Hidden features of C

Asked 16 years, 3 months ago Modified 7 years, 2 months ago

Viewed 93k times

141

votes

Locked. This question and its answers are locked because the question is off-topic but has historical significance. It is not currently accepting new answers or interactions.

I know there is a standard behind all C compiler implementations, so there should be no hidden features. Despite that, I am sure all C developers have hidden/secret tricks they use all the time.

c hidden-features

Share

edited Sep 25, 2017 at 20:52

community wiki
6 revs, 5 users 50%
bernardn

It'd be great if you/someone were to edit the “question” to indicate the pick of the best hidden features, such as in the C#

and Perl versions of this question. – Donal Fellows May 26, 2010 at 13:19

Comments disabled on deleted / locked posts / reviews

56 Answers

Sorted by: Highest score (default)

1 2 Next

115 votes

More of a trick of the GCC compiler, but you can give branch indication hints to the compiler (common in the Linux kernel)

```
#define likely(x)       __builtin_expect((x),1)
#define unlikely(x)     __builtin_expect((x),0)
```

see: http://kerneltrap.org/node/4705

What I like about this is that it also adds some expressiveness to some functions.

```
void foo(int arg)
{
     if (unlikely(arg == 0)) {
           do_this();
           return;
     }
     do_that();
     ...
}
```

Share

answered Sep 25, 2008 at 14:08

community wiki
tonylo

2 This trick is cool... :) Especially with the macros you define. :)
– Sundar R Oct 22, 2008 at 15:23

77
votes

```
int8_t
int16_t
int32_t
uint8_t
uint16_t
uint32_t
```

These are an optional item in the standard, but it must be a hidden feature, because people are constantly redefining them. One code base I've worked on (and still do, for now) has multiple redefinitions, all with different identifiers. Most of the time it's with preprocessor macros:

```
#define INT16 short
#define INT32  long
```

And so on. It makes me want to pull my hair out. ***Just use the freaking standard integer typedefs!***

Share

answered Sep 25, 2008 at 14:31

community wiki
Ben Collins

3 I think they are C99 or so. I haven't found a portable way to ensure these would be around. – akauppi Sep 25, 2008 at 19:25

3 They are an optional part of C99, but I know of no compiler vendors that don't implement this. – Ben Collins Sep 25, 2008 at 21:07

10 stdint.h isn't optional in C99, but following the C99 standard apparently is for some vendors (*cough* Microsoft). – Ben Combee Oct 22, 2008 at 17:54

5 @Pete, if you want to be anal: (1) This thread has nothig to do with any Microsoft product. (2) This thread never had anything to do with C++ at all. (3) There is no such thing as C++ 97. – Ben Collins Mar 1, 2009 at 21:20

5 Have a look at azillionmonkeys.com/qed/pstdint.h -- a close-to-portable stdint.h – gnud Apr 16, 2009 at 14:16

72 votes

The comma operator isn't widely used. It can certainly be abused, but it can also be very useful. This use is the most common one:

```
for (int i=0; i<10; i++, doSomethingElse())
{
  /* whatever */
}
```

But you can use this operator anywhere. Observe:

```
int j = (printf("Assigning variable j\n"), getValueFromS
```

Each statement is evaluated, but the value of the expression will be that of the last statement evaluated.

Share

answered Sep 25, 2008 at 14:37

community wiki
Ben Collins

7 In 20years of C I have NEVER seen that! – Martin Beckett Jun 22, 2009 at 14:55

11 In C++ you can even overload it. – Wouter Lievens Jul 1, 2009 at 10:45

6 can != should, of course. The danger with overloading it is that the built in applies to everything already, including void, so will never fail to compile for lack of available overload. Ie, gives programmer much rope. – Aaron Jul 6, 2009 at 17:45

The int inside the loop will not work with C: it's a C++ improvment. Is the "," the same operation as for (i=0,j=10;i<j; j--, i++) ? – Aif Nov 26, 2010 at 14:00

63 votes

initializing structure to zero

```
struct mystruct a = {0};
```

this will zero all stucture elements.

