wdv-notes 339
Stand: 1.APR.1995 (2.)
Wiss.Datenverarbeitung
FREIE UNIVERSITÄT BERLIN

Aren't you glad you're not using C ? The ultimate C IAQ.

© 1994–1995 Edited by *Karl-Heinz Dittberner*

Pro

Jedes Jahr zum 1. April wird im internationalen UseNet (*comp.lang.c*) die inzwischen schon berühmte IAQ (keine FAQ!) zu den Programmiersprachen C und C++ [1–5] publiziert.

In dieser IAQ (Infrequently Asked Questions) werden die vielen Ungereimtheiten dieser – bei vielen so beliebten – Programmiersprachen in etwas satirischer Form behandelt. Enjoy!

Annual IAQ Posting

By PETER SEEBACH
Digital Solutions Internet Consulting
seebs@taniemarie.solon.com
Posted to UseNet (Internet) – Apr 1, 1995

Certain topics never (well, hardly ever) come up on this newsgroup. They are stupid questions, to which the answers are immediately obvious, but they would be more fun to talk about than these arcane details of loop control. This article, which is posted yearly, attempts to answer these questions definitively, succinctly, and in such a way as to discourage further discussion. The answers have been carefully checked for periodic accuracy, and for blatant inaccuracy where relevant.

Contents

1. Null Statements
2. Arrays and Pointers
3. Memory Allocation
4. Expressions
5. ANSI C
6. C Preprocessor
7. Variable-Length Argument Lists
8. Boolean Expressions and Variables
9. Structs, Enums, and Unions
10. Declarations
11. Stdio
12. Library Subroutines
13. Lint
14. Style
15. Floating Point
16. System Dependencies
17. Miscellaneous (Fortran to C converters, YACC grammars, etc.)

Herewith, some infrequently-asked questions and their answers:

1: Null Statements

1.1: What is this infamous null statement, anyway?
A: A null statement is an expression statement consisting solely of the terminating semicolon. The optional expression is dropped. It can be distinguished from any other statement by byte count or study of side-effects.

1.2: How do I "get" a null statement in my programs?
A: In ANSI C, there are six types of statements; labelled statements, compound statements, expression-statements, selection statements, iteration statements, and jump statements. All of them, except the jump and expression statements, are defined in terms of optional preceding text, and other statements. The jump statements are never null statements. An expression statement is considered to be "a null statement" if the optional expression part of it has been left out. A null statement can appear on its own, or (most frequently) as the statement body of an iteration statement. These two null statements are equivalent, though neither of them is equivalent to any non-null statement.* You may accidentally get a null statement by deleting the body of a non-null statement.
* Actually, they are functionally equivalent to a large set of non-null statements, namely, those with no side-effects. However, the FDA has yet to approve any such, as their lack of side effects is conjectured, and not clinically proven. This applies only to the ANSI standard, and not the ISO standard, as the FDA has no jurisdiction outside the U.S.

1.3: Is there more than one null statement?
A: Sort of. You can use `;` or `0;` or `1;` – they will all act like a null statement. Only the first is a "true" null statement (all bits zero). They are basically equivalent. Note that `(void *) 0;` is a null statement of type pointer to void, for instance.

1.4: But I thought `{ }` was a null statement!
A: No. `{ statement-list[opt] }` is a compound statement. An empty block is not the same as a null statement, however, although it can be used in many of the same places. It's really a null block.

1.5: I use the statement
`#define NULLSTMT(F) (F) ;`
to allow me to cast a null statement to an appropriate type.
A: This trick, though popular in some circles, does not buy much. The resulting code is illegal, and will not compile. This (in the author's opinion) outweighs any arguable type consistency. It may be more common in industrial code. If it becomes common practice, C++ will probably legalize it.

1.6: I use the statement
`#define NULLSTMT(F) (F) 0;`
to allow me to cast a null statement to an appropriate type.
A: This trick will likely work, but think: what does it really buy you? Mostly, it will indicate to even the most casual observer that you are shaky on the concept of null statements, making it harder for them to check your code.

1.7: But wouldn't it be better to use `;` (rather than `0;`) in case the value of 0 changes, perhaps on a machine with nonzero no-op instructions?
A: No. The `0` of `0;` is not evaluated as an instruction, rather, it is just ignored. The advantages of `;` over `0;` have only to do with poor optimizers and savings of keystrokes.

1.8: Is a null statement a null pointer?
A: No. A null pointer is a pointer where all of the address bits are zero (no matter what the segment bits are), and can be obtained by typing `(char *) (int) 0`. A null statement is not a pointer to anything. They are not interchangeable, although you can combine them to get an effectively-null statement, such as `NULL;` This does not buy you anything.

