Can we have functions inside functions in C++?

Asked 9 years, 5 months ago Active 11 months ago Viewed 194k times

211

56

I mean something like:

```
int main()
{
  void a()
  {
      // code
  }
  a();

  return 0;
}
```

c++

edited Mar 19 '19 at 2:58
jww
1

asked Dec 1 '10 at 13:23
Rella
54.4k 97 328 605

1 Why are you trying to do this? Explaining your purpose might allow someone to tell you the right way to achieve your goal. – Thomas Owens Dec 1 '10 at 13:26

3 gcc supports nested functions as a non-standard extension. But better don't use it even if you are using gcc. And in C++ mode, it is not available anyway. – Sven Marnach Dec 1 '10 at 13:32

25 @Thomas: Because it would be good to reduce the scope of a? Functions in functions is a usual feature in other languages. – Johan Kotlinski Dec 1 '10 at 13:32

62 He's talking about nested functions. Similarly to being able to next classes inside classes, he wants to nest a function inside a function. Actually, I've had situations where I would have done so, too, if it were possible. There are languages (e.g. F#) which allows this, and I can tell you that it can make code much more clearer, readable and maintainable without polluting a library with dozens of helpers functions that are useless outside of a very specific context. ;) – Mephane Dec 1 '10 at 13:35

14 @Thomas - nested functions can be an excellent mechanism for breaking complex functions/algorithms *without* without filling the current scope with functions that are *not* of general use within the enclosing scope. Pascal and Ada have (IMO) lovely support for them. Same with Scala and many other old/new respected languages. Like any other feature, they can also be abused, but that's a function of the developer. IMO, they have been far more beneficial that detrimental. – luis.espinal Dec 1 '10 at 17:22

12 Answers

Active Oldest Votes

Modern C++ - Yes with lambdas!

252

In current versions of c++ (C++11, C++14, and C++17), you can have functions inside functions in the form of a lambda:

```
int main() {
    // This declares a lambda, which can be called just like a function
    auto print_message = [](std::string message)
    {
        std::cout << message << "\n";
    };

    // Prints "Hello!" 10 times
    for(int i = 0; i < 10; i++) {
        print_message("Hello!");
    }
}
```

Lambdas can also modify local variables through **capture-by-reference*. With capture-by-reference, the lambda has access to all local variables declared in the lambda's scope. It can modify and change them normally.

```
int main() {
    int i = 0;
    // Captures i by reference; increments it by one
    auto addOne = [&] () {
        i++;
    };

    while(i < 10) {
        addOne(); //Add 1 to i
        std::cout << i << "\n";
    }
}
```

C++98 and C++03 - Not directly, but yes with static functions inside local classes

C++ doesn't support that directly.

That said, you can have local classes, and they can have functions (non- `static` or `static`), so you can get this to some extend, albeit it's a bit of a kludge:

```
int main() // it's int, dammit!
{
  struct X { // struct's as good as class
    static void a()
    {
    }
  };

  X::a();

  return 0;
}
```

However, I'd question the praxis. Everyone knows (well, now that you do, anyway `:)`) C++ doesn't support local functions, so they are used to not having them. They are not used, however, to that kludge. I would spend quite a while on this code to make sure it's really only there to allow local functions. Not good.

edited May 8 '19 at 23:29
J. Antonio Perez
7,599 12 29

answered Dec 1 '10 at 13:24
sbi
194k 44 227 416

---

3 Main also takes two args if you're going to be pedantic about the return type. :) (Or is that optional but not the return these days? I can't keep up.) – Leo Davidson Dec 1 '10 at 13:29

3 This is just bad - it breaks every convention of good, clean code. I can't think of a single instance where this is a good idea. – Thomas Owens Dec 1 '10 at 13:29

17 @Thomas Owens: It's good if you need a callback function and don't want to pollute some other namespace with it. – Leo Davidson Dec 1 '10 at 13:31

