1 :: What is C language?

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

2 :: What is Duffs Device?

It's a devastatingly devious way of unrolling a loop, devised by Tom Duff while he was at Lucasfilm. In its ``classic" form, it was used to copy bytes, and looked like this: 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.")

3 :: Here is a good puzzle: how do you write a program which produces its own source code as output?

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 ought to be 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 has a few deficiencies, among other things neglecting to #include <stdio.h>, and assuming that the double-quote character " has the value 34, as it does in ASCII.)

```
#define q(k)main(){return!puts(#k"\nq("#k")");}
q(#define q(k)main(){return!puts(#k"\nq("#k")");})
```

4 :: Suggesting that there can be 62 seconds in a minute?

Q: Why can tm_sec in the tm structure range from 0 to 61, suggesting that there can be 62 seconds in a minute?

A: That's actually a buglet in the Standard. There can be 61 seconds in a minute during a leap

second. It's possible for there to be two leap seconds in a year, but it turns out that it's guaranteed that they'll never both occur in the same day (let alone the same minute).

5 :: Was 2000 a leap year?

Is (year % 4 == 0) an accurate test for leap years? (Was 2000 a leap year?)

No, it's not accurate (and yes, 2000 was a leap year). The actual rules for the present Gregorian calendar are that leap years occur every four years, but not every 100 years, except that they do occur every 400 years, after all. In C, these rules can be expressed as:
year % 4 == 0 && (year % 100 != 0 || year % 400 == 0)
Actually, if the domain of interest is limited (perhaps by the range of a time_t) such that the only century year it encompasses is 2000, the expression
(year % 4 == 0) /* 1901-2099 only */
is accurate, if less than robust.
If you trust the implementor of the C library, you can use mktime to determine whether a given year is a leap year;
Note also that the transition from the Julian to the Gregorian calendar involved deleting several days to make up for accumulated errors. (The transition was first made in Catholic countries under Pope Gregory XIII in October, 1582, and involved deleting 10 days. In the British Empire, eleven days were deleted when the Gregorian calendar was adopted in September 1752. A few countries didn't switch until the 20th century.) Calendar code which has to work for historical dates must therefore be especially careful.

6 :: How can I find the day of the week given the date?

Here are three methods:
1. Use mktime or localtime # . Here is a code fragment which computes the day of the week for February 29, 2000:

```
#include <stdio.h>
#include <time.h>

char *wday[] = {"Sunday", "Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday"};

struct tm tm;

tm.tm_mon = 2 - 1;
tm.tm_mday = 29;
tm.tm_year = 2000 - 1900;
tm.tm_hour = tm.tm_min = tm.tm_sec = 0;
tm.tm_isdst = -1;

if(mktime(&tm) != -1)
printf("%s\n", wday[tm.tm_wday]);
```

When using mktime like this, it's usually important to set tm_isdst to -1, as shown (especially if tm_hour is 0), otherwise a daylight saving time correction could push the time past midnight into another day. # Use Zeller's congruence, which says that if

J is the number of the century [i.e. the year / 100],
K the year within the century [i.e. the year % 100],
m the month,
q the day of the month,
h the day of the week [where 1 is Sunday];

7 :: What is hashing in C?

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 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. An extremely simple hash function for strings is simply to add up the values of all the characters:

```
unsigned hash(char *str)
{
unsigned int h = 0;
while(*str != '\0')
h += *str++;
return h % NBUCKETS;
}
```

A somewhat better hash function is

```
unsigned hash(char *str)
{
unsigned int h = 0;
while(*str != '\0')
h = (256 * h + *str++) % NBUCKETS;
return h;
}
```

8 :: I need a sort of an approximate strcmp routine ...

I need a sort of an ``approximate'' strcmp routine, for comparing two strings for close, but not necessarily exact, equality.

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.

9 :: What is C Programing language?

Is C++ a superset of C? What are the differences between C and C++? Can I use a C++ compiler to compile C code?

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; the two also define the meaning of some common constructs differently.
The most important feature of C++ not found in C is of course the extended structure known as a class which along with operator overloading makes object-oriented programming convenient. There are several other differences and new features: variables may be declared anywhere in a block; const variables may be true compile-time constants; structure tags are implicitly typedeffed; an & in a parameter declaration requests pass by reference; and the new and delete operators, along with per-object constructors and destructors, simplify dynamic data structure management. There are a host of mechanisms tied up with classes and object-oriented programming: inheritance, friends, virtual functions, templates, etc. (This list of C++ features is not intended to be complete; C++ programmers will notice many omissions.)

10 :: How can I call FORTRAN?

How can I call FORTRAN (C++, BASIC, Pascal, Ada, LISP) functions from C? (And vice versa?)

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. Besides arranging calling sequences correctly, you may also have to conspire between the various languages to get aggregate data structures declared compatibly.

In C++, a "C" modifier in an external function declaration indicates that the function is to be called using C calling conventions. In Ada, you can use the Export and Convention pragmas, and types from the package Interfaces.C, to arrange for C-compatible calls, parameters, and data structures.

11 :: What is assert and when would I use it?

It is a macro, defined in <assert.h>, for testing ``assertions". An assertion essentially documents an assumption being made by the programmer, an assumption which, if violated, would indicate a serious programming error. For example, a function which was supposed to be called with a non-null pointer could write
assert(p != NULL);
A failed assertion terminates the program. Assertions should not be used to catch expected errors, such as malloc or fopen failures.

12 :: Why doesnt C have nested functions?

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, though unsupported by qsort, is to pass around a pointer to a structure containing the necessary context.)

13 :: Does C have an equivalent to Pascals with statement?

No. The way in C to get quick and easy access to the fields of a structure is to declare a little local structure pointer variable (which, it must be admitted, is not quite as notationally convenient as a with statement and doesn't save quite as many keystrokes, though it is probably safer). That is, if you have something unwieldy like
structarray[complex_expression].a =
structarray[complex_expression].b +
structarray[complex_expression].c;

you can replace it with

struct whatever *p = &structarray[complex_expression];
p->a = p->b + p->c;

14 :: If the assignment operator were ...

If the assignment operator were :=, wouldn't it then be harder to accidentally write things like if(a = b) ?

Yes, but it would also be just a little bit more cumbersome to type all of the assignment statements which a typical program contains.
In any case, it's really too late to be worrying about this sort of thing now. The choices of = for assignment and == for comparison were made, rightly or wrongly, over two decades ago, and are not likely to be changed. (With respect to the question, many compilers and versions of lint will warn about if(a = b) and similar expressions;
As a point of historical interest, the choices were made based on the observation that assignment is more frequent than comparison, and so deserves fewer keystrokes. In fact, using = for

assignment in C and its predecessor B represented a change from B's own predecessor BCPL, which did use := as its assignment operator.

15 :: Is C a great language, or what?

Is C a great language, or what? Where else could you write something like a+++++b ?

Well, you can't meaningfully write it in C, either. The rule for lexical analysis is that at each point during a straightforward left-to-right scan, the longest possible token is determined, without regard to whether the resulting sequence of tokens makes sense. The fragment in the question is therefore interpreted as
a ++ ++ + b
and cannot be parsed as a valid expression.

16 :: Does C have circular shift operators?

No. (Part of the reason why is that the sizes of C's types aren't precisely defined----but a circular shift makes most sense when applied to a word of a particular known size.)
You can implement a circular shift using two regular shifts and a bitwise OR:
(x << 13) | (x > >3) /* circular shift left 13 in 16 bits */

17 :: There seem to be a few missing operators ...

There seem to be a few missing operators, like ^^, &&=, and ->=.

A logical exclusive-or operator (hypothetically ``^^'') would be nice, but it couldn't possibly have short-circuiting behavior analogous to && and || Similarly, it's not clear how short-circuiting would apply to hypothetical assignment operators &&= and ||=. (It's also not clear how often &&= and ||= would actually be needed.)
Though p = p->next is an extremely common idiom for traversing a linked list, -> is not a binary arithmetic operator. A hypothetical ->= operator therefore wouldn't really fit the pattern of the other assignment operators.
You can write an exclusive-or macro in several ways:

```
#define XOR(a, b) ((a) && !(b) || !(a) && (b)) /* 1 */
#define XOR(a, b) (!!(a) ^ !!(b)) /* 2 */
#define XOR(a, b) (!!(a) != !!(b)) /* 3 */
#define XOR(a, b) (!(a) ^ !(b)) /* 4 */
#define XOR(a, b) (!(a) != !(b)) /* 5 */
#define XOR(a, b) ((a) ? !(b) : !!(b)) /* 6 */
```

18 :: Why isnt there a numbered, multi-level break statement to break out

Why isn't there a numbered, multi-level break statement to break out of several loops at once? What am I supposed to use instead, a goto?

First, remember why it is that break and continue exist at all--they are, in effect, ``structured

gotos" used in preference to goto (and accepted as alternatives by most of those who shun goto) because they are clean and structured and pretty much restricted to a common, idiomatic usages. A hypothetical multi-level break, on the other hand, would rapidly lose the inherent cleanliness of the single break--programmers and readers of code would have to carefully count nesting levels to figure out what a given break did, and the insertion of a new intermediately-nested loop could, er, break things badly. (By this analysis, a numbered break statement can be even more confusing and error-prone than a goto/label pair.)
The right way to break out of several loops at once (which C also does not have) involves a syntax which allows the naming of loops, so that a break statement can specify the name of the loop to be broken out of.
If you do have to break out of more than one loop at once (or break out of a loop from inside a switch, where break would merely end a case label) yes, go ahead and use a goto. (But when you find the need for a multi-level break, it's often a sign that the loop should be broken out to its own function, at which point you can achieve roughly the same effect as that multi-level break by using a premature return.)

19 :: Why dont C comments nest?

Why don't C comments nest? How am I supposed to comment out code containing comments? Are comments legal inside quoted strings?

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).
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. (It is hard to imagine why anyone would want or need to place a comment inside a quoted string. It is easy to imagine a program needing to print "/*".)

20 :: Are the outer parentheses in return statements really optional?

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, under certain circumstances.)

21 :: Is there a way to have non-constant case labels (i.e. ranges or arbitrary expressions)?

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.

22 :: Is there a way to switch on strings?

Not directly. Sometimes, it's appropriate to use a separate function to map strings to integer codes, and then switch on those:

```
#define CODE_APPLE 1
#define CODE_ORANGE 2
#define CODE_NONE 0

switch(classifyfunc(string)) {
case CODE_APPLE:
...

case CODE_ORANGE:
...

case CODE_NONE:
...
}
```

where classifyfunc looks something like

```
static struct lookuptab {
char *string;
int code;
} tab[] = {
{"apple", CODE_APPLE},
{"orange", CODE_ORANGE},
};

classifyfunc(char *string)
{
int i;
for(i = 0; i < sizeof(tab) / sizeof(tab[0]); i++)
if(strcmp(tab[i].string, string) == 0)
return tab[i].code;

return CODE_NONE;
}
```

Otherwise, of course, you can fall back on a conventional if/else chain:

```
if(strcmp(string, "apple") == 0) {
...
```

```
} else if(strcmp(string, "orange") == 0) {
...
}
```

23 :: Which is more efficient, a switch statement or an if else chain?

The differences, if any, are likely to be slight. The switch statement was designed to be efficiently implementable, though the compiler may choose to use the equivalent of an if/else chain (as opposed to a compact jump table) if the case labels are sparsely distributed.
Do use switch when you can: it's certainly cleaner, and perhaps more efficient (and certainly should never be any less efficient).

24 :: How can I swap two values without using a temporary?

The standard hoary old assembly language programmer's trick is:

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

But this sort of code has little place in modern, HLL programming. Temporary variables are essentially free, and the idiomatic code using three assignments, namely

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

is not only clearer to the human reader, it is more likely to be recognized by the compiler and turned into the most-efficient code (e.g. perhaps even using an EXCH instruction). The latter code is obviously also amenable to use with pointers and floating-point values, unlike the XOR trick.

25 :: People claim that optimizing compilers are good and that we no longer have to write things in assembler for speed

People claim that optimizing compilers are good and that we no longer have to write things in assembler for speed, but my compiler can't even replace i/=2 with a shift.

Was i signed or unsigned? If it was signed, a shift is not equivalent (hint: think about the result if i is negative and odd), so the compiler was correct not to use it.

26 :: I have been replacing multiplications and divisions with shift operators, because shifting is more efficient.

This is an excellent example of a potentially risky and usually unnecessary optimization. Any compiler worthy of the name can replace a constant, power-of-two multiplication with a left shift, or a similar division of an unsigned quantity with a right shift. Furthermore, a compiler will make these optimizations only when they're correct; many programmers overlook the fact that

shifting a negative value to the right is not equivalent to division. (Therefore, when you need to make sure that these optimizations are performed, you may have to declare relevant variables as unsigned.)

27 :: Are pointers really faster than arrays?

Are pointers really faster than arrays? How much do function calls slow things down? Is ++i faster than i = i + 1?

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.
For conventional machines, it is usually faster to march through large arrays with pointers rather than array subscripts, but for some processors the reverse is true. (Better compilers should generate good code regardless of which notation you use, though it's arguably easier for a compiler to convert array indices to pointers than vice versa .)
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. (Actually, by reducing bulk, functions can improve performance.) Also, some compilers are able to expand small, critical-path functions in-line, either as an optimization or at the programmer's request.
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.