1.9: I'm still confused. I just can't understand all this null statement stuff.
A: Follow these two simple rules:
1. When you don't want to do anything in source code, don't write it.
2. If you need a null statement to round out an expression, use an unadorned '`;`' to provide it.
3. Send large donations, checks, and money orders to the author of the FAQ, or the moderator of the group, whichever you prefer. Then, cross the top question off the FAQ, answer the question at the bottom, and mail it to three people. Within two weeks, you will receive 729 answers to various questions! Do not break the chain; Emily Postnews broke the chain, and now no one listens to her.

„The last good thing written in C was Franz Schubert's Ninth Symphony."

2: Arrays and Pointers

2.1: I had the definition `char a[6]` in one source file, and in another I declared `extern char a[]`. Why did it work?
A: The declaration `extern char a[]` simply matches the actual definition. The type "array-of-type-T" is the same as "array-of-type-T". Go ahead and use `extern char a[]`. (For greater portability, use it in both files, not only in one of them.)

2.2: But I heard that `char a[]` was different from `char a[6]`.
A: This is true. However, the declaration `a[]` is compatible with the definition `a[6]`.

2.3: So what is meant by the "equivalence of pointers and arrays" in C?
A: Very little.

2.4: Then why are array and pointer declarations interchangeable as function formal parameters?
A: Classicism. We consider arrays "second class objects". They don't vote, and they get treated as pointers. Additionally, they're merely objects, not citizens. Marx wrote about this a lot.

2.5: Why doesn't `sizeof` properly report the size of an array which is a parameter to a function?
A: Part of the ANSI conspiracy to restrict people to passing pointers; this was undertaken after the first discovery that passing large arrays recursively could cause crashes. Since then, with the passing of MS-DOS, it has become a non-issue; since all serious machines have virtual memory, you can pass as much data as you want on the stack without detectable problems.

2.6: Someone explained to me that arrays were really just constant pointers.
A: Cool. Someone I know says he saw Elvis in a local bar.

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

A: About the difference between alcohol and marijuana; they have different characteristics, and that's not a problem if you don't mix them too carelessly.

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

A: It was added to allow people to avoid the character constant 'f' which may not be available on some systems. (Actually, it's a side-effect of the equivalence of arrays and pointers.)

3: Memory Allocation

3.1: Why doesn't this fragment work?

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

A: The semicolon after 'answer' is missing.

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

A: You probably returned a pointer to a local array. That doesn't work. Try using a temporary file, instead.

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

A: In interrupt-riddled code, it may be necessary to cast values to force the CPU to resolve pointer types.

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

A: Yes. However, what happens when you do is not clearly defined.

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

A: Interrupt 41h. On Macs, Amigas, and other "big-endian" processors, that would be interrupt 14h; be wary of portability problems.

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

A: Yup. Don't expect an answer though.

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

A: No. You just have to keep track of them somewhere else also.

3.8: Was Proust's masterwork, »A Remembrance of Things Past«, the basis for the C library's allocation scheme, based largely on contextual analysis?

A: The standard does not specify an allocation scheme; the famous author the allocation scheme is based on is implementation specified. Proust is a common choice, however.

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

A: You're probably not freeing the memory completely. Try replacing `free(foo);` with

```
free(foo); free(foo); free(foo);
```

in case the first `free()` frees the memory only partially. (Unix wizards may recognize the parallel with syncing three times before rebooting.)

Alternatively, `free(foo + 4);` may free the remaining four bytes. (Before using this, make sure `realloc(foo, 0)` returned 4).

4: Expressions

4.1: Why doesn't this code: `a[i]=i++;` work?

A: You didn't declare either i or a.

4.2: Under my compiler, the code

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

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

A: No. The only logical answer would be 81 – two postfix ++'s are automatically converted to prefix.

4.3: I've experimented with the code

```
int i = 2;
i = i++;
```

on several compilers. Some gave i the value 2, some gave 3, but one gave 4. I know the behaviour is undefined, but how could it give 4?

A: Because i is 2, the loop is executed twice.

4.4: People keep saying the behaviour is undefined, but I just tried it on an ANSI-conforming compiler, and got the results I expected.

A: They were probably wrong. Flame them mercilessly. Be sure before you do that your compiler is *really* ANSI conforming, though. If it turns out you were wrong, they get a legal claim on your first-born.

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

A: No. To force order of evaluation, you must threaten it. Take the comma operator hostage. Using it, you can force the other operators to do what you want.

4.6: But what about the `&&`, `||`, and comma operators? I see code like

```
if((c = getchar()) == EOF
    || c == '\n') …
```

A: As noted, once you've captured the comma operator, the others become docile.

