Questions with answers:

Section 1

1.      How do you decide which integer type to use?

A:      If you might need large values (tens of thousands), use long.

        Otherwise, if space is very important, use short.  Otherwise,

        use int.

2.      What should the 64-bit type on a machine that can support it?

A:      C9X specifies long long.

3.      What's the best way to declare and define global variables?

A:      The best arrangement is to place each definition in some

        relevant .c file, with an external declaration in a header file.

4.      What does extern mean in a function declaration?

A:      Nothing, really; the keyword extern is optional here.

5.      What's the auto keyword good for?

A:      Nothing.

6.      I can't seem to define a linked list node which contains a

        pointer to itself.

A:      Structures in C can certainly contain pointers to themselves;

        the discussion and example in section 6.5 of K&R make this

        clear.  Problems arise if an attempt is made to define (and use)

        a typedef in the midst of such a declaration; avoid this.

7.      How do I declare an array of N pointers to functions returning

        pointers to functions returning pointers to characters?

A:      char \*(\*(\*a[N])())();

        Using a chain of typedefs, or the cdecl program, makes these

        declarations easier.

8.      How can I declare a function that returns a pointer to a

        function of its own type?

A:      You can't quite do it directly.  Use a cast, or wrap a struct

        around the pointer and return that.

9.      My compiler is complaining about an invalid redeclaration of a

        function, but I only define it once.

A:      Calling an undeclared function declares it implicitly as

        returning int.

10.  What's the right declaration for main()?

A:      See questions 11.12a to 11.15.

11.  What am I allowed to assume about the initial values

        of variables which are not explicitly initialized?

A:      Uninitialized variables with "static" duration start out as 0,

        as if the programmer had initialized them.  Variables with

        "automatic" duration, and dynamically-allocated memory, start

        out containing garbage (with the exception of calloc).

12.  Why can't I initialize a local array with a string?

A:      Perhaps you have a pre-ANSI compiler.

13.  What's wrong with "char \*p = malloc(10);" ?

A:      Function calls are not allowed in initializers for global or

        static variables.

14.  What is the difference between char a[] = "string"; and

        char \*p = "string"; ?

A:      The first declares an initialized and modifiable array; the

        second declares a pointer initialized to a not-necessarily-

        modifiable constant string.

15.  How do I initialize a pointer to a function?

A:      Use something like "extern int func(); int (\*fp)() = func;" .

Section 2. Structures, Unions, and Enumerations

16.  What's the difference between struct x1 { ... }; and

        typedef struct { ... } x2; ?

A:      The first structure is named by a tag, the second by a typedef

        name.

17.  Why doesn't "struct x { ... }; x thestruct;" work?

A:      C is not C++.

18.  an a structure contain a pointer to itself?

A:      See question 1.14.

19.  What's the best way of implementing opaque (abstract) data types

        in C?

A:      One good way is to use structure pointers which point to

        structure types which are not publicly defined.

20.  I came across some code that declared a structure with the last

        member an array of one element, and then did some tricky

        allocation to make it act like the array had several elements.

        Is this legal or portable?

A:      An official interpretation has deemed that it is not strictly

        conforming with the C Standard.

21.  I heard that structures could be assigned to variables and

        passed to and from functions, but K&R1 says not.

A:      These operations are supported by all modern compilers.

22.  Is there a way to compare structures automatically?

A:      No.

23.  Can I pass constant values to functions which accept structure

        arguments?

A:      Not yet.  As of this writing, C has no way of generating

        anonymous structure values.

24.  How can I read/write structures from/to data files?

A:      It is relatively straightforward to use fread and fwrite.

25.  How can I turn off structure padding?

A:      There is no standard method.

26.  Why does sizeof report a larger size than I expect for a

        structure type?

A:      The alignment of arrays of structures must be preserved.

27.  How can I determine the byte offset of a field within a

        structure?

A:      ANSI C defines the offsetof() macro, which should be used if

        available.

28.  How can I access structure fields by name at run time?

A:      Build a table of names and offsets, using the offsetof() macro.

29.  I have a program which works correctly, but dumps core after it

        finishes.  Why?

