000, info@aib.com
.
```

18.3: What's a free or cheap C compiler I can use?

A: A popular and high-quality free C compiler is the FSF's GNU C compiler, or gcc. It is available by anonymous ftp from prep.ai.mit.edu in directory pub/gnu, or at several other FSF archive sites. An MS-DOS port, djgpp, is also available; it can be found in the Simtel and Oakland archives and probably many others, usually in a directory like pub/msdos/djgpp/ or simtel/msdos/djgpp/.

There is a shareware compiler called PCC, available as PCC12C.ZIP .

A very inexpensive MS-DOS compiler is Power C from Mix Software, 1132 Commerce Drive, Richardson, TX 75801, USA, 214-783-6001.

Another recently-developed compiler is lcc, available for anonymous ftp from ftp.cs.princeton.edu in pub/lcc.

Archives associated with comp.compilers contain a great deal of information about available compilers, interpreters, grammars, etc. (for many languages). The comp.compilers archives (including an FAQ list), maintained by the moderator, John R. Levine, are at iecc.com . A list of available compilers and related resources, maintained by Mark Hopkins, Steven Robenalt, and David Muir Sharnoff, is at ftp.idiom.com in pub/compilers- list/. (See also the comp.compilers directory in the news.answers archives at rtfm.mit.edu and ftp.uu.net; see question 20.40.)

See also question 18.16.

18.4: I just typed in this program, and it's acting strangely. Can you see anything wrong with it?

A: See if you can run lint first (perhaps with the -a, -c, -h, -p or other options). Many C compilers are really only half- compilers, electing not to diagnose numerous source code difficulties which would not actively preclude code generation.

See also questions 16.5 and 16.8.

References: Ian Darwin, _Checking C Programs with lint_ .

18.5: How can I shut off the "warning: possible pointer alignment problem" message which lint gives me for each call to malloc()?

A: The problem is that traditional versions of lint do not know, and cannot be told, that malloc() "returns a pointer to space suitably aligned for storage of any type of object." It is possible to provide a pseudoimplementation of malloc(), using a #define inside of #ifdef lint, which effectively shuts this warning off, but a simpleminded definition will also suppress meaningful messages about truly incorrect invocations. It may be easier simply to ignore the message, perhaps in an automated way with grep -v. (But don't get in the habit of ignoring too many lint messages, otherwise one day you'll overlook a significant one.)

18.7: Where can I get an ANSI-compatible lint?

A: Products called PC-Lint and FlexeLint (in "shrouded source form," for compilation on 'most any system) are available from

```
Gimpel Software
3207 Hogarth Lane
Collegeville, PA  19426  USA
(+1) 610 584 4261
gimpel@netaxs.com
```

The Unix System V release 4 lint is ANSI-compatible, and is available separately (bundled with other C tools) from UNIX Support Labs or from System V resellers.

Another ANSI-compatible lint (which can also perform higher- level formal verification) is LCLint, available via anonymous ftp from larch.lcs.mit.edu in pub/Larch/lclint/.

In the absence of lint, many modern compilers do attempt to diagnose almost as many problems as lint does.

18.8: Don't ANSI function prototypes render lint obsolete?

A: Not really. First of all, prototypes work only if they are present and correct; an inadvertently incorrect prototype is worse than useless. Secondly, lint checks consistency across multiple source files, and checks data declarations as well as functions. Finally, an independent program like lint will probably always be more scrupulous at enforcing compatible, portable coding practices than will any particular, implementation-specific, feature- and extension-laden compiler.

If you do want to use function prototypes instead of lint for cross-file consistency checking, make sure that you set the prototypes up correctly in header files. See questions 1.7 and 10.6.

18.9: Are there any C tutorials or other resources on the net?

A: There are several of them:

```
        "Notes for C programmers," by Christopher Sawtell, are
        available from svr-ftp.eng.cam.ac.uk in misc/sawtell_C.shar and
        garbo.uwasa.fi in /pc/c-lang/c-lesson.zip .

        Tim Love's "C for Programmers" is available by ftp from svr-
        ftp.eng.cam.ac.uk in the misc directory.  An html version is at

http://club.eng.cam.ac.uk/help/tpl/languages/C/teaching_C/teaching_C.ht
ml .

        The Coronado Enterprises C tutorials are available on Simtel
        mirrors in pub/msdos/c/.

        Rick Rowe has a tutorial which is available from ftp.netcom.com
        as pub/rowe/tutorde.zip or ftp.wustl.edu as
        pub/MSDOS_UPLOADS/programming/c_language/ctutorde.zip .

        There is evidently a web-based course at
        http://www.strath.ac.uk/CC/Courses/CCourse/CCourse.html .

        Finally, on some Unix machines you can try typing learn c at
the
        shell prompt.

        [Disclaimer: I have not reviewed these tutorials; I have heard
        that at least one of them contains a number of errors.  Also,
        this sort of information rapidly becomes out-of-date; these
        addresses may not work by the time you read this and try them.]
```

```
        Several of these tutorials, plus a great deal of other
        information about C, are accessible via the web at
        http://www.lysator.liu.se/c/index.html .

