1. Declarations and Initializations

1.1 How do you decide which integer type to use?

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

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

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

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

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

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

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

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

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

`#include` the same header file, so that the compiler can check that the definition matches the declarations.

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

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

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

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

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

1.11 What does `extern` mean in a function declaration?

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

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

and

```
int f();
```

1.12 What's the `auto` keyword good for?

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

1.14 I can't seem to define a linked list node which contains a pointer to itself.

Repeated Question

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

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

1. `char *(*(*a[N])())();`
2. Build the declaration up incrementally, using typedefs:

```
  typedef char *pc;      /* pointer to char */
  typedef pc fpc();      /* function returning pointer to char */
```

```
5.   typedef fpc *pfpc;     /* pointer to above */
6.   typedef pfpc fpfpc();  /* function returning... */
7.   typedef fpfpc *pfpfpc; /* pointer to... */
8.   pfpfpc a[N];           /* array of... */
```

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

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

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

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

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

1.22 How can I declare a function that returns a pointer to a function of its own type?

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

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

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

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

1.30 What can I safely assume about the initial values of variables which are not explicitly initialized?

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

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

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

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

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

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

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

1.32 What is the difference between `char a[] = "string";` and `char *p = "string";` ?

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

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

1.34 How do I initialize a pointer to a function?

Use something like

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

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

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

---

2. Structures, Unions, and Enumerations

2.1 What's the difference between `struct x1 { ... };` and `typedef struct { ... } x2;` ?

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

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

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

2.3 Can a structure contain a pointer to itself?

Most certainly

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

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

2.6 I came across some code that declared a structure 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?

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

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

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

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

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

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

2.8 Why can't you compare structures?

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

2.9 How are structure passing and returning implemented?

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

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

2.10 Can I pass constant values to functions which accept structure arguments?

C has no way of generating anonymous structure values. You will have to use a temporary structure variable or a little structure-building function; see question 14.11 for an example. (`gcc` provides structure constants as an extension, and the mechanism will probably be added to a future revision of the C Standard.) See also question 4.10.

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

It is relatively straightforward to write a structure out using `fwrite`:

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

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

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

2.12 How can I turn off structure padding?

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

2.13 Why does `sizeof` report a larger size than I expect for a structure type?

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

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

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

```
#define offsetof(type, mem) ((size_t) \
        ((char *)&((type *)0)->mem - (char *)(type *)0))
```

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

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

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

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

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

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

2.18 I have a program which works correctly, but dumps core after it finishes. Why?

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

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

/* Here is the main program. */

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

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

2.20 Can I initialize unions?

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

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

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

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

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

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

---

3. Expressions

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

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

3.2 Under my compiler, the code "`int i = 7; printf("%d\n", i++ * i++);`" prints 49. Regardless of the order of evaluation, shouldn't it print 56?

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

The behavior of code which contains multiple, ambiguous side effects has always been undefined. (Loosely speaking, by ``multiple, ambiguous side effects'' we mean any combination of `++`, `--`, `=`, `+=`, `-=`, etc. in a single expression which causes the same object either to be modified twice or modified and then inspected. This is a rough definition; see question 3.8 for a precise one, and question 11.33 for the meaning of ``undefined.'') Don't even try to find out how your compiler implements such things (contrary to the ill-

advised exercises in many C textbooks); as K&R wisely point out, ``if you don't know *how* they are done on various machines, that innocence may help to protect you."

3.3 How could the code "`int i = 3; i = i++;`" ever give 7?

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

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

3.4 Don't precedence and parentheses dictate order of evaluation?

Not in general.

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

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

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

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

3.5 But what about the `&&` and `||` operators?

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

3.8 What's a ``sequence point"?

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

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

The second sentence can be difficult to understand. It says that if an object is written to within a full expression, any and all accesses to it within the same expression must be for

the purposes of computing the value to be written. This rule effectively constrains legal expressions to those in which the accesses demonstrably precede the modification.

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

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

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

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

3.14 Why doesn't the code "`int a = 1000, b = 1000; long int c = a * b;`" work?

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

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

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

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

3.16 Can I use `?:` on the left-hand side of an assignment expression?

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

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

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

although this is admittedly not as pretty.