A:      Check to see if a structure type declaration just before main()

        is missing its trailing semicolon, causing main() to be declared

        as returning a structure.  See also questions 10.9 and 16.4.

30.  Can I initialize unions?

A:      The current C Standard allows an initializer for the first-named

        member.

31.  What is the difference between an enumeration and a set of

        preprocessor #defines?

A:      At the present time, there is little difference.  The C Standard

        states that enumerations are compatible with integral types.

32.  Is there an easy way to print enumeration values symbolically?

A:      No.

Section 3.Expressions

33.  Why doesn't the code "a[i] = i++;" work?

A:      The variable i is both referenced and modified in the same expression.

34.  Under my compiler, the code "int i = 7;

        printf("%d\n", i++ \* i++);" prints 49.  Regardless of the order

        of evaluation, shouldn't it print 56?

A:      The operations implied by the postincrement and postdecrement operators ++ and -- are performed at some time after the operand's former values are yielded and before the end of the expression, but not necessarily immediately after, or before other parts of the expression are evaluated.

35.  What should the code "int i = 3; i = i++;" do?

A:      The expression is undefined.

36.  Here's a slick expression: "a ^= b ^= a ^= b".  It swaps a and b

        without using a temporary.

A:      Not portably; its behavior is undefined.

37.  Don't precedence and parentheses dictate order of evaluation?

38.  Here's a slick expression: "a ^= b ^= a ^= b".  It swaps a and b without using a temporary.

A:      Not portably; its behavior is undefined.

39.  Don't precedence and parentheses dictate order of evaluation?

A:      Operator precedence and explicit parentheses impose only a partial ordering on the evaluation of an expression, which does not generally include the order of side effects.

40.  But what about the && and || operators?

A:      There is a special exception for those operators: left-to-right evaluation is guaranteed.

41.  What's a "sequence point"?

A:      A point (at the end of a full expression, or at the ||, &&, ?: or comma operators, or just before a function call) at which all side effects are guaranteed to be complete.

42.  So given a[i] = i++; we don't know which cell of a[] gets written to, but i does get incremented by one, right?

A:      \*No\*.  Once an expression or program becomes undefined, \*all\* aspects of it become undefined.

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

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

44.  Why doesn't the code "int a = 1000, b = 1000;

long int c = a \* b;" work?

A:      You must manually cast one of the operands to (long).

45.  Can I use ?: on the left-hand side of an assignment expression?

A:      No.

Section 4. Pointers

46.  What's wrong with "char \*p; \*p = malloc(10);"?

A:      The pointer you declared is p, not \*p.

47.  Does \*p++ increment p, or what it points to?

A:      \*p++ increments p.  To increment the value pointed to by p, use

        (\*p)++ .

48.  I want to use a char \* pointer to step over some ints. Why doesn't "((int \*)p)++;" work?

A:      In C, a cast operator is a conversion operator, and by definition it yields an rvalue, which cannot be assigned to, or incremented with ++.

49.  I have a function which accepts, and is supposed to initialize, a pointer, but the pointer in the caller remains unchanged.

A:      The called function probably altered only the passed copy of the pointer.

50.  Can I use a void \*\* pointer as a parameter so that a function can accept a generic pointer by reference?

A:      Not portably.

51.  I have a function which accepts a pointer to an int.  How can I pass a constant like 5 to it?

A:      You will have to declare a temporary variable.

52.  Does C even have "pass by reference"?

A:      Not really, though it can be simulated.

53.  I've seen different methods used for calling functions via

A:      The extra parentheses and explicit \* are now>

Transfer interrupted!

older implementations require them.

Section 5. Null Pointers

5.1:    What is this infamous null pointer, anyway?

A:      For each pointer type, there is a special value -- the "null pointer" -- which is distinguishable from all other pointer values and which is not the address of any object or function.

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

A:      A constant 0 in a pointer context is converted into a null pointer at compile time.  A "pointer context" is an initialization, assignment, or comparison with one side a variable or expression of pointer type, and (in ANSI standard C) a function argument which has a prototype in scope declaring a certain parameter as being of pointer type.  In other contexts (function arguments without prototypes, or in the variable part of variadic function calls) a constant 0 with an appropriate explicit cast is required.