9 @Leo: The standard says there are two permissible forms for main: `int main()` and `int main(int argc, char* argv[])` – John Dibling Dec 1 '10 at 13:33

8 The standard says `int main()` and `int main(int argc, char* argv[])` must be supported and others may be supported but they all have return int. – JoeG Dec 1 '10 at 13:37

For all intents and purposes, C++ supports this via lambdas:[1]

259

```
int main() {
    auto f = []() { return 42; };
    std::cout << "f() = " << f() << std::endl;
}
```

Here, `f` is a lambda object that acts as a local function in `main`. Captures can be specified to allow the function to access local objects.

Behind the scenes, `f` is a function object (i.e. an object of a type that provides an `operator()`). The function object type is created by the compiler based on the lambda.

[1] since C++11

edited May 14 '18 at 13:15

answered Dec 1 '10 at 13:31

Konrad Rudolph
445k 111 847 1089

5 Ah, that's neat! I didn't think of it. This is much better than my idea, `+1` from me. – sbi Dec 1 '10 at 13:32

1 @sbi: I've actually used local structs to simulate this in the past (yes, I'm suitably ashamed of myself). But the usefulness is limited by the fact that local structs don't create a closure, i.e. you cannot access local variables in them. You need to pass and store them explicitly via a constructor. – Konrad Rudolph Dec 1 '10 at 13:34

1 @Konrad: Another problem with them is that in C++98 you mustn't use local types as template parameters. I think C++1x has lifted that restriction, though. (Or was that C++03?) – sbi Dec 1 '10 at 13:36

3 @luis: I must agree with Fred. You are attaching a meaning to lambdas which they simply don't have (neither in C++ nor in other languages that I've worked with – which *don't* include Python and Ada, for the record). Furthermore, making that distinction is just not meaningful in C++ because C++ does not have local functions, period. It only has lambdas. If you want to limit the scope of a function-like thing to a function, your only choices are lambdas or the local struct mentioned in other answers. I'd say that the latter is rather too convoluted to be of any practical interest. – Konrad Rudolph Dec 2 '10 at 10:04

2 @AustinWBryan No, lambdas in C++ are just syntactic sugar for functors and have zero overhead. There's a question with more detail somewhere on this website. – Konrad Rudolph Jun 14 '18 at 0:10

42

Local classes have already been mentioned, but here is a way to let them appear even more as local functions, using an operator() overload and an anonymous class:

```
int main() {
    struct {
        unsigned int operator() (unsigned int val) const {
            return val<=1 ? 1 : val*(*this)(val-1);
        }
    } fac;

    std::cout << fac(5) << '\n';
}
```

I don't advise on using this, it's just a funny trick (can do, but imho shouldn't).

2014 Update:

With the rise of C++11 a while back, you can now have local functions whose syntax is a little reminiscient of JavaScript:

```
auto fac = [] (unsigned int val) {
    return val*42;
};
```

edited Aug 26 '14 at 7:30

answered Dec 1 '10 at 14:38

Sebastian Mach

34.1k 3 83 121

1 Should be `operator () (unsigned int val)`, your missing a set of parentheses. – Joe D Dec 1 '10 at 14:55

1 Actually, this is a perfectly reasonable thing to do if you need to pass this functor to an stl function or algorithm, like `std::sort()`, or `std::for_each()`. – Dima Dec 1 '10 at 16:18

1 @Dima: Unfortunately, in C++03, locally defined types cannot be used as template arguments. C++0x fixes this, but also provides the much nicer solutions of lambdas, so you still wouldn't do that. – Ben Voigt Dec 2 '10 at 20:31

Oops, you are right. My bad. But still, this is not just a funny trick. It would have been a useful thing if it were allowed. :) – Dima Dec 2 '10 at 20:34

3 Recursion is supported. However, you can't use `auto` to declare the variable. Stroustrup gives the example: `function<void(char*b, char*e)> rev=[](char*b, char*e) { if( 1<e-b ) { swap( *b, *--e); rev(++b,e); } };` for reversing a string given begin and end pointers. – Eponymous Aug 20 '14 at 15:11

17

No.