Share

answered Oct 1, 2008 at 0:34

community wiki
mike511

2 It doesn't zero the padding, if any, however. – Mikeage Mar 1, 2009 at 13:49

2 @simonn, no it doesn't do undefined behavior if the structure contains non-integral types. memset with 0 on the memory of a float/double will still be zero when you interpret the float/double (float/double are designed like that on purpose). – Trevor Boyd Smith Jun 11, 2009 at 13:59

6 @Andrew: `memset` / `calloc` do "all bytes zero" (i.e. physical zeroes), which is indeed not defined for all types. `{ 0 }` is guaranteed to intilaize *everything* with proper *logical* zero values. Pointers, for example, are guranteed to get their proper null values, even if the null-value on the given platform is `0xBAADFOOD`. – AnT stands with Russia Oct 28, 2009 at 10:12

1 @nvl: You get *physical* zero when you just forcefully set all memory occupied by the object to all-bits-zero state. This is what `memset` does (with `0` as second argument). You get *logical* zero when you initialize/assign `0` (or `{ 0 }`) to the object in the source code. These two kinds of zeros do not necessarily produce the same result. As in the example with pointer. When you do `memset` on a pointer, you get a `0x0000` pointer. But when you assign `0` to a pointer, you get *null pointer value*, which at the physical level might be `0xBAADF00D` or anything else. – AnT stands with Russia Feb 26, 2010 at 16:29

3 @nvl: Well, in practice the difference is often only conceptual. But in theory, virtually any type can have it. For example, `double`. Usually it is implemented in accordance with IEEE-

754 standard, in which the logical zero and physical zero are the same. But IEEE-754 is not required by the language. So it might happen that when you do `double d = 0;` (logical zero), physically some bits in memory occupied by `d` will not be zero. – AnT stands with Russia Feb 26, 2010 at 19:17

62
votes

✓

Function pointers. You can use a table of function pointers to implement, e.g., fast indirect-threaded code interpreters (FORTH) or byte-code dispatchers, or to simulate OO-like virtual methods.

Then there are hidden gems in the standard library, such as qsort(),bsearch(), strpbrk(), strcspn() [the latter two being useful for implementing a strtok() replacement].

A misfeature of C is that signed arithmetic overflow is undefined behavior (UB). So whenever you see an expression such as x+y, both being signed ints, it might potentially overflow and cause UB.

Share

answered Sep 25, 2008 at 10:29

community wiki
zvrba

29 But if they had specified behaviour on overflow, it would have made it very slow on architectures where that was not the normal behaviour. Very low runtime overhead has always been a design goal of C, and that has meant that a lot of things like this are undefined. – Mark Baker Oct 17, 2008 at 8:38

9 I'm very well aware of *why* overflow is UB. It is still a misfeature, because the standard should have at least provided library routines that can test for arithmetic overflow (of all basic operations) w/o causing UB. – zvrba Jan 20, 2009 at 20:51

2 @zvrba, "library routines that can test for arithmetic overflow (of all basic operations)" if you had added this then you would have incurred significant performance hit for any integer arithmetic operations. ===== Case study Matlab specifically ADDS the feature of controlling integer overflow behavior to wrapping or saturate. And it also throws an exception whenever overflow occurs ==> Performance of Matlab integer operations: VERY SLOW. My own conclusion: I think Matlab is a compelling case study that shows why you don't want integer overflow checking. – Trevor Boyd Smith Jun 11, 2009 at 13:35

15 I said that the standard should have provided *library* support for checking for arithmetic overflow. Now, how can a library routine incur a performance hit if you never use it? – zvrba Jun 12, 2009 at 18:52

5 A big negative is that GCC does not have a flag to catch signed integer overflows and throw a runtime exception. While there are x86 flags for detecting such cases, GCC does not utilize them. Having such a flag would allow non-performance-critical (especially legacy) applications the benefit of security with minimal to no code review and refactoring. – Andrew Keeton Jun 22, 2009 at 0:23

52

votes

Multi-character constants:

```
int x = 'ABCD';
```

This sets `x` to `0x41424344` (or `0x44434241`, depending on architecture).