        Vinit Carpenter maintains a list of resources for learning C
and
        C++; it is posted to comp.lang.c and comp.lang.c++, and
archived
        where this FAQ list is (see question 20.40), or on the web at
        http://vinny.csd.mu.edu/ .
```

See also question 18.10 below.

18.10: What's a good book for learning C?

A: There are far too many books on C to list here; it's impossible to rate them all. Many people believe that the best one was also the first: _The C Programming Language_, by Kernighan and Ritchie ("K&R," now in its second edition). Opinions vary on K&R's suitability as an initial programming text: many of us did learn C from it, and learned it well; some, however, feel that it is a bit too clinical as a first tutorial for those without much programming background.

An excellent reference manual is _C: A Reference Manual_, by Samuel P. Harbison and Guy L. Steele, now in its fourth edition.

Though not suitable for learning C from scratch, this FAQ list has been published in book form; see the Bibliography.

Mitch Wright maintains an annotated bibliography of C and Unix books; it is available for anonymous ftp from ftp.rahul.net in directory pub/mitch/YABL/.

This FAQ list's editor maintains a collection of previous answers to this question, which is available upon request. See also question 18.9 above.

18.13: Where can I find the sources of the standard C libraries?

A: One source (though not public domain) is _The Standard C Library_, by P.J. Plauger (see the Bibliography). Implementations of all or part of the C library have been written and are readily available as part of the netBSD and GNU (also Linux) projects. See also question 18.16.

18.14: I need code to parse and evaluate expressions.

A: Two available packages are "defunc," posted to comp.sources.misc in December, 1993 (V41 i32,33), to alt.sources in January, 1994, and available from sunsite.unc.edu in pub/packages/development/libraries/defunc-1.3.tar.Z, and "parse," at lamont.ldgo.columbia.edu. Other options include the S-Lang interpreter, available via anonymous ftp from amy.tch.harvard.edu in pub/slang, and the shareware Cmm ("C-minus-minus" or "C minus the hard stuff"). See also question 18.16.

There is also some parsing/evaluation code in _Software Solutions in C_ (chapter 12, pp. 235-55).

18.15: Where can I get a BNF or YACC grammar for C?

A: The definitive grammar is of course the one in the ANSI standard; see question 11.2. Another grammar (along with one for C++) by Jim Roskind is in pub/c++grammar1.1.tar.Z at ics.uci.edu . A fleshed-out, working instance of the ANSI grammar (due to Jeff Lee) is on ftp.uu.net (see question 18.16) in usenet/net.sources/ansi.c.grammar.Z (including a companion lexer). The FSF's GNU C compiler contains a grammar, as does the appendix to K&R2.

The comp.compilers archives contain more information about grammars; see question 18.3.

References: K&R1 Sec. A18 pp. 214-219; K&R2 Sec. A13 pp. 234- 239; ANSI Sec. A.2; ISO Sec. B.2; H&S pp. 423-435 Appendix B.

18.15a: Does anyone have a C compiler test suite I can use?

A: Plum Hall (formerly in Cardiff, NJ; now in Hawaii) sells one; other packages are Ronald Guilmette's RoadTest(tm) Compiler Test Suites (ftp to netcom.com, pub/rfg/roadtest/announce.txt for information) and Nullstone's Automated Compiler Performance Analysis Tool (see http://www.nullstone.com). The FSF's GNU C (gcc) distribution includes a c-torture-test which checks a number of common problems with compilers. Kahan's paranoia test, found in netlib/paranoia on netlib.att.com, strenuously tests a C implementation's floating point capabilities.

18.16: Where and how can I get copies of all these freely distributable programs?

A: As the number of available programs, the number of publicly accessible archive sites, and the number of people trying to access them all grow, this question becomes both easier and more difficult to answer.

There are a number of large, public-spirited archive sites out there, such as ftp.uu.net, archive.umich.edu, oak.oakland.edu, sumex-aim.stanford.edu, and wuarchive.wustl.edu, which have huge amounts of software and other information all freely available. For the FSF's GNU project, the central distribution site is prep.ai.mit.edu . These well-known sites tend to be extremely busy and hard to reach, but there are also numerous "mirror" sites which try to spread the load around.

On the connected Internet, the traditional way to retrieve files from an archive site is with anonymous ftp. For those without ftp access, there are also several ftp-by-mail servers in operation. More and more, the world-wide web (WWW) is being used to announce, index, and even transfer large data files. There are probably yet newer access methods, too.

Those are some of the easy parts of the question to answer. The hard part is in the details -- this article cannot begin to track or list all of the available archive sites or all of the various ways of accessing them. If you have access to the net at all, you probably have access to more up-to-date information about active sites and useful access methods than this FAQ list does.

The other easy-and-hard aspect of the question, of course, is simply *finding* which site has what you're looking for. There is a tremendous amount of work going on in this area, and there are probably new indexing services springing up every day. One of the first was "archie": for any program or resource available on the net, if you know its name, an archie server can usually tell you which anonymous ftp sites have it. Your system may have an archie command, or you can send the mail message "help" to archie@archie.cs.mcgill.ca for information.

If you have access to Usenet, see the regular postings in the comp.sources.unix and comp.sources.misc newsgroups, which describe the archiving policies for those groups and how to access their archives. The comp.archives newsgroup contains numerous announcements of anonymous ftp availability of various items. Finally, the newsgroup comp.sources.wanted is generally a more appropriate place to post queries for source availability, but check *its* FAQ list, "How to find sources," before posting there.

See also question 14.12.

System Dependencies

19.1: How can I read a single character from the keyboard without waiting for the RETURN key? How can I stop characters from being echoed on the screen as they're typed?

A: Alas, there is no standard or portable way to do these things in C. Concepts such as screens and keyboards are not even mentioned in the Standard, which deals only with simple I/O "streams" of characters.

At some level, interactive keyboard input is usually collected and presented to the requesting program a line at a time. This gives the operating system a chance to support input line editing (backspace/delete/rubout, etc.) in a consistent way, without requiring that it be built into every program. Only when the user is satisfied and presses the RETURN key (or equivalent) is the line made available to the calling program. Even if the calling program appears to be reading input a character at a time (with getchar() or the like), the first call blocks until the user has typed an entire line, at which point potentially many characters become available and many character requests (e.g. getchar() calls) are satisfied in quick succession.

When a program wants to read each character immediately as it arrives, its course of action will depend on where in the input stream the line collection is happening and how it can be disabled. Under some systems (e.g. MS-DOS, VMS in some modes), a program can use a different or modified set of OS-level input calls to bypass line-at-a-time input processing. Under other systems (e.g. Unix, VMS in other modes), the part of the operating system responsible for serial input (often called the "terminal driver") must be placed in a mode which turns off line- at-a-time processing, after which all calls to the usual input routines (e.g. read(), getchar(), etc.) will return characters immediately. Finally, a few systems (particularly older, batch- oriented mainframes) perform input processing in peripheral processors which cannot be told to do anything other than line-at-a-time input.

Therefore, when you need to do character-at-a-time input (or disable keyboard echo, which is an analogous problem), you will have to use a technique specific to the system you're using, assuming it provides one. Since comp.lang.c is oriented towards topics that C does deal with, you will usually get better answers to these questions by referring to a system-specific newsgroup such as comp.unix.questions or comp.os.msdos.programmer, and to the FAQ lists for these groups. Note that the answers are often not unique even across different variants of a system; bear in mind when answering system- specific questions that the answer that applies to your system may not apply to everyone else's.

However, since these questions are frequently asked here, here are brief answers for some common situations.

Some versions of curses have functions called cbreak(), noecho(), and getch() which do what you want. If you're specifically trying to read a short password without echo, you might try getpass(). Under Unix, you can use ioctl() to play with the terminal driver modes (CBREAK or RAW under "classic" versions; ICANON, c_cc[VMIN] and c_cc[VTIME] under System V or POSIX systems; ECHO under all versions), or in a pinch, system() and the stty command. (For more information, see and tty(4) under classic versions, and termio(4) under System V, or and termios(4) under POSIX.) Under MS- DOS, use getch() or getche(), or the corresponding BIOS interrupts. Under VMS, try the Screen Management (SMG$) routines, or curses, or issue low-level $QIO's with the IO$_READVBLK function code (and perhaps IO$M_NOECHO, and others) to ask for one character at a time. (It's also possible to set character-at-a-time or "pass through" modes in the VMS terminal driver.) Under other operating systems, you're on your own.

(As an aside, note that simply using setbuf() or setvbuf() to set stdin to unbuffered will *not* generally serve to allow character-at-a-time input.)

If you're trying to write a portable program, a good approach is to define your own suite of three functions to (1) set the terminal driver or input system into character-at-a-time mode (if necessary), (2) get characters, and (3) return the terminal driver to its initial state when the program is finished. (Ideally, such a set of functions might be part of the C Standard, some day.) The extended versions of this FAQ list (see question 20.40) contain examples of such functions for several popular systems.

See also question 19.2.

References: PCS Sec. 10 pp. 128-9, Sec. 10.1 pp. 130-1; POSIX Sec. 7.

19.2: How can I find out if there are characters available for reading (and if so, how many)? Alternatively, how can I do a read that will not block if there are no characters available?

A: These, too, are entirely operating-system-specific. Some versions of curses have a nodelay() function. Depending on your system, you may also be able to use "nonblocking I/O", or a system call named "select" or "poll", or the FIONREAD ioctl, or c_cc[VTIME], or kbhit(), or rdchk(), or the O_NDELAY option to open() or fcntl(). See also question 19.1.

19.3: How can I display a percentage-done indication that updates itself in place, or show one of those "twirling baton" progress indicators?

A: These simple things, at least, you can do fairly portably. Printing the character '\r' will usually give you a carriage return without a line feed, so that you can overwrite the current line. The character '\b' is a backspace, and will usually move the cursor one position to the left.

References: ANSI Sec. 2.2.2; ISO Sec. 5.2.2.

19.4: How can I clear the screen? How can I print things in inverse video? How can I move the cursor to a specific x, y position?

A: Such things depend on the terminal type (or display) you're using. You will have to use a library such as termcap, terminfo, or curses, or some system-specific routines, to perform these operations.

For clearing the screen, a halfway portable solution is to print a form-feed character ('\f'), which will cause some displays to clear. Even more portable would be to print enough newlines to scroll everything away. As a last resort, you could use system() (see question 19.27) to invoke an operating system clear-screen command.

References: PCS Sec. 5.1.4 pp. 54-60, Sec. 5.1.5 pp. 60-62.

19.5: How do I read the arrow keys? What about function keys?

A: Terminfo, some versions of termcap, and some versions of curses have support for these non-ASCII keys. Typically, a special key sends a multicharacter sequence (usually beginning with ESC, '\033'); parsing these can be tricky. (curses will do the parsing for you, if you call keypad() first.)

Under MS-DOS, if you receive a character with value 0 (*not* '0'!) while reading the keyboard, it's a flag indicating that the next character read will be a code indicating a special key. See any DOS programming guide for lists of keyboard codes. (Very briefly: the up, left, right, and down arrow keys are 72, 75, 77, and 80, and the function keys are 59 through 68.)

References: PCS Sec. 5.1.4 pp. 56-7.

19.6: How do I read the mouse?

A: Consult your system documentation, or ask on an appropriate system-specific newsgroup (but check its FAQ list first). Mouse handling is completely different under the X window system, MS- DOS, the Macintosh, and probably every other system.

References: PCS Sec. 5.5 pp. 78-80.

19.7: How can I do serial ("comm") port I/O?

A: It's system-dependent. Under Unix, you typically open, read, and write a device file in /dev, and use the facilities of the terminal driver to adjust its characteristics. (See also questions 19.1 and 19.2.) Under MS-DOS, you can use the predefined stream stdaux, or a special file like COM1, or some primitive BIOS interrupts, or (if you require decent performance) any number of interrupt-driven serial I/O packages. Several netters recommend the book _C Programmer's Guide to Serial Communications_, by Joe Campbell.

19.8: How can I direct output to the printer?

A: Under Unix, either use popen() (see question 19.30) to write to the lp or lpr program, or perhaps open a special file like /dev/lp. Under MS-DOS, write to the (nonstandard) predefined stdio stream stdprn, or open the special files PRN or LPT1.

References: PCS Sec. 5.3 pp. 72-74.

19.9: How do I send escape sequences to control a terminal or other device?

A: If you can figure out how to send characters to the device at all (see question 19.8 above), it's easy enough to send escape sequences. In ASCII, the ESC code is 033 (27 decimal), so code like