What are you trying to do?

workaround:

```
int main(void)
{
  struct foo
  {
    void operator()() { int a = 1; }
  };

  foo b;
  b(); // call the operator()
}
```

answered Dec 1 '10 at 13:26

Nim
31.3k 2 52 95

2 Note that the class instantiation approach comes with a memory allocation and is therefore dominated by the static approach. – ManuelSchneid3r May 18 '18 at 19:34

14

Old answer: You can, sort-of, but you have to cheat and use a dummy class:

```
void moo()
{
    class dummy
    {
    public:
         static void a() { printf("I'm in a!\n"); }
    };

    dummy::a();
    dummy::a();
}
```

Newer answer: Newer versions of C++ also support lambdas to do this better/properly. See answers higher up the page.

edited Aug 1 '18 at 16:59

answered Dec 1 '10 at 13:26

Leo Davidson
5,645 1 23 28

Not sure you can, except by creating an object instead (which adds just as much noise, IMO). Unless there's some clever thing you can do with namespaces, but I can't think of it and it's probably not a good idea to abuse the language any more than what we are already. :) – Leo Davidson Dec 2 '10 at 12:31

The getting-rid-of-dummy:: is in one of the other answers. – Sebastian Mach Feb 22 '11 at 14:23

8

As others have mentioned, you can use nested functions by using the gnu language extensions in gcc. If you (or your project) sticks to the gcc toolchain, your code will be mostly portable across the different architectures targeted by the gcc compiler.

However, if there is a possible requirement that you might need to compile code with a different toolchain, then I'd stay away from such extensions.

I'd also tread with care when using nested functions. They are a beautiful solution for managing the structure of complex, yet cohesive blocks of code (the pieces of which are not meant for external/general use.) They are also very helpful in controlling namespace pollution (a very real concern with naturally complex/long classes in verbose languages.)

But like anything, they can be open to abuse.

It is sad that C/C++ does not support such features as an standard. Most pascal variants and Ada do (almost all Algol-based languages do). Same with JavaScript. Same with modern languages like Scala. Same with venerable languages like Erlang, Lisp or Python.

And just as with C/C++, unfortunately, Java (with which I earn most of my living) does not.

I mention Java here because I see several posters suggesting usage of classes and class' methods as alternatives to nested functions. And that's also the typical workaround in Java.

Short answer: No.

Doing so tend to introduce artificial, needless complexity on a class hierarchy. With all things being equal, the ideal is to have a class hierarchy (and its encompassing namespaces and scopes) representing an actual domain as simple as possible.

Nested functions help deal with "private", within-function complexity. Lacking those facilities, one should try to avoid propagating that "private" complexity out and into one's class model.

In software (and in any engineering discipline), modeling is a matter of trade-offs. Thus, in real life, there will be justified exceptions to those rules (or rather guidelines). Proceed with care, though.

edited Dec 1 '10 at 17:54

answered Dec 1 '10 at 17:47

luis.espinal
9,310 5 32 52

7

You can't have local functions in C++. However, C++11 has lambdas. Lambdas are basically variables that work like functions.

A lambda has the type `std::function` (actually that's not quite true, but in most cases you can suppose it is). To use this type, you need to `#include <functional>` . `std::function` is a template, taking as template argument the return type and the argument types, with the syntax `std::function<ReturnType(ArgumentTypes)` . For example, `std::function<int(std::string, float)>` is a lambda returning an `int` and taking two arguments, one `std::string` and one `float` . The most common one is `std::function<void()>` , which returns nothing and takes no arguments.

Once a lambda is declared, it is called just like a normal function, using the syntax `lambda(arguments)` .

