What is a lambda expression in C++11?

Ask Question

1319

705

What is a lambda expression in C++11? When would I use one? What class of problem do they solve that wasn't possible prior to their introduction?

A few examples, and use cases would be useful.

c++ lambda c++11 c++-faq

edited Nov 2 '11 at 21:12

hugomg
50.4k 17 121 206

asked Oct 2 '11 at 14:58

Nawaz
255k 88 566 756

9 I've seen a case where the lambda was very useful: A colleague of me was doing code that has millions of iterations to solve a space optimization problem. The algorithm was much more speedy when using a lambda than a proper function! The compiler is Visual C++ 2013. –

sergiol Feb 12 '17 at 20:03

8 Answers

1338

The problem

C++ includes useful generic functions like `std::for_each` and `std::transform`, which can be very handy. Unfortunately they can also be quite cumbersome to use, particularly if the functor you would like to apply is unique to the particular function.

```
#include <algorithm>
#include <vector>

namespace {
  struct f {
    void operator()(i
      // do something
    }
  };
}

void func(std::vector
  f f;
  std::for_each(v.beg
}
```

If you only use `f` once and in that specific place it seems overkill to be writing a whole class just to do something trivial and one off.

In C++03 you might be tempted to write something like the following, to keep the functor local:

```
void func2(std::vecto
  struct {
    void operator()(i
       // do somethin
    }
  } f;
  std::for_each(v.beg
}
```

however this is not allowed, `f` cannot be passed to a template function in C++03.

The new solution

C++11 introduces lambdas allow you to write an inline, anonymous functor to replace the `struct f`. For small simple examples this can be cleaner to read (it keeps everything in one place) and potentially simpler to maintain, for example in the simplest form:

```
void func3(std::vecto
  std::for_each(v.beg
}
```

Lambda functions are just syntactic sugar for anonymous functors.

Return types

In simple cases the return type of the lambda is deduced for you, e.g.:

```
void func4(std::vecto
  std::transform(v.be
                 [](d
                 );
}
```

however when you start to write more complex lambdas you will quickly encounter cases where the return type cannot be deduced by the compiler, e.g.:

```
void func4(std::vecto
    std::transform(v.
        [](double d) 
            if (d < 0
                retur
            } else {
                retur
            }
        });
}
```

To resolve this you are allowed to explicitly specify a return type for a lambda function, using `-> T` :

```
void func4(std::vecto
    std::transform(v.
        [](double d)
            if (d < 0
                retur
            } else {
                retur
            }
        });
}
```

"Capturing" variables

So far we've not used anything other than what was passed to the lambda within it, but we can also use other variables, within the lambda. If you want to access other variables you can use the capture clause (the `[]` of the

expression), which has so far been unused in these examples, e.g.:

```
void func5(std::vecto
    std::transform(v.
        [epsilon](doul
            if (d < e
                retur
            } else {
                retur
            }
        });
}
```

You can capture by both reference and value, which you can specify using `&` and `=` respectively:

- `[&epsilon]` capture by reference
- `[&]` captures all variables used in the lambda by reference
- `[=]` captures all variables used in the lambda by value
- `[&, epsilon]` captures variables like with [&], but epsilon by value
- `[=, &epsilon]` captures variables like with [=], but epsilon by reference

The generated `operator()` is `const` by default, with the implication that captures will be `const` when you access them by

default. This has the effect that each call with the same input would produce the same result, however you can mark the lambda as `mutable` to request that the `operator()` that is produced is not `const`.

edited Aug 14 '18 at 9:50

AAEM
1,026 4 20

answered Oct 2 '11 at 15:21

Flexo ♦
70.1k 21 148 232

9 @Yakk you have been trapped. lambdas without a capture have an implicit conversion to function type pointers. the conversion function is `const` always... – Johannes Schaub - l Mar 31 '13 at 22:17

2 @JohannesSchaub -litb oh sneaky -- and it happens when you invoke `()` -- it is passed as a zero-argument lambda, but because `() const` doesn't match the lambda, it looks for a type conversion which allows it, which includes implicit-cast-to-function-pointer, and then calls that! Sneaky! – Yakk - Adam Nevrau Apr 1 '13 at 0:55

2 Interesting - I originally thought that lambdas were

anonymous *functions* rather than functors, and was confused about how captures worked. – immibis Mar 9 '14 at 1:39

34 If you want to use lambdas as variables in your program, you can use: `std::function<double(int, bool)> f = [](int a, bool b) -> double { ... };` But usually, we let the compiler deduce the type: `auto f = [](int a, bool b) -> double { ... };` (and don't forget to `#include <functional>`) – evertheylen Apr 10 '15 at 16:15

8 I suppose not everyone understands why `return d < 0.00001 ? 0 : d;` is guaranteed to return double, when one of the operands is an integer constant (it is because of an implicit promotion rule of the ?: operator where the 2nd and 3rd operand are balanced against each other through the usual arithmetic conversions no matter which one that gets picked). Changing to `0.0 : d` would perhaps make the example easier to understand. – Lundin Dec 17 '15 at 7:32

778

What is a lambda function?