EDIT: This technique is not portable, especially if you serialize the int. However, it can be extremely useful to create self-documenting enums. e.g.

```
enum state {
    stopped = 'STOP',
    running = 'RUN!',
    waiting = 'WAIT',
};
```

This makes it much simpler if you're looking at a raw memory dump and need to determine the value of an enum without having to look it up.

Share

edited Sep 10, 2011 at 20:27

community wiki
5 revs, 3 users 88%
Ferruccio

I'm pretty sure this is not a portable construct. The result of creating a multi-character constant is implementation-defined.
– Mark Bessey Oct 17, 2008 at 4:18

8 The "not portable" comments miss the point entirely. It is like criticizing a program for using INT_MAX just because INT_MAX is "not portable" :) This feature is as portable as it needs to be. Multi-char constant is an extremely useful feature that provides readable way to for generating unique integer IDs.
– AnT stands with Russia Oct 28, 2009 at 10:36

1 @Chris Lutz - I'm pretty sure the trailing comma goes all the way back to K&R. It's described in the second edition (1988).
– Ferruccio Oct 28, 2009 at 11:15

1 @Ferruccio: You must be thinking about the trailing comma in the aggregate initailizer lists. As for the trailing comma in enum declarations - it's a recent addition, C99.
– AnT stands with Russia Oct 28, 2009 at 17:59

3 You forgot 'HANG' or 'BSOD' :-) – JBRWilkinson Nov 2, 2009 at 16:30

44 votes

I never used bit fields but they sound cool for ultra-low-level stuff.

```
struct cat {
    unsigned int legs:3;  // 3 bits for legs (0-4 fit ir
    unsigned int lives:4; // 4 bits for lives (0-9 fit i
    // ...
};

cat make_cat()
{
    cat kitty;
    kitty.legs = 4;
    kitty.lives = 9;
    return kitty;
}
```

This means that `sizeof(cat)` can be as small as `sizeof(char)`.

Incorporated comments by Aaron and leppie, thanks guys.

Share

edited May 23, 2017 at 12:17

community wiki
4 revs
Motti

The combination of structs and unions is even more interesting - on embedded systems or low level driver code. An example is when you like to parse the registers of an SD card, you can read it in using union (1) and read it out using union (2) which is a struct of bitfields. – ComSubVie Sep 25, 2008 at 11:09

5 Bitfields are not portable -- the compiler can choose freely whether, in your example, legs will be allocated the most significant 3 bits, or the least significant 3 bits. – zvrba Sep 25, 2008 at 14:25

3 Bitfields are an example of where the standard gives implementations so much freedom in how they're inplemented, that in practice, they're nearly useless. If you care how many bits a value takes up, and how it's stored, you're better off using bitmasks. – Mark Bessey Oct 17, 2008 at 4:13

26 Bitfields are indeed portable as long as you treat them as the structure elements they are, and not "pieces of integers." Size, not location, matters in an embedded system with limited

memory, as each bit is precious ... but most of today's coders are too young to remember that. :-) – Adam Liss Oct 26, 2008 at 0:41

5 @Adam: location may well matter in an embedded system (or elsewhere), if you are depending on the position of the bitfield within its byte. Using masks removes any ambiguity. Similarly for unions. – Steve Melnikoff Mar 1, 2009 at 22:51

37 votes

C has a standard but not all C compilers are fully compliant (I've not seen any fully compliant C99 compiler yet!).

That said, the tricks I prefer are those that are non-obvious and portable across platforms as they rely on the C semantic. They usually are about macros or bit arithmetic.

