Should Local Variable Initialisation Be Mandatory? [closed]

Asked 16 years, 2 months ago Modified 2 months ago Viewed 3k times

11

Closed. This question is opinion-based. It is not currently accepting answers.

Want to improve this question? Update the question so it can be answered with facts and citations by editing this post.

Closed 2 months ago.

Improve this question

The maintenance problems that uninitialised locals cause (particularly pointers) will be obvious to anyone who has done a bit of c/c++ maintenance or enhancement, but I still see them and occasionally hear performance implications given as their justification.

It's easy to demonstrate in c that redundant initialisation is optimised out:

```
$ less test.c
#include <stdio.h>
main()
{
#ifdef INIT_LOC
    int a = 33;
    int b;
    memset(&b,66,sizeof(b));
#else
    int a;
    int b;
#endif
    a = 0;
    b = 0;
    printf ("a = %i, b = %i\n", a, b);
}

$ gcc --version
gcc (GCC) 3.4.4 (cygming special, gdc 0.12, using dmd 0.125)
```

[Not Optimised:]

```
$ gcc test.c -S -o no_init.s; gcc test.c -S -D INIT_LOC=1 -o init.s; diff no_in
it.s init.s
22a23,28
>       movl    $33, -4(%ebp)
```

```
>       movl    $4, 8(%esp)
>       movl    $66, 4(%esp)
>       leal    -8(%ebp), %eax
>       movl    %eax, (%esp)
>       call    _memset
33a40
>       .def    _memset;        .scl    3;      .type   32;     .endef
```

[Optimised:]

```
$ gcc test.c -O -S -o no_init.s; gcc test.c -O -S -D INIT_LOC=1 -o init.s; diff
 no_init.s init.s
$
```

So WRT performance under what circumstances is mandatory variable initialisation NOT a good idea?

IF applicable, no need to restrict answers to c/c++ but please be clear about the language/environment (and reproducible evidence much preferred over speculation!)

c++ **c** **performance** **optimization**

Share

Improve this question

Follow

edited Nov 12, 2008 at 3:26

Robert Gamble

109k ● 25 ● 147 ● 138

asked Sep 26, 2008 at 14:02

schmick

119 ● 1 ● 4

I would have preferred if if C++ required the user to explicitly declare a variable uninitialized (e.g. `uninitialized bool myVar;` or perhaps `bool myVar = uninitialized;`) and made zero-initialization the default behavior; but it's too late to change it now. – Jeremy Friesner Oct 7 at 13:26

@JeremyFriesner: In both C and C++, the current behavior is undefined. Defining that is always possible, because you cannot break programs with undefined behavior. The main technical challenge is the required syntax - both C and C++ dislike adding new keywords. `= void` would work for both. – MSalters Oct 7 at 13:36

18 Answers

Sorted by: Highest score (default)

11

Short answer: declare the variable as close to first use as possible and initialize to "zero" if you still need to.

Long answer: If you declare a variable at the start of a function, and don't use it until later, you should reconsider your placement of the variable to as local a scope as possible. You can then usually assign to it the needed value right away.

If you must declare it uninitialized because it gets assigned in a conditional, or passed by reference and assigned to, initializing it to a null-equivalent value is a good idea. The compiler can sometimes save you if you compile under -Wall, as it will warn if you read from a variable before initializing it. However, it fails to warn you if you pass it to a function.

If you play it safe and set it to a null-equivalent, you have done no harm if the function you pass it to overwrites it. If, however, the function you pass it to uses the value, you can pretty much be guaranteed failing an assert (if you have one), or at least segfaulting the second you use a null object. Random initialization can do all sorts of bad things, including "work".

Share

Improve this answer

Follow

edited Sep 26, 2008 at 15:16

answered Sep 26, 2008 at 14:10

hazzen

17.5k ● 7 ● 43 ● 33

Sometimes you need to pass a reference to function to be assigned to. In this case the variable has to be declared first. – Dima Sep 26, 2008 at 14:12

common case of a variable declared before a loop/conditional structure so you can set the value in the 'loop' and keep the value afterwards – ShoeLace Sep 26, 2008 at 14:12

you only get the warning with -Wall if you read from a variable that you never have written to. That's different to "initialize just to be sure"-style coding. – Nils Pipenbrinck Sep 26, 2008 at 14:14

Thanks for the comments; I've fixed it up. Should have tested the -Wall :(– hazzen Sep 26, 2008 at 15:17

@ShoeLace: So if the loop doesn't iterate at all, what is the value of your variable? I would say that is exactly the kind of thing that "initialize always" helps to prevent/highlight. – Richard Corden Sep 26, 2008 at 15:22

6

If you think that an initialization is redundant, it is. My goal is to write code that is as humanly readable as possible. Unnecessary initialization confuses future reader.