The C++ concept of a lambda function originates in the lambda calculus and functional programming. A lambda is an unnamed function that is useful (in actual programming, not theory) for short snippets of code that are impossible to reuse and are not worth naming.

In C++ a lambda function is defined like this

```
[]() { } // barebone l
```

or in all its glory

```
[]() mutable -> T { }
```

`[]` is the capture list, `()` the argument list and `{}` the function body.

The capture list

The capture list defines what from the outside of the lambda should be available inside the function

body and how. It can be either:

1. a value: [x]
2. a reference [&x]
3. any variable currently in scope by reference [&]
4. same as 3, but by value [=]

You can mix any of the above in a comma separated list `[x, &y]`.

The argument list

The argument list is the same as in any other C++ function.

The function body

The code that will be executed when the lambda is actually called.

Return type deduction

If a lambda has only one return statement, the return type can be omitted and has the implicit type of `decltype(return_statement)`.

Mutable

If a lambda is marked mutable (e.g. `[]() mutable { }`) it is allowed to mutate the

values that have been captured by value.

Use cases

The library defined by the ISO standard benefits heavily from lambdas and raises the usability several bars as now users don't have to clutter their code with small functors in some accessible scope.

C++14

In C++14 lambdas have been extended by various proposals.

Initialized Lambda Captures

An element of the capture list can now be initialized with `=`. This allows renaming of variables and to capture by moving. An example taken from the standard:

```
int x = 4;
auto y = [&r = x, x =
            r += 2;
            return x+2
         }();  // Upda
```

and one taken from Wikipedia showing how to capture with `std::move`:

```
auto ptr = std::make_u
auto lambda = [ptr = s
```

Generic Lambdas

Lambdas can now be generic (`auto` would be equivalent to `T` here if `T` were a type template argument somewhere in the surrounding scope):

```
auto lambda = [](auto
```

Improved Return Type Deduction

C++14 allows deduced return types for every function and does not restrict it to functions of the form `return expression;` . This is also extended to lambdas.

edited Jun 3 '14 at 9:26

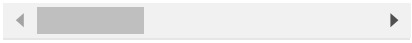
answered Oct 2 '11 at 15:43

pmr
47.2k 7 86 139

1 In your example for initialized lambda captures above, why do you end the lamba function with the ();? This appears like [](){}(); instead of [](){};. Also shouldn't the value of x be 5? – Ramakrishnan Kanna Jun 9 '16 at 13:25

3 @RamakrishnanKannan: 1) the () are there to call the lambda right after defining it and give y its return value. The

variable y is an integer, not the lambda. 2) No, x=5 is local to the lambda (a capture by value which just happens to have the same name as the outer scope variable x), and then x+2 = 5+2 is returned. The reassignment of the outer variable x happens through the reference r: `r = &x; r += 2;` , but this happens to the original value of 4. – The Vee Jul 14 '16 at 13:40



162

Lambda expressions are typically used to encapsulate algorithms so that they can be passed to another function. However, **it is possible to execute a lambda immediately upon definition**:

```
[&](){ ...your code...
```

is functionally equivalent to

```
{ ...your code... } //
```

This makes lambda expressions **a powerful tool for refactoring complex functions.** You start by wrapping a code section in a lambda function as shown above. The process of explicit parameterization can

then be performed gradually with intermediate testing after each step. Once you have the code-block fully parameterized (as demonstrated by the removal of the `&`), you can move the code to an external location and make it a normal function.