---

4. Pointers

4.2 What's wrong with "`char *p; *p = malloc(10);`"?

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

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

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

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

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

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

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

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

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

or (since `p` is a `char *`) simply

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

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

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

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

But when I call it like this:

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

the pointer in the caller remains unchanged.

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

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

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

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

I have a function

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

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

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

doesn't seem to work.

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

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

4.11 Does C even have ``pass by reference''?

Not really. Strictly speaking, C always uses pass by value. You can simulate pass by reference yourself, by defining functions which accept pointers and then using the `&` operator when calling, and the compiler will essentially simulate it for you when you pass an array to a function (by passing a pointer instead, see question 6.4 et al.), but C has nothing truly equivalent to formal pass by reference or C++ reference parameters. (However, function-like preprocessor macros do provide a form of ``call by name''.)

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

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

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

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

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

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

---

5. Null Pointers

5.1 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, the address-of operator & will never yield a null pointer, nor will a successful call to malloc. (malloc does return a null pointer when it fails, and this is a typical use of null pointers: as a ``special" pointer value with some other meaning, usually ``not allocated" or ``not pointing anywhere yet.")

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

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

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

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

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

(See also question 5.3.)

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

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

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

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

Summary:

```
Unadorned 0 okay:       Explicit cast required:

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

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

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

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

```
if(expr)
```

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

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

Substituting the trivial pointer expression ``p'' for ``expr,'' we have

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

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

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

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

which leads to the conclusion that

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

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

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

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

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

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

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

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

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

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

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

could fail.

Nevertheless, ANSI C allows the alternate definition

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

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

5.9 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. (See also questions 9.2 and 17.10.) C programmers must understand that NULL and 0 are interchangeable in pointer contexts, and that an uncast 0 is perfectly acceptable. Any usage of NULL (as opposed to 0) should be considered a gentle reminder that a pointer is involved; programmers should not depend on it (either for their own understanding or the compiler's) for distinguishing pointer 0's from integer 0's.

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

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

if you must.

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

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

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

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

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

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

1. 1. The conceptual null pointer, the abstract language concept defined in question 5.1. It is implemented with...
2. 2. The internal (or run-time) representation of a null pointer, which may or may not be all-bits-0 and which may be different for different pointer types. The actual values should be of concern only to compiler writers. Authors of C programs never see them, since they use...
3. 3. The null pointer constant, which is a constant integer 0 (see question 5.2). It is often hidden behind...
4. 4. The `NULL` macro, which is `#defined` to be `0` or `((void *)0)` (see question 5.4). Finally, as red herrings, we have...
5. 5. The ASCII null character (NUL), which does have all bits zero, but has no necessary relation to the null pointer except in name; and...
6. 6. The ``null string,'' which is another name for the empty string (`""`). Using the term ``null string'' can be confusing in C, because an empty string involves a null (`'\0'`) character, but *not* a null pointer, which brings us full circle...

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

5.14 Why is there so much confusion surrounding null pointers?

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

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

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

Follow these two simple rules:

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

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

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

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

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

mentioned in question 5.14, the urge to assume an internal zero representation would not even arise.)

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

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

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

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

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

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

The Symbolics Lisp Machine, a tagged architecture, does not even have conventional numeric pointers; it uses the pair `<NIL, 0>` (basically a nonexistent <object, offset> handle) as a C null pointer.

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

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

5.20 What does a run-time ``null pointer assignment'' error mean?

This message, which typically occurs with MS-DOS compilers (see, therefore, section 19) means that you've written, via a null (perhaps because uninitialized) pointer, to location 0. (See also question 16.8.) A debugger may let you set a data breakpoint or watchpoint or something on location 0. Alternatively, you could write a bit of code to stash away a copy of 20 or so bytes from location 0, and periodically check that the memory at location 0 hasn't changed

---

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?

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

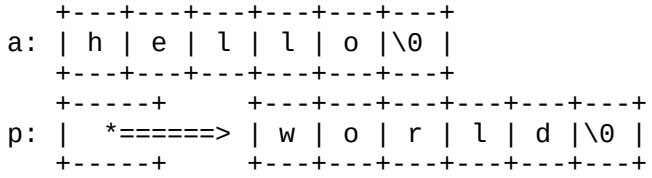
6.2 But I heard that `char a[]` was identical to `char *a`.

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

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

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

would initialize data structures which could be represented like this:



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

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

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

``Equivalence" refers to the following key definition:

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

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

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