```
fprintf(ofd, "\033[J");
```

sends the sequence ESC [J .

19.10: How can I do graphics?

A: Once upon a time, Unix had a fairly nice little set of device- independent plot routines described in plot(3) and plot(5), but they've largely fallen into disuse.

If you're programming for MS-DOS, you'll probably want to use libraries conforming to the VESA or BGI standards.

If you're trying to talk to a particular plotter, making it draw is usually a matter of sending it the appropriate escape sequences; see also question 19.9. The vendor may supply a C- callable library, or you may be able to find one on the net.

If you're programming for a particular window system (Macintosh, X windows, Microsoft Windows), you will use its facilities; see the relevant documentation or newsgroup or FAQ list.

References: PCS Sec. 5.4 pp. 75-77.

19.11: How can I check whether a file exists? I want to warn the user if a requested input file is missing.

A: It's surprisingly difficult to make this determination reliably and portably. Any test you make can be invalidated if the file is created or deleted (i.e. by some other process) between the time you make the test and the time you try to open the file.

Three possible test routines are stat(), access(), and fopen(). (To make an approximate test for file existence with fopen(), just open for reading and close immediately.) Of these, only fopen() is widely portable, and access(), where it exists, must be used carefully if the program uses the Unix set-UID feature.

Rather than trying to predict in advance whether an operation such as opening a file will succeed, it's often better to try it, check the return value, and complain if it fails. (Obviously, this approach won't work if you're trying to avoid overwriting an existing file, unless you've got something like the O_EXCL file opening option available, which does just what you want in this case.)

References: PCS Sec. 12 pp. 189,213; POSIX Sec. 5.3.1, Sec. 5.6.2, Sec. 5.6.3.

19.12: How can I find out the size of a file, prior to reading it in?

A: If the "size of a file" is the number of characters you'll be able to read from it in C, it is difficult or impossible to determine this number exactly).

Under Unix, the stat() call will give you an exact answer. Several other systems supply a Unix-like stat() which will give an approximate answer. You can fseek() to the end and then use ftell(), but these tend to have the same problems: fstat() is not portable, and generally tells you the same thing stat() tells you; ftell() is not guaranteed to return a byte count except for binary files. Some systems provide routines called filesize() or filelength(), but these are not portable, either.

Are you sure you have to determine the file's size in advance? Since the most accurate way of determining the size of a file as a C program will see it is to open the file and read it, perhaps you can rearrange the code to learn the size as it reads.

References: ANSI Sec. 4.9.9.4; ISO Sec. 7.9.9.4; H&S Sec. 15.5.1; PCS Sec. 12 p. 213; POSIX Sec. 5.6.2.

19.13: How can a file be shortened in-place without completely clearing or rewriting it?

A: BSD systems provide ftruncate(), several others supply chsize(), and a few may provide a (possibly undocumented) fcntl option F_FREESP. Under MS-DOS, you can sometimes use write(fd, "", 0). However, there is no portable solution, nor a way to delete blocks at the beginning. See also question 19.14.

19.14: How can I insert or delete a line (or record) in the middle of a file?

A: Short of rewriting the file, you probably can't. The usual solution is simply to rewrite the file. (Instead of deleting records, you might consider simply marking them as deleted, to avoid rewriting.) See also questions 12.30 and 19.13.

19.15: How can I recover the file name given an open stream or file descriptor?

A: This problem is, in general, insoluble. Under Unix, for instance, a scan of the entire disk (perhaps involving special permissions) would theoretically be required, and would fail if the descriptor were connected to a pipe or referred to a deleted file (and could give a misleading answer for a file with multiple links). It is best to remember the names of files yourself when you open them (perhaps with a wrapper function around fopen()).

19.16: How can I delete a file?

A: The Standard C Library function is remove(). (This is therefore one of the few questions in this section for which the answer is *not* "It's system-dependent.") On older, pre-ANSI Unix systems, remove() may not exist, in which case you can try unlink().

References: K&R2 Sec. B1.1 p. 242; ANSI Sec. 4.9.4.1; ISO Sec. 7.9.4.1; H&S Sec. 15.15 p. 382; PCS Sec. 12 pp. 208,220- 221; POSIX Sec. 5.5.1, Sec. 8.2.4.

19.17: Why can't I open a file by its explicit path? The call

