

# Are there any legitimate use-cases for "goto" in a language that supports loops and functions?

Asked 16 years, 4 months ago    Modified 1 year, 2 months ago

Viewed 60k times



I've long been under the impression that `goto` should never be used if possible.

**226**



However, while perusing libavcodec (which is written in C) the other day, I was surprised to notice multiple uses of it.



Is it ever advantageous to use `goto` in a language that supports loops and functions? If so, why? Please provide a concrete example that clearly justifies the use of a `goto`.

language-agnostic

goto

Share

Improve this question

Follow

edited Nov 4, 2021 at 4:04



Cody Gray ♦

244k ● 52 ● 501 ● 581

asked Aug 23, 2008 at 18:18



Landon

15.6k ● 12 ● 39 ● 30

## 25 Answers

Sorted by:

Highest score (default)



954



Everybody who is anti-`goto` cites, directly or indirectly, Edsger Dijkstra's [GoTo Considered Harmful](#) article to substantiate their position. Too bad Dijkstra's article has virtually *nothing* to do with the way `goto` statements are used these days and thus what the article says has little to no applicability to the modern programming scene. The `goto`-less meme verges now on a religion, right down to its scriptures dictated from on high, its high priests and the shunning (or worse) of perceived heretics.

Let's put Dijkstra's paper into context to shed a little light on the subject.

When Dijkstra wrote his paper the popular languages of the time were unstructured procedural ones like BASIC, FORTRAN (the earlier dialects) and various assembly languages. It was quite common for people using the higher-level languages to jump *all over their code base* in twisted, contorted threads of execution that gave rise to the term "spaghetti code". You can see this by hopping on over to [the classic Trek game](#) written by Mike Mayfield and trying to figure out how things work. Take a few moments to look that over.

**THIS** is "the unbridled use of the go to statement" that Dijkstra was railing against in his paper in 1968. **THIS** is the environment he lived in that led him to write that paper. The ability to jump anywhere you like in your code

at any point you liked was what he was criticising and demanding be stopped. Comparing that to the anaemic powers of `goto` in C or other such more modern languages is simply risible.

I can already hear the raised chants of the cultists as they face the heretic. "But," they will chant, "you can make code very difficult to read with `goto` in C." Oh yeah? You can make code very difficult to read without `goto` as well. Like this one:

[illegible]

Not a `goto` in sight, so it must be easy to read, right? Or how about this one:

```

a[900];      b;c;d=1      ;e=1;f;      g;h;0;
main(k,
l)char*      *l;{g=      atoi(*      ++l);
for(k=
0;k*k<      g;b=k      ++>>1)      ;for(h=
0;h*h<=
g;++h);      - -h;c=(      (h+=g>h      *(h+1))
-1)>>1;
while(d      <=g){      ++0;for      (f=0;f<
0&& d<=g
; ++f)a[      b<<5|c]      =d++,b+=      e;for(
f=0;f<0
&& d<=g;      ++f)a[b      <<5|c]=      d++,c+=
e;e= -e
;}for(c      =0;c<h;      ++c){      for(b=0
;b<k;++
b){if(b      <k/2)a[      b<<5|c]      ^=a[(k      -
(b+1))
<<5|c]^=      a[b<<5      |c]^=a[      (k-(b+1      ))
<<5|c]
;printf(      a[b<<5|c      ]?"%-4d"      :"      "
,a[b<<5
|c]);}      putchar(      '\n');}}      /*Mike
Laman*/

```

No `goto` there either. It must therefore be readable.

What's my point with these examples? It's not language features that make unreadable, unmaintainable code. It's not syntax that does it. It's bad programmers that cause this. And bad programmers, as you can see in that above item, can make **any** language feature unreadable and unusable. Like the `for` loops up there. (You can see them, right?)

Now to be fair, some language constructs are easier to abuse than others. If you're a C programmer, however, I'd

peer far more closely at about 50% of the uses of `#define` long before I'd go on a crusade against `goto`!

So, for those who've bothered to read this far, there are several key points to note.

1. Dijkstra's paper on `goto` statements was written for a programming environment where `goto` was a *lot* more potentially damaging than it is in most modern languages that aren't an assembler.
2. Automatically throwing away all uses of `goto` because of this is about as rational as saying "I tried to have fun once but didn't like it so now I'm against it".
3. There are legitimate uses of the modern (anaemic) `goto` statements in code that cannot be adequately replaced by other constructs.
4. There are, of course, illegitimate uses of the same statements.
5. There are, too, illegitimate uses of the modern control statements like the "`godo`" abomination where an always-false `do` loop is broken out of using `break` in place of a `goto`. These are often worse than judicious use of `goto`.

Share Improve this answer

Follow

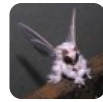
edited Mar 5, 2019 at 21:11



TylerH

21.2k ● 76 ● 79 ● 110

answered May 11, 2010 at 9:53



JUST MY correct  
OPINION

36.1k ● 17 ● 79 ● 99

- 
- 23 ... @sgm has already said it: that's one huge pile of logical fallacies. I just want to comment on one thing: You said you have used `goto` 3 times in the past x years – tell me, how does this compare to *any* other “general-purpose” language feature (I explicitly exclude specialized, yet sometimes necessary language features such as `volatile` which some people just don't need)? Isn't it odd that a *general-purpose* language feature gets used so little in good code? For me, that's a strong indication that `goto` doesn't bring much to the table. I.e., that it is defective. Go figure.  
– [Konrad Rudolph](#) May 12, 2010 at 8:35
- 
- 30 -1 for "Other things are bad, so `goto` can't be" arguments. I'm pretty sure the people arguing against `goto` would also be against the code you listed. – [David Stone](#) Apr 30, 2012 at 17:11
- 
- 44 @pocjoc: Writing tail-recursive optimization in C, for example. This comes up in implementation of functional language interpreters/runtimes, but is representative of an interesting category of problems. Most compilers written in C make use of `goto` for this reason. Its a niche use that doesn't occur in most situations, of course, but in the situations it is called for it makes a *lot* of sense to use. – [zxq9](#) Dec 30, 2013 at 2:46
- 
- 75 Why does everybody talk about Dijkstra's "Go to considered harmful" paper without mentioning Knuth's reply to it, "Structured programming with go to statements"? – [Oblomov](#) Dec 31, 2013 at 17:33
- 
- 36 -1 for making a very large argument why Dijkstra's remark is not applicable, while forgetting to show what the advantages

of `goto` actually are (which is the question posted)

– [Maarten Bodewes](#) Jun 29, 2014 at 23:17

---



There are a few reasons for using the "goto" statement that I'm aware of (some have spoken to this already):

**283**

### **Cleanly exiting a function**



Often in a function, you may allocate resources and need to exit in multiple places. Programmers can simplify their code by putting the resource cleanup code at the end of the function, and all "exit points" of the function would goto the cleanup label. This way, you don't have to write cleanup code at every "exit point" of the function.



### **Exiting nested loops**

If you're in a nested loop and need to break out of *all* loops, a goto can make this much cleaner and simpler than break statements and if-checks.