C compilers are getting pretty good at catching usage of unitialized variables, so the danger of that is now minimal.

Don't forget, by making "fake" initialization, you trade one danger - crashing on using garbage (which leads to a bug that is very easy to find and fix) on another - program taking wrong action based on fake value (which leads to a bug that is very difficult to find). The choice depends on the application. For some, it is critical never to crash. For majority, it is better to catch the bug ASAP.

Share

edited Sep 26, 2008 at 15:26

answered Sep 26, 2008 at 14:40

Improve this answer

Follow

buti-oxa
11.4k ● 5 ● 36 ● 44

Fake values can be hard to spot if you can't tell they're fake. For example, a pointer initialized to zero leads to a nice fail-fast situation if you dereference it. But a non-initialized pointer may point to memory that you can sometimes use, leading to confusing crashes from time to time. – Mr. Shiny and New 安宇 Sep 26, 2008 at 15:34

1 Actually, pointer initialized to zero (NULL), often means something. You are better off initializing with something like 0xbaadf00d – buti-oxa Sep 26, 2008 at 22:58

@B Not sure this is a true trade off, in my experience tracking down a bug caused by an undefined garbage variable is MUCH harder than tracking one down caused by a consistently reproducible incorrect initialisation. – schmick Sep 28, 2008 at 13:51

I like to use the IBM xlc compiler "initauto" option to help with this. For debug builds, I have all automatic variables initialized to some byte pattern that's easy to spot. It's then turned off for release builds for better performance. – Anthony Giorgio Sep 16, 2009 at 14:49

Schmick is right, “garbage using” bugs are often difficult to debug. I was talking about discovering the bug in testing, not locating the problem during debugging. Too often, “incorrect initialization” bugs are not found until the product ships. – buti-oxa Sep 16, 2009 at 18:15

5

This is a great example of **Premature optimization is the root of all evil**

The full quote is:

> There is no doubt that the grail of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a *strong negative impact when debugging and maintenance are considered*. **We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.** Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only after that code has been identified.

This came from Donald Knuth. who are you going to believe...your colleagues or Knuth?
I know where my money is...

To get back to the original question: "Should we MANDATE initialization?"
I would phrase it as so:

> Variables **should** be initialize, except in situation where it can be demonstrated there is a *significant* performance gain to be realized by not

> initializing. Come armed with hard numbers...

Share

Improve this answer

Follow

edited Sep 28, 2008 at 15:14

answered Sep 26, 2008 at 14:19

Benoit

38.9k ● 24 ● 85 ● 117

Donald Knuth scoffs at parallel progamming, and multi threading. informit.com/articles/article.aspx?p=1193856 – J.J. Sep 26, 2008 at 14:24

I hate it when people only quote half of the sentence and take it out of context btw... – Nils Pipenbrinck Sep 26, 2008 at 14:26

Doesn't invalidate his point...I don't think initializing a var is the critical 3% – Benoit Sep 26, 2008 at 14:28

1 Performance isn't really the problem here, it's usage. You don't want to use (read) a variable that hasn't been initialized yet. – Herms Sep 26, 2008 at 14:35

@JJ Interesting article Thanks! I think scoffs is a bit strong...healthily dubious might be better :) Note that he admits his ignorance on the subject and says don't listen to me... – Benoit Sep 26, 2008 at 14:39

4

I'm not sure if it is necessary to "make them mandatory", but I personally think it is always better to initialize variables. If the purpose of the application is to be as tight as possible then C/C++ is open for that purpose. However, I think many of us have been burned a time or two by not initializing a variable and assuming it contains a valid value (e.g. pointer) when it really doesn't. A pointer with an address of zero is much easier to check for than if it has random garbage from the last memory contents at that particular location. I think in most cases, it is no longer a matter of performance but a matter of clarity and safety.

Share Improve this answer Follow

answered Sep 26, 2008 at 14:09

Jordan Parmer

37.2k ● 30 ● 99 ● 120

4

Let me tell you a story about a product I worked on in 1992 and later that, for the purposes of this story, we'll call Stackrobat. I was assigned a bug that caused the application to crash on the Mac, but not on Windows, oh and the bug was not reproducible reliably. It took QA the better part of a week to come up with a recipe that worked maybe 1 in 10 times.

It was hell tracking down the root cause since the actual crash happened well after the action that did it.

