

# CS2013: Programación III

## Teoría: Librería Estándar I

---

José Chávez

UTEC

# Contenido

- *Callables*
- Funciones *Callback*
- Functors
- Expresiones Lambda
- La librería *<algorithm>*

# Callables

---

# Callables

Un objeto *llamable* en C++ es aquel que puede ser utilizado como función. Dentro de este tipo de objetos tenemos:

- Funciones
- Punteros a Funcion
- Functors
- Lambdas

# Callables

Cada objeto llamable tiene su propia **firma**:

```
int (int, int)
```

# Callables

Cada objeto llamable tiene su propia

`int (int, int)`

Esta podría ser la llamada de una función que toma como entrada dos enteros y retorna un entero

# Callables

Cada objeto llamable tiene su propia **firma**:

`int (int, int)`

La firma detalla el tipo de dato que retorna la llamada y el tipo de parámetros que esta requiere.

# Funciones Callback

---



# Funciones Callback

Una función callback es aquella que se pasa como parámetro a una función externa.

# Funciones Callback

Una función callback es aquella que se pasa como parámetro a una función externa. Esto con el objetivo de llamar a la función dentro de la externa.

sumar()  calculadora()

# Funciones Callback

Para realizar una función callback se necesita de un *Puntero a Función*.

```
int suma(int x1, int x2){return x1 + x2;}

void calculadora(int (*f)(int, int), int x1, int x2){
    cout << "El resultado es: " << f(x1, x2) << endl;
}
```

# Funciones Callback

Para realizar una función callback se necesita de un *Puntero a Función*.

```
int suma(int x1, int x2){return x1 + x2;}
```

Un puntero a la función como  
parámetro

```
void calculadora(int (*f)(int, int), int x1, int x2){  
    cout << "El resultado es: " << f(x1, x2) << endl;  
}
```

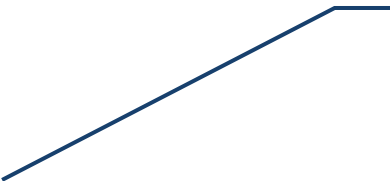
Lo que pasa a la función externa es en realidad **el puntero a la función**.

# Funciones Callback

```
int suma(int x1, int x2){return x1 + x2;}  
int resta(int x1, int x2){return x1 - x2;}  
  
void calculadora(int (*f)(int, int), int x1, int x2){  
    cout << "El resultado es: " << f(x1, x2) << endl;  
}
```

Esto permite utilizar distintas funciones para "f"

# Funciones Callback: Sintaxis

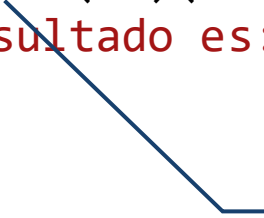


Un puntero a la función

```
void calculadora(int (*f)(int, int), int x1, int x2){  
    cout << "El resultado es: " << f(x1, x2) << endl;  
}
```

# Funciones Callback: Sintaxis

```
void calculadora(int (*f)(int, int), int x1, int x2){  
    cout << "El resultado es: " << f(x1, x2) << endl;  
}
```



El tipo de dato de  
retorno para la función

# Funciones Callback: Sintaxis

```
void calculadora(int (*f)(int, int), int x1, int x2){  
    cout << "El resultado es: " << f(x1, x2) << endl;  
}
```

Los tipos de datos que  
requiere la función

El tipo de retorno para la función



# Funciones Callback: Sintaxis

Parámetros adicionales

```
void calculadora(int (*f)(int, int), int x1, int x2){  
    cout << "El resultado es: " << f(x1, x2) << endl;  
}
```

# Ejemplo 1

```
int suma(int x1, int x2){return x1 + x2;}
int resta(int x1, int x2){return x1 - x2;}

void calculadora(int (*f)(int, int), int x1, int x2){
    cout << "El resultado es: " << f(x1, x2);
}

int main(){
    int x1=3, x2=2;
    calculadora(suma, x1, x2);
    calculadora(resta, x1, x2);

    return 0;
}
```