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

A:      Yes.  The construction "if(p)" works, regardless of the internal representation of null pointers, because the compiler essentially rewrites it as "if(p != 0)" and goes on to convert 0 into the correct null pointer.

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

A:      NULL is simply a preprocessor macro, #defined as 0 (or ((void \*)0)), which is used (as a stylistic convention, in  preference to unadorned 0's) to generate null pointers.

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

A:      The same as on any other machine: as 0.  (The compiler makes the translation, upon seeing a 0, not the preprocessor; see also question 5.4.)

5.6:    If NULL were defined as "((char \*)0)," wouldn't that make function calls which pass an uncast NULL work?

A:      Not in general.  The complication is that there are machines which use different internal representations for pointers to different types of data.  A cast is still required to tell the compiler which kind of null pointer is required, since it may be different from (char \*)0.

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

A:      Either; the distinction is entirely stylistic.

5.10:   But wouldn't it be better to use NULL, in case the value of NULL changes?

A:      No.  NULL is a constant zero, so a constant zero is equally sufficient.

5.12:   I use the preprocessor macro "#define Nullptr(type) (type \*)0" to help me build null pointers of the correct type.

A:      This trick, though valid, does not buy much.

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

A:      A "null pointer" is a language concept whose particular internal value does not matter.  A null pointer is requested in source code with the character "0".  "NULL" is a preprocessor macro, which is always #defined as 0 (or ((void \*)0)).

5.14:   Why is there so much confusion surrounding null pointers?

A:      The fact that null pointers are represented both in source code, and internally to most machines, as zero invites unwarranted assumptions.  The use of a preprocessor macro (NULL) may seem to suggest that the value could change some day, or on some weird machine.

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

A:      A simple rule is, "Always use `0' or `NULL' for null pointers, and always cast them when they are used as arguments in function calls."

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

A:      Such a requirement would accomplish little.

5.17:   Seriously, have any actual machines really used nonzero null pointers?

A:      Machines manufactured by Prime, Honeywell-Bull, and CDC, as well as Symbolics Lisp Machines, have done so.

5.20:   What does a run-time "null pointer assignment" error mean?

A:      It means that you've written, via a null pointer, to an invalid location.  (See also question 16.8.)

Section 6. Arrays and Pointers

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

A:      The declaration extern char \*a simply does not match the actual definition.  Use extern char a[].

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

A:      Not at all.  Arrays are not pointers.  A reference like x[3] generates different code depending on whether x is an array or a pointer.

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

A:      An lvalue of type array-of-T which appears in an expression decays into a pointer to its first element; the type of the resultant pointer is pointer-to-T.  So for an array a and pointer p, you can say "p = a;" and then p[3] and a[3] will access the same element.

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

A:      It's supposed to be a convenience.

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

A:      An array is not a "modifiable lvalue."

6.8:    What is the real difference between arrays and pointers?

A:      Arrays automatically allocate space which is fixed in size and location; pointers are dynamic.

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

A:      An array name is "constant" in that it cannot be assigned to, but an array is \*not\* a pointer.

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

A:      Yes, array subscripting is commutative in C.  The array subscripting operation a[e] is defined as being identical to \*((a)+(e)).

6.12:   What's the difference between array and &array?

A:      The type.

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

A:      Usually, you don't want to.  Consider using a pointer to one of the array's elements instead.

6.14:   How can I set an array's size at run time?

A:      It's straightforward to use malloc() and a pointer.

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

A:      Until recently, you couldn't; array dimensions had to be compile- time constants.  C9X will fix this.

6.16:   How can I dynamically allocate a multidimensional array?

A:      The traditional solution is to allocate an array of pointers, and then initialize each pointer to a dynamically-allocated "row."  See the full list for code samples.

6.17:   Can I simulate a non-0-based array with a pointer?

A:      Not if the pointer points outside of the block of memory it is intended to access.

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

A:      The rule by which arrays decay into pointers is not applied recursively.  An array of arrays (i.e. a two-dimensional array in C) decays into a pointer to an array, not a pointer to a pointer.

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

A:      It's not always particularly easy.