### **Low-level performance improvements**

This is only valid in perf-critical code, but goto statements execute very quickly and can give you a boost when moving through a function. This is a double-edged sword, however, because a compiler typically cannot optimize code that contains gotos.

Note that in all these examples, gotos are restricted to the scope of a single function.

Share Improve this answer

edited Jul 9, 2012 at 13:05

Follow



Noctis Skytower

22k ● 16 ● 84 ● 121

answered Aug 23, 2008 at 18:42



Chris Gillum

15k ● 5 ● 50 ● 66

---

17 The right way to exit nested loops is to refactor the inner loop into a separate method. – [jason](#) Feb 2, 2010 at 20:08

---

152 @Jason - Bah. That's a load of bull. Replacing `goto` with `return` is just silly. It's not "refactoring" anything, it's just "renaming" so that people who grew up in a `goto` - suppressed environment (i.e. all of us) feel better about using what morally amounts to a `goto`. I much prefer to see the loop *where I use it* and see a little `goto`, which by itself is *just a tool*, than see someone having moved the loop somewhere unrelated just to avoid a `goto`.  
– [Chris Lutz](#) Feb 2, 2010 at 20:18

---

21 There are situations where gotos are important: for instance, an exception-less C++ environment. In the Silverlight source we have tens of thousands (or more) of goto statements for safe function exist through the use of macros - key media codecs and libraries often work through return values and never exceptions, and it's difficult to combine these error handling mechanisms in a single performant way. – [Jeff Wilcox](#) May 11, 2010 at 9:15

---

94 It's worth noting that all `break`, `continue`, `return` are basically `goto`, just in nice packaging.  
– [el.pescado - нет войне](#) Jun 11, 2010 at 8:51

---

25 I don't see how `do{...}while(0)` is supposed to be a better idea than goto, except for the fact it works in Java.  
– [Jeremy List](#) Apr 28, 2014 at 13:20

---





172



Obeying best practices blindly is not a best practice. The idea of avoiding `goto` statements as one's primary form of flow control is to avoid producing unreadable spaghetti code. If used sparingly in the right places, they can sometimes be the simplest, clearest way of expressing an idea. Walter Bright, the creator of the Zortech C++ compiler and the D programming language, uses them frequently, but judiciously. Even with the `goto` statements, his code is still perfectly readable.

Bottom line: Avoiding `goto` for the sake of avoiding `goto` is pointless. What you really want to avoid is producing unreadable code. If your `goto`-laden code is readable, then there's nothing wrong with it.

Share Improve this answer

edited Feb 2, 2010 at 20:05

Follow

answered Feb 2, 2010 at 19:59



[dsimcha](#)

68.6k ● 55 ● 219 ● 340



44



Well, there's one thing that's always worse than `goto's`; strange use of other programflow operators to avoid a `goto`:

**Examples:**



```
// 1
try{
    ...
    throw NoErrorException;
    ...
} catch (const NoErrorException& noe){
    // This is the worst
}
```

```
// 2
do {
    ...break;
    ...break;
} while (false);
```

```
// 3
for(int i = 0;...) {
    bool restartOuter = false;
    for (int j = 0;...) {
        if (...)
            restartOuter = true;
        if (restartOuter) {
            i = -1;
        }
    }
}
```

etc  
etc

Share Improve this answer

answered May 11, 2010 at 9:06

Follow



**Viktor Sehr**

**13.1k** ● 6 ● 62 ● 94

---

3 `do{}while(false)` I think can be considered idiomatic.  
You are not allowed to disagree :D – [Thomas Eding](#) Nov 8, 2011 at 1:06

---

47 @trinithis: If it's "idiomatic", that's only because of the anti-goto cult. If you look closely at it, you'll realize it's just a way of saying `goto after_do_block;` without actually saying that. Otherwise...a "loop" that runs exactly one time? I'd call that abuse of control structures. – [cHao](#) Dec 9, 2011 at 19:18



7 @ThomasEding Eding There's one exception to your point. If you ever done some C/C++ programming and had to use `#define`, you would know, that `do{}while(0)` is one of standards for encapsulating multiple lines of code. For example: `#define do{memcpy(a,b,1); something++;}while(0)` is safer and better than `#define memcpy(a,b,1); something++` – [Ignas2526](#) Apr 8, 2014 at 10:51



18 @Ignas2526 You've just very nicely shown how `#define` s are many, many times much worse than using `goto` once in a while :D – [Luaan](#) Jun 16, 2014 at 8:05

5 +1 for a good list of ways that people try to avoid gotos, to an extreme; I've seen mentions of such techniques elsewhere, but this is a great, succinct, list. Pondering why, in the past 15 years, my team has never needed any of those techniques, nor used gotos. Even in a client/server application with hundreds of thousands of lines of code, by multiple programmers. C#, java, PHP, python, javascript. Client, server, and app code. Not bragging, nor proselytizing for one position, just truly curious why some people encounter situations that beg for gotos as clearest solution, and others don't... – [ToolmakerSteve](#) Aug 7, 2014 at 4:45



Since `goto` makes reasoning about program flow hard<sup>1</sup> (aka. "spaghetti code"), `goto` is generally only used to compensate for missing features: The use of `goto` may actually be acceptable, but only if the language doesn't



offer a more structured variant to obtain the same goal.  
Take Doubt's example:



The rule with `goto` that we use is that `goto` is okay to for jumping forward to a single exit cleanup point in a function.

This is true – but only if the language doesn't allow structured exception handling with cleanup code (such as `RAII` or `finally`), which does the same job better (as it is specially built for doing it), or when there's a good reason not to employ structured exception handling (but you will never have this case except at a very low level).

In most other languages, the only acceptable use of `goto` is to exit nested loops. And even there it is almost always better to lift the outer loop into an own method and use `return` instead.

Other than that, `goto` is a sign that not enough thought has gone into the particular piece of code.

---

<sup>1</sup> Modern languages which support `goto` implement some restrictions (e.g. `goto` may not jump into or out of functions) but the problem fundamentally remains the same.

Incidentally, the same is of course also true for other language features, most notably exceptions. And there are usually strict rules in place to only use these features

where indicated, such as the rule not to use exceptions to control non-exceptional program flow.