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

A: `++i`. Only losers and idiots use `i++`. This is different if your native language would idiomatically use "i increment", but in English and related languages, you must use "`++i`". Note that a modern program must use both, dependent on the current locale.

4.8: Why is `i = ++i` undefined?

A: Because it is unclear whether it is shorthand for `i = 42;` or

```
i = (char *) "forty two";
```

Given the ambiguity, the standards committee decided to leave it undefined.

5: ANSI C

5.1: What is the "ANSI C Standard?"

A: A whiny bunch of lousers who haven't written as many books as Herbert Schildt.

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

A: ftp to *ftp.borland.com*.

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

A: A router helps, but your best bet is still the band saw. Quick, efficient, and powerful.

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

A: This is because "3" is not a legal integral constant in C – it's a string constant.

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

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

A: Because you're not using C++.

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

A: One " " character. There are some trivial differences having to do with the distinction between a pointer to a constant, and a constant pointer, but since you can cast either to a `(char *)` it hardly matters.

5.7: Can I declare main as `void`, to shut off these annoying "main returns no value" messages? (I'm calling `exit()`, so main doesn't return.)

A: Certainly. You can also declare it as `double`. It may not compile, or it may crash, but who cares?

5.8: Why does the ANSI Standard not guarantee more than six monocase characters of external identifier significance?

A: Because none of the members of the committee had names over six letters, or in which letters other than the first were capitalized.

5.9: What is the difference between `memcpy` and `memmove`?

A: `memmove` moves memory, and `memcpy` copies it. `memmove` may not be supported on machines without internal robot arms. Do not use `memmove` while the machine is powered up – you can destroy your memory.

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

A: The Frobozz Magic Company lies through its teeth. Consider: does Flood Control Dam #3 actually control floods? Didn't think so. The wands are excellent for making useless via casts of Float, though.

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

A: You're too big and clumsy. When you try to push the numbers together, you lose your balance. Perhaps you should get some angels from the rave over on pin 3.

5.12: What are `#pragmas` and what are they good for?

A: They are useful ways to eliminate compiler features which are not helpful to your goals; contrast `#utility`, which introduces useful compiler features, and `#absolutist`, which introduces those compiler features believed to be right.

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

A: It means that your program will only run once; it's used to create "crippled demos".

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

A: There isn't really one; people just enjoy flaming over nits. (To be technical, one has a hyphen, one has a space, and one is a single word.)

5.15: Is C an acronym?

A: Yes, it stands for "C". It's another of those funky recursive acronyms.

wdv-notes 339
Stand: 1.APR.1995 (2.)
Wiss.Datenverarbeitung
FREIE UNIVERSITÄT BERLIN

Aren't you glad you're not using C ? The ultimate C IAQ.

© 1994–1995 Edited by *Karl-Heinz Dittberner*

Pro

6: C Preprocessor

6.1: How can I use a preprocessor `#if` *expression* to tell if a machine is big-endian or little-endian?
A: `#ifdef __BIG_ENDIAN` should work on all known machines; Borland defines it.

6.2: I've got this tricky processing I want to do at compile time and I can't figure out a way to get cpp to do it.
A: Poor baby.

6.3: How can I list all of the pre#defined identifiers?
A: `#define __ALL_CPP_IDS` – put this in a source file, and run it through your C preprocessor.

6.4: How can I write a cpp macro which takes a variable number of arguments?
A: `#utility varargs define …`

6.5: Shouldn't the following code:

```
#define ROSE 1
#define CHRYSANTHEMUM 2
#define RHODODENDRON 3
#define WATER_LILY 4
printf("%d\n",CHRYSATHNEMUM);
```

print "2"?
A: You misspelled CHRYSANTHEMUM. Use abbreviations for long flower names in C code.

7: Variable-Length Argument Lists

7.1: How can I write a function that takes a variable number of arguments?
A: `#utility varargs int foo()`

7.2: How can I write a function that takes a format string and a variable number of arguments, like `printf`, and passes them to `printf` to do most of the work?
A: Redefine `printf;` the call to "`printf`" inside yours will be resolved to the library version, because the C language doesn't allow recursion.

7.3: How can I discover how many arguments a function was actually called with?
A: `_args` is an external integer constant. It evaluates to three times the number of arguments the current function was called with.
You can then look at `_argdata[args]` to get the address of the last arg, `_argdata[args - 1]` to get the size of the last arg, and `_argdata[args - 2]` to get the type of the last arg (as an int). N.B. You MUST not refer to `_args` or `_argdata` between the `()`'s of a function call; their value will be indeterminate. Use temporary storage.