Ultimately, I tracked it down by writing a custom code profiler for the compiler. The compiler would quite happily inject calls to global prof_begin() and prof_end() functions and you were free to implement them yourselves. I wrote a profiler that took the return address from the stack, found the stack frame creation instruction, located the block on the stack that represented the locals for the function and coated them with a tasty layer of crap that would cause a bus error if any element was dereferenced.

This caught something like a half dozen errors of pointers being used before initialization, including the bug I was looking for.

What happened was that most of the time the stack happened to have values that were apparently benign if they were dereferenced. Other times the values would cause the app to shotgun its own heap, taking out the app sometime much later.

I spent more than two weeks trying to find this bug.

Lesson: initialize your locals. If someone barks performance at you, show them this comment and tell them that you'd rather spend two weeks running profiling code and fixing bottlenecks rather than having to track down bugs like this. Debugging tools and heap checkers have gotten way better since I had to do this, but quite frankly they got better to compensate for bugs from poor practices like this.

Unless you're running on a tiny system (embedded, etc), initialization of locals should be nearly free. MOVE/LOAD instructions are very, very fast. Write the code to be solid and maintainable first. Refactor it to be performant second.

Share Improve this answer Follow

answered Sep 30, 2008 at 14:54

plinth

49.1k ● 11 ● 83 ● 123

1 This story should be taught on the first lesson in any class for software developers. – Davyd Geyl May 23, 2014 at 5:20

3

It should be *mostly* mandatory. The reason for this has nothing to do with *performance* but rather the danger of using an unitialized variable. However, there are cases where it simply looks ridiculous. For example, I have seen:

```
struct stat s;
s.st_dev = -1;
s.st_ino = -1;
s.st_mode = S_IRWXU;
s.st_nlink = 0;
s.st_size = 0;
// etc...
```

```
s.st_st_ctime = -1;
if(stat(path, &s) != 0) {
   // handle error
   return;
}
```

WTF???

Note that we are handling the error right away, so there is no question about what happens if the stat fails.

Share

Improve this answer

Follow

edited Sep 27, 2008 at 0:27

answered Sep 26, 2008 at 14:39

Andrew Stein

13.1k ● 5 ● 38 ● 44

memset(&s, CHOOSE_YOUR_INITIALIZER, sizeof(struct stat)); – Adam Liss Nov 12, 2008 at 5:22

2

Sometimes you need a variable as a placeholder (e.g. using the `ftime` functions), so it doesn't make sense to initialize them before calling the initialization function.

However it wouldn't be bad, in my opinion, to annotate the fact that you are aware of the pitfalls, something in the way of

```
uninitialized time_t t;
time( &t );
```

Share Improve this answer Follow

answered Sep 26, 2008 at 14:07

xtofl

41.5k ● 12 ● 109 ● 201

1 If you made it mandatory, you could just do this which I think makes just as much sense: time_t t = {0}; – schmick Sep 28, 2008 at 13:41

2

This pertains to C++ only, but there is a definite distinction between the two methods. Let's assume you have a class `MyStuff`, and you want to initialize it by another class. You could do something like:

```
// Initialize MyStuff instance y
// ...
MyStuff x = y;
// ...
```

What this actually does is call the copy constructor of x. It's the same as:

```
MyStuff x(y);
```

This is different than this code:

```
MyStuff x; // This calls the MyStuff default constructor.
x = y; // This calls the MyStuff assignment operator.
```

Of course, completely different code is called when copy constructing vs. default constructing + assigning. Also, a single call to the copy constructor is likely to be more efficient than construction followed by assignment.

Share
Improve this answer
Follow

edited Sep 26, 2008 at 15:27

community wiki
2 revs, 2 users 97%
Marcin

1

Performance? Nowadays? Maybe back when CPUs ran at 10mhz it did make sense, but today its hardly a problem. Always initialise them.

Share Improve this answer Follow

answered Sep 26, 2008 at 14:12

gbjbaanb
52.6k ● 12 ● 110 ● 154

There are people among us that still write for these kind of processors. There is a whole lot of engineering going on in the embedded world. – Nils Pipenbrinck Sep 26, 2008 at 14:15

There are also people writing heavy-duty scientific number crunching code that runs for days/weeks. Even a 1% improvement is significant. But I agree that you compiler should be setup to mark this as a warning by default. – KeithB Sep 26, 2008 at 14:41

1 Gah, it is thinking like this that makes our modern computers just as slow as our old ones. Newer programmers write bloated code with the mentality "the CPU is fast now, we can afford it." – Evan Teran Nov 12, 2008 at 5:57