For example: swapping two unsigned integer without using a temporary variable:

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

or "extending C" to represent finite state machines like:

```
FSM {
  STATE(x) {
    ...
    NEXTSTATE(y);
  }

  STATE(y) {
    ...
    if (x == 0)
      NEXTSTATE(y);
      else
        NEXTSTATE(x);
    }
  }
```

that can be achieved with the following macros:

```
#define FSM
#define STATE(x)      s_##x :
#define NEXTSTATE(x)  goto s_##x
```

In general, though, I don't like the tricks that are clever but make the code unnecessarily complicated to read (as the swap example) and I love the ones that make the code clearer and directly conveying the intention (like the FSM example).

Share

answered Sep 25, 2008 at 9:20

community wiki
Remo.D

18 C supports chaining, so you can do a ^= b ^= a ^= b; – OJ. Sep 25, 2008 at 10:06

4 Strictly speaking, the state example is a tick of the preprocessor, and not the C language - it is possible to use the former without the latter. – Greg Whitfield Sep 25, 2008 at 13:10

15 OJ: actually what you suggest is undefined behavior because of sequence point rules. It may work on most compilers, but is not correct or portable. – Evan Teran Sep 25, 2008 at 14:14

5 Xor swap could actually be less efficient in the case of a free register. Any decent optimizer would make the temp variable be a register. Depending on implementation (and need for parallelism support) the swap might actually use real memory instead of a register (which would be the same).
– Paul de Vrieze Oct 17, 2008 at 9:49

27 please don't ever actually do this: en.wikipedia.org/wiki/…
– Christian Oudard Jan 2, 2009 at 18:55

37 votes

Interlacing structures like Duff's Device:

```
strncpy(to, from, count)
char *to, *from;
int count;
{
    int n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
               } while (--n > 0);
    }
}
```

Share

edited Oct 14, 2010 at 7:28

community wiki
6 revs, 4 users 88%
ComSubVie

29 @ComSubVie, anyone who uses Duff's Device is a script kiddy who saw Duff's Device and thought their code would look 1337 if they used Duff's Device. (1.) Duff's Device doesn't offer any performance increases on modern processor because modern processors have zero-overhead-looping. In other words it is an obsolete piece of code. (2.) Even if your processor doesn't offer zero-overhead-looping, it will probably have something like SSE/altivec/vector-processing which will put your Duff's Device to shame when you use memcpy(). (3.) Did I mention that other that doing memcpy() duff's is not useful?
– Trevor Boyd Smith Jun 11, 2009 at 13:50

2 @ComSubVie, please meet my Fist-of-death (en.wikipedia.org/wiki/...) – Trevor Boyd Smith Jun 11, 2009 at 13:52

12 @Trevor: so only script kiddies program 8051 and PIC microcontrollers, right? – SF. Feb 19, 2010 at 11:13

6 @Trevor Boyd Smith : While the Duff's Device appears outdated, it's still an historical curiosity, which validates ComSubVie's answer. Anyway, quoting Wikipedia : *"When numerous instances of Duff's device were removed from the XFree86 Server in version 4.0, there was a notable improvement in performance."*... – paercebal May 8, 2010 at 9:01

2 On Symbian, we once evaluated various loops for fast pixel coding; the duff's device, in assembler, was the fastest. So it still had relevance on the mainstream ARM cores on your smartphones today. – Will Oct 25, 2010 at 8:37

33 votes

I'm very fond of designated initializers, added in C99 (and supported in gcc for a long time):

```
#define FOO 16
#define BAR 3

myStructType_t myStuff[] = {
    [FOO] = { foo1, foo2, foo3 },
    [BAR] = { bar1, bar2, bar3 },
    ...
```

The array initialization is no longer position dependent. If you change the values of FOO or BAR, the array initialization will automatically correspond to their new value.