28 :: What is the best way of making my program efficient?

By picking good algorithms, implementing them carefully, and making sure that your program isn't doing any extra work. For example, the most microoptimized character-copying loop in the world will be beat by code which avoids having to copy characters at all.
When worrying about efficiency, it's important to keep several things in perspective. First of all, although efficiency is an enormously popular topic, it is not always as important 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 usually 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 seemingly ``inefficient'' code may be quite efficiently compilable, and 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.

29 :: What is the most efficient way to count the number of bits which are set in an integer?

Many ``bit-fiddling'' problems like this one can be sped up and streamlined using lookup tables (but see question 20.13). Here is a little function which computes the number of bits in a value, 4

bits at a time:

```
static int bitcounts[] =
{0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};

int bitcount(unsigned int u)
{
int n = 0;

for(; u != 0; u >>= 4)
n += bitcounts[u & 0x0f];

return n;
}
```

30 :: Can I use base-2 constants (something like 0b101010)? Is there a printf format for binary?

No, on both counts, although there are various preprocessor tricks you can try. You can convert base-2 string representations to integers with strtol. If you need to print numbers out in base 2, .

31 :: How can I convert integers to binary or hexadecimal?

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, either in the form of a source code constant or in the form of I/O performed by a program.

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: if numbers are being read or written as individual bytes (typically with getc or putc), or as multi-byte words (typically with fread or fwrite), it is meaningless to ask what ``base'' they are in.

If what you need is formatted binary conversion, it's easy enough to do. Here is a little function for formatting a number in a requested base:

32 :: How do I swap bytes?

V7 Unix had a swab function, but it seems to have been forgotten.

A problem with explicit byte-swapping code is that you have to decide whether to call it or not, based on the byte order of the data and the byte order of the machine in use.

A better solution is to define functions which convert between the known byte order of the data and the (unknown) byte order of the machine in use, and to arrange for these functions to be no-ops on those machines which already match the desired byte order. A set of such functions, introduced with the BSD networking code but now in wide use, is ntohs, htons, ntohl, and htonl.

These are intended to convert between ``network'' and ``host'' byte orders, for ``short'' or ``long'' integers, where ``network'' order is always big-endian, and where ``short'' integers are always 16 bits and ``long'' integers are 32 bits. (This is not the C definition, of course, but it's compatible with the C definition) So if you know that the data you want to convert from or to is big-endian, you can use these functions. (The point is that you always call the functions, making your code much cleaner. Each function either swaps bytes if it has to, or does nothing. The decision to swap or not to swap gets made once, when the functions are implemented for a particular machine, rather than being made many times in many different calling programs.)

33 :: How can I determine whether a machines byte order is big-endian or little-endian?

The usual techniques are to use a pointer:

```
int x = 1;
if(*(char *)&x == 1)
printf("little-endian\n");
else printf("big-endian\n");
```

or a union:

```
union {
int i;
char c[sizeof(int)];
} x;
x.i = 1;
if(x.c[0] == 1)
printf("little-endian\n");
else printf("big-endian\n");
```

(Note that there are also byte order possibilities beyond simple big-endian and little-endian

34 :: How can I implement sets or arrays of bits?

Use arrays of char or int, with a few macros to access the desired bit in the proper cell of the array. Here are some simple macros to use with arrays of char:

```
#include <limits.h> /* 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 BITCLEAR(a, b) ((a)[BITSLOT(b)] &= ~BITMASK(b))
#define BITTEST(a, b) ((a)[BITSLOT(b)] & BITMASK(b))
#define BITNSLOTS(nb) ((nb + CHAR_BIT - 1) / CHAR_BIT)
```

(If you don't have <limits.h>, try using 8 for CHAR_BIT.)

Here are some usage examples. To declare an ``array'' of 47 bits:
char bitarray[BITNSLOTS(47)];

To set the 23rd bit:

BITSET(bitarray, 23);

To test the 35th bit:

if(BITTEST(bitarray, 35)) ...
To compute the union of two bit arrays and place it in a third array (with all three arrays declared as above):
for(i = 0; i < BITNSLOTS(47); i++)
array3[i] = array1[i] | array2[i];

To compute the intersection, use & instead of |.
As a more realistic example, here is a quick implementation of the Sieve of Eratosthenes, for computing prime numbers:

```
#include <stdio.h>
#include <string.h>

#define MAX 10000

int main()
{
char bitarray[BITNSLOTS(MAX)];
int i, j;

memset(bitarray, 0, BITNSLOTS(MAX));

for(i = 2; i < MAX; i++) {
if(!BITTEST(bitarray, i)) {
printf("%d\n", i);
for(j = i + i; j < MAX; j += i)
BITSET(bitarray, j);
}
}
return 0;
}
```

35 :: How can I manipulate individual bits?

Bit manipulation is straightforward in C, and commonly done. To extract (test) a bit, use the bitwise AND (&) operator, along with a bit mask representing the bit(s) you're interested in:
value & 0x04
To set a bit, use the bitwise OR (| or |=) operator:

value |= 0x04

To clear a bit, use the bitwise complement (~) and the AND (& or &=) operators:
value &= ~0x04

(The preceding three examples all manipulate the third-least significant, or 2**2, bit, expressed as the constant bitmask 0x04.)
To manipulate an arbitrary bit, use the shift-left operator (<<) to generate the mask you need:

value & (1 << bitnumber)
value |= (1 << bitnumber)
value &= ~(1 << bitnumber)

Alternatively, you may wish to precompute an array of masks:

unsigned int masks[] =
{0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

value & masks[bitnumber]
value |= masks[bitnumber]
value &= ~masks[bitnumber]

To avoid surprises involving the sign bit, it is often a good idea to use unsigned integral types in code which manipulates bits and bytes.

36 :: If I have a char * variable pointing to the name of a function ...

If I have a char * variable pointing to the name of a function, how can I call that function? Code like
extern int func(int, int);
char *funcname = "func";
int r = (*funcname)(1, 2);

or
r = (*(int (*)(int, int))funcname)(1, 2);

doesn't seem to work.

By the time a program is running, information about the names of its functions and variables (the ``symbol table'') is no longer needed, and may therefore not be available. The most straightforward thing to do, therefore, is to maintain that information yourself, with a correspondence table of names and function pointers:
int one_func(), two_func();
int red_func(), blue_func();

```
struct { char *name; int (*funcptr)(); } symtab[] = {
"one_func", one_func,
"two_func", two_func,
"red_func", red_func,
"blue_func", blue_func,
};
```

Then, search the table for the name, and call via the associated function pointer, with code like this:

```
#include <stddef.h>
int (*findfunc(char *name))()
{
int i;

for(i = 0; i < sizeof(symtab) / sizeof(symtab[0]); i++) {
if(strcmp(name, symtab[i].name) == 0)
return symtab[i].funcptr;
}

return NULL;
}

...
char *funcname = "one_func";
int (*funcp)() = findfunc(funcname);
if(funcp != NULL)
(*funcp)();
```

37 :: How can I write data files which can be read on other machines with different word size, byte order, or floating point formats?

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.

38 :: What is the right way to use errno?

In general, you should detect errors by checking return values, and use errno only to distinguish among the various causes of an error, such as ``File not found'' or ``Permission denied''. (Typically, you use perror or strerror to print these discriminating error messages.) It's only

necessary to detect errors with errno when a function does not have a unique, unambiguous, out-of-band error return (i.e. because all of its possible return values are valid; one example is atoi). In these cases (and in these cases only; check the documentation to be sure whether a function allows this), you can detect errors by setting errno to 0, calling the function, then testing errno. (Setting errno to 0 first is important, as no library function ever does that for you.)
To make error messages useful, they should include all relevant information. Besides the strerror text derived from errno, it may also be appropriate to print the name of the program, the operation which failed (preferably in terms which will be meaningful to the user), the name of the file for which the operation failed, and, if some input file (script or source file) is being read, the name and current line number of that file.

39 :: What is a good data structure to use for storing lines of text?

What's a good data structure to use for storing lines of text? I started to use fixed-size arrays of arrays of char, but they're just too restrictive.

One good way of doing this is with a pointer (simulating an array) to a set of pointers (each simulating an array) of char. This data structure is sometimes called a ``ragged array,'' and looks something like this:
[FIGURE GOES HERE]
You could set up the tiny array in the figure above with these simple declarations:

```
char *a[4] = {"this", "is", "a", "test"};
char **p = a;
```

(where p is the pointer-to-pointer-to-char and a is an intermediate array used to allocate the four pointers-to-char).
To really do dynamic allocation, you'd of course have to call malloc:

```
#include <stdlib.h>
char **p = malloc(4 * sizeof(char *));
if(p != NULL) {
p[0] = malloc(5);
p[1] = malloc(3);
p[2] = malloc(2);
p[3] = malloc(5);

if(p[0] && p[1] && p[2] && p[3]) {
strcpy(p[0], "this");
strcpy(p[1], "is");
strcpy(p[2], "a");
strcpy(p[3], "test");
}
}
```

(Some libraries have a strdup function which would streamline the inner malloc and strcpy calls. It's not Standard, but it's obviously trivial to implement something like it.)
Here is a code fragment which reads an entire file into memory, using the same kind of ragged

array.

```
#include <stdio.h>
#include <stdlib.h>
extern char *agetline(FILE *);
FILE *ifp;
```

40 :: How can I return multiple values from a function?

There are several ways of doing this. (These examples show hypothetical polar-to-rectangular coordinate conversion functions, which must return both an x and a y coordinate.)

1. Pass pointers to several locations which the function can fill in:

```
#include <math.h>

polar_to_rectangular(double rho, double theta,
double *xp, double *yp)
{
*xp = rho * cos(theta);
*yp = rho * sin(theta);
}

...
double x, y;
polar_to_rectangular(1., 3.14, &x, &y);
```

2. Have the function return a structure containing the desired values:

```
struct xycoord { double x, y; };

struct xycoord
polar_to_rectangular(double rho, double theta)
{
struct xycoord ret;
ret.x = rho * cos(theta);
ret.y = rho * sin(theta);
return ret;
}

...
struct xycoord c = polar_to_rectangular(1., 3.14);
```

3. Use a hybrid: have the function accept a pointer to a structure, which it fills in:

```
polar_to_rectangular(double rho, double theta,
struct xycoord *cp)
{
cp->x = rho * cos(theta);
cp->y = rho * sin(theta);
```

```
}

...
struct xycoord c;
polar_to_rectangular(1., 3.14, &c);
```

(Another example of this technique is the Unix system call stat.)

4. In a pinch, you could theoretically use global variables (though this is rarely a good idea).

41 :: Why isnt any of this standardized in C? Any real program has to do some of these things.

Actually, some standardization has occurred along the way. In the beginning, C did not have a standard library at all; programmers always had to ``roll their own'' utility routines. After several abortive attempts, a certain set of library routines (including the str* and stdio families of routines) became a de facto standard, at least on Unix systems, but the library was not yet a formal part of the language. Vendors could (and occasionally did) provide completely different routines along with their compilers.
In ANSI/ISO Standard C, a library definition (based on the 1984 /usr/group standard, and largely compatible with the traditional Unix library) was adopted with as much standing as the language itself. The Standard C library's treatment of file and device I/O is, however, rather minimal. It states how streams of characters are written to and read from files, and it provides a few suggestions about the display behavior of control characters like \b, \r, and \t, but beyond that it is silent. (Many of these issues are, however, addressed and standardized in Posix.)
If the Standard were to attempt to define standard mechanisms for accessing things like keyboards and displays, it might seem to be a big convenience for programmers. But it would be a monumental task: there is already a huge variety of display devices, and huge variation among the operating systems through which they are usually accessed. We cannot assume that the years to come will offer any less variety.

42 :: But I cant use all these nonstandard, system-dependent functions, because my program has to be ANSI compatible!

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; it is a language standard, not an operating system standard. An international standard which does address many of these issues is POSIX (IEEE 1003.1, ISO/IEC 9945-1), and many operating systems (not just Unix) now have POSIX-compatible programming interfaces.
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 either heavily #ifdeffed or rewritten entirely for each system ported to;

43 :: What are near and far pointers?

These days, they're pretty much obsolete; they're definitely system-specific. They had to do with 16-bit programming under MS-DOS and perhaps some early versions of Windows. If you really need to know, see a DOS- or Windows-specific programming reference. If you're using a machine which doesn't require (or permit) making the near/far pointer distinction, just delete the unnecessary ``near'' and ``far'' keywords (perhaps using the preprocessor: ``#define far /* nothing */'').

44 :: I am trying to compile this program

I'm trying to compile this program, but the compiler is complaining that ``union REGS'' is undefined, and the linker is complaining that int86 is undefined.

Those have to do with MS-DOS interrupt programming. They don't exist on other systems.

45 :: How can I ensure that integer arithmetic doesnt overflow?

The usual approach is to test the operands against the limits in the header file <limits.h> before doing the operation. For example, here is a ``careful'' addition function:

```
int
chkadd(int a, int b)
{
if(INT_MAX - b < a) {
fputs("int overflow\n", stderr);
return INT_MAX;
}
return a + b;
}
```

46 :: How can I handle floating-point exceptions gracefully?