Yes, erm, I develop a fair amount of code on a 40MHz box. I may have to port it to an 8MHz box. Please not to be sloppy of performance. – Paul Nathan Nov 12, 2008 at 22:32

1 No, initialising variables is not a performance-killer. For that you need to look at libraries/languages that have memory copies at any opportunity as a feature. Don't optimise prematurely, variable initialisation is not going to be your problem. – gbjbaanb Nov 13, 2008 at 23:52

Always initialize local variables to zero at least. As you saw, there's no real performance it.

```
int i = 0;
struct myStruct m = {0};
```

You're basically adding 1 or 2 assembly instructions, if that. In fact, many C runtimes will do this for you on a "Release" build and you won't be changing a thing.

But you should initalize it because you will now have that guarantee.

One reason not to initialize has to do with debugging. Some runtimes, eg. MS CRT, will initialize memory with predetermined and documented patterns that you can identify. So when you're pouring through memory, you can see that the memory is indeed uninitialized and that hasn't been used and reset. That can be helpful in debugging. But that's during debugging.

Share Improve this answer Follow

answered Sep 26, 2008 at 14:26

kervin

11.9k ● 6 ● 46 ● 60

In C/C++ I totally agree with you.

In Perl when I create a variable it is automatically put to a default value.

```
my ($val1, $val2, $val3, $val4);
print $val1, "\n";
print $val1 + 1, "\n";
print $val2 + 2, "\n";
print $val3 = $val3 . 'Hello, SO!', "\n";
print ++$val4 +4, "\n";
```

They are all set to undef initially. Undef is a false value, and a place holder. Due to the dynamic typing if I add a number to it, it assumes that my variable is a number and replaces undef with the eqivilent false value 0. If i do string operations a false version of a string is an empty string, and that gets automatically substituted.

```
[jeremy@localhost Code]$ ./undef.pl

1
2
Hello, SO!
5
```

So for Perl at least declare early and don't worry. Especially as most programs have many variables. You use less lines and it looks cleaner without explicit initializing.

```
my($x, $y, $z);
```

:-)

```
my $x = 0;
my $y = 0;
my $z = 0;
```

Share

Improve this answer

Follow

edited Sep 26, 2008 at 15:16

answered Sep 26, 2008 at 14:18

J.J.

4,892 ● 1 ● 26 ● 29

1

Yes: *always* initialize your variables unless you have a *very* good reason not to. If my code doesn't require a particular initial value, I'll often initialize a variable to a value that will *guarantee* a blatant error if the code that follows is broken.

Share Improve this answer Follow

answered Nov 12, 2008 at 5:37

Adam Liss

48.3k ● 13 ● 113 ● 152

1

Dissenting opinion:

Never initialize any variable before you have a meaningful value to assign to it.

The reason for this has absolutely nothing to do with performance: The reason for this is ***correctness***.

Specifically, this is about allowing your compiler to give you meaningful warnings about uninitialized variables.

Modern compilers of most mainstream programming languages do extensive data flow analysis, and they are capable of unfailingly pointing out to you any situation where you might use a variable without having initialized it first. Thus, accidental use of uninitialized variables is not supposed to be a problem today.

- If you say "but I do not see any such warnings" then you are trying to do programming without first having figured out how to enable all warnings that your

compiler can give. Do not do this. I do not care if you are writing software that will save the world, stop whatever it is that you are doing, figure out how to enable all warnings, enable them, and only then continue coding.

- If you say "but my compiler is incapable of issuing such warnings" then you are using the wrong compiler. Throw away that compiler, and find one that is capable.
- If you say "but there is no such compiler for my language" then throw away everything and start from scratch with a different language. I do not care what it takes; you cannot be programming without warnings in the 3rd millennium.

Once you have warnings about uninitialized variables, then superfluously initializing a variable before you have any meaningful value to assign to it automatically becomes a bad practice, because it circumvents the checks of the compiler. Once you initialize a variable with a bogus value, (say, null, zero, the empty string, etc.) then as far as the compiler can tell, this variable has been initialized, so the compiler will not issue a warning if you forget to give that variable a meaningful non-null, non-zero, or non-empty value further down.

Share

Improve this answer

Follow

edited Oct 11 at 0:07

answered Oct 7 at 13:17

Mike Nakis

61.8k ● 11 ● 124 ● 163

0

As you've showed with respect to performacne it does not make a difference. The compiler will (in optimized builds) detect if a local variable is written without beeing read from and remove the code unless it has other side-effects.