Share Improve this answer

edited Nov 17, 2014 at 14:31

Follow

answered Aug 23, 2008 at 18:40



Konrad Rudolph

545k ● 138 ● 956 ● 1.2k

---

4 Just curious here, but what about the case of using gotos for clean up code. By clean-up, I mean, not only deallocation of memory, but also, say error logging. I was reading through a bunch of posts, and apparently, no one writes code that prints logs.. hmm?! – [shiva](#) Apr 3, 2009 at 21:32

---

22 The admonition against goto comes from the structured programming era, where early returns were also considered evil. Moving nested loops into a function trades one "evil" for another, and creates a function that has no real reason to exist (outside of "it allowed me to avoid using a goto!"). It's true that goto is the single most powerful tool for producing spaghetti code if used without constraints, but this is a perfect application for it; it simplifies the code. Knuth argued for this usage, and Dijkstra said "Don't fall into the trap of believing that I am terribly dogmatical about goto". – [Mud](#) May 5, 2012 at 19:30

---

7 `finally` ? So using exceptions for things other than error handling is good but using `goto` is bad? I think *exceptions* are quite aptly named. – [Christian](#) Aug 30, 2013 at 15:26

---

2 @Christian `finally` is for cleanup after an exception was thrown. How is that abuse? – [Konrad Rudolph](#) Aug 30, 2013 at 15:53

---

- 3 @Mud: in the cases that I encounter (on a daily basis), creating a function "that has no real reason to exist" **is** the best solution. Reason: it removes detail from the top-level function, such that the result has easy-to-read control flow. I don't personally encounter situations where a goto produces the most readable code. On the other hand, I use multiple returns, in simple functions, where someone else might prefer to goto a single exit point. I would love to see counter-examples where goto is more readable than refactoring into well-named functions. – [ToolmakerSteve](#) Aug 7, 2014 at 3:19
- 



31

In C# [switch](#) statement [doest not allow fall-through](#). So [goto](#) is used to transfer control to a specific switch-case label or the [default](#) label.



For example:



```
switch(value)
{
    case 0:
        Console.WriteLine("In case 0");
        goto case 1;
    case 1:
        Console.WriteLine("In case 1");
        goto case 2;
    case 2:
        Console.WriteLine("In case 2");
        goto default;
    default:
        Console.WriteLine("In default");
        break;
}
```

*Edit: There is one exception on "no fall-through" rule. Fall-through is allowed if a case statement has no code.*

Share Improve this answer

edited Nov 3, 2021 at 10:53

Follow



woof

83 ● 2 ● 8

answered Aug 23, 2008 at 19:10



Jakub Štunc

35.8k ● 25 ● 91 ● 115

- 
- 3 Switch fall-through is supported in .NET 2.0 - [msdn.microsoft.com/en-us/library/06tc147t\(VS.80\).aspx](https://msdn.microsoft.com/en-us/library/06tc147t(VS.80).aspx)  
– [rjzii](#) Sep 16, 2008 at 17:28
- 
- 10 Only if the case doesn't have a code body. If it does have code then you must use the goto keyword.  
– [Matthew Whited](#) Dec 9, 2009 at 18:07
- 
- 31 This answer is so funny - C# removed fall-through because many see it as harmful, and this example uses goto (also seen as harmful by many) to revive the original, supposedly harmful, behaviour, BUT the overall result is actually less harmful (because the code makes it clear that the fallthrough is intentional!). – [thomasrutter](#) May 31, 2010 at 7:15 ✎
- 
- 9 Just because a keyword is written with the letters G-O-T-O does not make it a goto. This presented case is not a goto. It is a switch statement construct for fallthrough. Then again, I don't really know C# very well, so I could be wrong.  
– [Thomas Eding](#) Nov 8, 2011 at 1:00 ✎
- 
- 1 Well, in this case it's a bit more than fall-through (because you can say `goto case 5:` when you're in case 1). It seems like Konrad Rudolph's answer is correct here: `goto` is compensating for a missing feature (and is less clear than the real feature would be). If what we really want is fall-through, perhaps the best default would be no fall-through, but something like `continue` to explicitly request it.  
– [David Stone](#) Apr 30, 2012 at 17:16 ✎
-







17



I've written more than a few lines of assembly language over the years. Ultimately, every high level language compiles down to gotos. Okay, call them "branches" or "jumps" or whatever else, but they're gotos. Can anyone write goto-less assembler?

Now sure, you can point out to a Fortran, C or BASIC programmer that to run riot with gotos is a recipe for spaghetti bolognaise. The answer however is not to avoid them, but to use them carefully.

A knife can be used to prepare food, free someone, or kill someone. Do we do without knives through fear of the latter? Similarly the goto: used carelessly it hinders, used carefully it helps.

Share Improve this answer

answered Sep 16, 2008 at 17:25

Follow



**bugmagnet**

7,749 ● 8 ● 72 ● 136

- 
- 1 Perhaps you want to read why I believe this is fundamentally wrong at [stackoverflow.com/questions/46586/...](http://stackoverflow.com/questions/46586/...)  
– Konrad Rudolph Sep 17, 2008 at 18:46
- 
- 21 Anybody who advances the "it all compiles down to JMP anyway!" argument basically doesn't understand the point behind programming in a higher level language.  
– Nietzsche-jou May 12, 2010 at 5:03
- 
- 10 All you really need is subtract-and-branch. Everything else is for convenience or performance. – David Stone Apr 30, 2012 at 17:19
-



15



```
#ifdef TONGUE_IN_CHEEK
```

Perl has a `goto` that allows you to implement poor-man's tail calls. :-P

```
sub factorial {  
    my ($n, $acc) = (@_, 1);  
    return $acc if $n < 1;  
    @_ = ($n - 1, $acc * $n);  
    goto &factorial;  
}
```

```
#endif
```

Okay, so that has nothing to do with C's `goto`. More seriously, I agree with the other comments about using `goto` for cleanups, or for implementing [Duff's device](#), or the like. It's all about using, not abusing.

(The same comment can apply to `longjmp`, exceptions, [call/cc](#), and the like---they have legitimate uses, but can easily be abused. For example, throwing an exception purely to escape a deeply-nested control structure, under completely non-exceptional circumstances.)

Share Improve this answer

answered Aug 25, 2008 at 2:53

Follow



**C. K. Young**

223k ● 47 ● 390 ● 443

---

I think this is the only reason to use `goto` in Perl.

– [Brad Gilbert](#) Nov 5, 2008 at 1:28

---



Take a look at [When To Use Goto When Programming in C](#):

11



Although the use of goto is almost always bad programming practice (surely you can find a better way of doing XYZ), there are times when it really isn't a bad choice. Some might even argue that, when it is useful, it's the best choice.

Most of what I have to say about goto really only applies to C. If you're using C++, there's no sound reason to use goto in place of exceptions. In C, however, you don't have the power of an exception handling mechanism, so if you want to separate out error handling from the rest of your program logic, and you want to avoid rewriting clean up code multiple times throughout your code, then goto can be a good choice.

What do I mean? You might have some code that looks like this:

```
int big_function()
{
    /* do some work */
    if([error])
    {
        /* clean up*/
        return [error];
    }
    /* do some more work */
    if([error])
```

```

    {
        /* clean up*/
        return [error];
    }
    /* do some more work */
    if([error])
    {
        /* clean up*/
        return [error];
    }
    /* do some more work */
    if([error])
    {
        /* clean up*/
        return [error];
    }
    /* clean up*/
    return [success];
}

```

This is fine until you realize that you need to change your cleanup code. Then you have to go through and make 4 changes. Now, you might decide that you can just encapsulate all of the cleanup into a single function; that's not a bad idea. But it does mean that you'll need to be careful with pointers -- if you plan to free a pointer in your cleanup function, there's no way to set it to then point to NULL unless you pass in a pointer to a pointer. In a lot of cases, you won't be using that pointer again anyway, so that may not be a major concern. On the other hand, if you add in a new pointer, file handle, or other thing that needs cleanup, then you'll need to change your cleanup function again; and then you'll need to change the arguments to that function.