Share

answered Sep 25, 2008 at 14:15

community wiki
DGentry

The syntax gcc has supported for a long time is not the same as the standard C99 syntax. – Mark Baker Oct 17, 2008 at 8:36

28
votes

C99 has some awesome any-order structure initialization.

```
struct foo{
  int x;
  int y;
  char* name;
};

void main(){
   struct foo f = { .y = 23, .name = "awesome", .x = -38
}
```

Share

edited Jan 19, 2012 at 15:20

community wiki
2 revs, 2 users 83%
Jason

27
votes

anonymous structures and arrays is my favourite one. (cf. http://www.run.montefiore.ulg.ac.be/~martin/resources/kung-f00.html)

```
setsockopt(yourSocket, SOL_SOCKET, SO_REUSEADDR, (int[])
```

or

```
void myFunction(type* values) {
    while(*values) x=*values++;
}
myFunction((type[]){val1,val2,val3,val4,0});
```

it can even be used to instanciate linked lists...

Share

answered Sep 25, 2008 at 10:16

community wiki
PypeBros

3 This feature is usually called "compound literals". Anonymous (or unnamed) structures designate nested structures that have no member names. – calandoa Jun 22, 2009 at 11:47

according to my GCC, "ISO C90 forbids compound literals". – jmtd Jun 22, 2009 at 15:55

"ISO C99 supports compound literals." "As an extension, GCC supports compound literals in C89 mode and in C++" (dixit info gcc). Plus, "As a GNU extension, GCC allows initialization of objects with static storage duration by compound literals (which is not possible in ISO C99, because the initializer is not a constant)." – PypeBros Feb 3, 2011 at 17:31

24 votes

gcc has a number of extensions to the C language that I enjoy, which can be found here. Some of my favorites are function attributes. One extremely useful example is the format attribute. This can be used if you define a custom function that takes a printf format string. If you enable this function attribute, gcc will do checks on your arguments to ensure that your format string and arguments match up and will generate warnings or errors as appropriate.

```
int my_printf (void *my_object, const char *my_format, .
            __attribute__ ((format (printf, 2, 3)));
```

Share

answered Sep 29, 2008 at 0:27

community wiki
Russell Bryant

24
votes

the (hidden) feature that "shocked" me when I first saw is about printf. this feature allows you to use variables for formatting format specifiers themselves. look for the code, you will see better:

```
#include <stdio.h>

int main() {
    int a = 3;
    float b = 6.412355;
    printf("%.*f\n",a,b);
    return 0;
}
```

the * character achieves this effect.

Share

answered Jun 22, 2009 at 11:10

community wiki
kolistivra

24
votes

Well... I think that one of the strong points of C language is its portability and standardness, so whenever I find some "hidden trick" in the implementation I am currently using, I try not to use it because I try to keep my C code as standard and portable as possible.

Share

edited Nov 3, 2009 at 15:00

community wiki
2 revs, 2 users 67%
Giacomo Degli Esposti

But in reality, how often do you have to compile your code with another compiler? – Joe D Jun 7, 2010 at 18:26

3 @Joe D if its a cross platform project like Windows/OSX/Linux, probably a bit, and also there's different arch such as x86 vs x86_64 and etc... – Pharaun Nov 11, 2010 at 17:13

@JoeD Unless you're in a very narrowminded project that's happy to marry one compiler vendor, very. You might want to avoid actually having to switch compilers, but you do want to keep that option open. With embedded systems, you don't always get a choice, though. AHS, ASS. – XTL Feb 17, 2012 at 7:34

19 votes

Compile-time assertions, as already discussed here.

```
//--- size of static_assertion array is negative if cond
#define STATIC_ASSERT(condition) \
    typedef struct { \
        char static_assertion[condition ? 1 : -1]; \
    } static_assertion_t

//--- ensure structure fits in
STATIC_ASSERT(sizeof(mystruct_t) <= 4096);
```

Share

edited May 23, 2017 at 12:09

community wiki
2 revs
philant

16
votes

Constant string concatenation

I was quite surprised not seeing it allready in the answers, as all compilers I know of support it, but many programmers seems to ignore it. Sometimes it's really handy and not only when writing macros.