7.4: Why doesn't

```
printf("hello, ","world!",
'\n');
```

work? I thought `printf()` took a variable number of arguments.
A: It will probably work some of the time; the number of arguments used by `printf()` may vary.

8: Boolean Expressions and Variables

8.1: What is the right type to use for boolean values in C? Why isn't it a standard type? Should #defines or enums be used for the true and false values?
A: `int (*)(int, char **)` makes a good boolean type. You can use "`main`" for true, and "`exit`" for false. On some compilers, you may need to cast `exit()` to an appropriate type.

8.2: Isn't #defining TRUE to be 1 dangerous, since any nonzero value is considered "true" in C? What if a built-in boolean or relational operator "returns" something other than 1?
A: Yes. In fact, my aunt was killed by a true value other than one. However, even more dangerous is defining true to be 0x256. (All kidding aside, folks, the stupidist programming error I ever saw had to do with `#define IS_TRUE(x) ((x) & 0x256)` – it was intended to help the programmer work with Fortran, which had a non-1 true value.)

9: Structs, Enums, and Unions

9.1: What is the difference between an enum and a series of preprocessor #defines?
A: The enum doesn't require the preprocessor.

9.2: I heard that structures could be assigned to variables and passed to and from functions, but K&R I says not.
A: K&R I was wrong; they hadn't actually learned C very well before writing the book. Later, Ritchie got a job at Bell Labs, and worked closely with the authors of C, allowing the 2nd edition of the book [1] to be much more accurate.

9.3: How does struct passing and returning work?
A: The structures are put into the low part of the VGA card's VRAM. They are then removed before the next video update. This is why struct passing was not supported for a long time; VGA cards were prohibitively expensive.

9.4: Why can't you compare structs?
A: Compare them to what? A summer's day?

9.5: How can I read/write structs from/to data files?
A: Loop with `putchar`. Be careful; if your machine uses signed chars by default, all of the sign bits in your structure elements will be reversed.

9.6: How can I determine the byte offset of a field within a structure?
A: It's generally four times the number of members of the structure. It may be more or less on some machines.

9.7: How can I access structure fields by name at run time?
A: `foo."name"` should work. You may need to overload the `.` operator, which, in turn, may overload your C compiler.

9.8: Why does `sizeof` report a larger size than I expect for a structure type, as if there was padding at the end?
A: Because there's padding at the end. »DUH«.

9.9: My compiler is leaving holes in structures, which is wasting space and preventing "binary" I/O to external data files. Can I turn off the padding, or otherwise control the alignment of structs?
A: The holes were left by bullets; your computer has probably been in a serious firefight.

9.10: Can I initialize unions?
A: Depends. They may go on strike when provoked. Luckily, if your program involves air traffic control, the ISO standard guarantees that Ronald Reagan will fire any unions that go on strike, and replace them with structs, which should be close enough.

9.11: How can I pass constant values to routines which accept struct arguments?
A: Try `foo((struct foo) 3)`.

How do you decide which integer type to use?

Flip a coin.
Heads are short, tails are long, and the edge is int.

10: Declarations

10.1: How do you decide which integer type to use?
A: Flip a coin. Heads are `short`, tails are `long`, and the edge is `int`.

10.2: What should the 64-bit type on new, 64-bit machines be?
A: `extern volatile short auto char`.

10.3: If I write the code `int i, j;` can I assume that `(&i + 1) == &j`?
A: Only sometimes. It's not portable, because in EBCDIC, i and j are not adjacent.

10.4: What's the best way to declare and define global variables?
A: In headers; this way, you can get link errors when you include the same header twice.

10.5: What does `extern` mean in a function declaration?
A: It refers to a variable which is not actually in your program.

10.6: I finally figured out the syntax for declaring pointers to functions, but now how do I initialize one?
A: With the assignment operator. You were perhaps expecting a screwdriver?

10.7: I've seen different methods used for calling through pointers to functions. What's the story?
A: There is no story. Nothing to see. Move along.

10.8: What's the auto keyword good for?
A: Declaring vehicles.

11: Stdio

11.1: What's wrong with this code:

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

A: It's stupid. It contains an obvious bug.

11.2: How can I print a '%' character in a printf format string? I tried "\%" but it didn't work.
A: Break the % sign out, i.e.,

```
fprintf("foo " "%" "%d\n", foo);
```

11.3: Why doesn't the code `scanf("%d", i);` work?
A: You probably didn't include `<stdio.h>`

11.4: Once I've used `freopen`, how can I get the original `stdout` (or `stdin`) back?
A: Call `main()` – the environment will be restored.

11.5: Why won't this code work?