By using `goto`, it will be

```

int big_function()
{
    int ret_val = [success];
    /* do some work */
    if([error])
    {
        ret_val = [error];
        goto end;
    }
    /* do some more work */
    if([error])
    {
        ret_val = [error];
        goto end;
    }
    /* do some more work */
    if([error])
    {
        ret_val = [error];
        goto end;
    }
    /* do some more work */
    if([error])
    {
        ret_val = [error];
        goto end;
    }
end:
    /* clean up*/
    return ret_val;
}

```

The benefit here is that your code following end has access to everything it will need to perform cleanup, and you've managed to reduce the number of change points considerably. Another benefit is that you've gone from having multiple exit points for your function to just one; there's no chance you'll accidentally return from the function without cleaning up.

Moreover, since goto is only being used to jump to a single point, it's not as though you're creating a mass of spaghetti code jumping back and forth in an attempt to simulate function calls. Rather, goto actually helps write more structured code.

---

**In a word, `goto` should always be used sparingly, and as a last resort -- but there is a time and a place for it. The question should be not "do you have to use it" but "is it the best choice" to use it.**

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12



Community Bot

1 • 1

answered Jan 26, 2014 at 5:29



herohuyongtao

50.6k • 30 • 137 • 176

- 
- 1 The CPython [/Modules](#) directory has thousands of uses of `goto`. It's almost entirely for error handling, `goto bail;`, `goto error`, `goto Fail`, `goto nomemory`, `goto invalid_string_error`, `goto exit`, `goto overflow`, `goto finally`, `goto cleanup`, `goto abort` are a sampling of the labels. – [ggorlen](#) Sep 23, 2022 at 18:13
- 



8

The rule with goto that we use is that goto is okay to for jumping forward to a single exit cleanup point in a function. In really complex functions we relax that rule to allow other jump forwards. In both cases we are avoiding



deeply nested if statements that often occur with error code checking, which helps readability and maintenance.



Share Improve this answer

answered Aug 23, 2008 at 18:30

Follow



[Ari Pernick](#)

469 ● 4 ● 8

---

2 I could see something like that being useful in a language like C. However, when you have the power of C++ constructors / destructors, that's generally not so useful. – [David Stone](#) Jul 10, 2012 at 1:51

---

1 *"In really complex functions we relax that rule to allow other jump forwards"* Without an example, that sounds like, if you are making complex functions even more complex by using jumps. Wouldn't be the "better" approach be to refactor and split those complex functions? – [MikeMB](#) Sep 11, 2015 at 14:17

---

1 This was the last purpose for which I used goto. I don't miss goto in Java, because it has try-finally that can do the same job. – [Patricia Shanahan](#) Jun 4, 2016 at 12:17

---



8



The most thoughtful and thorough discussion of goto statements, their legitimate uses, and alternative constructs that can be used in place of "virtuous goto statements" but can be abused as easily as goto statements, is Donald Knuth's article "[Structured Programming with goto Statements](#)", in the December 1974 Computing Surveys (volume 6, no. 4. pp. 261 - 301).



Not surprisingly, some aspects of this 39-year old paper are dated: Orders-of-magnitude increases in processing power make some of Knuth's performance improvements unnoticeable for moderately sized problems, and new programming-language constructs have been invented since then. (For example, try-catch blocks subsume Zahn's Construct, although they are rarely used in that way.) But Knuth covers all sides of the argument, and should be required reading before anyone rehashes the issue yet again.

Share Improve this answer

answered Jan 2, 2014 at 16:44

Follow



nhcohen

151 ● 1 ● 5



8



I find it funny that some people will go as far as to give a list of cases where goto is acceptable, saying that all other uses are unacceptable. Do you really think that you know every case where goto is the best choice for expressing an algorithm?



To illustrate, I'll give you an example that no one here has shown yet:



Today I was writing code for inserting an element in a hash table. The hash table is a cache of previous calculations which can be overwritten at will (affecting performance but not correctness).

Each bucket of the hash table has 4 slots, and I have a bunch of criteria to decide which element to overwrite



when a bucket is full. Right now this means making up to three passes through a bucket, like this:

```
// Overwrite an element with same hash key if it exists
for (add_index=0; add_index < ELEMENTS_PER_BUCKET; add_index++)
    if (slot_p[add_index].hash_key == hash_key)
        goto add;

// Otherwise, find first empty element
for (add_index=0; add_index < ELEMENTS_PER_BUCKET; add_index++)
    if ((slot_p[add_index].type == TT_ELEMENT_EMPTY)
        goto add;

// Additional passes go here...

add:
// element is written to the hash table here
```

Now if I didn't use goto, what would this code look like?

Something like this:

```
// Overwrite an element with same hash key if it exists
for (add_index=0; add_index < ELEMENTS_PER_BUCKET; add_index++)
    if (slot_p[add_index].hash_key == hash_key)
        break;

if (add_index >= ELEMENTS_PER_BUCKET) {
    // Otherwise, find first empty element
    for (add_index=0; add_index < ELEMENTS_PER_BUCKET; add_index++)
        if ((slot_p[add_index].type == TT_ELEMENT_EMPTY)
            break;
    if (add_index >= ELEMENTS_PER_BUCKET)
```

```
// Additional passes go here (nested
further)...
}

// element is written to the hash table here
```

It would look worse and worse if more passes are added, while the version with goto keeps the same indentation level at all times and avoids the use of spurious if statements whose result is implied by the execution of the previous loop.

So there's another case where goto makes the code cleaner and easier to write and understand... I'm sure there are many more, so don't pretend to know all the cases where goto is useful, dissing any good ones that you couldn't think of.

[Share](#) [Improve this answer](#)

[Follow](#)

edited Aug 23, 2018 at 8:33



[Klutt](#)

31.2k ● 19 ● 61 ● 105

answered May 11, 2010 at 8:54



[Ricardo](#)

97 ● 1 ● 2

- 
- 2 In the example you gave, I'd prefer to refactor that pretty significantly. In general, I try to avoid one-liner comments that say what the next chunk of code is doing. Instead, I break that out into its own function that is named similar to the comment. If you were to make such a transformation, then this function would give a high level overview of what the function is doing, and each of the new functions would state how you do each step. I feel that much more important than

any opposition to `goto` is having each function be at the same level of abstraction. That it avoids `goto` is a bonus.

– [David Stone](#) Apr 30, 2012 at 17:26

---

- 3 You haven't really explained how adding more functions gets rid of the `goto`, the multiple indentation levels and spurious if statements... – [Ricardo](#) Jul 9, 2012 at 10:30
- 

It would look something like this, using standard container notation: `container::iterator it =`

`slot_p.find(hash_key); if (it != slot_p.end())`

`it->overwrite(hash_key); else it =`

`slot_p.find_first_empty();` I find that sort of

programming much easier to read. Each function in this case could be written as a pure function, which is much easier to reason about. The main function now explains what the code does just by the name of the functions, and then if you want, you can look at their definitions to find out how it does it.