Use case I have in my current code: I have a `#define PATH "/some/path/"` in a configuration file (really it is setted by the makefile). Now I want to build the full path including filenames to open ressources. It just goes to:

```
fd = open(PATH "/file", flags);
```

Instead of the horrible, but very common:

```
char buffer[256];
snprintf(buffer, 256, "%s/file", PATH);
fd = open(buffer, flags);
```

Notice that the common horrible solution is:

- three times as long
- much less easy to read
- much slower

- less powerfull at it set to an arbitrary buffer size limit (but you would have to use even longer code to avoid that without constant strings contatenation).
- use more stack space

Share

answered May 8, 2010 at 8:22

community wiki
kriss

1 It is also useful to split a string constant on multiple source lines without using dirty `\`. – dolmen Mar 28, 2011 at 21:35

15 votes

Well, I've never used it, and I'm not sure whether I'd ever recommend it to anyone, but I feel this question would be incomplete without a mention of Simon Tatham's co-routine trick.

Share

answered Oct 17, 2008 at 8:54

community wiki
Mark Baker

12 votes

When initializing arrays or enums, you can put a comma after the last item in the initializer list. e.g:

```
int x[] = { 1, 2, 3, };

enum foo { bar, baz, boom, };
```

This was done so that if you're generating code automatically you don't need to worry about eliminating the last comma.

Share

answered Jun 11, 2009 at 11:14

community wiki
Ferruccio

This is also important in a multi-developer environment where, for instance, Eric adds in "baz," and then George adds in "boom,". If Eric decides to pull his code out for the next project build, it still compiles with George's change. Very important for multi-branch source code control and overlapping development schedules. – Harold Bamford Jun 30, 2009 at 21:50

Enums may be C99. Array initializers & the trailing comma are K&R. – Ferruccio Nov 24, 2009 at 11:27

Plain enums were in c89, AFAIK. At least they've been around for ages. – XTL Feb 17, 2012 at 7:48

12
votes

Struct assignment is cool. Many people don't seem to realize that structs are values too, and can be assigned around, there is no need to use `memcpy()`, when a simple assignment does the trick.

For example, consider some imaginary 2D graphics library, it might define a type to represent an (integer) screen coordinate:

```
typedef struct {
   int x;
   int y;
} Point;
```

Now, you do things that might look "wrong", like write a function that creates a point initialized from function arguments, and returns it, like so:

```
Point point_new(int x, int y)
{
  Point p;
  p.x = x;
  p.y = y;
  return p;
}
```

This is safe, as long (of course) as the return value is copied by value using struct assignment:

```
Point origin;
origin = point_new(0, 0);
```

In this way you can write quite clean and object-oriented-ish code, all in plain standard C.

Share edited Oct 25, 2010 at 8:32

community wiki
2 revs
unwind

4 Of course, there are performance implications to passing round large structs in this way; it's often useful (and is indeed something a lot of people don't realise you can do) but you need to consider whether passing pointers is better.
– Mark Baker Oct 17, 2008 at 8:47

1 Of course, there *might* be. Ít's also quite possible for the compiler to detect the usage and optimize it. – unwind Oct 17, 2008 at 8:57

Be careful if any of the elements are pointers, as you'll be copying the pointers themselves, not their contents. Of course, the same is true if you use memcpy(). – Adam Liss Oct 26, 2008 at 0:27

The compiler can't optimize this converting by-value passing with by-referenece, unless it can do global optimizations.
– Blaisorblade Jan 19, 2009 at 0:20

It's probably worth noting that in C++ the standard specifically allows optimizing away the copy (the standard has to allow for it for compilers to implement it because it means the copy constructor which may have side effects may not be called), and since most C++ compilers are also C compilers, there's a good chance your compiler does do this optimization.
– Joseph Garvin Feb 21, 2010 at 17:05