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

A: Because the end of file character is not detected on files named "`infp`". (Introverted-iNtuitive-Feeling-Perceptive, that is.) Also, it may be that the file was opened in text mode, where an end of file is read as a capital 'Z' on most machines, and `feof()` only looks for '`control Z`'.

11.6: Why does everyone say not to use `gets()`?
A: Because they're trying to spoil your fun. `gets()` can make an otherwise droll and predictable program a lot more exciting.

11.7: Why does `errno` contain ENOTTY after a call to `printf`?
A: Because `stdout` is not a mammal.

11.8: My program's prompts and intermediate output don't always show up on the screen, especially when I pipe the output through another program.
A: Have you turned your monitor on?

11.9: How can I read one character at a time, without waiting for the RETURN key?
A: Ask the user to press enter after hitting a single character.

11.10: People keep telling me that `getch()` is not standard, but my C compiler has it. Are they wrong?
A: They've been programming more than ten years. You haven't. Draw your own conclusions. That's right! They hadn't noticed it. No doubt their compilers have it too, and its behaviour is identical everywhere else in the world, also. That would explain everything.

11.11: What does it matter that `getch()` isn't standard; it works, doesn't it?
A: Well, that would depend on the definition you're using for "works".

11.12: I tried to port some code from a PC to a Unix machine, and now it crashes immediately on startup. It isn't using `getch()` – it's reading directly from the keyboard. How can this be wrong?
A: This is why we keep telling you nonstandard things don't work; because they don't.

11.13: How can I redirect `stdin` or `stdout` to a file from within a program?
A: `ececlv("main()" "> file", argv);`

11.14: How can I recover the file name given an open file descriptor?
A: `fname(fd)`.

11.15: How do I open Flood Control Dam #3?
A: PUSH THE YELLOW BUTTON.
TURN THE BOLT WITH THE WRENCH.
[You must have the wrench, first!]

12: Library Subroutines

12.1: How can I convert numbers to strings (the opposite of `atoi`)? Is there an `itoa` function?
A: There's frequently an `itoa` function. Better yet, write your own; it'll be good practice. On some implementations, `(char *) x;` will convert x to a string.

12.2: How can I get the current date or time of day in a C program?
A: `fprintf(stderr, "please enter the current time and date…"); fflush(stderr); gets(stdin);`

12.3: I need a random number generator.
A: Count errors in Herbert Schildt's C books [2]. No one has detected any consistent pattern.

12.4: How can I get random integers in a certain range?
A: `random(n)` returns random numbers between n and INT_MAX.

12.5: Each time I run my program, I get the same sequence of numbers back from `rand()`.
A: This is so your results will be reproducible.

12.6: I need a random true/false value, so I'm taking `rand() % 2`, but it's just alternating 0, 1, 0, 1, 0 …
A: That seems pretty random to me.

12.7: I need some code to do regular expression matching.
A: So do I. Let me know if you find some.

12.8: I read through the standard library, but there's no function to multiply two floating point numbers! Help!
A: Many C compilers offer an extension "`mult`" to do just this. If your compiler doesn't, just hang tight; ANSI is likely to add it in the next revision. For now, you can try

```
float mult(float m, n)
{
   float i = 0, j = 0;
   for (i = 0; i < n; ++i)
               j += m;
   return j;
}
```

which is fine as long as n is an integer.

12.9: How do I get past the snake?
A: Release the bird. You will have to drop the rod to get the bird in the cage.

13: Lint

13.1: I just typed in this program, and it's acting strangely. Can you see anything wrong with it?
A: Yes. There's too much lint in it. You should get a shop vac.

13.2: How can I shut off the "warning: possible pointer alignment problem" message LINT gives me for each call to `malloc`?
A: Don't run LINT [4].

13.3: Where can I get an ANSI-compatible LINT?
A: There is an LCLint package, but it depends on a garbage collecting `malloc` that may not run on your system. You may wish to check your spouse's bellybutton occasionally, especially if your spouse works for a standards committee.

13.4: What does LINT stand for, anyway?
A: Laser Interactive Neutopian Technology – it's a kind of virtual reality developed by the readers of *alt.religion.kibology*.

14: Style

14.1: Here's a neat trick:

```
if(!strcmp(s1, s2))
```

Is this good style?
A: Excellent. It's short, it's terse, and experienced programmers will understand it. (True Confessions time: I use this all the time. Oh, well. Habits.)