To define a lambda, use the syntax `[captures](arguments){code}` (there are other ways of doing it, but I won't mention them here). `arguments` is what arguments the lambda takes, and `code` is the code that should be run when the lambda is called. Usually you put `[=]` or `[&]` as captures. `[=]` means that you capture all variables in the scope in which the value is defined by value, which means that they will keep the value that they had when the lambda was declared. `[&]` means that you capture all variables in the scope by reference, which means that they will always have their current value, but if they are erased from memory the program will crash. Here are some examples:

```
#include <functional>
#include <iostream>

int main(){
    int x = 1;

    std::function<void()> lambda1 = [=](){
        std::cout << x << std::endl;
    };
    std::function<void()> lambda2 = [&](){
        std::cout << x << std::endl;
    };

    x = 2;
    lambda1();    //Prints 1 since that was the value of x when it was captured and x
was captured by value with [=]
    lambda2();    //Prints 2 since that's the current value of x and x was captured by
value with [&]

    std::function<void()> lambda3 = [](){}, lambda4 = [](){};    //I prefer to
initialize these since calling an uninitialized lambda is undefined behavior.
                                                                  //[](){} is the empty
lambda.

    {
        int y = 3;    //y will be deleted from the memory at the end of this scope
        lambda3 = [=](){
            std::cout << y << endl;
        };
        lambda4 = [&](){
            std::cout << y << endl;
        };
    }

    lambda3();    //Prints 3, since that's the value y had when it was captured

    lambda4();    //Causes the program to crash, since y was captured by reference and y
doesn't exist anymore.
                  //This is a bit like if you had a pointer to y which now points
nowhere because y has been deleted from the memory.
                  //This is why you should be careful when capturing by reference.

    return 0;
}
```

You can also capture specific variables by specifying their names. Just specifying their name will capture them by value, specifying their name with a `&` before will capture them by reference. For example, `[=, &foo]` will capture all variables by value except `foo` which will be captured by reference, and `[&, foo]` will capture all variables by reference except `foo` which will be captured by value. You can also capture only specific variables, for example `[&foo]` will capture `foo` by reference and will capture no other variables. You can also capture no variables at all by using `[]`. If you try to use a variable in a lambda that you didn't capture, it won't compile. Here is an example:

```
#include <functional>

int main(){
    int x = 4, y = 5;

    std::function<void(int)> myLambda = [y](int z){
        int xSquare = x * x;    //Compiler error because x wasn't captured
        int ySquare = y * y;    //OK because y was captured
        int zSquare = z * z;    //OK because z is an argument of the lambda
    };
    return 0;
}
```

You can't change the value of a variable that was captured by value inside a lambda (variables captured by value have a `const` type inside the lambda). To do so, you need to capture the variable by reference. Here is an exampmle:

```
#include <functional>

int main(){
    int x = 3, y = 5;
    std::function<void()> myLambda = [x, &y](){
        x = 2;    //Compiler error because x is captured by value and so it's of type
const int inside the lambda
        y = 2;    //OK because y is captured by reference
    };
    x = 2;    //This is of course OK because we're not inside the lambda
    return 0;
}
```

Also, calling uninitialized lambdas is undefined behavior and will usually cause the program to crash. For example, never do this:

```
std::function<void()> lambda;
lambda();    //Undefined behavior because lambda is uninitialized
```

Examples

Here is the code for what you wanted to do in your question using lambdas:

```
#include <functional>    //Don't forget this, otherwise you won't be able to use the
std::function type

int main(){
    std::function<void()> a = [](){
        // code
    }
    a();
    return 0;
}
```

Here is a more advanced example of a lambda:

```
#include <functional>    //For std::function
#include <iostream>      //For std::cout

int main(){
    int x = 4;
    std::function<float(int)> divideByX = [x](int y){
        return (float)y / (float)x;    //x is a captured variable, y is an argument
    }
    std::cout << divideByX(3) << std::endl;    //Prints 0.75
    return 0;
}
```

answered Oct 4 '17 at 13:39

Donald Duck

5,281 14 48 71

No, it's not allowed. Neither C nor C++ support this feature by default, however TonyK points out (in the comments) that there are extensions to the GNU C compiler that enable this behavior in C.