```
fopen("c:\newdir\file.dat", "r")
```

is failing.

A: The file you actually requested -- with the characters \n and \f in its name -- probably doesn't exist, and isn't what you thought you were trying to open.

In character constants and string literals, the backslash \ is an escape character, giving special meaning to the character following it. In order for literal backslashes in a pathname to be passed through to fopen() (or any other routine) correctly, they have to be doubled, so that the first backslash in each pair quotes the second one:

```
fopen("c:\\newdir\\file.dat", "r");
```

Alternatively, under MS-DOS, it turns out that forward slashes are also accepted as directory separators, so you could use

```
fopen("c:/newdir/file.dat", "r");
```

(Note, by the way, that header file names mentioned in preprocessor #include directives are *not* string literals, so you may not have to worry about backslashes there.)

19.18: I'm getting an error, "Too many open files". How can I increase the allowable number of simultaneously open files?

A: There are actually at least two resource limitations on the number of simultaneously open files: the number of low-level "file descriptors" or "file handles" available in the operating system, and the number of FILE structures available in the stdio library. Both must be sufficient. Under MS-DOS systems, you can control the number of operating system file handles with a line in CONFIG.SYS. Some compilers come with instructions (and perhaps a source file or two) for increasing the number of stdio FILE structures.

19.20: How can I read a directory in a C program?

A: See if you can use the opendir() and readdir() routines, which are part of the POSIX standard and are available on most Unix variants. Implementations also exist for MS-DOS, VMS, and other systems. (MS-DOS also has FINDFIRST and FINDNEXT routines which do essentially the same thing.) readdir() only returns file names; if you need more information about the file, try calling stat(). To match filenames to some wildcard pattern, see question 13.7.

References: K&R2 Sec. 8.6 pp. 179-184; PCS Sec. 13 pp. 230-1; POSIX Sec. 5.1; Schumacher, ed., _Software Solutions in C_ Sec. 8.

19.22: How can I find out how much memory is available?

A: Your operating system may provide a routine which returns this information, but it's quite system-dependent.

19.23: How can I allocate arrays or structures bigger than 64K?

A: A reasonable computer ought to give you transparent access to all available memory. If you're not so lucky, you'll either have to rethink your program's use of memory, or use various system-specific techniques.

64K is (still) a pretty big chunk of memory. No matter how much memory your computer has available, it's asking a lot to be able to allocate huge amounts of it contiguously. (The C Standard does not guarantee that a single object can be larger than 32K.) Often it's a good idea to use data structures which don't require that all memory be contiguous. For dynamically- allocated multidimensional arrays, you can use pointers to pointers, as illustrated in question 6.16. Instead of a large array of structures, you can use a linked list, or an array of pointers to structures.

If you're using a PC-compatible (8086-based) system, and running up against a 640K limit, consider using "huge" memory model, or expanded or extended memory, or malloc variants such as halloc() or farmalloc(), or a 32-bit "flat" compiler (e.g. djgpp, see question 18.3), or some kind of a DOS extender, or another operating system.

References: ANSI Sec. 2.2.4.1; ISO Sec. 5.2.4.1.

19.24: What does the error message "DGROUP data allocation exceeds 64K" mean, and what can I do about it? I thought that using large model meant that I could use more than 64K of data!

A: Even in large memory models, MS-DOS compilers apparently toss certain data (strings, some initialized global or static variables) into a default data segment, and it's this segment that is overflowing. Either use less global data, or, if you're already limiting yourself to reasonable amounts (and if the problem is due to something like the number of strings), you may be able to coax the compiler into not using the default data segment for so much. Some compilers place only "small" data objects in the default data segment, and give you a way (e.g. the /Gt option under Microsoft compilers) to configure the threshold for "small."

19.25: How can I access memory (a memory-mapped device, or graphics memory) located at a certain address?

A: Set a pointer, of the appropriate type, to the right number (using an explicit cast to assure the compiler that you really do intend this nonportable conversion):

```
unsigned int *magicloc = (unsigned int *)0x12345678;
```

Then, *magicloc refers to the location you want. (Under MS-DOS, you may find a macro like MK_FP() handy for working with segments and offsets.)

References: K&R1 Sec. A14.4 p. 210; K&R2 Sec. A6.6 p. 199; ANSI Sec. 3.3.4; ISO Sec. 6.3.4; Rationale Sec. 3.3.4; H&S Sec. 6.2.7 pp. 171-2.

19.27: How can I invoke another program (a standalone executable, or an operating system command) from within a C program?

A: Use the library function system(), which does exactly that. Note that system's return value is the command's exit status, and usually has nothing to do with the output of the command. Note also that system() accepts a single string representing the command to be invoked; if you need to build up a complex command line, you can use sprintf(). See also question 19.30.

References: K&R1 Sec. 7.9 p. 157; K&R2 Sec. 7.8.4 p. 167, Sec. B6 p. 253; ANSI Sec. 4.10.4.5; ISO Sec. 7.10.4.5; H&S Sec. 19.2 p. 407; PCS Sec. 11 p. 179.

19.30: How can I invoke another program or command and trap its output?

A: Unix and some other systems provide a popen() routine, which sets up a stdio stream on a pipe connected to the process running a command, so that the output can be read (or the input supplied). (Also, remember to call pclose().)

If you can't use popen(), you may be able to use system(), with the output going to a file which you then open and read.

If you're using Unix and popen() isn't sufficient, you can learn about pipe(), dup(), fork(), and exec().

(One thing that probably would *not* work, by the way, would be to use freopen().)

References: PCS Sec. 11 p. 169.

19.31: How can my program discover the complete pathname to the executable from which it was invoked?

A: argv[0] may contain all or part of the pathname, or it may contain nothing. You may be able to duplicate the command language interpreter's search path logic to locate the executable if the name in argv[0] is present but incomplete. However, there is no guaranteed solution.

References: K&R1 Sec. 5.11 p. 111; K&R2 Sec. 5.10 p. 115; ANSI Sec. 2.1.2.2.1; ISO Sec. 5.1.2.2.1; H&S Sec. 20.1 p. 416.

19.32: How can I automatically locate a program's configuration files in the same directory as the executable?

A: It's hard; see also question 19.31 above. Even if you can figure out a workable way to do it, you might want to consider making the program's auxiliary (library) directory configurable, perhaps with an environment variable. (It's especially important to allow variable placement of a program's configuration files when the program will be used by several people, e.g. on a multiuser system.)

19.33: How can a process change an environment variable in its caller?

A: It may or may not be possible to do so at all. Different operating systems implement global name/value functionality similar to the Unix environment in different ways. Whether the "environment" can be usefully altered by a running program, and if so, how, is system-dependent.

Under Unix, a process can modify its own environment (some systems provide setenv() or putenv() functions for the purpose), and the modified environment is generally passed on to child processes, but it is *not* propagated back to the parent process.

19.36: How can I read in an object file and jump to routines in it?

A: You want a dynamic linker or loader. It may be possible to malloc some space and read in object files, but you have to know an awful lot about object file formats, relocation, etc. Under BSD Unix, you could use system() and ld -A to do the linking for you. Many versions of SunOS and System V have the -ldl library which allows object files to be dynamically loaded. Under VMS, use LIB$FIND_IMAGE_SYMBOL. GNU has a package called "dld". See also question 15.13.

19.37: How can I implement a delay, or time a user's response, with sub-second resolution?

A: Unfortunately, there is no portable way. V7 Unix, and derived systems, provided a fairly useful ftime() routine with resolution up to a millisecond, but it has disappeared from System V and POSIX. Other routines you might look for on your system include clock(), delay(), gettimeofday(), msleep(), nap(), napms(), setitimer(), sleep(), times(), and usleep(). (A routine called wait(), however, is at least under Unix *not* what you want.) The select() and poll() calls (if available) can be pressed into service to implement simple delays. On MS- DOS machines, it is possible to reprogram the system timer and timer interrupts.

Of these, only clock() is part of the ANSI Standard. The difference between two calls to clock() gives elapsed execution time, and if CLOCKS_PER_SEC is greater than 1, the difference will have subsecond resolution. However, clock() gives elapsed processor time used by the current program, which on a multitasking system may differ considerably from real time.

If you're trying to implement a delay and all you have available is a time-reporting function, you can implement a CPU-intensive busy-wait, but this is only an option on a single-user, single- tasking machine as it is terribly antisocial to any other processes. Under a multi-tasking operating system, be sure to use a call which puts your process to sleep for the duration, such as sleep() or select(), or pause() in conjunction with alarm() or setitimer().

For really brief delays, it's tempting to use a do-nothing loop like