14.2: Here's an even neater trick:

```
volatile int True_Tester=3;
#define TRUE (!True_Tester
      == !True_Tester)
#define FALSE ((!TRUE) !=
      (!TRUE))
#define STR_DISSIMILAR(x, y)
(strcmp((x), (y)) != FALSE)
```

Isn't this cool?
A: Very impressive. The volatile int type assures that even seemingly redundant calculations involving True_Tester will be performed, making sure that if the compiler's ANSI-compliant values of 0 for false and 1 for true vary during runtime, your program will detect it - and producing meaningful error messages if this change occurs during a boolean computation!
Similarly, the STR_DISSIMILAR macro allows you to make quite clear what the real effects of `strcmp()` are.

14.3: What's the best style for code layout in C?
A: 1TBS.

14.4: Is goto a good thing or a bad thing?
A: Yes.

14.5: No, really, should I use goto statements in my code?
A: Any loop control construct can be written with gotos; similarly, any goto can be emulated by some loop control constructs and additional logic. However, gotos are unclean. For instance, compare the following two code segments:

```
do {                foo();
   foo();           if (bar())
   if (bar())          goto SKIP;
      break;        baz();
   baz();           quux();
   quux();
} while(1==0);     SKIP:
buz();             buz();
```

Note how the loop control makes it quite clear that the statements inside it will be looped on for a while, where the goto statement gives the impression that, if `bar()` returned a nonzero value, the statements `baz()` and `quux()` will be skipped.

14.6: What's this "white space" I keep hearing about?
A: White space is a racist, segregational term. Implicitly, "dark" or "colored" space (i.e., the '_' character) is not good enough to separate tokens. More interestingly, the white space characters keep the other tokens apart. They say it's for parsing, but there's ample evidence the goal of white space is to keep the other characters from "taking over" the program. This is disguised by the description of C as "white space insensitive" – a simple ploy for sympathy.

15: Floating Point

15.1: My floating-point calculations are acting strangely and giving me different answers on different machines.
A: One of the machines has probably an Intel Pentium inside. [Intel has built about 6 million Pentium chips with a FDIV bug inside.] Scrap it and get a real machine.

15.2: I'm trying to do some simple trig, and I am

Aren't you glad you're not using C ?
The ultimate C IAQ.

© 1994–1995 Edited by *Karl-Heinz Dittberner*

#including <math.h>, but I keep getting "undefined: _sin" compilation errors.

A: You forgot to define the sin() function. Most math texts should cover it in some detail. The easiest way to fix this should be:

```
double sin(double x) {
   return sqrt(1 - cos(x) *
         cos(x)); }
```

15.3: Why doesn't C have an exponentiation operator?

A: It does. It looks like the multiplication operator, but you use it more. For instance, the C way of expressing "x squared" is "x*x". "x cubed" would be "x*x*x". Easy, isn't it?

15.4: How do I round numbers?

A: Multiply by 10. »Numerical Recipes in C« has a section on this, but there's reputedly a bug in their algorithm.

15.5: How do I test for IEEE NaN and other special values?

A: Using an electron microscope; the patterns are obvious once you know them.

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

A: Turbo C is notoriously buggy. Get a compiler with floating point support.

15.7: What is so "unsafe" about floating point?

A: Have you tried EXAMINE STICK? The stick has a sharp point, which punctures the raft, which no longer floats. Don't bring the stick into the raft with you.

15.8: Which is larger, "2" or "2.0"?

A: The »Numerical Recipes in C« has a function for comparing two values to see which is greater.
It may have a slight bug, where it would report incorrect results if the numbers differ by less than FLOAT_MAX / INT_MAX.

16: System Dependencies

16.1: How can I read a single character from the keyboard without waiting for a newline?

A: Try `stty eol ^M` to wait for a carriage return.

16.2: How can I find out if there are characters available for reading (and if so, how many)? Alternatively, how can I do a read that will not block if there are no characters available?

A: The buffer is normally at `&main - 0x100`. Lower if you have more than 256 characters of typeahead.

16.3: How can I clear the screen? How can I print things in inverse video?

A: You can clear the screen by sending several formfeed characters. Additionally, some operating systems (like NetBSD) support a feature called "white-outs".

16.4: How do I read the mouse?

A: Flip it over, put on your reading glasses.

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

A: By asking the user.

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

A: Only by force.

16.7: How can I check whether a file exists? I want to query the user before overwriting existing files.

A: Time an attempt to truncate it to zero length; if it takes more than 20–30 ms, the file existed. The exact values will depend on the system and the load; before testing, create several large files and time attempts to truncate them, for calibration.

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

A: There are two good ways:
1. Vernier callipers work well.
2. `mmap()` the file, then use `sizeof()`.

16.9: How can I implement a delay, or time a user's response, with sub-second resolution?

A: Time writes of large files to disks; then you can wait for a certain amount of time by writing a certain amount of data, and time a response by how much you could write before the response arrived.