On many systems, you can define a function matherr which will be called when there are certain floating-point errors, such as errors in the math routines in <math.h>. You may also be able to use signal to catch SIGFPE

47 :: How can I trap or ignore keyboard interrupts like control-C?

The basic step is to call signal, either as

```
#include <signal.h>
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; 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.

48 :: How can I implement a delay, or time a users response, with sub-second resolution?

Unfortunately, there is no portable way. Routines you might look for on your system include clock, delay, ftime, gettimeofday, msleep, nap, napms, nanosleep, setitimer, sleep, Sleep, times, and usleep. (A function 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 may even have subsecond resolution, if CLOCKS_PER_SEC is greater than 1. However, clock gives elapsed processor time used by the current program, which on a multitasking system (or in a non-CPU-intensive program) 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 multitasking 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++)
;

49 :: How can I read in an object file and jump to locations in it?

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., and this approach can't work if code and data reside in separate address spaces or if code is otherwise privileged.
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 containing routines like dlopen and dlsym which

allow object files to be dynamically loaded. Under VMS, use LIB$FIND_IMAGE_SYMBOL. GNU has a package called ``dld''.

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

Yes and no. The Standard says that a return from the initial call to main is equivalent to calling exit. However, a return from main cannot be expected to work if data local to main might be needed during cleanup; A few very old, nonconforming systems may once have had problems with one or the other form. (Finally, the two forms are obviously not equivalent in a recursive call to main.)

51 :: How can I open files mentioned on the command line, and parse option flags?

Here is a skeleton which implements a traditional Unix-style argv parse, handling option flags beginning with -, and optional filenames. (The two flags accepted by this example are -a and -b; -b takes an argument.)

```
#include <stdio.h>
#include <string.h>
#include <errno.h>

main(int argc, char *argv[])
{
int argi;
int aflag = 0;
char *bval = NULL;

for(argi = 1; argi < argc && argv[argi][0] == '-'; argi++) {
char *p;
for(p = &argv[argi][1]; *p != '\0'; p++) {
switch(*p) {
case 'a':
aflag = 1;
printf("-a seen\n");
break;

case 'b':
bval = argv[++argi];
printf("-b seen (\"%s\")\n", bval);
break;

default:
fprintf(stderr,
"unknown option -%c\n", *p);
}
}
```

```
}

if(argi >= argc) {
/* no filename arguments; process stdin */
printf("processing standard input\n");
} else {
/* process filename arguments */

for(; argi < argc; argi++) {
FILE *ifp = fopen(argv[argi], "r");
if(ifp == NULL) {
fprintf(stderr, "can't open %s: %s\n",
argv[argi], strerror(errno));
continue;
}

printf("processing %s\n", argv[argi]);

fclose(ifp);
}
}

return 0;
}
```

52 :: How can a process change an environment variable in its caller?

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. (The environment of the parent process can only be altered if the parent is explicitly set up to listen for some kind of change requests. The conventional execution of the BSD ``tset'' program in .profile and .login files effects such a scheme.) Under MS-DOS, it's possible to manipulate the master copy of the environment, but the required techniques are arcane.

53 :: How can I automatically locate a programs configuration files in the same directory as the executable?

It's hard, in general; 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.)

54 :: How can my program discover the complete pathname to the executable from which it was invoked?

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.

55 :: How can I invoke another program or command and trap its output?

Unix and some other systems provide a popen function, which sets up a stdio stream on a pipe connected to the process running a command, so that the calling program can read the output (or alternatively supply the input). Using popen,

```
extern FILE *popen();
sprintf(cmdbuf, "sort < %s", datafile);

fp = popen(cmdbuf, "r");

/* ...now read sorted data from fp... */

pclose(fp);
```

(Do be sure to call pclose, as shown; leaving it out will seem to work at first but may eventually run you out of processes or file descriptors.)
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.)

56 :: How do I get an accurate error status return from system on MS-DOS?

You can't; COMMAND.COM doesn't tend to provide one. If you don't need COMMAND.COM's services (i.e. if you're just trying to invoke a simple program, without I/O redirection and such) try one of the spawn routines, instead.

57 :: How can I call system when parameters (filenames, etc.) of the executed command arent known until run time?

Just use sprintf (or perhaps strcpy and strcat) to build the command string in a buffer, then call system with that buffer. (Make sure the buffer is allocated with enough space;
Here is a contrived example suggesting how you might build a data file, then sort it (assuming the existence of a sort utility, and Unix- or MS-DOS-style input/output redirection):

```
char *datafile = "file.dat";
char *sortedfile = "file.sort";
char cmdbuf[50];
FILE *fp = fopen(datafile, "w");
```

```
/* ...write to fp to build data file... */

fclose(fp);

sprintf(cmdbuf, "sort < %s > %s", datafile, sortedfile);
system(cmdbuf);

fp = fopen(sortedfile, "r");
/* ...now read sorted data from fp... */
```

58 :: How can I invoke another program (a standalone executable, or an operating system command) from within a C program?

Use the library function system, which does exactly that.
Some systems also provide a family of spawn routines which accomplish approximately the same thing. system is more ``portable'' in that it is required under the ANSI C Standard, although the interpretation of the command string--its syntax and the set of commands accepted--will obviously vary tremendously.
The system function ``calls'' a command in the manner of a subroutine, and control eventually returns to the calling program. If you want to overlay the calling program with another program (that is, a ``chain'' operation) you'll need a system-specific routine, such as the exec family on Unix.
Note that system's return value is at best the command's exit status (although even that is not guaranteed), and usually has nothing to do with the output of the command.

59 :: How can I do PEEK and POKE in C?

How can I access memory (a memory-mapped device, or graphics memory) located at a certain address? How can I do PEEK and POKE in C?

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. If the location is a memory-mapped I/O register, you will probably also want to use the volatile qualifier: ``volatile unsigned int *magicloc''. (If you want to refer to a byte at a certain address rather than a word, use unsigned char *.)
Under MS-DOS, you may find a macro like MK_FP() handy for working with segments and offsets. As suggested by Gary Blaine, you can also declare tricky array pointers which allow you to access screen memory using array notation. For example, on an MS-DOS machine in an 80x25 text mode, given the declaration

```
unsigned short (far * videomem)[80] =
(unsigned short (far *)[80])0xb8000000;
```

you can access the character and attribute byte at row i, column j with videomem[i][j].
Many operating systems execute user-mode programs in a protected mode where direct access to

I/O devices (or to any address outside the running process) is simply not possible. In such cases you will have to ask the operating system to carry out I/O operations for you.

60 :: I thought that using large model meant that I could use more than 64K of data!

Q: 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.''

61 :: How can I allocate arrays or structures bigger than 64K?

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 single objects can be 32K or larger, or 64K for C99.) 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, 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 64K or 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 , or some kind of a DOS extender, or another operating system.

62 :: How can I find out how much memory is available?

Your operating system may provide a routine which returns this information, but it's quite system-dependent. (Also, the number may vary over time.) If you're trying to predict whether you'll be able to allocate a certain amount of memory, just try it--call malloc (requesting that amount) and check the return value.

63 :: How do I create a directory? How do I remove a directory (and its contents)?

If your operating system supports these services, they are likely to be provided in C via functions named mkdir and rmdir. Removing a directory's contents as well will require listing them and

calling remove . If you don't have these C functions available, try system along with your operating system's delete command(s).

64 :: How can I read a directory in a C program?

See if you can use the opendir and readdir functions, 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, and MS Windows has FindFirstFile and FindNextFile.) readdir returns just the file names; if you need more information about the file, try calling stat. To match filenames to some wildcard pattern,
Here is a tiny example which lists the files in the current directory:

```
#include <stdio.h>
#include <sys/types.h>
#include <dirent.h>

main()
{
struct dirent *dp;
DIR *dfd = opendir(".");
if(dfd != NULL) {
while((dp = readdir(dfd)) != NULL)
printf("%s\n", dp->d_name);
closedir(dfd);
}
return 0;
}
```

(On older systems, the header file to #include may be <direct.h> or <dir.h>, and the pointer returned by readdir may be a struct direct *. This example assumes that "." is a synonym for the current directory.)
In a pinch, you could use popen to call an operating system list-directory program, and read its output. (If you only need the filenames displayed to the user, you could conceivably use system

65 :: How can I find out how much free space is available on disk?

There is no portable way. Under some versions of Unix you can call statfs. Under MS-DOS, use interrupt 0x21 subfunction 0x36, or perhaps a routine such as diskfree. Another possibility is to use popen to invoke and read the output of a ``disk free" command (df on Unix).
(Note that the amount of free space apparently available on a disk may not match the size of the largest file you can store, for all sorts of reasons.)

66 :: How can I increase the allowable number of simultaneously open files?

Q; I'm getting an error, ``Too many open files". How can I increase the allowable number of simultaneously open files?

A: There are typically 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.

67 :: fopen isnt letting me open files like "$HOME/.profile" and "~/.myrcfile".

fopen isn't letting me open files like "$HOME/.profile" and "~/.myrcfile".

Under Unix, at least, environment variables like $HOME, along with the home-directory notation involving the ~ character, are expanded by the shell, and there's no mechanism to perform these expansions automatically when you call fopen.

68 :: Why cant I open a file by its explicit path?

Why can't I open a file by its explicit path? The call
fopen("c:\newdir\file.dat", "r")
is failing.

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 function) 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.)

69 :: How do I copy files?

Either use system() to invoke your operating system's copy utility, or open the source and destination files (using fopen or some lower-level file-opening system call), read characters or blocks of characters from the source file, and write them to the destination file. Here is a simple example:

```
#include <stdio.h>
int copyfile(char *fromfile, char *tofile)
{
FILE *ifp, *ofp;
int c;
```

```
if((ifp = fopen(fromfile, "r")) == NULL) return -1;
if((ofp = fopen(tofile, "w")) == NULL) { fclose(ifp); return -1; }

while((c = getc(ifp)) != EOF)
putc(c, ofp);

fclose(ifp);
fclose(ofp);

return 0;
}
```

To copy a block at a time, rewrite the inner loop as

```
while((r = fread(buf, 1, sizeof(buf), ifp))> 0)
fwrite(buf, 1, r, ofp);
```

where r is an int and buf is a suitably-sized array of char.

70 :: How can I delete a file?

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.

71 :: How can I recover the file name given an open stream or file descriptor?

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 as you open them (perhaps with a wrapper function around fopen).

72 :: ow can I insert or delete a line (or record) in the middle of a file?

In general, there is no way to do this. The usual solution is simply to rewrite the file.
When you find yourself needing to insert data into an existing file, here are a few alternatives you can try:
* Rearrange the data file so that you can append the new information at the end.
* Put the information in a second file.
* Leave some blank space (e.g. a line of 80 spaces, or a field like 0000000000) in the file when it is first written, and overwrite it later with the final information . (This technique is most portable in binary mode; on some systems, overwriting a text file may truncate it.) * Use a database instead of a flat file.

Instead of actually deleting records, you might consider just marking them as deleted, and having the code which reads the file ignore them. (You could run a separate coalescion program once in a while to rewrite the file, finally discarding the deleted records. Or, if the records are all the same length, you could take the last record and use it to overwrite the record to be deleted, then truncate the file.)

73 :: How can I find the modification date and time of a file?

The Unix and POSIX function is stat, which several other systems supply as well.

74 :: How can I find out the size of a file, prior to reading it in?

If the ``size of a file'' is the number of characters you'll be able to read from it in C (or which were written to it by a previous program), it can be difficult or impossible to determine this number exactly (other than by reading the whole file).
Under Unix, the stat call (specifically, the st_size field of the stat structure) will give you an exact answer.Several other systems supply a Unix-like stat call, but the sizes reported for text files may be approximate (due to differing end-of-line representations; . You can open the file and use fstat, or fseek to the end of the file 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 (but, strictly speaking, binary files don't necessarily support fseek to SEEK_END at all). Some systems provide functions called filesize or filelength, but these are obviously 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. (In general, your program should behave gracefully if the number of characters actually read does not match prior expectations, since any advance determination of the size might be approximate.)

75 :: How can I check whether a file exists? I want to warn the user if a requested input file is missing.

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 functions are stat, access, and fopen. (To make an approximate test using fopen, just open for reading and close immediately, although failure does not necessarily indicate nonexistence.) 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. (If you have the choice, the best compromise is probably one of the stat functions.)
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.)

76 :: How can I send mail from within a C program?

Under Unix, open a pipe to the mail program, or perhaps /usr/lib/sendmail.

77 :: How can I do graphics in C?

Once upon a time, Unix had a fairly nice little set of device-independent plot functions described in plot(3) and plot(5). The GNU libplot library, written by Robert Maier, maintains the same spirit and supports many modern plot devices; see
http://www.gnu.org/software/plotutils/plotutils.html.
A modern, platform-independent graphics library (which also supports 3D graphics and animation) is OpenGL. Other graphics standards which may be of interest are GKS and PHIGS.
A modern, platform-independent graphics library (which also supports 3D graphics and animation) is OpenGL. Other graphics standards which may be of interest are GKS and PHIGS.
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; 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;

78 :: How can I access an I O board directly?

At one level, at least, it's quite simple: you have a device register which is actually wired up so that the bits written to it get coverted to actual voltage levels in the real world that you can do interesting things with. In general, there are two ways to get the bits in and out. (A particular I/O board will use one method or the other; you'll need to consult its documentation for details.)

1. If the device is accessed via a dedicated ``I/O port'', use system-specific functions to communicate with it. Under MS-DOS, for example, there were quasistandard ``inport'' and ``outport'' instructions.
2. If the device uses ``memory-mapped I/O'', that is, if the device register(s) are accessed as if they were normal memory at particular, known locations within the processor's addressing space, use contrived pointer variables to access those locations.

79 :: How do I send escape sequences to control a terminal or other device?

If you can figure out how to send characters to the device at all , 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 .
Some programmers prefer to parameterize the ESC code, like this:
#define ESC 033
fprintf(ofd, "%c[J", ESC);

80 :: How can I direct output to the printer?

Under Unix, either use popen 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. Under some circumstances, another (and perhaps the only) possibility is to use a window manager's screen-capture function, and print the resulting bitmap.

81 :: How can I do serial comm port I O?

How can I do serial (``comm'') port I/O?

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.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.

82 :: How do I read the arrow keys? What about function keys?

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 scan 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.)

83 :: How can I make it pause before closing the program output window?

I'm compiling some test programs on a windows-based system, and the windows containing my program's output are closing so quickly after my program calls exit that I can't see the output. How can I make it pause before closing?

After wondering why the author of your compiler's run-time system didn't take care of this for you, simply add the lines
printf("Hit RETURN to exit"\n");
fflush(stdout); (void)getchar();

just before the end of your program. (If you want to wait for any keystroke, not just the RETURN key

84 :: How can I display a percentage-done indication that updates itself in place, or show one of those twirling baton progress indicators?

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.

Using these characters, you can print a percentage-done indicator:

```
for(i = 0; i < lotsa; i++) {
printf("\r%3d%%", (int)(100L * i / lotsa));
fflush(stdout);
do_timeconsuming_work();
}
printf("\ndone.\n");
```

or a baton:

```
printf("working: ");
for(i = 0; i < lotsa; i++) {
printf("%c\b", "|/-\\"[i%4]);
fflush(stdout);
do_timeconsuming_work();
}
printf("done.\n");
```

85 :: How can I find out if there are characters available for reading?

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?

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. You can also try setting an alarm to cause a blocking read to time out after a certain interval (under Unix, look at alarm, signal, and maybe setitimer). If what you're trying to do is read input from several sources without blocking, you will definitely want to use some kind of a ``select" call, because a busy-wait, polling loop is terribly inefficient on a multitasking system.

86 :: I need code to parse and evaluate expressions.

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").

87 :: Dont ANSI function prototypes render lint obsolete?

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.

88 :: Where can I get an ANSI-compatible lint?

Products called PC-Lint and FlexeLint are available from Gimpel Software.
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 Splint (formerly lclint).
In the absence of lint, many modern compilers do attempt to diagnose almost as many problems as lint does. (Many netters recommend gcc -Wall -pedantic

89 :: How can I shut off the warning ...

How can I shut off the ``warning: possible pointer alignment problem'' message which lint gives me for each call to malloc?

A modern lint shouldn't be complaining about this.
Once upon a time, lint did not and could not know that malloc ``returns a pointer to space suitably aligned for storage of any type of object.'' There were various kludgey workarounds for this problem, but today, the void * type exists precisely to encapsulate the notion of a ``generic'' pointer, and an ANSI-compatible lint should understand this.

90 :: I just typed in this program, and it is acting strangely. Can you see anything wrong with it?

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, taking the attitude that it's not their problem if you didn't say what you meant, or if what you said is virtually guaranteed not to work. (But do also see if your compiler has extra warning levels which can be optionally requested.)

91 :: People always say that good style is important

People always say that good style is important, but when they go out of their way to use clear techniques and make their programs readable, they seem to end up with less efficient programs. Since efficiency is so important, isn't it necessary to sacrifice some style and readability?

It's true that grossly inefficient programs are a problem, but the blind zeal with which many programmers often chase efficiency is also a problem. Cumbersome, obscure programming tricks not only destroy readability and maintainability, but they may actually lead to slimmer long-term efficiency improvements than would more appropriate design or algorithm choices. With care, it is possible to design code which is both clean and efficient.

92 :: What is Hungarian Notation? Is it worthwhile?

Hungarian Notation is a naming convention, invented by Charles Simonyi, which encodes information 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.

93 :: Should I use symbolic names like TRUE and FALSE for Boolean constants, or plain 1 and 0?

It's your choice. Preprocessor macros like TRUE and FALSE (and, of course, NULL) are used for code readability, not because the underlying values might ever change. It's a matter of style, not correctness, whether to use symbolic names or raw 1/0 values.
On the one hand, using a symbolic name like TRUE or FALSE reminds the reader that a Boolean value is involved. 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.

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

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. 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.
It is only in pointer contexts that NULL and 0 are equivalent. 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.

95 :: I came across some code that puts a (void) cast before each call to printf. Why?

printf does return a value (the number of characters printed, or an error code), 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

96 :: I have seen function declarations that look like this

I've seen function declarations that look like this:
extern int func __((int, int));
What are those extra parentheses and underscores for?

They're part of a trick which allows the prototype part of the function declaration to be turned off for a pre-ANSI compiler. Somewhere else is a conditional definition of the __ macro like this:
#ifdef __STDC__
#define __(proto) proto
#else
#define __(proto) ()
#endif

The extra parentheses in the invocation
extern int func __((int, int));
are required so that the entire prototype list (perhaps containing many commas) is treated as the single argument expected by the macro.

97 :: Why do some people write if(0 == x) instead of if(x == 0)?

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 for some people to remember to reverse the test than to remember to type the doubled = sign. (To be sure, accidentally using = instead of == is a typo which even the most experienced C programmer can make.)
On the other hand, some people find these reversed tests ugly or distracting, and argue that a compiler should warn about if(x = 0). (In fact, many compilers do warn about assignments in conditionals, though you can always write if((x = expression)) or if((x = expression) != 0) if you really mean it.)

98 :: Here is a neat trick for checking whether two strings are equal

Here's a neat trick for checking whether two strings are equal:
if(!strcmp(s1, s2))
Is this good style?

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.
Another option is to define a macro:
#define Streq(s1, s2) (strcmp((s1), (s2)) == 0)
which you can then use like this:
if(Streq(s1, s2))

Another option (which borders on preprocessor abuse;is to define
#define StrRel(s1, op, s2) (strcmp(s1, s2) op 0)
after which you can say things like
if(StrRel(s1, ==, s2)) ...
if(StrRel(s1, !=, s2)) ...
if(StrRel(s1, >=, s2)) ...

99 :: How should functions be apportioned among source files?

Usually, related functions are put together in one file. Sometimes (as when developing libraries) it is appropriate to have exactly one source file (and, consequently, one object module) per independent function. Other times, and especially for some programmers, numerous source files can be cumbersome, and it may be tempting (or even appropriate) to put most or all of a program in a few big source files. When it is desired to limit the scope of certain functions or global variables by using the static keyword, source file layout becomes more constrained: the static functions and variables and the functions sharing access to them must all be in the same file.
In other words, there are a number of tradeoffs, so it is difficult to give general rules.

100 :: What is the best style for code layout in C?

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.
Each of the various popular styles has its good and bad points. Putting the open brace on a line by itself wastes vertical space; combining it with the following line makes it hard to edit; combining it with the previous line prevents it from lining up with the close brace and may make it harder to see.
Indenting by eight columns per level is most common, but often gets you uncomfortably close to the right margin (which may be a hint that you should break up the function). If you indent by one tab but set tabstops at something other than eight columns, you're requiring other people to read your code with the same software setup that you used. .
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.

101 :: Why does this code crash?

Why does this code:
char *p = "hello, world!";
p[0] = 'H';
crash?

String constants are in fact constant. The compiler may place them in nonwritable storage, and it is therefore not safe to modify them. When you need writable strings, you must allocate writable

memory for them, either by declaring an array, or by calling malloc. Try
char a[] = "hello, world!";
By the same argument, a typical invocation of the old Unix mktemp routine
char *tmpfile = mktemp("/tmp/tmpXXXXXX");
is nonportable; the proper usage is
char tmpfile[] = "/tmp/tmpXXXXXX";
mktemp(tmpfile);

102 :: This program runs perfectly on one machine ...

This program runs perfectly on one machine, but I get weird results on another. Stranger still, adding or removing a debugging printout changes the symptoms.
Lots of things could be going wrong; here are a few of the more common things to check:
* uninitialized local variables
integer overflow, especially on 16-bit machines, especially of an intermediate result when doing things like a * b / c
* undefined evaluation order
* omitted declaration of external functions, especially those which return something other than int, or have ``narrow'' or variable arguments
* dereferenced null pointers
* improper malloc/free use: assuming malloc'ed memory contains 0, assuming freed storage persists, freeing something twice, corrupting the malloc arena
* pointer problems in general
* mismatch between printf format and arguments, especially trying to print long ints using %d
* trying to allocate more memory than an unsigned int can count, especially on machines with limited memory
* array bounds problems, especially of small, temporary buffers, perhaps used for constructing strings with sprintf
* invalid assumptions about the mapping of typedefs, especially size_t
* floating point problems
* 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.

103 :: I have a program that seems to run correctly

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?

There are at least three things to look for:
1. If a semicolon in a previous declaration is missing, main might be inadvertently declared as returning a structure, conflicting with the run-time startup code's expectations.
2. If setbuf or setvbuf is called, and if the supplied buffer is an automatic, local variable of main (or any function), the buffer may not exist any more by the time the stdio library tries to perform

its final cleanup.
3. A cleanup function registered by atexit may have an error. Perhaps it is trying to reference data local to main or to some other function which no longer exists.

104 :: This program crashes before it even runs!

This program crashes before it even runs! (When single-stepping with a debugger, it dies before the first statement in main.)

You probably have one or more very large (kilobyte or more) local arrays. Many systems have fixed-size stacks, and even 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;
Other possibilities are that your program has been linked incorrectly (combining object modules compiled with different compilation options, or using improper dynamic libraries), or that run-time dynamic library linking is failing for some reason, or that you have somehow misdeclared main.

105 :: Why isnt my procedure call working? The compiler seems to skip right over it

Does the code look like this?
myprocedure;
C has only functions, and function calls always require parenthesized argument lists, even if empty. Use
myprocedure();
Without the parentheses, the reference to the function name simply generates a pointer to the function, which is then discarded.

106 :: I am getting baffling syntax errors which make no sense at all

I'm getting baffling syntax errors which make no sense at all, and it seems like large chunks of my program aren't being compiled.

Check for unclosed comments, mismatched #if/#ifdef/#ifndef/#else/#endif directives, and perhaps unclosed quotes; remember to check header files, too.

107 :: Why is this loop always executing once?

Why is this loop always executing once?

```
for(i = start; i < end; i++);
{
printf("%d\n", i);
}
```

The accidental extra semicolon hiding at the end of the line containing the for constitutes a null statement which is, as far as the compiler is concerned, the loop body. The following brace-enclosed block, which you thought (and the indentation suggests) was a loop body, is actually the next statement, and it is traversed exactly once, regardless of the number of loop iterations.

108 :: How can I call a function with an argument list built up at run time?

There is no guaranteed or portable way to do this.
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.)

109 :: I cant get va_arg to pull in an argument of type pointer-to-function.

Try using a typedef for the function pointer type.
The type-rewriting games which the va_arg macro typically plays are stymied by overly-complicated types such as pointer-to-function. To illustrate, a simplified implementation of va_arg is

```
#define va_arg(argp, type) \
(*(type *)(((argp) += sizeof(type)) - sizeof(type)))
```

where argp's type (va_list) is char *. When you attempt to invoke

```
va_arg(argp, int (*)())
```

the expansion is

```
(*(int (*)() *)(((argp) += sizeof(int (*)())) - sizeof(int (*)())))
```

which is a syntax error (the first cast (int (*)() *) is meaningless).
If you use a typedef for the function pointer type, however, all will be well. Given

```
typedef int (*funcptr)();
```

the expansion of

```
va_arg(argp, funcptr)
```

is

```
(*(funcptr *)(((argp) += sizeof(funcptr)) - sizeof(funcptr)))
```

which works correctly.

110 :: I have a varargs function which accepts a float parameter

I have a varargs function which accepts a float parameter. Why isn't

