

Lambda Expressions en C++



TPV2
Samir Genaim

Lecture Notes
Samir Genaim
TPV2 2021/22
Facultad de Informática
Universidad Complutense de Madrid

Callable

- ◆ **Callable** es cualquier “valor” en un lenguaje de programación al que se puede llamar (como función).
- ◆ Se usa normalmente para callbacks: en lugar de esperar la respuesta de alguien, **le paso un callable** y cuando tenga la respuesta usa ese callable para devolvérmela
- ◆ Se usa también como una abstracción sobre operaciones (te paso una función que devuelve true/false depende si alguna condición cumple – abstracción sobre esa condición).
- ◆ En lenguajes de programación hay varias maneras de hacerlo, p.ej., la posibilidad de pasar una función como se fuera un valor.
- ◆ En programación orientada a objetos se puede pasar un objeto y llamar a sus métodos.

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;  
}
```

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

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

se puede llamar a foo con una función o otra

Callbacks usando objetos ...

```
class SomeTask {  
    void f(int x, int y) {  
        ...  
    }  
}
```

```
void foo( SomeTask *t ) {  
    ...  
    t->f(...)  
    ...  
}
```

```
int main() {  
    foo( new SomeTask() );  
}
```

Clase que representa la tarea,
para callback ...

Pasar una tarea es pasar
un objeto

Callable que usan el contexto

```
void foo(... f) {  
    ...  
    f(...)  
    ...  
}
```

```
int d = 1;  
int c = 3;
```

A veces la tarea que queremos pasar depende o usa información que del entorno donde pasamos ese callable. **Esto no se puede hacer con funciones C**, pero se puede simular con classes (pasando esa información a la constructora) pero es muy incomodo.

```
// función que recibe 'a1' y 'a2' y devuelve a1+a2+d+c;  
auto f1 = ...
```

```
// función que recibe 'a1' y 'a2' y actualiza 'c' a a1*a2;  
auto f2 = ...
```

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

Lambda expression

- ◆ Es un mecanismo que nos permite “crear” un callable con (o sin) referencias al contexto de manera bastante sencilla.
- ◆ El nombre “lambda” viene de programación funcional, o de modelos teóricos de programación funcional, donde a las funciones se llaman λ expressions y pueden tener referencias al contexto en el momento de crearlos, etc.

Lambda expressions

```
template<typename T>
void foo(T 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);
}
```

Una forma más general para definir/pasar funciones como parámetros

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)

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 λ -expression.

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

Es más cómodo de usar un parámetro de tipo, y también permite definir containers de λ -expressions , p.ej.,
`std::vector<function<void(int)>>`

Sintaxis de lambda expressions

Capture: que 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.
- **constexp**, **consteval**, **noexcept**, etc.

Lambda expressions - capture

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



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)

A veces, si se puede compilar 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)

Cuando el λ -expr usa algo del contexto, 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);  
    ...  
}
```

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

```
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);  
    ...  
}
```

Salvo autorización previa por escrito, los materiales entregados a estudiantes durante la carrera sólo podrán ser utilizados para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución posterior (incluida la divulgación en redes sociales) o servicios de compartición en Internet puede vulnerar la normativa de protección de datos y/o la de propiedad intelectual y generar responsabilidad civil y penal. Si encuentras este material en otra persona intelectual que no tenga la extensión ucm.es, avísanos en denunciacontenido@ucm.es o reportcontent@ucm.es

Cómo se implementa std::function?

Representa una clase que tiene un “application operator” con salida de tipo RT y parámetros de tipo Ts, es decir, un callable que recibe parámetros de tipo como en Ts y devuelve algo de tipo RT

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

Cómo se implementa std::function?

Recibe un callable de tipo F en la constructora, y lo almacena como atributo. Su application operator llama a f_. Eso obliga que f_ reciba parámetros de tipo Ts y devuelva valor de tipo RT, en otro caso no compila.

```
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)...);
    }
};
```

La clave está en que todas las instancias de `callable` que son distintas sólo en F, se pueden ver como `callable_base<RT, Ts...>`

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)...);  
    }  
};
```

Creamos como **callable** y lo almacenamos en **c_** como **callable_base** (**F** "desaparece")

La clave está en que **F** no es un parámetro del template de **func**, sino del "method template" de la constructora.