– [David Stone](#) Jul 10, 2012 at 1:43 

---

- 6 The fact that anyone has to give examples, of how certain algorithms should naturally use a `goto` - is a sad reflection on how little algorithmic thinking goes on today!! Of course, @Ricardo's example is (one of many) perfect examples of where `goto` is elegant and obvious. – [Fattie](#) Oct 5, 2014 at 14:42
-



7



One of the reasons goto is bad, besides coding style is that you can use it to create *overlapping*, but *non-nested* loops:

```
loop1:
  a
loop2:
  b
  if(cond1) goto loop1
  c
  if(cond2) goto loop2
```

This would create the bizarre, but possibly legal flow-of-control structure where a sequence like (a, b, c, b, a, b, a, b, ...) is possible, which makes compiler hackers unhappy. Apparently there are a number of clever optimization tricks that rely on this type of structure not occurring. (I should check my copy of the dragon book...) The result of this might (using some compilers) be that other optimizations aren't done for code that contains `goto S`.

It might be useful if you *know* it just, "oh, by the way", happens to persuade the compiler to emit faster code. Personally, I'd prefer to try to explain to the compiler about what's probable and what's not before using a trick like goto, but arguably, I might also try `goto` before hacking assembler.

Share Improve this answer

edited Aug 23, 2008 at 19:50

Follow

answered Aug 23, 2008 at 19:44



[Anders Eurenus](#)

4,226 ● 2 ● 25 ● 20

---

3 Well...takes me back to the days of programming FORTRAN in a financial information company. In 2007. – [Marcin](#) Oct 22, 2008 at 7:18

---

8 Any language structure can be abused in ways that render it unreadable or poorly-performing.  
– [JUST MY correct OPINION](#) May 11, 2010 at 12:22

---

@JUST: The point is not about readability, or poor performance, but assumptions and guarantees about the flow-of-control graph. Any abuse of goto would be for *increased* performance (or readability). – [Anders Eurenus](#) May 28, 2010 at 7:31

---

3 I'd argue one of the reasons `goto` is *useful* is that it allows you to construct loops like this, that would require a bunch of logical contortions otherwise. I'd further argue that if the optimizer doesn't know how to rewrite this, then *good*. A loop like this shouldn't be done for performance or readability, but because that is exactly the order in which things need to happen. In which case i wouldn't particularly *want* the optimizer screwing around with it. – [cHao](#) Jan 6, 2014 at 15:43

---

3 ...a straightforward translation of the required algorithm into GOTO-based code may be much easier to validate than a mess of flag variables and abused looping constructs.  
– [supercat](#) Dec 29, 2014 at 19:22

---



Some say there is no reason for goto in C++. Some say that in 99% cases there are better alternatives. **This is not reasoning, just irrational impressions.** Here's a

6

solid example where goto leads to a nice code, something like enhanced do-while loop:

```
int i;

PROMPT_INSERT_NUMBER:
    std::cout << "insert number: ";
    std::cin >> i;
    if(std::cin.fail()) {
        std::cin.clear();
        std::cin.ignore(1000, '\n');
        goto PROMPT_INSERT_NUMBER;
    }

std::cout << "your number is " << i;
```

Compare it to goto-free code:

```
int i;

bool loop;
do {
    loop = false;
    std::cout << "insert number: ";
    std::cin >> i;
    if(std::cin.fail()) {
        std::cin.clear();
        std::cin.ignore(1000, '\n');
        loop = true;
    }
} while(loop);

std::cout << "your number is " << i;
```

I see these differences:

- nested `{ }` block is needed (albeit `do { ... } while` looks more familiar)

- extra `loop` variable is needed, used in four places
- it takes longer time to read and understand the work with the `loop`
- the `loop` does not hold any data, it just controls the flow of the execution, which is less comprehensible than simple label

There is another example

```
void sort(int* array, int length) {
  SORT:
    for(int i=0; i<length-1; ++i)
  if(array[i]>array[i+1]) {
    swap(data[i], data[i+1]);
    goto SORT; // it is very easy to understand
  }
  this code, right?
}
```

Now let's get rid of the "evil" goto:

```
void sort(int* array, int length) {
  bool seemslegit;
  do {
    seemslegit = true;
    for(int i=0; i<length-1; ++i)
  if(array[i]>array[i+1]) {
    swap(data[i], data[i+1]);
    seemslegit = false;
  }
  } while(!seemslegit);
}
```

You see it is the same type of using goto, it is well structured pattern and it is not forward goto as many

promote as the only recommended way. Surely you want to avoid "smart" code like this:

```
void sort(int* array, int length) {
    for(int i=0; i<length-1; ++i)
    if(array[i]>array[i+1]) {
        swap(data[i], data[i+1]);
        i = -1; // it works, but WTF on the first
glance
    }
}
```

The point is that goto can be easily misused, but goto itself is not to blame. Note that label has function scope in C++, so it does not pollute global scope like in pure assembly, in which *overlapping loops* have its place and are very common - like in the following code for 8051, where 7segment display is connected to P1. The program loops lightning segment around:

```
; P1 states loops
; 11111110 <-
; 11111101 |
; 11111011 |
; 11110111 |
; 11101111 |
; 11101111 |
; 11011111 |
; |_____|

init_roll_state:
    MOV P1,#11111110b
    ACALL delay
next_roll_state:
    MOV A,P1
    RL A
    MOV P1,A
    ACALL delay
```



```
JNB P1.5, init_roll_state
SJMP next_roll_state
```

There is another advantage: goto can serve as named loops, conditions and other flows:

```
if(valid) {
    do { // while(loop)

        // more than one page of code here
        // so it is better to comment the meaning
        // of the corresponding curly bracket

    } while(loop);
} // if(valid)
```

Or you can use equivalent goto with indentation, so you don't need comment if you choose the label name wisely:

```
if(!valid) goto NOTVALID;
    LOOPBACK:

    // more than one page of code here

    if(loop) goto LOOPBACK;
NOTVALID;;
```

[Share](#) [Improve this answer](#)

[edited Nov 16, 2018 at 10:10](#)

[Follow](#)

answered Oct 13, 2016 at 22:08



[Jan Turoň](#)

32.8k ● 23 ● 137 ● 177



5

I have come across a situation where a `goto` was a good solution, and I have not seen this example here or anywhere.



I had a switch case with a few cases which all needed to call the same function in the end. I had other cases which all needed to call a different function in the end.