```
va_arg(argp, float)
```

working?

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. (For analogous reasons, the last ``fixed'' argument, as handed to va_start, should not be widenable, either.)

111 :: My compiler isnt letting me declare a function

My compiler isn't letting me declare a function
int f(...)
{
}

i.e. accepting a variable number of arguments, but with no fixed arguments at all.

A: Standard C requires at least one fixed argument, in part so that you can hand it to va_start. (In any case, you often need a fixed argument to determine the number, and perhaps the types, of the variable arguments.)

112 :: How can I discover how many arguments a function was actually called with?

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 . Finally, if the 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 supply).

113 :: How can I write a function analogous to scanf

How can I write a function analogous to scanf, i.e. that accepts similar arguments, and calls scanf to do most of the work?

C99 (but not any earlier C Standard) supports vscanf, vfscanf, and vsscanf.

114 :: How can I write a function that takes a format string and a variable number of arguments

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?

Use vprintf, vfprintf, or vsprintf. These routines are like their counterparts printf, fprintf, and

sprintf, except that instead of a variable-length argument list, they accept a single va_list pointer. As an example, here is an error function which prints an error message, preceded by the string ``error: '' and terminated with a newline:

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

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

115 :: I had a frustrating problem which turned out to be caused by the line

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.

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. The programmer must make sure that arguments match, or must manually insert explicit casts.
In the case of printf-like functions, some compilers (including gcc) and some versions of lint are able to check the actual arguments against the format string, as long as the format string is an immediate string literal.

116 :: How can f be used for both float and double arguments in printf? Are not they different types?

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. Therefore, printf's %f format always sees a double. (Similarly, %c always sees an int, as does %hd.)

117 :: I heard that you have to include stdio.h before calling printf. Why?

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.

118 :: I am having trouble with a Turbo C program which crashes

I'm having trouble with a Turbo C program which crashes and says something like ``floating point formats not linked."

Some compilers for small machines, including Turbo C (and Ritchie's original PDP-11 compiler), leave out certain floating point support if it looks like it will not be needed. In particular, the non-floating-point versions of printf and scanf save space by not including code to handle %e, %f, and %g. It happens that Borland's heuristics for determining whether the program uses floating point are insufficient, and the programmer must sometimes insert a dummy call to a floating-point library function (such as sqrt; any will do) to force loading of floating-point support.
A partially-related problem, resulting in a similar error message (perhaps ``floating point not loaded") can apparently occur under some MS-DOS compilers when an incorrect variant of the floating-point library is linked. Check your compiler manual's description of the various floating-point libraries.

119 :: What is a good way to implement complex numbers in C?

It is straightforward to define a simple structure and some arithmetic functions to manipulate them. C99 supports complex as a standard type. Here is a tiny example, to give you a feel for it:

```
typedef struct {
double real;
double imag;
} complex;

#define Real(c) (c).real
#define Imag(c) (c).imag

complex cpx_make(double real, double imag)
{
complex ret;
ret.real = real;
ret.imag = imag;
return ret;
}

complex cpx_add(complex a, complex b)
{
return cpx_make(Real(a) + Real(b), Imag(a) + Imag(b));
}
```

You can use these routines with code like

```
complex a = cpx_make(1, 2);
complex b = cpx_make(3, 4);
complex c = cpx_add(a, b);

or, even more simply,
complex c = cpx_add(cpx_make(1, 2), cpx_make(3, 4));
```

120 :: The predefined constant M_PI seems to be missing from my machines copy of math.h.

That constant (which is apparently supposed to be the value of pi, accurate to the machine's precision), is not standard; in fact a standard-conforming copy of should not #define a symbol M_PI. If you need pi, you'll have to define it yourself, or compute it with 4*atan(1.0) or acos(-1.0). (You could use a construction like

```
#ifndef M_PI
#define M_PI 3.1415926535897932385
#endif
```

to provide your own #definition only if some system header file has not.)

121 :: Why does not C have an exponentiation operator?

One reason is probably that few processors have a built-in exponentiation instruction. C has a pow function (declared in <math.h>) for performing exponentiation, although explicit multiplication is usually better for small positive integral exponents. In other words, pow(x, 2.) is probably inferior to x * x. (If you're tempted to make a Square() macro, though, check

122 :: How do I round numbers?

The simplest and most straightforward way is with code like

```
(int)(x + 0.5)
```

C's floating to integer conversion truncates (discards) the fractional part, so adding 0.5 before truncating arranges that fractions >= 0.5 will be rounded up. (This technique won't work properly for negative numbers, though, for which you could use something like (int)(x < 0 ? x - 0.5 : x + 0.5), or play around with the floor and ceil functions.)

You can round to a certain precision by scaling:

```
(int)(x / precision + 0.5) * precision
```

Handling negative numbers, or implementing even/odd rounding, is slightly trickier.

Note that because truncation is otherwise the default, it's usually a good idea to use an explicit rounding step when converting floating-point numbers to integers. Unless you're careful, it's quite possible for a number which you thought was 8.0 to be represented internally as 7.999999 and to be truncated to 7.

123 :: I am sure I have got the trig functions declared correctly, but they are still giving me wrong answers.

You weren't handing them angles in degrees, were you? C's trig functions (like FORTRAN's and most other languages) accept angles in radians. The conversion from degrees to radians is simple enough:
sin(degrees * pi / 180)

124 :: My floating-point calculations are acting strangely and giving me different answers on different machines.

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. Among other things, the associative and distributive laws do not hold completely; that is, order of operation may be important, and repeated addition is not necessarily equivalent to multiplication. 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; Beware that some machines have more precision available in floating-point computation registers than in double values stored in memory, which can lead to floating-point inequalities when it would seem that two values just have to be equal.
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" , otherwise a compiler for a machine without the ``right" model would have to do prohibitively expensive emulations.
This document cannot begin to list the pitfalls associated with, and workarounds appropriate for, floating-point work. A good numerical programming text should cover the basics; (Beware, though, that subtle problems can occupy numerical analysts for years.)

125 :: I am trying to do some simple trig, and I am #including <math.h>, but the linker keeps complaining that functions like sin and cos are undefined

Make sure you're actually linking with the math library. For instance, due to a longstanding bug in Unix and Linux systems, you usually need to use an explicit -lm flag, at the end of the command line, when compiling/linking.

126 :: When I set a float variable to, say, 3.1, why is printf printing it as 3.0999999?

Most computers use base 2 for floating-point numbers as well as for integers, and just as for base 10, not all fractions are representable exactly in base 2. It's well-known that in base 10, a fraction like 1/3 = 0.333333... repeats infinitely. It turns out that in base 2, one tenth is also an infinitely-repeating fraction (0.0001100110011...), so exact decimal fractions such as 3.1 cannot be represented exactly in binary. 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 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).

127 :: What does it mean when the linker says that _end is undefined?

That message is a quirk of the old Unix linkers. You get an error about _end being undefined only when other symbols are undefined, too--fix the others, and the error about _end will disappear.

128 :: What is the difference between memcpy and memmove?

memmove offers guaranteed behavior if the memory regions pointed to by 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.
It seems simple enough to implement memmove; the overlap guarantee apparently requires only an additional test:

```
void *memmove(void *dest, void const *src, size_t n)
{
register char *dp = dest;
register char const *sp = src;
if(dp < sp) {

while(n-- > 0)
*dp++ = *sp++;
} else {
dp += n;
sp += n;
while(n-- >> 0)
*--dp = *--sp;
}

return dest;
}
```

The problem with this code is in that additional test: the comparison (dp < sp) is not quite portable (it compares two pointers which do not necessarily point within the same object) and may not be as cheap as it looks. On some machines (particularly segmented architectures), it may be tricky and significantly less efficient to implement.

129 :: How can I generate floating-point random numbers?

drand48 is a Unix System V routine which returns floating point random numbers (presumably with 48 bits of precision) in the half-open interval [0, 1). (Its companion seed routine is srand48; neither is in the C Standard.) It's easy to write a low-precision replacement:

```
#include <stdlib.h>
double drand48()
{
return rand() / (RAND_MAX + 1.); }
```

To more accurately simulate drand48's semantics, you can try to give it closer to 48 bits worth of precision:

```
#define PRECISION 2.82e14 /* 2**48, rounded up */
double drand48()
{
double x = 0;
double denom = RAND_MAX + 1.;
double need;

for(need = PRECISION; need > 1;
need /= (RAND_MAX + 1.)) {
x += rand() / denom;
denom *= RAND_MAX + 1.;
}

return x;
}
```

Before using code like this, though, beware that it is numerically suspect, particularly if (as is usually the case) the period of rand is on the order of RAND_MAX. (If you have a longer-period random number generator available, such as BSD random, definitely use it when simulating drand48.)

130 :: How can I retHow can I return a sequence of random numbers which dont repeat at all?

What you're looking for is often called a ``random permutation'' or ``shuffle.'' One way is to initialize an array with the values to be shuffled, then randomly interchange each of the cells with another one later in the array:

```
int a[10], i, nvalues = 10;

for(i = 0; i < nvalues; i++)
a[i] = i + 1;

for(i = 0; i < nvalues-1; i++) {
int c = randrange(nvalues-i);
int t = a[i]; a[i] = a[i+c]; a[i+c] = t; /* swap */
}
```

where randrange(N) is rand() / (RAND_MAX/(N) + 1)

131 :: How can I get random integers in a certain range?

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.
A better method is something like
(int)((double)rand() / ((double)RAND_MAX + 1) * N)
If you'd rather not use floating point, another method is
rand() / (RAND_MAX / N + 1)
If you just need to do something with probability 1/N, you could use
if(rand() < (RAND_MAX+1u) / N)
All these methods obviously require knowing RAND_MAX (which ANSI #defines in <stdlib.h>), and assume that N is much less than RAND_MAX.
When N is close to RAND_MAX, and if the range of the random number generator is not a multiple of N (i.e. if (RAND_MAX+1) % N != 0), all of these methods break down: some outputs occur more often than others. (Using floating point does not help; the problem is that rand returns RAND_MAX+1 distinct values, which cannot always be evenly divvied up into N buckets.) If this is a problem, about the only thing you can do is to call rand multiple times, discarding certain values:

```
unsigned int x = (RAND_MAX + 1u) / N;
unsigned int y = x * N;
unsigned int r;
do {
r = rand();
} while(r >= y);
return r / x;
```

132 :: Did C have any Year 2000 problems?

No, although poorly-written C programs might have.
The tm_year field of struct tm holds the value of the year minus 1900; this field therefore contains the value 100 for the year 2000. Code that uses tm_year correctly (by adding or subtracting 1900 when converting to or from human-readable 4-digit year representations) has no problems at the turn of the millennium. Any code that used tm_year incorrectly, however, such as by using it directly as a human-readable 2-digit year, or setting it from a 4-digit year with code like

```
tm.tm_year = yyyy % 100; /* WRONG */
```

or printing it as an allegedly human-readable 4-digit year with code like

```
printf("19%d", tm.tm_year); /* WRONG */
```

would have had grave y2k problems indeed.
(The y2k problem is now mostly old history; all we have left to do is fix all the 32-bit time_t problems by 2038...)

133 :: How can I sort more data than will fit in memory?

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:

134 :: How can I sort a linked list?

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.

135 :: How can I get the current date or time of day in a C program?

Just use the time, ctime, localtime and/or strftime functions. Here is a simple example:

```
#include <stdio.h>
#include <time.h>

int main()
{
time_t now;
time(&now);
printf("It's %s", ctime(&now));
return 0;
}
```

Calls to localtime and strftime look like this:

```
struct tm *tmp = localtime(&now);
char fmtbuf[30];
printf("It's %d:%02d:%02d\n",
tmp->tm_hour, tmp->tm_min, tmp->tm_sec);
strftime(fmtbuf, sizeof fmtbuf, "%A, %B %d, %Y", tmp);
printf("on %s\n", fmtbuf);
```

(Note that these functions take a pointer to the time_t variable, even when they will not be modifying it.

136 :: How can I split up a string into whitespace-separated fields?

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?

The only Standard function 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.) Here is a usage example, which simply prints each field as it's extracted:

```
#include <stdio.h>
#include <string.h>
char string[] = "this is a test"; /* not char *;
```

```
char *p;
for(p = strtok(string, " \t\n"); p != NULL;
p = strtok(NULL, " \t\n"))
printf("\"%s\"\n", p);
```

As an alternative, here is a routine I use for building an argv all at once:

```
#include <ctype.h>
int makeargv(char *string, char *argv[], int argvsize)
{
char *p = string;
int i;
int argc = 0;

for(i = 0; i < argvsize; i++) {
/* skip leading whitespace */
while(isspace(*p))
p++;

if(*p != '\0')
argv[argc++] = p;
else {
argv[argc] = 0;
break;
}

/* scan over arg */
while(*p != '\0' && !isspace(*p))
p++;
/* terminate arg: */
if(*p != '\0' && i < argvsize-1)
*p++ = '\0';
}

return argc;
}
```

Calling makeargv is straightforward:

```
char *av[10];
int i, ac = makeargv(string, av, 10);
for(i = 0; i < ac; i++)
printf("\"%s\"\n", av[i]);
```

137 :: Why do some versions of toupper act strangely if given an upper-case letter?

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

In earlier times, toupper was a function-like preprocessor macro and was defined to work only on lower-case letters; it misbehaved if applied to digits, punctuation, or letters which were already upper-case. Similarly, tolower worked only on upper-case letters. Therefore, old code (or code written for wide portability) tends to call islower before toupper, and isupper before tolower. The C Standard, however, says that toupper and tolower must work correctly on all characters, i.e. characters which don't need changing are left alone.

138 :: How do I convert a string to all upper or lower case?

Some libraries have routines strupr and strlwr or strupper and strlower, but these are not Standard or portable. It's a straightforward exercise to write upper/lower-case functions in terms of the toupper and tolower macros in <ctype.h>;(The only tricky part is that the function will either have to modify the string in-place or deal with the problem of returning a new string;
(Note also that converting characters and strings to upper or lower case is vastly more complicated when multinational character sets are being used.)

139 :: why isnt it being handled properly?

I'm reading strings typed by the user into an array, and then printing them out later. When the user types a sequence like \n, why isn't it being handled properly?

Character sequences like \n are interpreted at compile time. When a backslash and an adjacent n appear in a character constant or string literal, they are translated immediately into a single newline character. (Analogous translations occur, of course, for the other character escape sequences.) When you're reading strings from the user or a file, however, no interpretation like this is performed: a backslash is read and printed just like any other character, with no particular interpretation.
(Some interpretation of the newline character may be done during run-time I/O, but for a completely different reason;

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

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. What happens here is that fwrite receives a pointer to the structure, and writes (or fread correspondingly reads) the memory image of the structure as a stream of bytes. The sizeof operator determines how many bytes the structure occupies.
(The call to fwrite above is correct under an ANSI compiler as long as a prototype for fwrite is in scope, usually because <stdio.h> is #included.
However, data files written as memory images in this way will not be portable, particularly if they contain floating-point fields or pointers. The memory layout of structures is machine and compiler dependent. Different compilers may use different amounts of padding , and the sizes

and byte orders of fundamental types vary across machines. Therefore, structures written as memory images cannot necessarily be read back in by programs running on other machines (or even compiled by other compilers), and this is an important concern if the data files you're writing will ever be interchanged between machines.
Also, if the structure contains any pointers (char * strings, or pointers to other data structures), only the pointer values will be written, and they are most unlikely to be valid when read back in.

141 :: How can I change their mode to binary?

I'm writing a ``filter'' for binary files, but stdin and stdout are preopened as text streams. How can I change their mode to binary?

There is no standard way to do this. On Unix-like systems, there is no text/binary distinction, so there is no need to change the mode. Some MS-DOS compilers supply a setmode call. Otherwise, you're on your own.

142 :: What is the difference between text and binary I/O?

In text mode, a file is assumed to consist of lines of printable characters (perhaps including tabs). The routines in the stdio library (getc, putc, and all the rest) translate between the underlying system's end-of-line representation and the single \n used in C programs. C programs which simply read and write text therefore don't have to worry about the underlying system's newline conventions: when a C program writes a '\n', the stdio library writes the appropriate end-of-line indication, and when the stdio library detects an end-of-line while reading, it returns a single '\n' to the calling program.
In binary mode, on the other hand, bytes are read and written between the program and the file without any interpretation. (On MS-DOS systems, binary mode also turns off testing for control-Z as an in-band end-of-file character.)
Text mode translations also affect the apparent size of a file as it's read. Because the characters read from and written to a file in text mode do not necessarily match the characters stored in the file exactly, the size of the file on disk may not always match the number of characters which can be read from it. Furthermore, for analogous reasons, the fseek and ftell functions do not necessarily deal in pure byte offsets from the beginning of the file.

143 :: How can I read a binary data file properly?

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

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". (Under operating systems such as Unix which don't distinguish between text and binary files, "b" may not be required, but is harmless.)
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.

144 :: How can I get back to the interactive keyboard if stdin is redirected?

I'm trying to write a program like ``more.'' How can I get back to the interactive keyboard if stdin is redirected?

There is no portable way of doing this. Under Unix, you can open the special file /dev/tty. Under MS-DOS, you can try opening the ``file'' CON, or use routines or BIOS calls such as getch which may go to the keyboard whether or not input is redirected.

145 :: Once I have used freopen, how can I get the original stdout (or stdin) back?

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. For example, declare a global
FILE *ofp;
and replace all calls to printf(...) with fprintf(ofp, ...). (Obviously, you'll have to check for calls to putchar and puts, too.) Then you can set ofp to stdout or to anything else.
You might wonder if you could skip freopen entirely, and do something like
FILE *savestdout = stdout; stdout = fopen(file, "w"); /* WRONG */

leaving yourself able to restore stdout later by doing
stdout = savestdout; /* WRONG */
but code like this is not likely to work, because stdout (and stdin and stderr) are typically constants which cannot be reassigned (which is why freopen exists in the first place). It may be possible, in a nonportable way, to save away information about a stream before calling freopen to open some file in its place, such that the original stream can later be restored. The most straightforward and reliable way is to manipulate the underlying file descriptors using a system-specific call such as dup or dup2, if available Another is to copy or inspect the contents of the FILE structure, but this is exceedingly nonportable and unreliable.

146 :: How can I recover the file name given an open stream?

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 as you open them (perhaps with a wrapper function around fopen).

147 :: How can I insert or delete a line (or record) in the middle of a file?

In general, there is no way to do this. The usual solution is simply to rewrite the file.
When you find yourself needing to insert data into an existing file, here are a few alternatives you can try:
* Rearrange the data file so that you can append the new information at the end.

* Put the information in a second file.
* Leave some blank space (e.g. a line of 80 spaces, or a field like 0000000000) in the file when it is first written, and overwrite it later with the final information
* (This technique is most portable in binary mode; on some systems, overwriting a text file may truncate it.) * Use a database instead of a flat file.
Instead of actually deleting records, you might consider just marking them as deleted, and having the code which reads the file ignore them. (You could run a separate coalescion program once in a while to rewrite the file, finally discarding the deleted records. Or, if the records are all the same length, you could take the last record and use it to overwrite the record to be deleted, then truncate the file.)

148 :: If fflush wont work, what can I use to flush input?

If fflush wont work, what can I use to flush input?

C Interview Questions and Answers

(Continued from previous question...)

If fflush wont work, what can I use to flush input?

It depends on what you're trying to do. If you're trying to get rid of an unread newline or other unexpected input after calling scanf you really need to rewrite or replace the call to scanf. Alternatively, you can consume the rest of a partially-read line with a simple code fragment like
while((c = getchar()) != '\n' && c != EOF)
/* discard */ ;

(You may also be able to use the curses flushinp function.)
There is no standard way to discard unread characters from a stdio input stream. Some vendors do implement fflush so that fflush(stdin) discards unread characters, although portable programs cannot depend on this. (Some versions of the stdio library implement fpurge or fabort calls which do the same thing, but these aren't standard, either.) Note, too, that flushing stdio input buffers is not necessarily sufficient: unread characters can also accumulate in other, OS-level input buffers. If you're trying to actively discard input (perhaps in anticipation of issuing an unexpected prompt to confirm a destructive action, for which an accidentally-typed ``y'' could be disastrous), you'll have to use a system-specific technique to detect the presence of typed-ahead input; Keep in mind that users can become frustrated if you discard input that happened to be typed too quickly.

149 :: Why does everyone say not to use gets?

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 if an input line is longer than expected--and Murphy's Law says that, sooner or later, a larger-than-expected input line will occur. (It's possible to convince yourself that, for some reason or another, input lines longer than some maximum are impossible, but it's also possible to be mistaken, and in any case it's just as easy to use fgets.)
The Standard fgets function is a vast improvement over gets(), although it's not perfect, either. (If

long lines are a real possibility, their proper handling must be carefully considered.)
One other difference between fgets() and gets() is that fgets() retains the '\n', but it is straightforward to strip it out.for a code fragment illustrating the replacement of gets() with fgets().

150 :: What is the deal on sprintfs return value?

What's the deal on sprintf's return value? Is it an int or a char *?

The Standard says that it returns an int (the number of characters written, just like printf and fprintf). Once upon a time, in some C libraries, sprintf returned the char * value of its first argument, pointing to the completed result (i.e. analogous to strcpy's return value).

151 :: Why does everyone say not to use scanf? What should I use instead?

scanf has a number of problems, its %s format has the same problem that gets() has --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. You have very little opportunity to do any error recovery.
Yet interactive user input is the least structured input there is. A well-designed user interface will allow for the possibility of the user typing just about anything--not just letters or punctuation when digits were expected, but also more or fewer characters than were expected, or no characters at all (i.e. just the RETURN key), or premature EOF, or anything. It's nearly impossible to deal gracefully with all of these potential problems when using scanf; it's far easier to read entire lines (with fgets or the like), then interpret them, either using sscanf or some other techniques. (Functions like strtol, strtok, and atoi are often useful; If you do use any scanf variant, be sure to check the return value to make sure that the expected number of items were found. Also, if you use %s, be sure to guard against buffer overflow.

152 :: How can I read data from data files with particular formats?

How can I read data from data files with particular formats?
How can I read ten floats without having to use a jawbreaker scanf format like "%f %f %f %f %f %f %f %f %f %f"?
How can I read an arbitrary number of fields from a line into an array?

In general, there are three main ways of parsing data lines:
1. Use fscanf or sscanf, with an appropriate format string. Despite the limitations mentioned in this section, the scanf family is quite powerful. Though whitespace-separated fields are always the easiest to deal with, scanf format strings can also be used with more compact, column oriented, FORTRAN-style data. For instance, the line
1234ABC5.678
could be read with "%d%3s%f".
, then deal with each field individually, perhaps with functions like atoi and atof. (Once the line

is broken up, the code for handling the fields is much like the traditional code in main() for handling the argv array;
Break the line into fields separated by whitespace (or some other delimiter), using strtok or the equivalent This method is particularly useful for reading an arbitrary (i.e. not known in advance) number of fields from a line into an array.
Here is a simple example which copies a line of up to 10 floating-point numbers (separated by whitespace) into an array:

```
#define MAXARGS 10
char line[] = "1 2.3 4.5e6 789e10";
char *av[MAXARGS];
int ac, i;
double array[MAXARGS];

ac = makeargv(line, av, MAXARGS);
for(i = 0; i < ac; i++)
array[i] = atof(av[i]);
```

153 :: Why doesnt that code work?

Why doesn't the code

```
short int s;
scanf("%d", &s);
```

work?

When converting %d, scanf expects a pointer to an int. To convert to a short int, use %hd .

154 :: Why doesnt this code work?

Why doesn't this code:

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

work?

Unlike printf, scanf uses %lf for values of type double, and %f for float.%f tells scanf to expect a pointer-to-float, not the pointer-to-double you gave it. Either use %lf, or declare the receiving variable as a float.

155 :: Why does the call char scanf work?

Why does the call