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

A:      There is no single perfect method, but see the full list for some ideas.

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

A:      The sizeof operator reports the size of the pointer parameter which the function actually receives.

Section 7. Memory Allocation

7.1:    Why doesn't the code "char \*answer; gets(answer);" work?

A:      The pointer variable answer has not been set to point to any

        valid storage.  The simplest way to correct this fragment is to use a local array, instead of a pointer.

7.2:    I can't get strcat() to work.  I tried "char \*s3 = strcat(s1, s2);" but I got strange results.

A:      Again, the main problem here is that space for the concatenated result is not properly allocated.

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

A:      In general, when using pointers you \*always\* have to consider memory allocation, if only to make sure that the compiler is doing it for you.

7.3b:   I just tried the code "char \*p; strcpy(p, "abc");" and it worked.  Why didn't it crash?

A:      You got "lucky".

7.3c:   How much memory does a pointer variable allocate?

A:      Only enough memory to hold the pointer itself, not any memory for the pointer to point to.

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

A:      Make sure that the pointed-to memory is properly (i.e. not locally) allocated.

7.5b:   So what's the right way to return a string?

A:      Return a pointer to a statically-allocated buffer, a buffer passed in by the caller, or memory obtained with malloc().

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

A:      Have you #included <stdlib.h>?

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

A:      Before ANSI/ISO C, these casts were required to silence certain warnings.

7.8:    Why does so much code leave out the multiplication by sizeof(char) when allocating strings?

A:      Because sizeof(char) is, by definition, exactly 1.

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

A:      It's hard to say.

7.16:   I'm allocating a large array for some numeric work, but malloc() is acting strangely.

A:      Make sure the number you're trying to pass to malloc() isn't bigger than a size\_t can hold.

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

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

7.19:   My program is crashing, apparently somewhere down inside malloc.

A:      Make sure you aren't using more memory than you malloc'ed, especially for strings (which need strlen(str) + 1 bytes).

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

        can you?

A:      No.  Some early documentation implied otherwise, but the claim is no longer valid.

7.21:   Why isn't a pointer null after calling free()?

A:      C's pass-by-value semantics mean that called functions can never permanently change the values of their arguments.

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

A:      Yes.

7.23:   When I free a dynamically-allocated structure containing pointers, do I also have to free each subsidiary pointer?

A:      Yes.

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

A:      You shouldn't have to.

7.25:   Why doesn't my program's memory usage go down when I free memory?

A:      Most implementations of malloc/free do not return freed memory to the operating system.

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

A:      The malloc/free implementation remembers the size of each block as it is allocated.

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

A:      Not portably.

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

A:      ANSI C sanctions this usage, although several earlier implementations do not support it.

7.31:   What's the difference between calloc() and malloc()?

A:      calloc() takes two arguments, and initializes the allocated memory to all-bits-0.

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

A:      alloca() allocates memory which is automatically freed when the function which called alloca() returns.  alloca() cannot be written portably, is difficult to implement on machines without a stack, and fails under certain conditions if implemented simply.

Section 8. Characters and Strings

8.1:    Why doesn't "strcat(string, '!');" work?

A:      strcat() concatenates \*strings\*, not characters.

8.2:    Why won't the test if(string == "value") correctly compare string against the value?

A:      It's comparing pointers.  To compare two strings, use strcmp().

8.3:    Why can't I assign strings to character arrays?

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

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

A:      In C, if you have the character, you have its value.

8.9:    Why is sizeof('a') not 1?

A:      Character constants in C are of type int.

Section 9. Boolean Expressions and Variables

9.1:    What is the right type to use for Boolean values in C?

A:      There's no one right answer; see the full list for some discussion.

9.2:    What if a built-in logical or relational operator "returns" something other than 1?

A:      When a Boolean value is generated by a built-in operator, it is guaranteed to be 1 or 0.  (This is \*not\* true for some library routines such as isalpha.)

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

A:      Yes.  See question 5.3.

Section 10. C Preprocessor

10.2:   I've got some cute preprocessor macros that let me write C code that looks more like Pascal.  What do y'all think?

A:      Bleah.

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

A:      There is no good answer to this question.  The best all-around solution is probably to forget about using a macro.