Similarly, you can use lambda expressions to **initialize variables based on the result of an algorithm**...

```
int a = []( int b ){ i
```

As **a way of partitioning your program logic**, you might even find it useful to pass a lambda expression as an argument to another lambda expression...

```
[&]( std::function<voi
   {
   ...your wrapper cod
   algorithm();
   ...your wrapper cod
   }
([&]() // algorithm se
   {
   ...your algorithm c
   });
```

Lambda expressions also let you create named **nested functions**, which can be a convenient way of avoiding duplicate logic. Using named lambdas also tends to be a little easier on the

eyes (compared to anonymous inline lambdas) when passing a non-trivial function as a parameter to another function. *Note: don't forget the semicolon after the closing curly brace.*

```
auto algorithm = [&](
   {
   return m*x+b;
   };

int a=algorithm(1,2,3)
```

If subsequent profiling reveals significant initialization overhead for the function object, you might choose to rewrite this as a normal function.

edited Nov 24 '15 at 9:22

NightFurry
129 14

answered Mar 1 '13 at 8:08

nobar
27.1k 10 86 100

11 Have you realized that this question was asked 1.5 years ago and that the last activity was almost 1 year ago? Anyway, you're contributing some interesting ideas I haven't seen before! – Piotr99 Mar 1 '13 at 8:32

7 Thanks for the simultaneous define-and-execute tip! I think it's worth noting that that works in as a contidion for `if` statements: `if`

`([i]{ for (char j : i) if (!isspace(j)) return false ; return true ; }()) // i is all whitespace ,` assuming `i` is an `std::string` – Blacklight Shining Mar 2 '13 at 1:13

69 So the following is a legal expression: `[](){}();` . – nobar Apr 13 '13 at 22:35

8 Ugh! Python's `(lambda: None)()` syntax is so much more legible. – dan04 May 30 '13 at 3:28

8 @nobar - you're right, I mistyped. This is legal (I tested it this time) `main() {{{{{{{[](){{}}()))));}}}}}` – Mark Lakata May 2 '14 at 16:05

Answers

37

Q: What is a lambda expression in C++11?

A: Under the hood, it is the object of an autogenerated class with overloading **operator() const**. Such object is called *closure* and created by compiler. This 'closure' concept is near with the bind concept from C++11. But lambdas typically generate better code. And calls through closures allow full inlining.

Q: When would I use one?

A: To define "simple and small logic" and ask compiler perform generation from previous question. You give a compiler some expressions which you want to be inside operator(). All other stuff compiler will generate to you.

Q: What class of problem do they solve that wasn't possible prior to their introduction?

A: It is some kind of syntax sugar like operators overloading instead of functions for custom *add, subrtact* operations...But it save more lines of unneeded code to wrap 1-3 lines of real logic to some classes, and etc.! Some engineers think that if the number of lines is smaller then there is a less chance to make errors in it (I'm also think so)

Example of usage

```
auto x = [=](int arg1)
void(*f)(int) = x;
f(1);
x(1);
```

Extras about lambdas, not covered by question. Ignore this section if you're not interest

1. Captured values. What you can to capture

1.1. You can reference to a variable with static storage duration in lambdas. They all are captured.

1.2. You can use lambda for capture values "by value". In such case captured vars will be copied to the function object (closure).

```
[captureVar1,captureVa
```

1.3. You can capture be reference. & -- in this context mean reference, not pointers.

```
    [&captureVar1,&capt
```

1.4. It exists notation to capture all non-static vars by value, or by reference

```
   [=](int arg1){} // c

   [&](int arg1){} // c
```

1.5. It exists notation to capture all non-static vars by value, or by reference and specify smth. more. Examples: Capture all not-static vars by value, but by reference capture Param2

```
[=,&Param2](int arg1){
```

Capture all not-static vars by reference, but by value capture Param2

```
[&,Param2](int arg1){}
```

2. Return type deduction

2.1. Lambda return type can be deduced if lambda is one expression. Or you can explicitly specify it.

```
[=](int arg1)->trailin
```

If lambda has more then one expression, then return type must be specified via trailing return type. Also, similar syntax can be applied to auto functions and member-functions

3. Captured values. What you can not capture

3.1. You can capture only local vars, not member variable of the object.

4. Conversions

4.1 !! Lambda is not a function pointer and it is not an anonymous function, but **capture-less** lambdas can be implicitly converted to a function pointer.

p.s.

1. More about lambda grammar information can

be found in Working draft for Programming Language C++ #337, 2012-01-16, 5.1.2. Lambda Expressions, p.88

2. In C++14 the extra feature which has named as "init capture" have been added. It allow to perform arbitarily declaration of closure data members:

```
auto toFloat = []
auto interpolate =
(value - min) / (
```

edited Oct 9 '18 at 19:57

answered Jun 3 '15 at 16:40

bruziuz
2,519 1 26 34

2 Did I not cover smth.? Why I retrived minus? – bruziuz Jun 3 '15 at 22:51

1 I append info about capturing, return type deduction in C++11 after somebody give me a minus. It was not mentioned in question, but I add it to answer! Maybe it was the reason of minus for my post without this "extra section...." – bruziuz Jun 4 '15 at 11:01