This looked a bit like this:

```
switch( x ) {  
  
    case 1: case1() ; doStuffFor123() ; break ;  
    case 2: case2() ; doStuffFor123() ; break ;  
    case 3: case3() ; doStuffFor123() ; break ;  
  
    case 4: case4() ; doStuffFor456() ; break ;  
    case 5: case5() ; doStuffFor456() ; break ;  
    case 6: case6() ; doStuffFor456() ; break ;  
  
    case 7: case7() ; doStuffFor789() ; break ;  
    case 8: case8() ; doStuffFor789() ; break ;  
    case 9: case9() ; doStuffFor789() ; break ;  
}
```

Instead of giving every case a function call, I replaced the `break` by a `goto`. The `goto` jumps to a label which is also inside the switch case.

```
switch( x ) {  
  
    case 1: case1() ; goto stuff123 ;  
    case 2: case2() ; goto stuff123 ;  
    case 3: case3() ; goto stuff123 ;  
  
    case 4: case4() ; goto stuff456 ;
```

```
case 5: case5() ; goto stuff456 ;
case 6: case6() ; goto stuff456 ;

case 7: case7() ; goto stuff789 ;
case 8: case8() ; goto stuff789 ;
case 9: case9() ; goto stuff789 ;

stuff123: doStuffFor123() ; break ;
stuff456: doStuffFor456() ; break ;
stuff789: doStuffFor789() ; break ;
}
```

cases 1 through 3 all must call `doStuffFor123()` and similarly cases 4 through 6 had to call `doStuffFor456()` etc.

In my opinion, gotos are perfectly fine if you use them correctly. In the end, any code is as clear as people write it. With gotos one can make spaghetti code, but that does not mean that gotos are the cause of the spaghetti code. That cause is us; programmers. I can also create spaghetti code with functions if I want to. The same goes for macros as well.

[Share](#) [Improve this answer](#)

[Follow](#)

edited Oct 26, 2021 at 14:32



[TylerH](#)

21.2k ● 76 ● 79 ● 110

answered Oct 22, 2020 at 8:16



[bask185](#)

397 ● 1 ● 4 ● 14



In a Perl module, you occasionally want to create subroutines or closures on the fly. The thing is, that once

4

you have created the subroutine, how do you get to it. You could just call it, but then if the subroutine uses `caller(.)` it won't be as helpful as it could be. That is where the `goto &subroutine` variation can be helpful.



## Here is a quick example:

```
sub AUTOLOAD{
  my($self) = @_;
  my $name = $AUTOLOAD;
  $name =~ s/.*:://;

  *{$name} = my($sub) = sub{
    # the body of the closure
  }

  goto $sub;

  # nothing after the goto will ever be executed.
}
```

You can also use this form of `goto` to provide a rudimentary form of tail-call optimization.

```
sub factorial($){
  my($n,$tally) = (@_,1);

  return $tally if $n <= 1;

  $tally *= $n--;
  @_ = ($n,$tally);
  goto &factorial;
}
```

( In [Perl 5 version 16](#) that would be better written as `goto` `SUB _;` )

There is a module that will import a `tail` modifier and one that will import `recur` if you don't like using this form of `goto`.

```
use Sub::Call::Tail;
sub AUTOLOAD {
    ...
    tail &$sub( @_ );
}

use Sub::Call::Recur;
sub factorial($){
    my($n,$tally) = (@_,1);

    return $tally if $n <= 1;
    recur( $n-1, $tally * $n );
}
```

---

**Most of the other reasons to use `goto` are better done with other keywords.**

Like `redo` ing a bit of code:

```
LABEL: ;
...
goto LABEL if $x;
```

```
{
    ...
    redo if $x;
}
```

Or going to the `last` of a bit of code from multiple places:

```
goto LABEL if $x;  
...  
goto LABEL if $y;  
...  
LABEL: ;
```

```
{  
  last if $x;  
  ...  
  last if $y  
  ...  
}
```

Share Improve this answer

edited Jun 20, 2020 at 9:12

Follow



Community Bot

1 • 1

answered Sep 16, 2008 at 16:43



Brad Gilbert

34.1k • 11 • 79 • 130



4



1) The most common use of goto that I know of is emulating exception handling in languages that don't offer it, namely in C. (The code given by Nuclear above is just that.) Look at the Linux source code and you'll see a bazillion gotos used that way; there were about 100,000 gotos in Linux code according to a quick survey conducted in 2013: <http://blog.regehr.org/archives/894>. Goto usage is even mentioned in the Linux coding style



guide:

<https://www.kernel.org/doc/Documentation/CodingStyle>.

Just like object-oriented programming is emulated using structs populated with function pointers, goto has its place in C programming. So who is right: Dijkstra or Linus (and all Linux kernel coders)? It's theory vs. practice basically.

There is however the usual gotcha for not having compiler-level support and checks for common constructs/patterns: it's easier to use them wrong and introduce bugs without compile-time checks. Windows and Visual C++ but in C mode offer exception handling via SEH/VEH for this very reason: exceptions are useful even outside OOP languages, i.e. in a procedural language. But the compiler can't always save your bacon, even if it offers syntactic support for exceptions in the language. Consider as example of the latter case the famous Apple SSL "goto fail" bug, which just duplicated one goto with disastrous consequences

(<https://www.imperialviolet.org/2014/02/22/applebug.html>)

:

```
if (something())
    goto fail;
goto fail; // copypasta bug
printf("Never reached\n");
fail:
    // control jumps here
```

You can have exactly the same bug using compiler-supported exceptions, e.g. in C++:

```
struct Fail {};  
  
try {  
    if (something())  
        throw Fail();  
    throw Fail(); // cypasta bug  
    printf("Never reached\n");  
}  
catch (Fail&) {  
    // control jumps here  
}
```

But both variants of the bug can be avoided if the compiler analyzes and warns you about unreachable code. For example compiling with Visual C++ at the /W4 warning level finds the bug in both cases. Java for instance forbids unreachable code (where it can find it!) for a pretty good reason: it's likely to be a bug in the average Joe's code. As long as the goto construct doesn't allow targets that the compiler can't easily figure out, like gotos to computed addresses(\*\*), it's not any harder for the compiler to find unreachable code inside a function with gotos than using Dijkstra-approved code.

(\*\*) Footnote: Gotos to computed line numbers are possible in some versions of Basic, e.g. GOTO 10\*x where x is a variable. Rather confusingly, in Fortran "computed goto" refers to a construct that is equivalent to a switch statement in C. Standard C doesn't allow computed gotos in the language, but only gotos to statically/syntactically declared labels. GNU C however has an extension to get the address of a label (the unary, prefix && operator) and also allows a goto to a variable of



type void\*. See

<https://gcc.gnu.org/onlinedocs/gcc/Labels-as-Values.html>

for more on this obscure sub-topic. The rest of this post isn't concerned with that obscure GNU C feature.

Standard C (i.e. not computed) gotos are not usually the reason why unreachable code can't be found at compile time. The usual reason is logic code like the following.

Given

```
int computation1() {
    return 1;
}

int computation2() {
    return computation1();
}
```

It's just as hard for a compiler to find unreachable code in any of the following 3 constructs:

```
void tough1() {
    if (computation1() != computation2())
        printf("Unreachable\n");
}

void tough2() {
    if (computation1() == computation2())
        goto out;
    printf("Unreachable\n");
out:;
}

struct Out{};

void tough3() {
    try {
```

```
    if (computation1() == computation2())  
        throw Out();  
    printf("Unreachable\n");  
}  
catch (Out&) {  
}  
}
```

(Excuse my brace-related coding style, but I tried to keep the examples as compact as possible.)