```
long int i;
for(i = 0; i < 1000000; i++)
        ;
```

but resist this temptation if at all possible! For one thing, your carefully-calculated delay loops will stop working next month when a faster processor comes out. Perhaps worse, a clever compiler may notice that the loop does nothing and optimize it away completely.

References: H&S Sec. 18.1 pp. 398-9; PCS Sec. 12 pp. 197-8,215- 6; POSIX Sec. 4.5.2.

19.38: How can I trap or ignore keyboard interrupts like control-C?

A: The basic step is to call signal(), either as

```
#include
signal(SIGINT, SIG_IGN);
```

to ignore the interrupt signal, or as

```
extern void func(int);
signal(SIGINT, func);
```

to cause control to transfer to function func() on receipt of an interrupt signal.

On a multi-tasking system such as Unix, it's best to use a slightly more involved technique:

```
extern void func(int);
if(signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, func);
```

The test and extra call ensure that a keyboard interrupt typed in the foreground won't inadvertently interrupt a program running in the background (and it doesn't hurt to code calls to signal() this way on any system).

On some systems, keyboard interrupt handling is also a function of the mode of the terminal-input subsystem; see question 19.1. On some systems, checking for keyboard interrupts is only performed when the program is reading input, and keyboard interrupt handling may therefore depend on which input routines are being called (and *whether* any input routines are active at all). On MS-DOS systems, setcbrk() or ctrlbrk() functions may also be involved.

References: ANSI Secs. 4.7,4.7.1; ISO Secs. 7.7,7.7.1; H&S Sec. 19.6 pp. 411-3; PCS Sec. 12 pp. 210-2; POSIX Secs. 3.3.1,3.3.4.

19.39: How can I handle floating-point exceptions gracefully?

A: On many systems, you can define a routine matherr() which will be called when there are certain floating-point errors, such as errors in the math routines in . You may also be able to use signal() (see question 19.38 above) to catch SIGFPE. See also question 14.9.

References: Rationale Sec. 4.5.1.

19.40: How do I... Use sockets? Do networking? Write client/server applications?

A: All of these questions are outside of the scope of this list and have much more to do with the networking facilities which you have available than they do with C. Good books on the subject are Douglas Comer's three-volume _Internetworking with TCP/IP_ and W. R. Stevens's _UNIX Network Programming_. (There is also plenty of information out on the net itself.)

19.40b: How do I use BIOS calls? How can I write ISR's? How can I create TSR's?

A: These are very particular to specific systems (PC compatibles running MS-DOS, most likely). You'll get much better information in a specific newsgroup such as comp.os.msdos.programmer or its FAQ list; another excellent resource is Ralf Brown's interrupt list.

19.41: But I can't use all these nonstandard, system-dependent functions, because my program has to be ANSI compatible!

A: You're out of luck. Either you misunderstood your requirement, or it's an impossible one to meet. ANSI/ISO Standard C simply does not define ways of doing these things. (POSIX defines a few.) It is possible, and desirable, for *most* of a program to be ANSI-compatible, deferring the system-dependent functionality to a few routines in a few files which are rewritten for each system ported to.

Miscellaneous

20.1: How can I return multiple values from a function?

A: Either pass pointers to several locations which the function can fill in, or have the function return a structure containing the desired values, or (in a pinch) consider global variables. See also questions 2.7, 4.8, and 7.5.

20.3: How do I access command-line arguments?

A: They are pointed to by the argv array with which main() is called.

References: K&R1 Sec. 5.11 pp. 110-114; K&R2 Sec. 5.10 pp. 114- 118; ANSI Sec. 2.1.2.2.1; ISO Sec. 5.1.2.2.1; H&S Sec. 20.1 p. 416; PCS Sec. 5.6 pp. 81-2, Sec. 11 p. 159, pp. 339-40 Appendix F; Schumacher, ed., _Software Solutions in C_ Sec. 4 pp. 75-85.

20.5: How can I write data files which can be read on other machines with different word size, byte order, or floating point formats?

A: The most portable solution is to use text files (usually ASCII), written with fprintf() and read with fscanf() or the like. (Similar advice also applies to network protocols.) Be skeptical of arguments which imply that text files are too big, or that reading and writing them is too slow. Not only is their efficiency frequently acceptable in practice, but the advantages of being able to interchange them easily between machines, and manipulate them with standard tools, can be overwhelming.

If you must use a binary format, you can improve portability, and perhaps take advantage of prewritten I/O libraries, by making use of standardized formats such as Sun's XDR (RFC 1014), OSI's ASN.1 (referenced in CCITT X.409 and ISO 8825 "Basic Encoding Rules"), CDF, netCDF, or HDF. See also questions 2.12 and 12.38.

References: PCS Sec. 6 pp. 86,88.

20.6: If I have a char * variable pointing to the name of a function, how can I call that function?

A: The most straightforward thing to do is to maintain a correspondence table of names and function pointers:

```
int func(), anotherfunc();

struct { char *name; int (*funcptr)(); } symtab[] = {
        "func",          func,
        "anotherfunc",   anotherfunc,
};
```

Then, search the table for the name, and call via the associated function pointer. See also questions 2.15 and 19.36.

References: PCS Sec. 11 p. 168.

20.8: How can I implement sets or arrays of bits?

A: Use arrays of char or int, with a few macros to access the desired bit at the proper index. Here are some simple macros to use with arrays of char:

```
#include              /* for CHAR_BIT */
```

```
#define BITMASK(b) (1 << ((b) % CHAR_BIT))
#define BITSLOT(b) ((b) / CHAR_BIT)
#define BITSET(a, b) ((a)[BITSLOT(b)] |= BITMASK(b))
#define BITTEST(a, b) ((a)[BITSLOT(b)] & BITMASK(b))
```

(If you don't have , try using 8 for CHAR_BIT.)

References: H&S Sec. 7.6.7 pp. 211-216.

20.9: How can I determine whether a machine's byte order is big-endian or little-endian?

A: One way is to use a pointer:

```
int x = 1;
if(*(char *)&x == 1)
        printf("little-endian\n");
else    printf("big-endian\n");
```

It's also possible to use a union.

See also question 10.16.

References: H&S Sec. 6.1.2 pp. 163-4.

20.10: How can I convert integers to binary or hexadecimal?

A: Make sure you really know what you're asking. Integers are stored internally in binary, although for most purposes it is not incorrect to think of them as being in octal, decimal, or hexadecimal, whichever is convenient. The base in which a number is expressed matters only when that number is read in from or written out to the outside world.

In source code, a non-decimal base is indicated by a leading 0 or 0x (for octal or hexadecimal, respectively). During I/O, the base of a formatted number is controlled in the printf and scanf family of functions by the choice of format specifier (%d, %o, %x, etc.) and in the strtol() and strtoul() functions by the third argument. During *binary* I/O, however, the base again becomes immaterial.

For more information about "binary" I/O, see question 2.11. See also questions 8.6 and 13.1.

References: ANSI Secs. 4.10.1.5,4.10.1.6; ISO Secs. 7.10.1.5,7.10.1.6.

20.11: Can I use base-2 constants (something like 0b101010)? Is there a printf() format for binary?

A: No, on both counts. You can convert base-2 string representations to integers with strtol().

20.12: What is the most efficient way to count the number of bits which are set in a value?

A: Many "bit-fiddling" problems like this one can be sped up and streamlined using lookup tables (but see question 20.13 below).

20.13: How can I make my code more efficient?

A: Efficiency, though a favorite comp.lang.c topic, is not important nearly as often as people tend to think it is. Most of the code in most programs is not time-critical. When code is not time-critical, it is far more important that it be written clearly and portably than that it be written maximally efficiently. (Remember that computers are very, very fast, and that even "inefficient" code can run without apparent delay.)

It is notoriously difficult to predict what the "hot spots" in a program will be. When efficiency is a concern, it is important to use profiling software to determine which parts of the program deserve attention. Often, actual computation time is swamped by peripheral tasks such as I/O and memory allocation, which can be sped up by using buffering and caching techniques.

Even for code that *is* time-critical, it is not as important to "microoptimize" the coding details. Many of the "efficient coding tricks" which are frequently suggested (e.g. substituting shift operators for multiplication by powers of two) are performed automatically by even simpleminded compilers. Heavyhanded optimization attempts can make code so bulky that performance is actually degraded, and are rarely portable (i.e. they may speed things up on one machine but slow them down on another). In any case, tweaking the coding usually results in at best linear performance improvements; the big payoffs are in better algorithms.

For more discussion of efficiency tradeoffs, as well as good advice on how to improve efficiency when it is important, see chapter 7 of Kernighan and Plauger's _The Elements of Programming Style_, and Jon Bentley's _Writing Efficient Programs_.

20.14: Are pointers really faster than arrays? How much do function calls slow things down? Is ++i faster than i = i + 1?

A: Precise answers to these and many similar questions depend of course on the processor and compiler in use. If you simply must know, you'll have to time test programs carefully. (Often the differences are so slight that hundreds of thousands of iterations are required even to see them. Check the compiler's assembly language output, if available, to see if two purported alternatives aren't compiled identically.)

It is "usually" faster to march through large arrays with pointers rather than array subscripts, but for some processors the reverse is true.

Function calls, though obviously incrementally slower than in- line code, contribute so much to modularity and code clarity that there is rarely good reason to avoid them.

Before rearranging expressions such as i = i + 1, remember that you are dealing with a compiler, not a keystroke-programmable calculator. Any decent compiler will generate identical code for ++i, i += 1, and i = i + 1. The reasons for using ++i or i += 1 over i = i + 1 have to do with style, not efficiency. (See also question 3.12.)

20.17: Is there a way to switch on strings?

A: Not directly. Sometimes, it's appropriate to use a separate function to map strings to integer codes, and then switch on those. Otherwise, of course, you can fall back on strcmp() and a conventional if/else chain. See also questions 10.12, 20.18, and 20.29.

References: K&R1 Sec. 3.4 p. 55; K&R2 Sec. 3.4 p. 58; ANSI Sec. 3.6.4.2; ISO Sec. 6.6.4.2; H&S Sec. 8.7 p. 248.

20.18: Is there a way to have non-constant case labels (i.e. ranges or arbitrary expressions)?

A: No. The switch statement was originally designed to be quite simple for the compiler to translate, therefore case labels are limited to single, constant, integral expressions. You *can* attach several case labels to the same statement, which will let you cover a small range if you don't mind listing all cases explicitly.

If you want to select on arbitrary ranges or non-constant expressions, you'll have to use an if/else chain.

See also questions question 20.17.

References: K&R1 Sec. 3.4 p. 55; K&R2 Sec. 3.4 p. 58; ANSI Sec. 3.6.4.2; ISO Sec. 6.6.4.2; Rationale Sec. 3.6.4.2; H&S Sec. 8.7 p. 248.

20.19: Are the outer parentheses in return statements really optional?

A: Yes.

Long ago, in the early days of C, they were required, and just enough people learned C then, and wrote code which is still in circulation, that the notion that they might still be required is widespread.

(As it happens, parentheses are optional with the sizeof operator, too, as long as its operand is a variable or a unary expression.)

References: K&R1 Sec. A18.3 p. 218; ANSI Sec. 3.3.3, Sec. 3.6.6; ISO Sec. 6.3.3, Sec. 6.6.6; H&S Sec. 8.9 p. 254.

20.20: Why don't C comments nest? How am I supposed to comment out code containing comments? Are comments legal inside quoted strings?

A: C comments don't nest mostly because PL/I's comments, which C's are borrowed from, don't either. Therefore, it is usually better to "comment out" large sections of code, which might contain comments, with #ifdef or #if 0 (but see question 11.19).

The character sequences /* and */ are not special within double- quoted strings, and do not therefore introduce comments, because a program (particularly one which is generating C code as output) might want to print them.

Note also that // comments, as in C++, are not currently legal in C, so it's not a good idea to use them in C programs (even if your compiler supports them as an extension).

References: K&R1 Sec. A2.1 p. 179; K&R2 Sec. A2.2 p. 192; ANSI Sec. 3.1.9 (esp. footnote 26), Appendix E; ISO Sec. 6.1.9, Annex F; Rationale Sec. 3.1.9; H&S Sec. 2.2 pp. 18-9; PCS Sec. 10 p. 130.

20.24: Why doesn't C have nested functions?

A: It's not trivial to implement nested functions such that they have the proper access to local variables in the containing function(s), so they were deliberately left out of C as a simplification. (gcc does allow them, as an extension.) For many potential uses of nested functions (e.g. qsort comparison functions), an adequate if slightly cumbersome solution is to use an adjacent function with static declaration, communicating if necessary via a few static variables. (A cleaner solution when such functions must communicate is to pass around a pointer to a structure containing the necessary context.)

20.25: How can I call FORTRAN (C++, BASIC, Pascal, Ada, LISP) functions from C? (And vice versa?)

A: The answer is entirely dependent on the machine and the specific calling sequences of the various compilers in use, and may not be possible at all. Read your compiler documentation very carefully; sometimes there is a "mixed-language programming guide," although the techniques for passing arguments and ensuring correct run-time startup are often arcane. More information may be found in FORT.gz by Glenn Geers, available via anonymous ftp from suphys.physics.su.oz.au in the src directory.

cfortran.h, a C header file, simplifies C/FORTRAN interfacing on many popular machines. It is available via anonymous ftp from zebra.desy.de (131.169.2.244).

In C++, a "C" modifier in an external function declaration indicates that the function is to be called using C calling conventions.

References: H&S Sec. 4.9.8 pp. 106-7.

20.26: Does anyone know of a program for converting Pascal or FORTRAN (or LISP, Ada, awk, "Old" C, ...) to C?

A: Several freely distributable programs are available:

p2c A Pascal to C converter written by Dave Gillespie, posted to comp.sources.unix in March, 1990 (Volume 21); also available by anonymous ftp from csvax.cs.caltech.edu, file pub/p2c-1.20.tar.Z .

ptoc Another Pascal to C converter, this one written in Pascal (comp.sources.unix, Volume 10, also patches in Volume 13?).

f2c A Fortran to C converter jointly developed by people from Bell Labs, Bellcore, and Carnegie Mellon. To find out more about f2c, send the mail message "send index from f2c" to netlib@research.att.com or research!netlib. (It is also available via anonymous ftp on netlib.att.com, in directory netlib/f2c.)

This FAQ list's maintainer also has available a list of a few other commercial translation products, and some for more obscure languages.

See also questions 11.31 and 18.16.

20.27: Is C++ a superset of C? Can I use a C++ compiler to compile C code?

A: C++ was derived from C, and is largely based on it, but there are some legal C constructs which are not legal C++. Conversely, ANSI C inherited several features from C++, including prototypes and const, so neither language is really a subset or superset of the other. In spite of the differences, many C programs will compile correctly in a C++ environment, and many recent compilers offer both C and C++ compilation modes.

References: H&S p. xviii, Sec. 1.1.5 p. 6, Sec. 2.8 pp. 36-7, Sec. 4.9 pp. 104-107.

20.28: I need a sort of an "approximate" strcmp routine, for comparing two strings for close, but not necessarily exact, equality.

A: Some nice information and algorithms having to do with approximate string matching, as well as a useful bibliography, can be found in Sun Wu and Udi Manber's paper "AGREP -- A Fast Approximate Pattern-Matching Tool."

Another approach involves the "soundex" algorithm, which maps similar-sounding words to the same codes. Soundex was designed for discovering similar-sounding names (for telephone directory assistance, as it happens), but it can be pressed into service for processing arbitrary words.

References: Knuth Sec. 6 pp. 391-2 Volume 3; Wu and Manber, "AGREP -- A Fast Approximate Pattern-Matching Tool" .

20.29: What is hashing?

A: Hashing is the process of mapping strings to integers, usually in a relatively small range. A "hash function" maps a string (or some other data structure) to a a bounded number (the "hash bucket") which can more easily be used as an index in an array, or for performing repeated comparisons. (Obviously, a mapping from a potentially huge set of strings to a small set of integers will not be unique. Any algorithm using hashing therefore has to deal with the possibility of "collisions.") Many hashing functions and related algorithms have been developed; a full treatment is beyond the scope of this list.

References: K&R2 Sec. 6.6; Knuth Sec. 6.4 pp. 506-549 Volume 3; Sedgewick Sec. 16 pp. 231-244.

20.31: How can I find the day of the week given the date?

A: Use mktime() or localtime() (see questions 13.13 and 13.14, but beware of DST adjustments if tm_hour is 0), or Zeller's congruence (see the sci.math FAQ list), or this elegant code by Tomohiko Sakamoto:

```
dayofweek(y, m, d)      /* 0 = Sunday */
int y, m, d;            /* 1 <= m <= 12, y > 1752 or so */
{
        static int t[] = {0, 3, 2, 5, 0, 3, 5, 1, 4, 6, 2, 4};
        y -= m < 3;
        return (y + y/4 - y/100 + y/400 + t[m-1] + d) % 7;
}
```

See also questions 13.14 and 20.32.

References: ANSI Sec. 4.12.2.3; ISO Sec. 7.12.2.3.

20.32: Will 2000 be a leap year? Is (year % 4 == 0) an accurate test for leap years?

A: Yes and no, respectively. The full expression for the present Gregorian calendar is

```
year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)
```

See a good astronomical almanac or other reference for details. (To forestall an eternal debate: references which claim the existence of a 4000-year rule are wrong.)

20.34: Here's a good puzzle: how do you write a program which produces its own source code as its output?

A: It is actually quite difficult to write a self-reproducing program that is truly portable, due particularly to quoting and character set difficulties.

Here is a classic example (which is normally presented on one line, although it will "fix" itself the first time it's run):

```
char*s="char*s=%c%s%c;main(){printf(s,34,s,34);}";
main(){printf(s,34,s,34);}
```

(This program, like many of the genre, assumes that the double- quote character " has the value 34, as it does in ASCII.)

20.35: What is "Duff's Device"?

A: It's a devastatingly deviously unrolled byte-copying loop, devised by Tom Duff while he was at Lucasfilm. In its "classic" form, it looks like:

```
register n = (count + 7) / 8;   /* count > 0 assumed */
switch (count % 8)
{
case 0:    do { *to = *from++;
case 7:         *to = *from++;
case 6:         *to = *from++;
case 5:         *to = *from++;
case 4:         *to = *from++;
case 3:         *to = *from++;
case 2:         *to = *from++;
case 1:         *to = *from++;
               } while (--n > 0);
}
```

where count bytes are to be copied from the array pointed to by from to the memory location pointed to by to (which is a memory- mapped device output register, which is why to isn't incremented). It solves the problem of handling the leftover bytes (when count isn't a multiple of 8) by interleaving a switch statement with the loop which copies bytes 8 at a time. (Believe it or not, it *is* legal to have case labels buried within blocks nested in a switch statement like this. In his announcement of the technique to C's developers and the world, Duff noted that C's switch syntax, in particular its "fall through" behavior, had long been controversial, and that "This code forms some sort of argument in that debate, but I'm not sure whether it's for or against.")

20.36: When will the next International Obfuscated C Code Contest (IOCCC) be held? How can I get a copy of the current and previous winning entries?

A: The contest schedule is tied to the dates of the USENIX conferences at which the winners are announced. At the time of this writing, it is expected that the yearly contest will open in October. To obtain a current copy of the rules and guidelines, send e-mail with the Subject: line "send rules" to:

```
{apple,pyramid,sun,uunet}!hoptoad!judges  or
```

```
judges@toad.com
```

(Note that these are *not* the addresses for submitting entries.)

Contest winners should be announced at the winter USENIX conference in January, and are posted to the net sometime thereafter. Winning entries from previous years (back to 1984) are archived at ftp.uu.net (see question 18.16) under the directory pub/ioccc/.

As a last resort, previous winners may be obtained by sending e- mail to the above address, using the Subject: "send YEAR winners", where YEAR is a single four-digit year, a year range, or "all".

20.37: What was the entry keyword mentioned in K&R1?

A: It was reserved to allow the possibility of having functions with multiple, differently-named entry points, a la FORTRAN. It was not, to anyone's knowledge, ever implemented (nor does anyone remember what sort of syntax might have been imagined for it). It has been withdrawn, and is not a keyword in ANSI C. (See also question 1.12.)

References: K&R2 p. 259 Appendix C.

20.38: Where does the name "C" come from, anyway?

A: C was derived from Ken Thompson's experimental language B, which was inspired by Martin Richards's BCPL (Basic Combined Programming Language), which was a simplification of CPL (Cambridge Programming Language). For a while, there was speculation that C's successor might be named P (the third letter in BCPL) instead of D, but of course the most visible descendant language today is C++.

20.39: How do you pronounce "char"?

A: You can pronounce the C keyword "char" in at least three ways: like the English words "char," "care," or "car;" the choice is arbitrary.

20.40: Where can I get extra copies of this list? What about back issues?

A: An up-to-date copy may be obtained from ftp.eskimo.com in directory u/s/scs/C-faq/. You can also just pull it off the net; it is normally posted to comp.lang.c on the first of each month, with an Expires: line which should keep it around all month. A parallel, abridged version is available (and posted), as is a list of changes accompanying each significantly updated version.

The various versions of this list are also posted to the newsgroups comp.answers and news.answers . Several sites archive news.answers postings and other FAQ lists, including this one; two sites are rtfm.mit.edu (directories pub/usenet/news.answers/C-faq/ and pub/usenet/comp.lang.c/) and ftp.uu.net (directory usenet/news.answers/C-faq/). An archie server (see question 18.16) should help you find others; ask it to "find C-faq". If you don't have ftp access, a mailserver at rtfm.mit.edu can mail you FAQ lists: send a message containing the single word help to mail-server@rtfm.mit.edu . See the meta-FAQ list in news.answers for more information.

A hypertext (HTML) version of this FAQ list is available on the World-Wide Web; the URL is http://www.eskimo.com/~scs/C-faq/top.html . URL's pointing at all FAQ lists (these may also allow topic searching) are http://www.cis.ohio-state.edu/hypertext/faq/usenet/FAQ-List.html and http://www.luth.se/wais/ .

An extended version of this FAQ list is being published by Addison-Wesley as _C Programming FAQs: Frequently Asked Questions_ (ISBN 0-201-84519-9). It should be available in November 1995.

This list is an evolving document of questions which have been Frequent since before the Great Renaming, not just a collection of this month's interesting questions. Older copies are obsolete and don't contain much, except the occasional typo, that the current list doesn't.

Bibliography

Americal National Standards Institute, _American National Standard for Information Systems -- Programming Language -- C_, ANSI X3.159-1989 (see question 11.2). [ANSI]

Americal National Standards Institute, _Rationale for American National Standard for Information Systems -- Programming Language -- C_ (see question 11.2). [Rationale]

Jon Bentley, _Writing Efficient Programs_, Prentice-Hall, 1982, ISBN 0- 13-970244-X.

G.E.P. Box and Mervin E. Muller, "A Note on the Generation of Random Normal Deviates," _Annals of Mathematical Statistics_, Vol. 29 #2, June, 1958, pp. 610-611.

David Burki, "Date Conversions," _The C Users Journal_, February 1993, pp. 29-34.

Ian F. Darwin, _Checking C Programs with lint_, O'Reilly, 1988, ISBN 0- 937175-30-7.

David Goldberg, "What Every Computer Scientist Should Know about Floating-Point Arithmetic," _ACM Computing Surveys_, Vol. 23 #1, March, 1991, pp. 5-48.

Samuel P. Harbison and Guy L. Steele, Jr., _C: A Reference Manual_, Fourth Edition, Prentice-Hall, 1995, ISBN 0-13-326224-3. [H&S]

Mark R. Horton, _Portable C Software_, Prentice Hall, 1990, ISBN 0-13- 868050-7. [PCS]

Institute of Electrical and Electronics Engineers, _Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API) [C Language_, IEEE Std. 1003.1, ISO/IEC 9945-1.

International Organization for Standardization, ISO 9899:1990 (see question 11.2). [ISO]

Brian W. Kernighan and P.J. Plauger, _The Elements of Programming Style_, Second Edition, McGraw-Hill, 1978, ISBN 0-07-034207-5.

Brian W. Kernighan and Dennis M. Ritchie, _The C Programming Language_, Prentice-Hall, 1978, ISBN 0-13-110163-3. [K&R1]

Brian W. Kernighan and Dennis M. Ritchie, _The C Programming Language_, Second Edition, Prentice Hall, 1988, ISBN 0-13-110362-8, 0-13-110370-9. [K&R2]

Donald E. Knuth, _The Art of Computer Programming_. Volume 1: _Fundamental Algorithms_, Second Edition, Addison-Wesley, 1973, ISBN 0- 201-03809-9. Volume 2: _Seminumerical Algorithms_, Second Edition, Addison-Wesley, 1981, ISBN 0-201-03822-6. Volume 3: _Sorting and Searching_, Addison-Wesley, 1973, ISBN 0-201-03803-X. [Knuth]

Andrew Koenig, _C Traps and Pitfalls_, Addison-Wesley, 1989, ISBN 0-201- 17928-8. [CT&P]

Stephen K. Park and Keith W. Miller, "Random Number Generators: Good Ones are Hard to Find," _Communications of the ACM_, Vol. 31 #10, October, 1988, pp. 1192-1201 (also technical correspondence August, 1989, pp. 1020-1024, and July, 1993, pp. 108-110).

P.J. Plauger, _The Standard C Library_, Prentice Hall, 1992, ISBN 0-13- 131509-9.

Thomas Plum, _C Programming Guidelines_, Second Edition, Plum Hall, 1989, ISBN 0-911537-07-4.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, _Numerical Recipes in C_, Second Edition, Cambridge University Press, 1992, ISBN 0-521-43108-5.

Dale Schumacher, Ed., _Software Solutions in C_, AP Professional, 1994, ISBN 0-12-632360-7.

Robert Sedgewick, _Algorithms in C_, Addison-Wesley, 1990, ISBN 0-201- 51425-7.

Charles Simonyi and Martin Heller, "The Hungarian Revolution," _Byte_, August, 1991, pp.131-138.

David Straker, _C Style: Standards and Guidelines_, Prentice Hall, ISBN 0-13-116898-3.

Steve Summit, _C Programming FAQs: Frequently Asked Questions_, Addison- Wesley, 1995, ISBN 0-201-84519-9. [The book version of this FAQ list.]

Sun Wu and Udi Manber, "AGREP -- A Fast Approximate Pattern-Matching Tool," USENIX Conference Proceedings, Winter, 1992, pp. 153-162.

There is another bibliography in the revised Indian Hill style guide (see question 17.9). See also question 18.10.