This `[&,=Param2](int arg1){}` doesn't seem to be

valid syntax. The correct form would be `[&,Param2](int arg1){}` – GetFree Apr 15 '17 at 8:00

Thanks. First I tried to compile this snippet. And it seems strange assymetry in allowable modificators in capture list // g++ -std=c++11 main.cpp -o test_bin; ./test_bin #include <stdio.h> int main() { #if 1 { int param = 0; auto f= [=,&param](int arg1) mutable {param = arg1;}; f(111); printf("%i\n", param); } #endif #if 0 { int param = 0; auto f= [&,=param](int arg1) mutable {param = arg1;}; f(111); printf("%i\n", param); } #endif return 0; } – bruziuz Apr 16 '17 at 13:02

Looks that new line in not supported in comment. Then I opened 5.1.2 Lambda expressions, p.88, "Working Draft, Standard for Programming Language C ++", Dcoument Number: #337, 2012-01-16. And looked into grammar syntax. And you're right. There is no exist such thing like capture via "=arg" – bruziuz Apr 16 '17 at 13:07

A lambda function is an anonymous function that you

13

create in-line. It can capture variables as some have explained, (e.g. http://www.stroustrup.com/C++11FAQ.html#lambda) but there are some limitations. For example, if there's a callback interface like this,

```
void apply(void (*f)(i
    f(10);
    f(20);
    f(30);
}
```

you can write a function on the spot to use it like the one passed to apply below:

```
int col=0;
void output() {
    apply([](int data)
        cout << data <
    });
}
```

But you can't do this:

```
void output(int n) {
    int col=0;
    apply([&col,n](int
        cout << data <
    });
}
```

because of limitations in the C++11 standard. If you want to use captures, you have to rely on the library and

```
#include <functional>
```

(or some other STL library like algorithm to get it indirectly) and then work with

std::function instead of passing normal functions as parameters like this:

```
#include <functional>
void apply(std::functi
    f(10);
    f(20);
    f(30);
}
void output(int width)
    int col;
    apply([width,&col]
        cout << data <
    });
}
```

edited Mar 11 '15 at 0:16

answered Mar 10 '15 at 22:36

Ted
131 1 3

1 the reason is, that a lambda can only convert to a function pointer, if it has no capture. if `apply` was a template that accepted a functor, it would work – sp2danny Mar 10 '15 at 23:50

1 But the problem is that if apply is an existing interface, you may not have the luxury of being able to declare it differently than a plain old function. The standard could have been designed to allow a new instance of a plain old function to be generated each time such a lambda expression is executed, with generated hard-coded references to the captured variables. It seems a

lambda function is generated at compile time. There are other consequences as well. e.g., If you declare a static variable, even if you re-evaluate the lambda expression, you don't get a new static variable. – Ted Mar 11 '15 at 0:29

1 function pointer are often meant to be saved, and a lambdas capture can go out of scope. that only capture-less lambdas convert to function-pointers was by design – sp2danny Mar 11 '15 at 0:37

1 You still have to pay attention to stack variables being deallocated for the same reason either way. See blogs.msdn.com/b/nativeconcurrency/archive/2012/01/29/… The example I wrote with output and apply is written so that if instead function pointers were allowed and used, they would work as well. The col remains allocated until after all of the function calls from apply have finished. How would you rewrite this code to work using the existing apply interface? Would you end up using global or static variables, or some more obscure transformation of the code? – Ted Mar 11 '15 at 1:34

1 or perhaps you simply mean that lambda expressions

are rvalues and therefore temporary, yet the code remains constant (singleton/static) so that it can be called in the future. In that case, perhaps the function should remain allocated as long as its stack-allocated captures remain allocated. Of course it could get messy unwinding it if for example many variations of the function are allocated in a loop. – Ted Mar 11 '15 at 1:49

10

One of the best explanation of `lambda expression` is given from author of C++ **Bjarne Stroustrup** in his book `***The C++ Programming Language***` chapter 11 (ISBN-13: 978-0321563842):

`What is a lambda expression?`

> A *lambda expression*, sometimes also referred to as a *lambda* function or (strictly speaking incorrectly, but colloquially) as a *lambda*, is a simplified notation for defining and using an **anonymous function object**. Instead of defining

> a named class with an operator(), later making an object of that class, and finally invoking it, we can use a shorthand.

When would I use one?

> This is particularly useful when we want to pass an operation as an argument to an algorithm. In the context of graphical user interfaces (and elsewhere), such operations are often referred to as *callbacks*.

What class of problem do they solve that wasn't possible prior to their introduction?