Visual C++ /W4 (even with /Ox) fails to find unreachable code in any of these, and as you probably know the problem of finding unreachable code is undecidable in general. (If you don't believe me about that:

<https://www.cl.cam.ac.uk/teaching/2006/OptComp/slides/lecture02.pdf>)

As a related issue, the C goto can be used to emulate exceptions only inside the body of a function. The standard C library offers a setjmp() and longjmp() pair of functions for emulating non-local exits/exceptions, but those have some serious drawbacks compared to what other languages offer. The Wikipedia article <http://en.wikipedia.org/wiki/Setjmp.h> explains fairly well this latter issue. This function pair also works on Windows (<http://msdn.microsoft.com/en-us/library/yz2ez4as.aspx>), but hardly anyone uses them there because SEH/VEH is superior. Even on Unix, I think setjmp and longjmp are very seldom used.

2) I think the second most common use of goto in C is implementing multi-level break or multi-level continue,

which is also a fairly uncontroversial use case. Recall that Java doesn't allow goto label, but allows break label or continue label. According to

<http://www.oracle.com/technetwork/java/simple-142616.html>, this is actually the most common use case of gotos in C (90% they say), but in my subjective experience, system code tends to use gotos for error handling more often. Perhaps in scientific code or where the OS offers exception handling (Windows) then multi-level exits are the dominant use case. They don't really give any details as to the context of their survey.

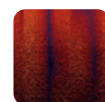
Edited to add: it turns out these two use patterns are found in the C book of Kernighan and Ritchie, around page 60 (depending on edition). Another thing of note is that both use cases involve only forward gotos. And it turns out that MISRA C 2012 edition (unlike the 2004 edition) now permits gotos, as long as they are only forward ones.

Share Improve this answer

edited Jul 22, 2014 at 11:06

Follow

answered Jul 14, 2014 at 6:42




Computer says 'no'--  
SOooooo

5,156 ● 1 ● 29 ● 53

- 
- 1 Right. The "elephant in the room" is that the Linux kernel, for goodness sake, as one example of a world-critical code base - is loaded with goto. Of course it is. Obviously. The "anti-goto-meme" is just a curiosity from decades ago. OF

COURSE, there are a number of things in programming (notably "statics", indeed "globals" and for example "elseif") which can be abused by non-professionals. So, if you're kid cousin is learnin 2 program, you tell them "oh don't use globals" and "never use elseif". – [Fattie](#) Oct 5, 2014 at 14:49

---

- 1 The goto fail bug has nothing to do with goto. The problem is caused by the if statement without braces afterward. Just about any statement copy pasted twice there would have caused a problem. I consider that type of naked braceless if to be much more harmful than goto. – [muusbolla](#) Jun 24, 2019 at 21:01 
- 



If so, why?

3



C has no multi-level/labelled break, and not all control flows can be easily modelled with C's iteration and decision primitives. gotos go a long way towards redressing these flaws.



Sometimes it's clearer to use a flag variable of some kind to effect a kind of pseudo-multi-level break, but it's not always superior to the goto (at least a goto allows one to easily determine where control goes to, unlike a flag variable), and sometimes you simply don't want to pay the performance price of flags/other contortions to avoid the goto.

libavcodec is a performance-sensitive piece of code. Direct expression of the control flow is probably a priority, because it'll tend to run better.

Share Improve this answer

answered Aug 23, 2008 at 19:14

Follow



DrPizza

18.3k ● 7 ● 42 ● 53



2

I find the `do{} while(false)` usage utterly revolting. It is conceivable might convince me it is necessary in some odd case, but never that it is clean sensible code.



If you must do some such loop, why not make the dependence on the flag variable explicit?



```
for (stepfailed=0 ; ! stepfailed ; /*empty*/)
```

Share Improve this answer

edited Nov 16, 2010 at 10:08

Follow



philant

35.7k ● 11 ● 73 ● 113

answered May 11, 2010 at 12:09



Sandy

61 ● 2

Shouldn't that `/*empty*/` be `stepfailed = 1` ? In any case, how is this better than a `do{}while(0)` ? In both, you need to `break` out of it (or in yours `stepfailed = 1; continue;` ). Seems unnecessary to me. – [Thomas Eding](#)  
Nov 8, 2011 at 1:13



2

The GOTO can be used, of course, but there is one more important thing than the code style, or if the code is or not



readable that you must have in mind when you use it: **the code inside may not be as robust as you think.**



For instance, look at the following two code snippets:



```
If A <> 0 Then A = 0 EndIf  
Write("Value of A:" + A)
```

An equivalent code with GOTO

```
If A == 0 Then GOTO FINAL EndIf  
  A = 0  
FINAL:  
Write("Value of A:" + A)
```

The first thing we think is that the result of both bits of code will be that "Value of A: 0" (we suppose an execution without parallelism, of course)

That's not correct: in the first sample, A will always be 0, but in the second sample (with the GOTO statement) A might not be 0. Why?

The reason is because from another point of the program I can insert a `GOTO FINAL` without controlling the value of A.

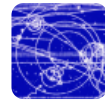
This example is very obvious, but as programs get more complicated, the difficulty of seeing those kind of things increases.

Related material can be found into the famous article from Mr. Dijkstra ["A case against the GO TO statement"](#)

Share Improve this answer

edited Sep 6, 2012 at 17:25

Follow



Beska

12.7k ● 14 ● 79 ● 113

answered Jan 4, 2012 at 12:35



pocjoc

510 ● 8 ● 19

- 
- 9 This was often the case in old-school BASIC. In modern variants, though, you're not allowed to jump into the middle of another function...or in many cases, even past the declaration of a variable. Basically (pun not intended), modern languages have largely done away with the "unbridled" GOTOs that Dijkstra was talking about...and the only way to use it as he was arguing against, is to commit certain other heinous sins. :) – [cHao](#) Jan 6, 2014 at 17:48
- 



It comes in handy for character-wise string processing from time to time.

2

Imagine something like this printf-esque example:



```
for cur_char, next_char in
    sliding_window(input_string) {
    if cur_char == '%' {
        if next_char == '%' {
            cur_char_index += 1
            goto handle_literal
        }
        # Some additional logic
        if chars_should_be_handled_literally() {
            goto handle_literal
        }
        # Handle the format
    }
    # some other control characters
```

```
    else {
        handle_literal:
            # Complicated logic here
            # Maybe it's writing to an array for some
            OpenGL calls later or something,
            # all while modifying a bunch of local
            variables declared outside the loop
        }
    }
```

You could refactor that `goto handle_literal` to a function call, but if it's modifying several different local variables, you'd have to pass references to each unless your language supports mutable closures. You'd still have to use a `continue` statement (which is arguably a form of `goto`) after the call to get the same semantics if your logic makes an else case not work.