```
char s[30];
scanf("%s", s);
```

work? I thought you always needed an & on each variable passed to scanf.

You always need a pointer; you don't necessarily need an explicit &. When you pass an array to scanf, you do not need the &, because arrays are always passed to functions as pointers, whether you use & or not.

156 :: Why doesnt the call scanf work?

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

The arguments you pass to scanf must always be pointers: for each value converted, scanf ``returns" it by filling in one of the locations you've passed pointers to. To fix the fragment above, change it to scanf("%d", &i) .

157 :: Why doesnt long int work?

Why doesn't
long int n = 123456;
printf("%d\n", n);

work?

Whenever you print long ints you must use the l (lower case letter ``ell") modifier in the printf format (e.g. %ld). printf can't know the types of the arguments which you've passed to it, so you must let it know by using the correct format specifiers.

158 :: What is wrong with this code?

What's wrong with this code?
char c;
while((c = getchar()) != EOF) ...

For one thing, the variable to hold getchar's return value must be an int. EOF is an ``out of band" return value from getchar: it is distinct from all possible char values which getchar can return. (On modern systems, it does not reflect any actual end-of-file character stored in a file; it is a signal that no more characters are available.) getchar's return value must be stored in a variable larger than char so that it can hold all possible char values, and EOF.
Two failure modes are possible if, as in the fragment above, getchar's return value is assigned to a char.
1. If type char is signed, and if EOF is defined (as is usual) as -1, the character with the decimal value 255 ('\377' or '\xff' in C) will be sign-extended and will compare equal to EOF, prematurely terminating the input.
2. If type char is unsigned, an actual EOF value will be truncated (by having its higher-order bits discarded, probably resulting in 255 or 0xff) and will not be recognized as EOF, resulting in effectively infinite input.
The bug can go undetected for a long time, however, if chars are signed and if the input is all 7-bit characters. (Whether plain char is signed or unsigned is implementation-defined.)

159 :: What does the message "Automatic aggregate intialization is an ANSI feature" mean?

What does the message Automatic aggregate intialization is an ANSI feature" mean? My compiler is complaining about valid ANSI code.

Messages like these are typically emitted by pre-ANSI compilers which have been upgraded just enough to detect (but not properly translate) new C features which were introduced with the ANSI Standard. The implication of the message is that you should pay your vendor more money for a copy of their real ANSI C compiler.

160 :: What was noalias and what ever happened to it?

noalias was another type qualifier, in the same syntactic class as const and volatile, which was intended to assert that an object was not pointed to (``aliased") by other pointers. The primary application, which is an important one, would have been for the formal parameters of functions designed to perform computations on large arrays. A compiler cannot usually take advantage of vectorization or other parallelization hardware (on supercomputers which have it) unless it can ensure that the source and destination arrays do not overlap.
The noalias keyword was not backed up by any ``prior art," and it was introduced late in the review and approval process. It was surprisingly difficult to define precisely and explain coherently, and sparked widespread, acrimonious debate, including a scathing pan by Dennis Ritchie. It had far-ranging implications, particularly for several standard library interfaces, for which easy fixes were not readily apparent.
Because of the criticism and the difficulty of defining noalias well, the Committee declined to adopt it, in spite of its superficial attractions. (When writing a standard, features cannot be introduced halfway; their full integration, and all implications, must be understood.) The need for an explicit mechanism to support parallel implementation of non-overlapping operations remains unfilled (although some work is being done on the problem)

161 :: What should malloc0 do? Return a null pointer or a pointer to 0 bytes?

The ANSI/ISO Standard says that it may do either; the behavior is implementation-defined Portable code must either take care not to call malloc(0), or be prepared for the possibility of a null return.

162 :: What are pragmas and what are they good for?

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

163 :: What is the correct declaration of main?

There are two valid declarations:
int main(void)

int main(int argc, char **argv)

although they can be written in a variety of ways. The second parameter may be declared char *argv[] , you can use any names for the two parameters, and you can use old-style syntax:
int main()
int main(argc, argv)
int argc; char **argv;

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

Doing so is legal (and can be useful for backwards compatibility), but requires a certain amount of care . Modern practice, however, is to use the prototyped form in both declarations and definitions. (The old-style syntax is marked as obsolescent, so official support for it may be removed some day.)

165 :: What is the ANSI C Standard?

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 standardized 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 formalized the C run-time library support routines.
A year or so later, the Standard was adopted as an international standard, ISO/IEC 9899:1990, and this ISO Standard replaced the earlier X3.159 even within the United States (where it was known as ANSI/ISO 9899-1990 [1992]). As an ISO Standard, it is subject to ongoing revision through the release of Technical Corrigenda and Normative Addenda.
In 1994, Technical Corrigendum 1 (TC1) amended the Standard in about 40 places, most of them minor corrections or clarifications, and Normative Addendum 1 (NA1) added about 50 pages of new material, mostly specifying new library functions for internationalization. In 1995, TC2 added a few more minor corrections.
Most recently, a major revision of the Standard, ``C99'', has been completed and adopted.

166 :: How can I list all of the predefined identifiers?

There's no standard way, although it is a common need. gcc provides a -dM option which works with -E, and other compilers may provide something similar. 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 predefined identifiers (e.g. ``unix'') are non-Standard (because they clash with the user's namespace) and are being removed or renamed. (In any case, as a general rule, it's considered wise to keep conditional compilation to a minimum.)

167 :: How can I use a preprocessorif expression to ?

How can I use a preprocessor #if expression to tell whether a machine's byte order is big-endian or little-endian?

You probably can't. The usual techniques for detecting endianness involve pointers or arrays of char, or maybe unions, but preprocessor arithmetic uses only long integers, and there is no concept of addressing. Another tempting possibility is something like #if 'ABCD' == 0x41424344 but this isn't reliable, either. At any rate, the integer formats used in preprocessor #if expressions are not necessarily the same as those that will be used at run time. Are you sure you need to know the machine's endianness explicitly? Usually it's better to write code which doesn't care

168 :: Is there anything like an ifdef for typedefs?

Unfortunately, no. (There can't be, because types and typedefs haven't been parsed at preprocessing time.) You may have to keep sets of preprocessor macros (e.g. MY_TYPE_DEFINED) recording whether certain typedefs have been declared.

169 :: What is the difference between #include <> and #include?

The <> syntax is typically used with Standard or system-supplied headers, while "" is typically used for a program's own header files.

170 :: What are the complete rules for header file searching?

The exact behavior is implementation-defined (which means that it is supposed to be documented; 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. (This last rule, that "" files are additionally searched for as if they were <> files, is the only rule specified by the Standard.)
Another distinction is the definition of ``current directory'' for "" files. 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 is the directory in which the compiler was initially invoked. (Compilers running on systems without directories or without the notion of a current directory may of course use still different rules.)
It is also common for there to be a way (usually a command line option involving capital I, or maybe an environment variable) to add additional directories to the list of standard places to search. Check your compiler documentation.

171 :: What is the right type to use for Boolean values in C? Is there a standard type? Should I use #defines or enums for the true and false values?

Traditionally, C did not provide a standard Boolean type, partly and partly to allow the programmer to make the appropriate space/time tradeoff. (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.)
However, C99 does define a standard Boolean type, as long as you include <stdbool.h>.

If you decide to define them yourself, the choice between #defines and enumeration constants for the true/false values is arbitrary and not terribly interestingUse 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.)
You may also want to use a typedef:
typedef int bool;
or
typedef char bool;
or
typedef enum {false, true} bool;

Some people prefer variants like
#define TRUE (1==1)
#define FALSE (!TRUE)
or define ``helper'' macros such as
#define Istrue(e) ((e) != 0)

172 :: Why does not strcat(string, "!"); work?

There is a very real difference between characters and strings, and strcat concatenates strings.
A character constant like '!' represents a single character. A string literal between double quotes usually represents multiple characters. A string literal like "!" seems to represent a single character, but it actually contains two: the ! you requested, and the \0 which terminates all strings in C.
Characters in C are represented by small integers corresponding to their character set values
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, "!");

173 :: Why is not a pointer null after calling free? How unsafe is it to use (assign, compare) a pointer value after it is been freed?

When you call free, the memory pointed to by the passed pointer is freed, but the value of the pointer in the caller probably remains unchanged, because C's pass-by-value semantics mean that called functions never permanently change the values of their arguments. A pointer value which has been freed is, strictly speaking, invalid, and any use of it, even if it is not dereferenced (i.e. even if the use of it is a seemingly innocuous assignment or comparison), can theoretically lead to trouble. (We can probably assume that as a quality of implementation issue, most

implementations will not go out of their way to generate exceptions for innocuous uses of invalid pointers, but the Standard is clear in saying that nothing is guaranteed, and there are system architectures for which such exceptions would be quite natural.)
When pointer variables (or fields within structures) are repeatedly allocated and freed within a program, it is often useful to set them to NULL immediately after freeing them, to explicitly record their state.

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

The ANSI/ISO Standard says that it may do either; the behavior is implementation-defined ortable code must either take care not to call malloc(0), or be prepared for the possibility of a null return.

175 :: How can I dynamically allocate arrays?

The equivalence between arrays and pointers allows a pointer to malloc'ed memory to simulate an array quite effectively. After executing
#include <stdlib.h>
int *dynarray;
dynarray = malloc(10 * sizeof(int));

(and if the call to malloc succeeds), you can reference dynarray[i] (for i from 0 to 9) almost as if dynarray were a conventional, statically-allocated array (int a[10]). The only difference is that sizeof will not give the size of the ``array''

176 :: What is wrong with this initialization?

What's wrong with this initialization?
char *p = malloc(10);
My compiler is complaining about an ``invalid initializer'', or something.

Is the declaration of a static or non-local variable? Function calls are allowed in initializers only for automatic variables (that is, for local, non-static variables).

177 :: How can this be legal C?

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

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. The ``proof'' looks like

a[e]
*((a) + (e)) (by definition)

*((e) + (a)) (by commutativity of addition)
e[a] (by definition)

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 .
Since strings in C are arrays of char, the expression "abcdef"[5] is perfectly legal, and evaluates to the character 'f'. You can think of it as a shorthand for
char *tmpptr = "abcdef";

... tmpptr[5] ...

178 :: Is a pointer a kind of array?

I'm still mystified. Is a pointer a kind of array, or is an array a kind of pointer?

An array is not a pointer, nor vice versa. An array reference (that is, any mention of an array in a value context), turns into a pointer
There are perhaps three ways to think about the situation:
1. Pointers can simulate arrays
2. There's hardly such a thing as an array (it is, after all, a ``second-class citizen"); the subscripting operator [] is in fact a pointer operator.
3. At a higher level of abstraction, a pointer to a block of memory is effectively the same as an array (though this says nothing about other uses of pointers).

But, to reiterate, here are two ways not to think about it:
4. ``They're completely the same."
5. ``Arrays are constant pointers.'

179 :: Why ca not I do something like this?

Why can't I do something like this?
extern char *getpass();
char str[10];
str = getpass("Enter password: ");

Arrays are ``second-class citizens" in C; one upshot of this prejudice is that you cannot assign to them . When you need to copy the contents of one array to another, you must do so explicitly. In the case of char arrays, the strcpy routine is usually appropriate:
strcpy(str, getpass("Enter password: "));
(When you want to pass arrays around without copying them, you can use pointers and simple assignment.

180 :: Is NULL valid for pointers to functions?

Yes There is no ``total generic pointer type."
void *'s are only guaranteed to hold object (i.e. data) pointers; it is not portable to convert a

function pointer to type void *. (On some machines, function addresses can be very large, bigger than any data pointers.)
It is guaranteed, however, that all function pointers can be interconverted, as long as they are converted back to an appropriate type before calling. Therefore, you can pick any function type (usually int (*)() or void (*)(), that is, pointer to function of unspecified arguments returning int or void) as a generic function pointer. When you need a place to hold object and function pointers interchangeably, the portable solution is to use a union of a void * and a generic function pointer (of whichever type you choose).

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

With a null pointer constant.

According to the language definition, an ``integral constant expression with the value 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)
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,"

182 :: What is this infamous null pointer, anyway?

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, a null pointer points definitively nowhere; it is not the address of any object or function. 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. 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.

183 :: What is the total generic pointer type?

What's the total generic pointer type? My compiler complained when I tried to stuff function pointers into a void *.

There is no ``total generic pointer type."
void *'s are only guaranteed to hold object (i.e. data) pointers; it is not portable to convert a function pointer to type void *. (On some machines, function addresses can be very large, bigger than any data pointers.)
It is guaranteed, however, that all function pointers can be interconverted, as long as they are converted back to an appropriate type before calling. Therefore, you can pick any function type (usually int (*)() or void (*)(), that is, pointer to function of unspecified arguments returning int or void) as a generic function pointer. When you need a place to hold object and function pointers interchangeably, the portable solution is to use a union of a void * and a generic function pointer (of whichever type you choose).

184 :: Can I initialize unions?

In the original ANSI C, an initializer was allowed only for the first-named member of a union. C99 introduces ``designated initializers" which can be used to initialize any member.

In the absence of designated initializers, if you're desperate, you can sometimes define several variant copies of a union, with the members in different orders, so that you can declare and initialize the one having the appropriate first member. (These variants are guaranteed to be implemented compatibly, so it's okay to ``pun" them by initializing one and then using the other.)

185 :: What are pointers really good for, anyway?

They're good for lots of things, such as:
* dynamically-allocated arrays
* generic access to several similar variables
* (simulated) by-reference function parameters
* malloc'ed data structures of all kinds, especially trees and linked lists
* walking over arrays (for example, while parsing strings)
* efficient, by-reference ``copies" of arrays and structures, especially as function parameters

186 :: Are enumerations really portable?

Are enumerations really portable? Aren't they Pascalish?

Enumerations were a mildly late addition to the language (they were not in K&R1), but they are definitely part of the language now: they're in the C Standard, and all modern compilers support them. They're quite portable, although historical uncertainty about their precise definition led to their specification in the Standard being rather weak

187 :: C is not C++. Typedef names are not automatically generated

Why doesn't
struct x { ... };
x thestruct;

work?

C is not C++. Typedef names are not automatically generated for structure tags. Either declare structure instances using the struct keyword:
struct x thestruct;
or declare a typedef when you declare a structure:
typedef struct { ... } x;
x thestruct;

188 :: What is the auto keyword good for?

Nothing; it's archaic. (It's a holdover from C's typeless predecessor language B, where in the absence of keywords like int a declaration always needed a storage class.)

189 :: What is wrong with this declaration?

What's wrong with this declaration?
char* p1, p2;
I get errors when I try to use p2.

Nothing is wrong with the declaration--except that it doesn't do what you probably want. The * in a pointer declaration is not part of the base type; it is part of the declarator containing the name being declared That is, in C, the syntax and interpretation of a declaration is not really
type identifier ;
but rather
base_type thing_that_gives_base_type ; where ``thing_that_gives_base_type"--the declarator--is either a simple identifier, or a notation like *p or a[10] or f() indicating that the variable being declared is a pointer to, array of, or function returning that base_type. (Of course, more complicated declarators are possible as well.)

In the declaration as written in the question, no matter what the whitespace suggests, the base type is char and the first declarator is ``* p1", and since the declarator contains a *, it declares p1 as a pointer-to-char. The declarator for p2, however, contains nothing but p2, so p2 is declared as a plain char, probably not what was intended. To declare two pointers within the same declaration, use

char *p1, *p2;

190 :: Linked Lists -- Can you tell me how to check whether a linked list is circular?

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

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

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

191 :: What does static variable mean in c?

there are 3 main uses for the static.
1. If you declare within a function:
It retains the value between function calls

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

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

192 :: What are the different storage classes in C?

C has three types of storage: automatic, static and allocated.

Variable having block scope and without static specifier have automatic storage duration.

Variables with block scope, and with static specifier have static scope. Global variables (i.e, file scope) with or without the the static specifier also have static scope.

Memory obtained from calls to malloc(), alloc() or realloc() belongs to allocated storage class.

193 :: What is hashing in C language?

To hash means to grind up, and that's essentially what hashing is all about. The heart of a hashing algorithm is a hash function that takes your nice, neat data and grinds it into some random-looking integer.
The idea behind hashing is that some data either has no inherent ordering (such as images) or is expensive to compare (such as images). If the data has no inherent ordering, you can't perform comparison searches.

If the data is expensive to compare, the number of comparisons used even by a binary search might be too many. So instead of looking at the data themselves, you'll condense (hash) the data to an integer (its hash value) and keep all the data with the same hash value in the same place. This task is carried out by using the hash value as an index into an array.
To search for an item, you simply hash it and look at all the data whose hash values match that of the data you're looking for. This technique greatly lessens the number of items you have to look at. If the parameters are set up with care and enough storage is available for the hash table, the number of comparisons needed to find an item can be made arbitrarily close to one.
One aspect that affects the efficiency of a hashing implementation is the hash function itself. It should ideally distribute data randomly throughout the entire hash table, to reduce the likelihood of collisions. Collisions occur when two different keys have the same hash value.

194 :: What is page thrashing?

Some operating systems (such as UNIX or Windows in enhanced mode) use virtual memory. Virtual memory is a technique for making a machine behave as if it had more memory than it really has, by using disk space to simulate RAM (random-access memory). In the 80386 and higher Intel CPU chips, and in most other modern microprocessors (such as the Motorola 68030, Sparc, and Power PC), exists a piece of hardware called the Memory Management Unit, or MMU.
The MMU treats memory as if it were composed of a series of pages. A page of memory is a block of contiguous bytes of a certain size, usually 4096 or 8192 bytes. The operating system sets up and maintains a table for each running program called the Process Memory Map, or PMM. This is a table of all the pages of memory that program can access and where each is really located.
Every time your program accesses any portion of memory, the address (called a virtual address) is processed by the MMU. The MMU looks in the PMM to find out where the memory is really located (called the physical address). The physical address can be any location in memory or on disk that the operating system has assigned for it. If the location the program wants to access is on disk, the page containing it must be read from disk into memory, and the PMM must be updated to reflect this action (this is called a page fault).

195 :: What is the benefit of using #define to declare a constant?

Using the #define method of declaring a constant enables you to declare a constant in one place and use it throughout your program. This helps make your programs more maintainable, because you need to maintain only the #define statement and not several instances of individual constants throughout your program.
For instance, if your program used the value of pi (approximately 3.14159) several times, you might want to declare a constant for pi as follows:
#define PI 3.14159
Using the #define method of declaring a constant is probably the most familiar way of declaring constants to traditional C programmers. Besides being the most common method of declaring constants, it also takes up the least memory. Constants defined in this manner are simply placed directly into your source code, with no variable space allocated in memory. Unfortunately, this is one reason why most debuggers cannot inspect constants created using the #define method.

196 :: How can I search for data in a linked list?

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

197 :: When should a type cast be used?

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

198 :: What is a null pointer in C?

There are times when it's necessary to have a pointer that doesn't point to anything. The macro NULL, defined in , has a value that's guaranteed to be different from any valid pointer. NULL is a literal zero, possibly cast to void* or char*. Some people, notably C++ programmers, prefer to use 0 rather than NULL.
The null pointer is used in three ways:
1) To stop indirection in a recursive data structure
2) As an error value
3) As a sentinel value

199 :: What is the stack in C?

The stack is where all the functions' local (auto) variables are created. The stack also contains some information used to call and return from functions.
A stack trace is a list of which functions have been called, based on this information. When you start using a debugger, one of the first things you should learn is how to get a stack trace.
The stack is very inflexible about allocating memory; everything must be deallocated in exactly the reverse order it was allocated in. For implementing function calls, that is all that's needed. Allocating memory off the stack is extremely efficient. One of the reasons C compilers generate such good code is their heavy use of a simple stack.
There used to be a C function that any programmer could use for allocating memory off the stack. The memory was automatically deallocated when the calling function returned. This was a dangerous function to call; it's not available anymore.

200 :: What is Preprocessor in C?

The preprocessor is used to modify your program according to the preprocessor directives in your source code. Preprocessor directives (such as #define) give the preprocessor specific instructions on how to modify your source code. The preprocessor reads in all of your include files and the source code you are compiling and creates a preprocessed version of your source

code. This preprocessed version has all of its macros and constant symbols replaced by their corresponding code and value assignments. If your source code contains any conditional preprocessor directives (such as #if), the preprocessor evaluates the condition and modifies your source code accordingly.
The preprocessor contains many features that are powerful to use, such as creating macros, performing conditional compilation, inserting predefined environment variables into your code, and turning compiler features on and off. For the professional programmer, in-depth knowledge of the features of the preprocessor can be one of the keys to creating fast, efficient programs.

201 :: What is the heap in C?

The heap is where malloc(), calloc(), and realloc() get memory.
Getting memory from the heap is much slower than getting it from the stack. On the other hand, the heap is much more flexible than the stack. Memory can be allocated at any time and deallocated in any order. Such memory isn't deallocated automatically; you have to call free(). Recursive data structures are almost always implemented with memory from the heap. Strings often come from there too, especially strings that could be very long at runtime. If you can keep data in a local variable (and allocate it from the stack), your code will run faster than if you put the data on the heap. Sometimes you can use a better algorithm if you use the heap faster, or more robust, or more flexible. It's a tradeoff.
If memory is allocated from the heap, it's available until the program ends. That's great if you remember to deallocate it when you're done. If you forget, it's a problem. A memory leak is some allocated memory that's no longer needed but isn't deallocated. If you have a memory leak inside a loop, you can use up all the memory on the heap and not be able to get any more. (When that happens, the allocation functions return a null pointer.) In some environments, if a program doesn't deallocate everything it allocated, memory stays unavailable even after the program ends.

202 :: What is the purpose of realloc( )?

The function realloc(ptr,n) uses two arguments.the first argument ptr is a pointer to a block of memory for which the size is to be altered.The second argument n specifies the new size.The size may be increased or decreased.If n is greater than the old size and if sufficient space is not available subsequent to the old region, the function realloc( ) may create a new region and all the old data are moved to the new region.

203 :: What is the purpose of main( ) function?

The function main( ) invokes other functions within it.It is the first function to be called when the program starts execution.
- It is the starting function
- It returns an int value to the environment that called the program
- Recursive call is allowed for main( ) also.
- It is a user-defined function
- Program execution ends when the closing brace of the function main( ) is reached.

- It has two arguments 1)argument count and 2) argument vector (represents strings passed).
- Any user-defined name can also be used as parameters for main( ) instead of argc and argv

204 :: What is the benefit of using const for declaring constants?

The benefit of using the const keyword is that the compiler might be able to make optimizations based on the knowledge that the value of the variable will not change. In addition, the compiler will try to ensure that the values won't be changed inadvertently.
Of course, the same benefits apply to #defined constants. The reason to use const rather than #define to define a constant is that a const variable can be of any type (such as a struct, which can't be represented by a #defined constant). Also, because a const variable is a real variable, it has an address that can be used, if needed, and it resides in only one place in memory

205 :: What is the easiest sorting method to use?

The answer is the standard library function qsort(). It's the easiest sort by far for several reasons:
It is already written.
It is already debugged.
It has been optimized as much as possible (usually).
Void qsort(void *buf, size_t num, size_t size, int (*comp)(const void *ele1, const void *ele2));

206 :: What is a const pointer in C?

The access modifier keyword const is a promise the programmer makes to the compiler that the value of a variable will not be changed after it is initialized. The compiler will enforce that promise as best it can by not enabling the programmer to write code which modifies a variable that has been declared const.
A const pointer, or more correctly, a pointer to const, is a pointer which points to data that is const (constant, or unchanging). A pointer to const is declared by putting the word const at the beginning of the pointer declaration. This declares a pointer which points to data that can't be modified. The pointer itself can be modified. The following example illustrates some legal and illegal uses of a const pointer:

```
const char *str = hello;
char c = *str /* legal */
str++; /* legal */
*str = 'a'; /* illegal */
str[1] = 'b'; /* illegal */
```

207 :: What is a pragma in C?

The #pragma preprocessor directive allows each compiler to implement compiler-specific features that can be turned on and off with the #pragma statement. For instance, your compiler might support a feature called loop optimization. This feature can be invoked as a command-line option or as a #pragma directive.
To implement this option using the #pragma directive, you would put the following line into your code:

#pragma loop_opt(on)
Conversely, you can turn off loop optimization by inserting the following line into your code:
#pragma loop_opt(off)

208 :: What is #line used for?

The #line preprocessor directive is used to reset the values of the _ _LINE_ _ and _ _FILE_ _ symbols, respectively. This directive is commonly used in fourth-generation languages that generate C language source files.

209 :: What is the difference between far and near in C?

Some compilers for PC compatibles use two types of pointers. near pointers are 16 bits long and can address a 64KB range. far pointers are 32 bits long and can address a 1MB range.
Near pointers operate within a 64KB segment. There's one segment for function addresses and one segment for data. far pointers have a 16-bit base (the segment address) and a 16-bit offset. The base is multiplied by 16, so a far pointer is effectively 20 bits long. Before you compile your code, you must tell the compiler which memory model to use. If you use a smallcode memory model, near pointers are used by default for function addresses.
That means that all the functions need to fit in one 64KB segment. With a large-code model, the default is to use far function addresses. You'll get near pointers with a small data model, and far pointers with a large data model. These are just the defaults; you can declare variables and functions as explicitly near or far.
far pointers are a little slower. Whenever one is used, the code or data segment register needs to be swapped out. far pointers also have odd semantics for arithmetic and comparison. For example, the two far pointers in the preceding example point to the same address, but they would compare as different! If your program fits in a small-data, small-code memory model, your life will be easier.

210 :: What is a method in C?

Method is a way of doing something, especially a systematic way; implies an orderly logical arrangement (usually in steps).

211 :: Are pointers integers in C?

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

212 :: How do you redirect a standard stream?

Most operating systems, including DOS, provide a means to redirect program input and output to and from different devices. This means that rather than your program output (stdout) going to the screen; it can be redirected to a file or printer port. Similarly, your program's input (stdin) can come from a file rather than the keyboard. In DOS, this task is accomplished using the redirection characters, < and >. For example, if you wanted a program named PRINTIT.EXE to

receive its input (stdin) from a file named STRINGS.TXT, you would enter the following command at the DOS prompt:
C:> PRINTIT <STRINGS.TXT
Notice that the name of the executable file always comes first. The less-than sign (<) tells DOS to take the strings contained in STRINGS.TXT and use them as input for the PRINTIT program.
The following example would redirect the program's output to the prn device, usually the printer attached on LPT1:
C :> REDIR > PRN
Alternatively, you might want to redirect the program's output to a file, as the following example shows:
C :> REDIR > REDIR.OUT
In this example, all output that would have normally appeared on-screen will be written to the file
REDIR.OUT.
Redirection of standard streams does not always have to occur at the operating system. You can redirect a standard stream from within your program by using the standard C library function named freopen(). For example, if you wanted to redirect the stdout standard stream within your program to a file named OUTPUT.TXT, you would

213 :: What is indirection in C?

If you declare a variable, its name is a direct reference to its value. If you have a pointer to a variable, or any other object in memory, you have an indirect reference to its value.

214 :: What is an lvalue in C?

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

215 :: Array is an lvalue or not?

An lvalue was defined as an expression to which a value can be assigned. Is an array an expression to which we can assign a value? The answer to this question is no, because an array is composed of several separate array elements that cannot be treated as a whole for assignment purposes.
The following statement is therefore illegal:
int x[5], y[5]; x = y;
Additionally, you might want to copy the whole array all at once. You can do so using a library function such as the memcpy() function, which is shown here:
memcpy(x, y, sizeof(y));
It should be noted here that unlike arrays, structures can be treated as lvalues. Thus, you can assign one structure variable to another structure variable of the same type, such as this:
typedef struct t_name
{

```
char last_name[25];
char first_name[15];
char middle_init[2];
} NAME;
...
NAME my_name, your_name;
...
your_name = my_name;
```

216 :: What is a void pointer in C?

A void pointer is a C convention for a raw address. The compiler has no idea what type of object a void Pointer really points to. If you write
int *ip;
ip points to an int. If you write
void *p;
p doesn't point to a void!
In C and C++, any time you need a void pointer, you can use another pointer type. For example, if you have a char*, you can pass it to a function that expects a void*. You don't even need to cast it. In C (but not in C++), you can use a void* any time you need any kind of pointer, without casting. (In C++, you need to cast it).
A void pointer is used for working with raw memory or for passing a pointer to an unspecified type.
Some C code operates on raw memory. When C was first invented, character pointers (char *) were used for that. Then people started getting confused about when a character pointer was a string, when it was a character array, and when it was raw memory.

217 :: What is a pointer variable in C language?

A pointer variable is a variable that may contain the address of another variable or any valid address in the memory.

218 :: What is a static function in C?

A static function is a function whose scope is limited to the current source file. Scope refers to the visibility of a function or variable. If the function or variable is visible outside of the current source file, it is said to have global, or external, scope. If the function or variable is not visible outside of the current source file, it is said to have local, or static, scope.

219 :: What is a pointer value and address in C?

A pointer value is a data object that refers to a memory location. Each memory locaion is numbered in the memory. The number attached to a memory location is called the address of the location.

220 :: How do we print only part of a string in C?

```
/* Use printf() to print the first 11 characters of source_str. */
printf(First 11 characters: ‘%11.11s’n, source_str);
```