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

A:      #define Func() do {stmt1; stmt2; ... } while(0) /\* (no trailing ;) \*/

10.6:   What are .h files and what should I put in them?

A:      Header files (also called ".h files") should generally contain common declarations and macro, structure, and typedef definitions, but not variable or function definitions.

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

A:      It's a question of style, and thus receives considerable debate.

10.8a:  What's the difference between #include <> and #include "" ?

A:      Roughly speaking, the <> syntax is for Standard headers and "" is for project headers.

10.8b:  What are the complete rules for header file searching?

A:      The exact behavior is implementation-defined; see the full list for some discussion.

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

A:      Perhaps there's a missing semicolon at the end of the last declaration in the last header file you're #including.

10.10b: I'm #including the header file for a function, but the linker keeps saying it's undefined.

A:      See question 13.25.

10.11:  Where can I get a copy of a missing header file?

A:      Contact your vendor, or see question 18.16 or the full list.

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

A:      You can't do it directly; try #defining several manifest constants and implementing conditionals on those.

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

A:      No.

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

A:      No.

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

A:      Unfortunately, no.

10.16:  How can I use a preprocessor #if expression to detect endianness?

A:      You probably can't.

10.18:  How can I preprocess some code to remove selected conditional compilations, without preprocessing everything?

A:      Look for a program called unifdef, rmifdef, or scpp.

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

A:      If the compiler documentation is unhelpful, try extracting printable strings from the compiler or preprocessor executable.

10.20:  I have some old code that tries to construct identifiers with a macro like "#define Paste(a, b) a/\*\*/b", but it doesn't work any more.

A:      Try the ANSI token-pasting operator ##.

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

A:      See question 11.18.

10.23-4: I'm having trouble using macro arguments inside string literals, using the `#' operator.

A:      See questions 11.17 and 11.18.

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

A:      Consider writing your own little special-purpose preprocessing tool, instead.

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

A:      Here is one popular trick.  Note that the parentheses around printf's argument list are in the macro call, not the definition.

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

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

Section 11. ANSI/ISO Standard C

11.1:   What is the "ANSI C Standard?"

A:      In 1983, the American National Standards Institute (ANSI) commissioned a committee to standardize the C language.  Their work was ratified as ANS X3.159-1989, and has since been adopted as ISO/IEC 9899:1990, and later amended.

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

A:      Copies are available from ANSI in New York, or from Global Engineering Documents in Englewood, CO, or from any national standards body, or from ISO in Geneva, or republished within one or more books.  See the unabridged list for details.

11.2b:  Where can I get information about updates to the Standard?

A:      See the full list for pointers.

11.3:   My ANSI compiler is complaining about prototype mismatches for parameters declared float.

A:      You have mixed the new-style prototype declaration "extern int func(float);" with the old-style definition "int func(x) float x;".  "Narrow" types are treated differently according to which syntax is used.  This problem can be fixed by avoiding narrow types, or by using either new-style (prototype) or old-style syntax consistently.

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

A:      Doing so is currently legal, for most argument types (see question 11.3).

11.5:   Why does the declaration "extern int f(struct x \*p);" give me a warning message?

A:      A structure declared (or even mentioned) for the first time within a prototype cannot be compatible with other structures declared in the same source file.

11.8:   Why can't I use const values in initializers and array dimensions?

A:      The value of a const-qualified object is \*not\* a constant expression in the full sense of the term.

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

A:      The former declares a pointer to a constant character; the latter declares a constant pointer to a character.

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

A:      The rule which permits slight mismatches in qualified pointer assignments is not applied recursively.

11.12a: What's the correct declaration of main()?

A:      int main(int argc, char \*argv[]) .

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

A:      No.

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

A:      It's a non-standard (though common) extension.

11.14:  I believe that declaring void main() can't fail, since I'm calling exit() instead of returning.

A:      It doesn't matter whether main() returns or not, the problem is that its caller may not even be able to \*call\* it correctly.

11.15:  The book I've been using always uses void main().

A:      It's wrong.

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

A:      Yes and no.  (See the full list for details.)

11.17:  How do I get the ANSI "stringizing" preprocessing operator `#' to stringize the macro's value instead of its name?