I have also used `gotos` judiciously in lexers, typically for similar cases. You don't need them most of the time, but they're nice to have for those weird cases.

[Share](#) [Improve this answer](#)

answered Jul 30, 2019 at 23:11

[Follow](#)



[Beefster](#)

769 ● 7 ● 22



1

In Perl, use of a label to "goto" from a loop - using a "last" statement, which is similar to break.

This allows better control over nested loops.



The traditional `goto label` is supported too, but I'm not sure there are too many instances where this is the only







way to achieve what you want - subroutines and loops should suffice for most cases.

Share Improve this answer

answered Aug 23, 2008 at 18:22

Follow



Abhinav

2,876 ● 3 ● 22 ● 22

---

I think the only form of goto that you would ever use in Perl is `goto &subroutine`. Which starts the subroutine with the current `@_`, while replacing the current subroutine in the stack. – [Brad Gilbert](#) Nov 5, 2008 at 1:27

---



1

I use goto in the following case: when needed to return from functions at different places, and before return some uninitialization needs to be done:



non-goto version:



```
int doSomething (struct my_complicated_stuff
*ctx)
{
    db_conn *conn;
    RSA *key;
    char *temp_data;
    conn = db_connect();

    if (ctx->smth->needs_alloc) {
        temp_data=malloc(ctx->some_size);
        if (!temp_data) {
            db_disconnect(conn);
            return -1;
        }
    }

    ...
}
```

```

if (!ctx->smth->needs_to_be_processed) {
    free(temp_data);
    db_disconnect(conn);
    return -2;
}

pthread_mutex_lock(ctx->mutex);

if (ctx->some_other_thing->error) {
    pthread_mutex_unlock(ctx->mutex);
    free(temp_data);
    db_disconnect(conn);
    return -3;
}

...

key=rsa_load_key(...);

```

goto version:

```

int doSomething_goto (struct
my_complicated_stuff *ctx)
{
    int ret=0;
    db_conn *conn;
    RSA *key;
    char *temp_data;
    conn = db_connect();

    if (ctx->smth->needs_alloc) {
        temp_data=malloc(ctx->some_size);
        if (!temp_data) {
            ret=-1;
            goto exit_db;
        }
    }

    ...

```

```
if (!ctx->smth->needs_to_be_processed) {
    ret=-2;
    goto exit_freemp;
}

pthread_mutex_lock(ctx->mutex);

if (ctx->some_other_thing->error) {
    ret=-3;
    goto exit;
}

...

key=rsa_load_key(...);
```

The second version makes it easier, when you need to change something in the deallocation statements (each is used once in the code), and reduces the chance to skip any of them, when adding a new branch. Moving them in a function will not help here, because the deallocation can be done at different "levels".

Share Improve this answer

answered Nov 19, 2013 at 20:45

Follow



Nuclear

1,398 ● 2 ● 11 ● 17

3 This is why we have **finally** blocks in C#

– [John Saunders](#) Nov 19, 2013 at 20:57

^ like @JohnSaunders said. This is an example of "using goto because a language lacks the appropriate control construct". HOWEVER it is a code smell to require MULTIPLE goto points near the return. There is a programming style that is safer (easier to not screw up) **AND** does not require gotos, which works fine even in languages

without "finally": design those "clean-up" calls so that they are harmless to do when no clean up is required. Factor everything but the cleanup into a method, which uses the multiple-return design. Call that method, then do the clean-up calls. – [ToolmakerSteve](#) Aug 7, 2014 at 5:00

---

Note that the approach I describe requires an extra level of method call (but only in a language which lacks `finally`). As an alternative, use `goto` s, but to a **common** exit point, which always does **all** the clean-up. But each clean-up method either can handle a value that is null or already clean, or is protected by a conditional test, so skipped when not appropriate. – [ToolmakerSteve](#) Aug 7, 2014 at 5:02

---

@ToolmakerSteve this is not a code smell; in fact, it is an extremely common pattern in C and is considered one of the most valid ways to use `goto`. You want me to create 5 methods, each with their own unnecessary if-test, just to handle cleanup from this function? You've now created code AND performance bloat. Or you could just use `goto`.

– [muusbolla](#) Jun 24, 2019 at 21:07

---

@muusbolla - 1) "create 5 methods" - No. I'm suggesting a *single* method that cleans up any resources that have a non-null value. OR see my alternative, that uses `goto` s that all go to **the same** exit point, which has the same logic (which requires an extra if per resource, as you say). But never mind, when using `C` you are right - whatever the reason the code is in C, its almost certainly a trade-off that favors the most "direct" code. (My suggestion handles complex situations where any given resource may or may not have been allocated. But yeah, overkill in this case.)

– [ToolmakerSteve](#) Jun 25, 2019 at 20:37

---



Use "goto" wherever it makes your code more readable or run faster. Just don't let it turn your code into spaghetti.

0

Share Improve this answer

answered Aug 23, 2022 at 7:22

Follow



Kellad

668 ● 1 ● 5 ● 10



-1



The problem with 'goto' and the most important argument of the 'goto-less programming' movement is, that if you use it too frequently your code, although it might behave correctly, becomes unreadable, unmaintainable, unreviewable etc. In 99.99% of the cases 'goto' leads to spaghetti code. Personally, I cannot think of any good reason as to why I would use 'goto'.

Share Improve this answer

answered Aug 23, 2008 at 18:30

Follow



cschol

13k ● 12 ● 68 ● 80

16 Saying "I cannot think of any reason" in your argument is, formally, [en.wikipedia.org/wiki/Argument\\_from\\_ignorance](https://en.wikipedia.org/wiki/Argument_from_ignorance), although I prefer the terminology "proof by lack of imagination". – JUST MY correct OPINION May 11, 2010 at 11:37

2 @JUST MY correct OPINION: It's a logical fallacy only if it's employed post-hoc. From a language designer's point of view it may be a valid argument to weigh the cost of including a feature ( `goto` ). @cschol's usage is similar: While maybe not designing a language right now, (s)he's basically evaluating the designer's effort. – Konrad Rudolph May 12, 2010 at 12:03

2 @KonradRudolph: IMHO, having a language allow `goto` except in contexts where it would bring variables into existence is apt to be cheaper than trying to support every kind of control structure someone might need. Writing code with `goto` may not be as nice as using some other structure, but being able to write such code with `goto` will help avoid "holes in expressiveness"--constructs for which a language is incapable of writing efficient code. – [supercat](#)  
Dec 29, 2014 at 19:27

---

1 @supercat I'm afraid we're from fundamentally different schools of language design. I object to languages being maximally expressive at the price of understandability (or correctness). I'd rather have a restrictive than a permissive language. – [Konrad Rudolph](#) Dec 30, 2014 at 10:33

---

1 @KonradRudolph: I tend to be of the philosophy that it's possible to write unreadable code in any language, and that icky code more often stems from a language's inability to express something nicely, than from its ability to express it poorly. – [supercat](#) Dec 30, 2014 at 19:16

---



**Highly active question.** Earn 10 reputation (not counting the [association bonus](#)) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.