```
p = a;
```

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

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

It's supposed to be a convenience.

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

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

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

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

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

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

> Repeated

6.8 What is the real difference between arrays and pointers?

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

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

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

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

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

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

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

The type.

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

6.13 How do I declare a pointer to an array?

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

Instead of a pointer to an array, consider using a pointer to one of the array's elements. Arrays of type T decay into pointers to type T (see question 6.3), which is convenient; subscripting or incrementing the resultant pointer will access the individual members of

the array. True pointers to arrays, when subscripted or incremented, step over entire arrays, and are generally useful only when operating on arrays of arrays, if at all. (See question 6.18.)

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

See also question 6.12.

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

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

```
#include <stdlib.h>
int *dynarray = (int *)malloc(10 * sizeof(int));
```

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

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

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

6.16 How can I dynamically allocate a multidimensional array?

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

```
#include <stdlib.h>

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

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

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

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

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

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

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

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

Finally, you could use pointers to arrays:

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

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

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

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

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

Here's a neat trick: if I write

```
        int realarray[10];
        int *array = &realarray[-1];
```

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

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

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

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

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

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

the function's declaration must match:

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

or

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

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

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

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

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

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

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

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

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

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

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

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

There is no single perfect method. Given the declarations

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

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

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

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

```
f1(array, NROWS, NCOLUMNS);
f1(array4, nrows, NCOLUMNS);

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

f3(array1, nrows, ncolumns);
f3(array2, nrows, ncolumns);
```

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

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

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

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

Rather than worrying about all of this, one approach to using multidimensional arrays of various sizes is to make them *all* dynamic, as in question 6.16. If there are no static

multidimensional arrays--if all arrays are allocated like `array1` or `array2` in question 6.16--then all functions can be written like `f3`.

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

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

---

7. Memory Allocation

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

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

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

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

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

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

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

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

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

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

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

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

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?

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

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

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

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

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

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

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

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

Have you `#included <stdlib.h>`, or otherwise arranged for `malloc` to be declared properly?

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

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

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

I see code like

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

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

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

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?

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

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

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

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

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

Notice that 256 x 256 is 65,536, which will not fit in a 16-bit `int`, even before you multiply it by `sizeof(double)`. If you need to allocate this much memory, you'll have to be careful. If `size_t` (the type accepted by `malloc`) is a 32-bit type on your machine, but

`int` is 16 bits, you might be able to get away with writing `256 * (256 * sizeof(double))` (see question 3.14). Otherwise, you'll have to break your data structure up into smaller chunks, or use a 32-bit machine, or use some nonstandard memory allocation routines. See also question 19.23.

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

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`.

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

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

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

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

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

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

7.21 Why isn't a pointer null after calling `free`?

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

A pointer value which has been freed is, strictly speaking, invalid, and *any* use of it, even if is not dereferenced can theoretically lead to trouble, though as a quality of

implementation issue, most implementations will probably not go out of their way to generate exceptions for innocuous uses of invalid pointers.

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

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

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

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

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

7.24 Must I free allocated memory before the program exits?

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

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

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

7.26 How does `free` know how many bytes to free?

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

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

Not portably.

7.30 Is it legal to pass a null pointer as the first argument to `realloc`?

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

7.31 What's the difference between `calloc` and `malloc`?

`calloc(m, n)` is essentially equivalent to

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

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

7.32 What is `alloca` and why is its use discouraged?

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

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

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

---

8. Characters and Strings

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

There is a very real difference between characters and strings, and `strcat` concatenates *strings*.

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

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

See also questions 1.32, 7.2, and 16.6.

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

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

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

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

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

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

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

See also questions 1.32, 4.2, and 7.2.

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

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

8.9 Why is `sizeof('a')` not 1?

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

---

9. Boolean Expressions and Variables

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

C does not provide a standard Boolean type, in part because picking one involves a space/time tradeoff which can best be decided by the programmer. (Using an `int` may be

faster, while using `char` may save data space. Smaller types may make the generated code bigger or slower, though, if they require lots of conversions to and from `int`.)

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

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

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

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

Some people prefer variants like

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

or define ``helper'' macros such as

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

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

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

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

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

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

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

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

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

Yes. See question 5.3.