A:      You can use a two-step #definition to force a macro to be expanded as well as stringized.

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

A:      Some pre-ANSI compilers/preprocessors expanded macro parameters even inside string literals and character constants.

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

A:      Under ANSI C, #ifdeffed-out text must still consist of "valid preprocessing tokens."  This means that there must be no newlines inside quotes, and no unterminated comments or quotes (i.e. no single apostrophes).

11.20:  What are #pragmas ?

A:      The #pragma directive provides a single, well-defined "escape hatch" which can be used for extensions.

11.21:  What does "#pragma once" mean?

A:      It is an extension implemented by some preprocessors to help make header files idempotent.

11.22:  Is char a[3] = "abc"; legal?

A:      Yes, in ANSI C.

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

A:      The compiler doesn't know the size of the pointed-to objects.

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

A:      memmove() offers guaranteed behavior if the source and destination arguments overlap.

11.26:  What should malloc(0) do?

A:      The behavior is implementation-defined.

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

A:      The problem is older linkers which cannot be forced (by mere words in a Standard) to upgrade.

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

A:      Perhaps it is a pre-ANSI compiler.

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

A:      Perhaps you don't have ANSI-compatible headers and libraries.

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

A:      See the full list for details.

11.32:  Why won't frobozz-cc, which claims to be ANSI compliant, accept this code?

A:      Are you sure that the code being rejected doesn't rely on some non-Standard extension?

11.33:  What's the difference between implementation-defined, unspecified, and undefined behavior?

A:      If you're writing portable code, ignore the distinctions. Otherwise, see the full list.

11.34:  I'm appalled that the ANSI Standard leaves so many issues undefined.

A:      In most of these cases, the Standard is simply codifying existing practice.

11.35:  I just tried some allegedly-undefined code on an ANSI-conforming compiler, and got the results I expected.

A:      A compiler may do anything it likes when faced with undefined behavior, including doing what you expect.

Section 12. Stdio

12.1:   What's wrong with the code "char c; while((c = getchar()) != EOF) ..."?

A:      The variable to hold getchar's return value must be an int.

12.2:   Why won't the code "while(!feof(infp)) {

        fgets(buf, MAXLINE, infp); fputs(buf, outfp); }" work?

A:      EOF is only indicated \*after\* an input routine fails.

12.4:   My program's prompts and intermediate output don't always show up on the screen.

A:      It's best to use an explicit fflush(stdout) whenever output should definitely be visible.

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

A:      See question 19.1.

12.6:   How can I print a '%' character with printf?

A:      "%%".

12.9:   How can printf() use %f for type double, if scanf() requires %lf?

A:      C's "default argument promotions" mean that values of type float are promoted to double.

12.9b:  What printf format should I use for a typedef when I don't know the underlying type?

A:      Use a cast to convert the value to a known type, then use the printf format matching that type.

12.10:  How can I implement a variable field width with printf?

A:      Use printf("%\*d", width, x).

12.11:  How can I print numbers with commas separating the thousands?

A:      There is no standard routine (but see <locale.h>).

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

A:      The arguments you pass to scanf() must always be pointers.

12.13:  Why doesn't the code "double d; scanf("%f", &d);" work?

A:      Unlike printf(), scanf() uses %lf for double, and %f for float.

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

A:      You can't.

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

A:      Try using "%d" instead of "%d\n".

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

A:      scanf() and gets() do not work well together.

12.19:  I'm re-prompting the user if scanf() fails, but sometimes it seems to go into an infinite loop.

A:      scanf() tends to "jam" on bad input since it does not discard it.

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

A:      scanf() has a number of problems.  Usually, it's easier to read entire lines and then interpret them.

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

A:      Use the new snprintf() function, if you can.

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

A:      It cannot be prevented from overflowing the input buffer.

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

A:      Don't worry about it.  It is only meaningful for a program to inspect the contents of errno after an error has been reported.

12.25:  What's the difference between fgetpos/fsetpos and ftell/fseek?

A:      fgetpos() and fsetpos() use a special typedef which may allow  them to work with larger files than ftell() and fseek().