10 votes

Strange vector indexing:

```
int v[100]; int index = 10;
/* v[index] it's the same thing as index[v] */
```

Share

answered Sep 25, 2008 at 9:30

community wiki
INS

4 It's even better... char c = 2["Hello"]; (c == 'l' after this). – yrp Sep 25, 2008 at 10:20

5 Not so strange when you consider that v[index] == *(v + index) and index[v] == *(index + v) – Ferruccio Sep 25, 2008 at 14:36

17 Please tell me you don't actually use this "all the time", like the question asks! – Tryke Sep 25, 2008 at 20:52

9 votes

C compilers implement one of several standards. However, having a standard does not mean that all aspects of the language are defined. Duff's device, for example, is a favorite 'hidden' feature that has become so popular that modern compilers have special purpose recognition code to ensure that optimization techniques do not clobber the desired effect of this often used pattern.

In general hidden features or language tricks are discouraged as you are running on the razor edge of whichever C standard(s) your compiler uses. Many such tricks do not work from one compiler to another, and often these kinds of features will fail from one version of a compiler suite by a given manufacturer to another version.

Various tricks that have broken C code include:

1. Relying on how the compiler lays out structs in memory.
2. Assumptions on *endianness* of integers/floats.
3. Assumptions on function ABIs.
4. Assumptions on the direction that stack frames grow.
5. Assumptions about order of execution within statements.
6. Assumptions about order of execution of statements in function arguments.
7. Assumptions on the bit size or precision of short, int, long, float and double types.

Other problems and issues that arise whenever programmers make assumptions about execution models that are all specified in most C standards as 'compiler dependent' behavior.

Share

answered Sep 25, 2008 at 9:26

community wiki
Kevin S.

To solve most of those, make those assumptions dependant on the characteristics of your platform, and describe each platform in his own header. Order execution is an exception - never rely on that; on the other ideas, each platform needs having a reliable decision. – Blaisorblade Jan 19, 2009 at 0:17

---

2 @Blaisorblade, Even better, use compile-time assertions to document your assumptions in a way that will make the compile fail on a platform where they are violated. – RBerteig Aug 1, 2010 at 0:06

I think one should combine both, so that your code works on multiple platforms (that was the original intention), and if the feature macros are set the wrong way, compile-time assertions will catch it. I'm not sure if, say, assumption on function ABIs are checkable as compile-time assertions, but it should be possible for most of the other (valid) ones (except order of execution ;-)). – Blaisorblade Aug 2, 2010 at 13:01

Function ABI checks should be handled by a test suite. – dolmen Mar 28, 2011 at 21:30

9 votes

When using sscanf you can use %n to find out where you should continue to read:

```
sscanf ( string, "%d%n", &number, &length );
string += length;
```

Apparently, you can't add another answer, so I'll include a second one here, you can use "&&" and "||" as conditionals:

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

int main()
{
   1 || puts("Hello\n");
   0 || puts("Hi\n");
   1 && puts("ROFL\n");
   0 && puts("LOL\n");
```

```
    exit( 0 );
}
```

This code will output:

```
Hi
ROFL
```

Share

edited May 26, 2011 at 2:20

community wiki
3 revs, 2 users 94%
onemasse

8
votes

using INT(3) to set break point at the code is my all time favorite

Share

answered Sep 25, 2008 at 9:06

community wiki
Dror Helper

3 I don't think it's portable. It will work on x86, but what about other platforms? – Cristian Ciupitu Sep 25, 2008 at 19:25

1 I have no idea - You should post a question about it – Dror Helper Dec 8, 2008 at 16:48

2 It's a good technique and it is X86 specific (although there are probably similar techniques on other platforms). However, this

is not a feature of C. It depends on non-standard C extensions or library calls. – Ferruccio Jun 22, 2009 at 12:28

1 In GCC there is __builtin_trap and for MSVC __debugbreak which will work on any supported architecture. – Axel Gneiting Jun 7, 2010 at 19:05

8 votes

My favorite "hidden" feature of C, is the usage of %n in printf to write back to the stack. Normally printf pops the parameter values from the stack based on the format string, but %n can write them back.

Check out section 3.4.2 here. Can lead to a lot of nasty vulnerabilities.

Share

answered Sep 26, 2008 at 20:18

community wiki
Sridhar Iyer

the link is not working anymore, in fact the site itself seems is not working. Can you provide another link? – thequark Jan 7, 2011 at 4:39

@thequark: Any article on "format string vulnerabilities" will have some info in it.. (e.g. crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf) .. However due to the nature of the field, the security websites themselves are a bit flaky and real academic articles are hard to come by (with implementation). – Sridhar Iyer Jan 7, 2011 at 7:08

8
votes

Compile-time assumption-checking using enums: Stupid example, but can be really useful for libraries with compile-time configurable constants.