That said: If you initialize stuff with simple statements just to be sure it's initialized it's fine to do so.. I personally don't do it, for a single reason:

It tricks the guys who may later maintain your code into thinking that the initialization is required. That little foo = 0; will increase the code-complexity. Other than that it's just a matter of taste.

If you unnessesary initialize variables via complex statements it may have a side-effect.

For example:

```
float x = sqrt(0);
```

May be optimized by your compiler if you are lucky and work with a clever compiler. With a not so clever compiler it may as well result in a costly and unnessesary function-call because sqrt can - as a side-effect - set the errno variable.

If you call functions that you have defined yourself my best bet is, that the compiler always assumes that they may have side-effects and don't optimize them out. That may be different if the function happen to be in the same translation unit or you have whole program optimization turned on.

Share Improve this answer Follow

answered Sep 26, 2008 at 14:12

Nils Pipenbrinck

86.2k ● 33 ● 155 ● 223

0

Sometimes a variable is used to "collect" the result of a longer block of nested ifs/elses... In those cases I sometimes keep the variable uninitialized, because it *should* be initialized later by one of the conditional branches.

The trick is: if I leave it uninitialized at first and then there's a bug in the long if/else block so the variable is never assigned, I can see that bug in Valgrind :-) which of course requires to frequently run the code (ideally the regular tests) through Valgrind.

Share Improve this answer Follow

answered Sep 26, 2008 at 14:25

oliver

6,751 ● 9 ● 50 ● 55

0

As a simple example, can you determine what this will be initialised to (C/C++)?

```
bool myVar;
```

We had an issue in a product that would sometimes draw an image on screen and sometimes not, usually depending on who's machine it was built with. It turned out that on my machine it was being initialised to false, and on a colleagues machine it was being initialised to true.

Share Improve this answer Follow

answered Sep 26, 2008 at 15:32

Mark Ingram

73.5k ● 53 ● 177 ● 233

If variable is local in scope then as per K&R 4.9 it has an “undefined (i.e. garbage) initial value”. I suspect machine on which it was BUILT had nothing to do with the variation in behavior; it was due to the state of the machine on which it was EXECUTED. Sorry if my question was unclear though. – schmick Sep 28, 2008 at 12:44

Ahh right. Thanks - that's cleared that up for me. Always wondered why. – Mark Ingram Sep 28, 2008 at 16:44

@schmick: It can certainly be affected by tiny differences in compiler versions or libraries. – Ben Voigt Oct 7 at 14:50

0

I think it is in most cases a bad idea to initialize variables with an default value, because it simply hides bugs, that are easily found with uninitialized variables. If you forget to get and set the actual value, or delete the get code by accident, you probably never notice it because 0 is in many cases a reasonable value. Mostly it is much easier to trigger those bugs with an value >> 0.

For example:

```
void func(int n)
{
    int i = 0;

    ... // Many lines of code

    for (;i < n; i++)
        do_something(i);
```

After some time you are going to add some other stuff.

```
void func(int n)
{
    int i = 0;

    for (i = 0; i < 3; i++)
        do_something_else(i);

    ... // Many lines of code

    for (;i < n; i++)
        do_something(i);
```

Now your second loop won't start with 0, but with 3, depending on what the function does it can be very difficult to find, that there is even a bug.

Share edited Sep 26, 2008 at 18:13 answered Sep 26, 2008 at 17:45

Improve this answer

Follow

quinmars
11.6k ● 8 ● 34 ● 41

This should fail a code review because 'i' is not explicitly initialized in a for() loop that *depends* on its initial value. Best practices can almost always be broken by worse ones. – Adam Liss Nov 12, 2008 at 5:31

then use a while loop or do {} while loop here or a function that takes i as an argument, my point is that relying on an initialization that isn't near the code where you use it, can bring you bad results if you extend the function later. – quinmars Nov 12, 2008 at 9:12

0

Just a secondary observation. Initializations are only EASILY optimized on primitive types or when assigned by const functions.

a= foo();

a= foo2();

Cannot be easily optimized because foo may have side effects.

Also heap allocations before time might result in huge performance hits. Take a code like

```
void foo(int x)
```

{

ClassA *instance= new ClassA();

//... do something not "instance" related... if(x>5) {

```
delete instance;

return;
```

}

//.. do something that uses instance

}

On that case, simply declare instance just when you will use it, and initialize it only there. And no The compiler Cannot optimize that for you since the constructor may have side effects that code reordering would change.

edit: I fail at using the code listing feature :P

Share Improve this answer Follow

answered Sep 30, 2008 at 14:26

user23415