Here i guess every action done with lambda expression can be solved without them, but with much more code and much bigger complexity. Lambda expression this is the way of optimization for your code and a way of making it more attractive. As sad by Stroustup :

> effective ways of optimizing

Some examples

via lambda expression

```
void print_modulo(cons
v[i]%m==0
{
    for_each(begin(v),
        [&os,m](int x)
           if (x%m==0)
         });
}
```

or via function

```
class Modulo_print {
         ostream& os;
     public:
         Modulo_print(
         void operator
           {
             if (x%m==
           }
};
```

or even

```
void print_modulo(cons
     // output v[i] to
{
    class Modulo_print
        ostream& os; /
        int m;
        public:
           Modulo_prin
           void operat
           {
               if (x%m
           }
     };
     for_each(begin(v)
}
```

if u need u can name `lambda expression` like below:

```
void print_modulo(cons
    // output v[i] to
{
      auto Modulo_prin
      for_each(begin(v
 }
```

Or assume another simple sample

```
void TestFunctions::si
    bool sensitive = t
    std::vector<int> v
```

```
    sort(v.begin(),v.e
         [sensitive](i
             printf("\
             return se
         });


    printf("sorted");
    for_each(v.begin()
             [](int x)
                 print
             }
             );
}
```

will generate next

> 0
>
> 1
>
> 0
>
> 1
>
> 0
>
> 1
>
> 0
>
> 1
>
> 0
>
> 1
>
> 0 sortedx - 1;x - 3;x - 4;x - 5;x - 6;x - 7;x - 33;

`[]` - this is capture list or `lambda introducer` : if `lambdas` require no access to their local environment we can use it.

Quote from book:

> The first character of a lambda expression is always **[**. A lambda

introducer can take various forms:

• **[]**: an empty capture list. This implies that no local names from the surrounding context can be used in the lambda body. For such lambda expressions, data is obtained from arguments or from nonlocal variables.

• **[&]**: implicitly capture by reference. All local names can be used. All local variables are accessed by reference.

• **[=]**: implicitly capture by value. All local names can be used. All names refer to copies of the local variables taken at the point of call of the lambda expression.

• **[capture-list]**: explicit capture; the capture-list is the list of names of local variables to be captured (i.e., stored in the object) by reference or by value. Variables with names preceded by & are captured by reference. Other variables are

captured by value. A capture list can also contain this and names followed by ... as elements.

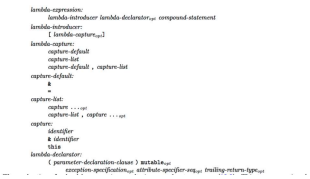
• **[&, capture-list]**: implicitly capture by reference all local variables with names not mentioned in the list. The capture list can contain this. Listed names cannot be preceded by &. Variables named in the capture list are captured by value.

• **[=, capture-list]**: implicitly capture by value all local variables with names not mentioned in the list. The capture list cannot contain this. The listed names must be preceded by &. Variables named in the capture list are captured by reference.

Note that a local name preceded by & is always captured by reference and a local name not preceded by & is always captured by value. Only capture by reference allows modification of variables in the calling environment.

Additional

Lambda expression format



Additional references:

- Wiki
- open-std.org, chapter 5.1.2

edited Nov 9 '16 at 11:11

answered Nov 9 '16 at 11:02

gbk
6,973 7 69 95

2

Well, one practical use I've found out is reducing boiler plate code. For example:

```
void process_z_vec(vec
{
  auto print_2d = [](c
  {
    for(int i = 0; i<b
    {
      for(int j=0; j<b
      {
        cout << board[
      }
      cout << "\n";
    }
  };
  // Do sth with the v
  print_2d(vec,x_size)
  // Do sth else with
  print_2d(vec,y_size)
  //...
}
```

Without lambda, you may need to do

something for different `bsize` cases. Of course you could create a function but what if you want to limit the usage within the scope of the soul user function? the nature of lambda fulfills this requirement and I use it for that case.

edited Mar 30 '17 at 4:24

Klik
1,198 1 12 34

answered Nov 23 '15 at 9:16

Misgevolution
586 4 17

1

One problem it solves: [Code simpler than lambda for a call in constructor that uses an output parameter function for initializing a const member]

You can initialize a const member of your class, with a call to a function that sets its value by giving back its output as an output parameter.

edited May 23 '17 at 12:18

Community ♦
1 1

answered Jun 27 '15 at 0:38

sergiol
2,632 2 26 66

This can also be done with a plain function, which is

even what the accepted answer to the question you linked to says to do. – SirGuy Sep 9 '16 at 2:50

protected by Matt Sep 21 '16 at 12:31

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?