

Lambda Expressions en C++

TPV2
Samir Genaim

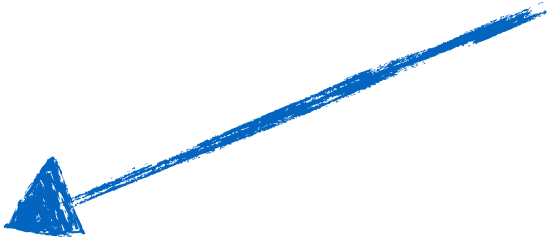
Funciones como parámetros en C

```
int p(int x, int y) {  
    return x+y;  
}
```

```
int m(int x, int y) {  
    return x*y;  
}
```

```
void foo( int (*f)(int,int) ) {  
    cout << f(3,4) << endl;  
}
```

foo recibe una función como parámetro - el parámetro f es una función que recibe 2 valores de tipo `int` y devuelve un `int`



```
int main() {  
    foo(p);  
    foo(m);  
}
```

se puede llamar a foo con una función o otra



Lambda expressions

```
template<typename T>
void foo(T f) {
    std::cout << f(3, 4) << std::endl;
}
```

Una forma más general para definir/pasar funciones como parámetros

```
void f() {
    int x = 1;
    int y = 3;

    auto f1 = [](int i, int j) {
        return i + j;
    };
}
```

Se escribe la función directamente, no hace falta definirla. Puede tener acceso a su contexto — por eso se llama lambda (como funciones en programación funcional)

```
auto f2 = [x, y](int i, int j) {
    return i + j + x + y;
};
```

```
foo(f1);
foo(f2);
}
```

Cuál es el tipo que se usa para λ -expression? No se sabe!! Por eso usamos un parámetro de tipo en el método foo — va a quedar claro cuando hablamos de como se compila, de momento usamos `std::function ...`

El tipo de un lambda expression

```
void foo(std::function<int(int, int)> f) {  
    std::cout << f(3, 4) << std::endl;  
}
```

```
void f() {  
    int x = 1;  
    int y = 3;
```

```
    auto f1 = [](int i, int j) {  
        return i + j;  
    };
```

```
    auto f2 = [x, y](int i, int j) {  
        return i + j + x + y;  
    };
```

```
    foo(f1);  
    foo(f2);  
}
```

Se puede usar

`std::function<int(int, int)>`

para representar un lambda-expression.

No es el tipo de verdad, es un wrapper que tiene dentro un λ -expression (o cualquier otro callable).

Es más cómodo de usar un parámetros de tipo ...

Sintaxis de lambda expressions

Capture: acceso tiene al contexto

Tipo de la salida (se puede omitir)

El cuerpo

[...]<tparams>(int i, int j, ...) lambda-specifiers -> retValue { ... };

Los parámetros (se pueden omitir)

Parámetros de tipo, a partir de c++20

- **mutable**: para poder modificar variables capturados por copia, aunque la modificación no se refleja en el contexto pero hay que especificarlo.
- **constexpr**, **constexpr**, **noexcept**, etc.

Lambda expressions - capture

`[c1,c2,...]<tparams>(int i, int j, ...) lambda-specifiers -> retValue { ... };`



1. [=] todas las variables por copia
2. [&] todas las variables por referencia
3. [=,&x] todas por copia excepto x
4. [&,x,z] todas por referencia excepto x y z
5. [x,&z] x por copia y z por referencia
6. [this] el objeto del contexto por referencia (para acceder/modificar atributos)
7. [*this] el objeto del contexto por copia (para acceder a los atributos) — desde c++17
8. hay otras formas (como move) – ver cppreference.com

variables por copia: la copia se crea con el valor en el momento de declaración, y se pueden asignar solo si usamos **mutable**