```
#define D 1
#define DD 2

enum CompileTimeCheck
{
    MAKE_SURE_DD_IS_TWICE_D = 1/(2*(D) == (DD)),
    MAKE_SURE_DD_IS_POW2    = 1/((((DD) - 1) & (DD)) ==
};
```

Share

answered Nov 11, 2009 at 13:42

community wiki
S.C. Madsen

2 +1 Neat. I used to use the CompilerAssert macro from Microsoft, but yours is not bad either. (`#define CompilerAssert(exp) extern char _CompilerAssert[(exp)?1:-1]`) – Patrick Schlüter Nov 27, 2010 at 12:52

1 I like the enumeration method. The approach I'd used before took advantage of dead code elimination: "if (something_bad) {void BLORG_IS_WOOZLED(void); BLORG_IS_WOOZLED();}" which didn't error until link time, though it did offer the advantage of letting the programmer know via error message that the blorg was woozled. – supercat Aug 25, 2011 at 21:28

8
votes

Gcc (c) has some fun features you can enable, such as nested function declarations, and the a?:b form of the ?: operator, which returns a if a is not false.

Share

edited May 26, 2011 at 2:32

community wiki
2 revs, 2 users 80%
Alex Brown

8
votes

I discoverd recently 0 bitfields.

```
struct {
  int    a:3;
  int    b:2;
  int     :0;
  int    c:4;
  int    d:3;
};
```

which will give a layout of

```
000aaabb 0ccccddd
```

instead of without the :0;

```
0000aaab bccccddd
```

The 0 width field tells that the following bitfields should be set on the next atomic entity (`char`)

Share

edited Sep 10, 2011 at 20:58

community wiki
2 revs
tristopia

7
votes

C99-style variable argument macros, aka

```
#define ERR(name, fmt, ...)   fprintf(stderr, "ERROR " #
                                  __VAR_ARGS__)
```

which would be used like

```
ERR(errCantOpen, "File %s cannot be opened", filename);
```

Here I also use the stringize operator and string constant concatentation, other features I really like.

Share

answered Oct 22, 2008 at 18:00

community wiki
Ben Combee

You have an extra 'R' in **VA_ARGS**. – Blaisorblade Jan 19, 2009 at 0:24

6
votes

Variable size automatic variables are also useful in some cases. These were added i nC99 and have been supported in gcc for a long time.

```
void foo(uint32_t extraPadding) {
    uint8_t commBuffer[sizeof(myProtocol_t) + extraPaddi
```

You end up with a buffer on the stack with room for the fixed-size protocol header plus variable size data. You can get the same effect with alloca(), but this syntax is more compact.

You have to make sure extraPadding is a reasonable value before calling this routine, or you end up blowing the stack. You'd have to sanity check the arguments before calling malloc or any other memory allocation technique, so this isn't really unusual.

Share

answered Sep 25, 2008 at 14:23

community wiki
DGentry

Will this also work correctly if a byte/char is not exactly 8 bits wide on the target platform? I know, those cases are rare, but still... :) – Stephan202 Apr 26, 2009 at 10:26