16.10: How can I read in an object file and jump to routines in it?

A: `fopen` and `goto`.

16.11: How can I invoke an operating system command from within a program?

A: Ask the user to open a new shell. The best way to do this is

```
system("echo Please open a
   new shell now.");
sprintf(cmdstring, "echo
   Enter the command '%s' in
   it.", cmd);
system(cmdstring);
```

This will not work if you haven't declared cmdstring properly.

16.12: How can I ensure objects of my class are always created via "new" rather than as locals or global/static objects?

A: Read the C++ FAQ.

17: Miscellaneous

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

A: They're always zero.

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

A: The traditional solution, pioneered by Microsoft, is to sell enough copies of your proprietary, *slow*, and *limited* software that everyone else supports your formats.

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

A: Using `fcntl()`, lock the line or record in the file exclusively. Now, using another thread, read the file, at each byte, trying to write that byte back. Whenever you succeed, write that byte into another file. Then copy the new file over the old file, releasing the lock first.

17.4: How can I return several values from a function?

A: Code like this ought to work.

```
int int foo() {return 2 3; }
```

Not all compilers will support this feature; complain if your vendor doesn't cover it. A compiler that does can probably be found at *ftp://127.0.0.1/pub/vapor/cc*

17.5: If I have a `char *` variable pointing to the name of a function as a string, how can I call that function?

A: `eval s;`

17.6: I seem to be missing the system header file <math.h>. Can someone send me a copy of this header file?

A: A lot of people claim that it is useless to send people headers from other machines. Not so! It can be informative, and can show you a lot about how blatantly stupid your request was, although it can't show you anything you wouldn't have known in an instant had you thought before posting.
Of course, we'd be happy to send you the header files…

```
------ cut here ------
/* math.h rev 7.0b (3/7/95) */
/* RCS log: #log% - can anyone tell me why this
 * doesn't work?
 * - joe, 2/12/93
 */

/*
 * Copyright 1995 Berserkley Software Systems
 */

/* Parts of this header, including in particular
 * the second and third clauses of the first
 * sentance of the third comment, were based on
 * copyright agreements from other sources,
 * including Xerox corporation.
 */

/*
 * math.h - math related macros and headers
 */

#ifndef _MATH_H
#define _MATH_H

/*
 * global data and definitions
 */

#ifdef __LITERAL_BIBLICAL_FUNDEMENTALISM
#define PI 3.0      /* 1 Kings 7:23 */
#endif

/*
 * common (portable) structures and functions
 */

/*
 * machine specific data
 */

#include <machine/math.h>
#endif /* _MATH_H // prevent multiple inclusion
 by using C++ comments*/
```

```
------ cut here ------
```

(Morons.)

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

A:
```
DO CALL FORTRAN;    fortran();
__LINE__ BASIC;     basic();
sub pascal;         pascal();
(((((lisp)))))      lithp(); *
```

* C is pass by value, of course.

17.8: Does anyone know of a program for converting Pascal or FORTRAN (or LISP, Ada, awk, "Old" C, ...) to C?
A: Nope. However, the psychic friends network may have a lead. And they're not just a psychic, they're a friend.

17.9: Is C++ a superset of C? Can I use a C++ compiler to compile C code?
A: C++ is a superset of something, we're not sure what. You can use a C++ compiler to compile C code, but the results may surprise you.

17.10: Where can I get copies of all these public-domain programs?
A: From *ftp.microsoft.com*. Some of the code may look copyrighted; don't worry! The small companies that wrote it in the first place are not available for comment.

17.11: When will the next International Obfuscated C Code Contest (IOCCC) be held? How can I get a copy of the current and previous winning entries?
A: Next week. You missed the deadline. Tough, sucker.

17.12: Why don't C comments nest? How am I supposed to comment out code containing comments? Are comments legal inside quoted strings?
A: We believe it has something to do with captivity; C comments in the wild mate and nest normally. The San Diego Zoo believes it has managed to convince some C comments to nest, but it's hard to tell how much of that is really in the preprocessor, and how much of it is just bovine fecal matter.

17.13: How can I get the ASCII value corresponding to a character, or vice versa?
A: `chr$(foo);` You would have known this if you had an integer basic in ROM.