12.26:  Will fflush(stdin) flush unread characters from the standard input stream?

A:      No.

12.30:  I'm trying to update a file in place, by using fopen mode "r+", but it's not working.

A:      Be sure to call fseek between reading and writing.

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

A:      Use freopen().

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

A:      There isn't a good way.  Try avoiding freopen.

12.36b: How can I arrange to have output go two places at once?

A:      You could write your own printf variant which printed everything twice.  See question 15.5.

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

A:      Be sure to specify "rb" mode when calling fopen().

Section 13. Library Functions

13.1:   How can I convert numbers to strings?

A:      Just use sprintf().

13.2:   Why does strncpy() not always write a '\0'?

A:      For mildly-interesting historical reasons.

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

A:      Older versions of toupper () and tolower () did not always work as expected in this regard.

13.6:   How can I split up a string into white space-separated fields?

A:      Try strtok().

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

A:      regexp libraries abound; see the full list for details.

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

A:      You'll have to write a "helper" comparison function which takes two generic pointer arguments, converts them to char \*\*, and dereferences them, yielding char \*'s which can be usefully compared.

13.9:   Now I'm trying to sort an array of structures, but the compiler is complaining that the function is of the wrong type for qsort().

A:      The comparison function must be declared as accepting "generic pointers" (const void \*) which it then converts to structure pointers.

13.10:  How can I sort a linked list?

A:      Algorithms like insertion sort and merge sort work well, or you can keep the list in order as you build it.

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

A:      You want an "external sort"; see the full list for details.

13.12:  How can I get the time of day in a C program?

A:      Just use the time(), ctime(), localtime() and/or strftime() functions.

13.13:  How can I convert a struct tm or a string into a time\_t?

A:      The ANSI mktime() function converts a struct tm to a time\_t.  No standard routine exists to parse strings.

13.14:  How can I perform calendar manipulations?

A:      The ANSI/ISO Standard C mktime() and difftime() functions provide some support for both problems.

13.14b: Does C have any Year 2000 problems?

A:      No, although poorly-written C programs do.  Make sure you know that tm\_year holds the value of the year minus 1900.

13.15:  I need a random number generator.

A:      The Standard C library has one: rand().

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

A:      One method is something like

                (int)((double)rand() / ((double)RAND\_MAX + 1) \* N)

13.17:  Each time I run my program, I get the same sequence of numbers back from rand().

A:      You can call srand() to seed the pseudo-random number generator with a truly random initial value.

13.18:  I need a random true/false value, so I'm just taking rand () % 2, but it's alternating 0, 1, 0, 1, 0...

A:      Try using the higher-order bits: see question 13.16.

13.20:  How can I generate random numbers with a normal or Gaussian distribution?

A:      See the longer versions of this list for ideas.

13.24:  I'm trying to port this old program.  Why do I get "undefined external" errors for some library functions?

A:      Some semi standard functions have been renamed or replaced over the years; see the full list for details.

13.25:  I get errors due to library functions being undefined even though I #include the right header files.

A:      You may have to explicitly ask for the correct libraries to be searched.

13.26:  I'm still getting errors due to library functions being undefined, even though I'm requesting the right libraries.

A:      Library search order is significant; usually, you must search the libraries last.

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

A:      You generally get that message only when other symbols are undefined, too.

Section 14. Floating Point

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

A:      Most computers use base 2 for floating-point numbers, and many fractions (including 0.1 decimal) are not exactly representable in base 2.

14.2:   Why is sqrt(144.) giving me crazy numbers?

A:      Make sure that you have #included <math.h>, and correctly declared other functions returning double.

14.3:   I keep getting "undefined: sin" compilation errors.

A:      Make sure you're actually linking with the math library.

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

A:      First, see question 14.2 above.  If the problem isn't that simple, see the full list for a brief explanation, or any good programming book for a better one.

14.5:   What's a good way to check for "close enough" floating-point equality?

A:      The best way is to use an accuracy threshold which is relative to the magnitude of the numbers being compared.

14.6:   How do I round numbers?

A:      For positive numbers, try (int)(x + 0.5) .

14.7:   Where is C's exponentiation operator?

A:      Try using the pow() function