7

edited Dec 1 '10 at 13:31

answered Dec 1 '10 at 13:25

Thomas Owens
103k 92 289 422

2 It is supported by the GNU C compiler, as a special extension. But only for C, not C++. – TonyK Dec 1 '10 at 13:29

Ah. I don't have any special extensions in my C compiler. That's good to know, though. I'll add that titbit to my answer. – Thomas Owens Dec 1 '10 at 13:30

I've used the gcc extension for support of nested functions (in C, though, not C++). Nested functions are a nifty thing (as in Pascal and Ada) for managing complex, yet cohesive structures that are not meant to be of general use. As long as one uses the gcc toolchain, it is assured to be *mostly* portable to all targeted architectures. But if there is change of having to compile the resulting code with a non-gcc compiler, then, it is best to avoid such extensions and stick as close as possible to the ansi/posix mantra. – luis.espinal Dec 1 '10 at 17:34

7

All this tricks just look (more or less) as local functions, but they don't work like that. In a local function you can use local variables of it's super functions. It's kind of semi-globals. Non of these tricks can do that. The closest is the lambda trick from c++0x, but it's closure is bound in definition time, not the use time.

edited Jul 6 '16 at 15:59

Alexis Wilke
13.4k 5 49 94

answered Dec 5 '10 at 8:04

royas
4,088 2 13 12

Now I think this is the best answer. Although it is possible to declare a function within a function (which I use all the time,) it is not a local function as defined in many other languages. It is still good to know of the possibility. – Alexis Wilke Jul 6 '16 at 16:05

6

You cannot define a free function inside another in C++.

answered Dec 1 '10 at 13:25
Prasoon Saurav
81.3k 41 227 332

1 Not with ansi/posix, but you can with gnu extensions. – luis.espinal Dec 1 '10 at 17:35

4

Let me post a solution here for C++03 that I consider the cleanest possible.*

```
#define DECLARE_LAMBDA(NAME, RETURN_TYPE, FUNCTION) \
    struct { RETURN_TYPE operator () FUNCTION } NAME;

...

int main(){
  DECLARE_LAMBDA(demoLambda, void, (){ cout<<"I'm a lambda!"<<endl; });
  demoLambda();

  DECLARE_LAMBDA(plus, int, (int i, int j){
    return i+j;
  });
  cout << "plus(1,2)=" << plus(1,2) << endl;
  return 0;
}
```

(*) in the C++ world using macros is never considered clean.

edited Jul 6 '16 at 19:02

answered Aug 10 '15 at 13:14

Barney

9,137 9 48 75

Alexis, you are right to say that it is not perfectly clean. It is still close to being clean as it well expresses what the programmer meant to do, without side-effects. I consider the art of programming is writing human-readably expressive that reads like a novel. – Barney Jul 6 '16 at 18:57

2

But we can declare a function inside main():

```
int main()
{
    void a();
}
```

Although the syntax is correct, sometimes it can lead to the "Most vexing parse":

```
#include <iostream>


struct U
{
    U() : val(0) {}
    U(int val) : val(val) {}

    int val;
};

struct V
{
    V(U a, U b)
    {
        std::cout << "V(" << a.val << ", " << b.val << ");\n";
    }
    ~V()
    {
        std::cout << "~V();\n";
    }
};

int main()
{
    int five = 5;
    V v(U(five), U());
}
```

=> no program output.

(Only Clang warning after compilation).

C++'s most vexing parse again

edited May 23 '17 at 12:26
Community ♦
1 1

answered Feb 15 '14 at 9:47
trig-ger
995 1 8 17