17.14: How can I implement sets and/or arrays of bits?
A: With linked lists of bitfields. You may also wish to simply use a large set of constants and some clever use of the switch statement, i.e.:

```
int zero = 0;
int one = 1;
int two = 2;
int three = 3;

int bitwise_or(int n, int m) {
    switch (n) {
    case 3:
            return three;
            break;
    case 2:
            switch (m) {
            case 1:  case 3: return
                    three; break;
            default: return two; break;
            }
            break;
    case 1:
            switch (m) {
            case 2:  case 3: return
                    three; break;
            default: return 1; break;
            }
            break;
    default:  case 0:
            switch (m) {
            case 1:  return one; break;
            case 2:  return two; break;
            case 3:  return three; break;
            default: return zero; break;
            }
            break;
    }
}
```

Obviously, you'll need to increase this slightly to deal with more than two bits. This is much more readable than the alleged "C" solution:

```
int bitwise_or(int n,int m)
{return n|m;}
```

Note how the lack of whitespace around operators obscures the functionality of the code. A clear argument for explicit statement of program logic over arcane operators, if I ever saw one.

17.15: What is the most efficient way to count the number of bits which are set in a value?
A: Start a counter at zero and add one to it for each bit set. Some operating systems may provide a call to do this. For values over INT_MAX/2, start the counter at `CHAR_BIT * sizeof(int)` and subtract one for each bit not set.

17.16: How can I make this code more efficient?
A: Remove the comments; the no-op instructions generated by comments can slow your code down significantly. Similarly, shorten variable names. Most compilers, to implement pass by value, actually pass the names of variables in the stack; shorter variable names will reduce stack usage, and consequently execution time. If your compiler has good loop optimization, replace

```
foo();  with
do {
      foo();
   } while (1 != 1);
```

which will likely receive more optimization.

17.17: Are pointers really faster than arrays? How much do function calls slow things down? Is `++i` faster than `i = i + 1`?
A: Yes. About 10 ms per call. Only on machines which feature pre-increment addressing.

Is C an acronym?
Yes, it stands for »C«.
It's another of those
funky recursive acronyms.

17.18: This program crashes before it even runs! (When single-stepping with a debugger, it dies before the first statement in main.)
A: You probably declared main as `void main(void)`. It's also possible that the first statement in main is `abort();` – by the as if rule, the compiler can abort at any time before then, too. Some compilers have bugs, and will produce buggy code for any module which includes the letters 'a', 'b', 'o', 'r', and 't' in that order before the first function declaration.

17.19: What do "Segmentation violation" and "Bus error" mean?
A: C programs are very territorial, and divide their code into segments. Violating these segments can trigger riots; similarly, pointers and integral constants are at the front of the bus, whereas arrays, strings, and other second-class data types are required to be at the rear of the bus. When they start forgetting their places, you can get a bus error.

17.20: My program is crashing, apparently somewhere down inside `malloc`, but I can't see anything wrong with it.
A: Your vendor's library is buggy; complain loudly. Don't send them any example code; they just ask for that so they can steal your trade secrets.

17.21: Does anyone have a C compiler test suite I can use?
A: Yes. Unfortunately, it's probably broken. It's hard to tell.

17.22: Where can I get a YACC grammar for C?
A: You can't; YACC is written in C.

17.23: I need code to parse and evaluate expressions.
A: Ask any first year CS student. You may also wish to use your C compiler.

17.24: I need a sort of an "approximate" `strcmp` routine, for comparing two strings for close, but not necessarily exact, equality.
A: Just try comparing pointers near the original pointers.

17.25: Will 2000 be a leap year?
A: Yes, you pathetic moron. Duh!

17.26: How do you pronounce "char"?
A: Like the first word of "`char *`". The accent is generally on the first syllable.

17.27: Is this FAQ for real?
A: »Sigh« I knew someone would ask that.

Credits

The original *comp.lang.c* FAQ is maintained by Steve Summit, and many of the questions were stolen from it. Some of the idiotic misconceptions here are original, but many are from other sources. The Zork series may well be trademarked, but it was certainly an excellent game. Some of the mistakes may look similar to things warned against in »C – Traps and Pitfalls« [3]. And, of course, if Dennis Ritchie hadn't written C, these jokes would all make a lot less sense.

Some References [added by the editor]

[1] Kernighan, B. W. and Ritchie, D. M.: The C Programming Language (2nd Edition). Prentice Hall. – ISBN: 0-13-110362-8.

[2] Schildt, H.: C – The Complete Reference. McGraw Hill. – ISBN: 0-07-881538-X.

[3] Koenig, A.: C – Traps and Pitfalls. Redwood City (USA): Addison-Wesley 1989. – ISBN: 0-201-17928-8.

[4] Darwin, I. F.: Checking C Programs with LINT. Sebastopol (USA): O'Reilly 1988. – ISBN: 0-937175-30-7.

[5] Dittberner, K.-H.: Der Problem-Fall »C«. FU Berlin (IfP): wdv-notes Nr. 220, 1991–1994.