Capture - variable por referencia

```
void foo(std::function<void(int)> f) {  
    int x = 3;  
    f(x);  
}
```

cambia el valor de x (del main) a 3

```
int main() {  
    int x = 1;  
    foo( [&x](int i) { x=i; } );  
    cout << x << endl;  
}
```

refiere al x del main

Capture - variable por referencia

```
std::function<int()> p() {  
    int x=1;  
    return [&x]() { return x; };  
}
```

p y q devuelven funciones

```
std::function<void(int)> q() {  
    int z=1;  
    return [&z](int i) { z=i; };  
}
```

```
int main() {  
    auto f = p();  
    auto h = q();  
    h(5);  
    cout << f() << endl;  
}
```

En el momento de ejecutar h y f, la variable x y z de p y q no existen, no se puede saber que escribe este programa

Capture - objeto por referencia

```
class A {  
public:  
    A(int x) :x_(x) { }  
    virtual ~A() { }  
    int getX() { return x_; }  
    void setX(int x) { x_=x; }  
  
    std::function<void(int)> createF() {  
        return [this](int i) {x_=i;};  
    }  
private:  
    int x_;  
};
```

Es por referencia porque **this** es un puntero!

```
int main() {  
    A a(5);  
    auto f = a.createF();  
    f(0);  
    cout << a.getX() << endl;  
}
```

Capture - objeto por copia

```
class A {  
public:  
    A(int x) :x_(x) { }  
    virtual ~A() { }  
    int getX() { return x_; }  
    void setX(int x) { x_=x; }  
  
    std::function<void(int)> createF() {  
        return [*this](int i) {x_=i;};  
    }  
  
private:  
    int x_;  
};
```

Es por copia porque **this* es un objeto no puntero!

```
A a(5);  
auto f = a.createF();  
f(5);  
cout << a.getX() << " " << endl;
```

Compilation de λ expressions (I)

Si se puede compilarlo usando una función (y un puntero a esa función), el compilador normalmente lo hace, p.ej., si el λ -expr no usa nada de contexto, es decir la lista de captura es [].

```
int main() {  
    ...  
    auto f1 = [](int i, int j) {  
        return i + j;  
    };  
    foo(f1);  
    ...  
}
```



```
int _lexp123_(int i, int j) {  
    return i + j;  
};  
  
void f() {  
    ...  
    auto f1 = _lexp123_;  
    foo(f1);  
    ...  
}
```

En este caso incluso se puede pasar el λ -expr a una función que acepta un puntero a una función (p.ej., `foo(void(*f)(int)) {...}`)

Compilation de λ expressions (II)

En general, se puede compilar usando una clase auxiliar y application operator

```
int main() {  
    int x;  
    int y;  
    auto f = [&x, y](int i, int j) {  
        x = x + 1;  
        return i + j + y;  
    };  
    foo(f);  
    ...  
}
```



```
class _lex123_ {  
    int &x;  
    int y;  
public:  
    _lex123_(int &x, int y) :  
        x(x), y(y) {  
    }  
    int operator()(int i, int j) {  
        x = x + 1;  
        return i + j + y;  
    }  
};
```

```
int main() {  
    int x;  
    int y;  
    auto f = _lex123_(x,y);  
    foo(f);  
    ...  
}
```

En este caso no se puede pasar el λ -expr a una función C que acepta un puntero a un función

Cómo se implementa std::function?

```
template<typename RT, typename ...Ts>
class callable_base {
public:
    virtual RT operator()(Ts&&...args) = 0;
    virtual ~callable_base() {
    }
};
```

```
template<typename F, typename RT, typename ... Ts>
class callable: callable_base<RT, Ts...> {
    F f_;
public:
    callable(F functor) :
        f_(functor) {
    }

    RT operator()(Ts&&...args) override {
        return f_(std::forward<Ts>(args)...);
    }
};
```

Cómo se implementa std::function?

```
template<typename, typename ...>
class func;

template<typename RT, typename ... Ts>
class func<RT(Ts...)> {
    std::unique_ptr<callable_base<RT, Ts...>> c_;
public:

    template<typename F>
    func(F f) {
        c_.reset(new callable<F, RT, Ts...>(f));
    }

    RT operator()(Ts&&... args) {
        return (*c_)(std::forward<Ts>(args)...);
    }
};
```