Las funciones "suma" y "resta" deben ser del mismo tipo que "f"

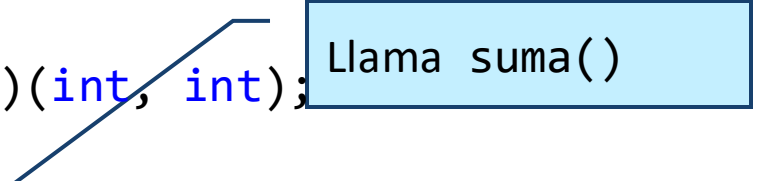
## Ejemplo 2

O podemos crear un puntero a función

```
int main(){  
    int (*pf)(int, int);  
  
    pf = suma;  
    cout << "El resultado es: " << pf(x1, x2) << endl;  
    pf = resta;  
    cout << "El resultado es: " << pf(x1, x2) << endl;  
  
    return 0;  
}
```

## Ejemplo 2

```
int main(){  
    int (*pf)(int, int);  
    pf = suma;  
    cout << "El resultado es: " << pf(x1, x2) << endl;  
    pf = resta;  
    cout << "El resultado es: " << pf(x1, x2) << endl;  
    return 0;  
}
```



A light blue rectangular box with a black border contains the text "Llama suma()". A blue line originates from the "suma" variable in the "pf = suma;" line of the code and points diagonally upwards and to the right towards the box.

# Functors

---

# Functor

Las clases que sobrecargan el operador de llamada "()" permiten que los objetos de su tipo sean utilizados como funciones

# Functor

Las clases que sobrecargan el operador de llamada "()" permiten que los objetos de su tipo sean utilizados como funciones

```
struct pow2{  
    int operator()(int val) {  
        return val * val;  
    }  
};
```

# Functor

Las clases que sobrecargan el operador de llamada "()" permiten que los objetos de su tipo sean utilizados como funciones

```
struct pow2{  
    int operator()(int val) {  
        return val * val;  
    }  
};
```



```
int i  = -5;  
pow2 powObj;  
int i2 = powObj(i); // 25
```



# Ejemplo 3

```
struct printString{  
    char _c;  
    printString(char c = ' '): _c(c) {}  
    void operator() (string s){  
        cout << s << _c;  
    }  
};
```

# Ejemplo 3

```
struct printString{  
    char _c;  
    printString(char c = ' '): _c(c) {}  
    void operator() (string s){  
        cout << s << _c;  
    }  
};
```



```
int main(){  
    printString line('\n');  
    line("Hola");  
  
    printString excl('!');  
    excl("Ingrese un entero valido");  
  
    return 0;  
}
```

# Ejemplo 3

```
struct printString{  
    char _c;  
    printString(char c = ' '): _c(c) {}  
    void operator() (string s){  
        cout << s << _c;  
    }  
};
```



```
int main(){  
    printString line('\n');  
    line("Hola");  
  
    printString excl('!');  
    excl("Ingresa un entero valido");  
  
    return 0;  
}
```

# Ejemplo 4

```
#include <algorithm>
```

```
int main(){  
    string v[4] = {"Hola", "Mundo", "en", "C++"};  
  
    for_each(v, v+4, printString('\n'));  
  
    return 0;  
}
```

# Expresiones Lambda

---

# Expresiones Lambda

C++ nos permite crear funciones anónimas (sin nombre). Similar a los Functors, estas son objetos de función.

# Expresiones Lambda

C++ nos permite crear funciones anónimas (sin nombre). Similar a los Functors, estas son objetos de función.


```
auto foo = [](){  
    cout << "Hola Mundo" << endl;  
};
```

# Expresiones Lambda

C++ nos permite crear funciones anónimas (sin nombre). Similar a los Functors, estas son objetos de función.

```
auto foo = [](){  
    cout << "Hola Mundo" << endl;  
};
```

```
void foo(){  
    cout << "Hola Mundo" << endl;  
}
```



Ambas funciones  
hacen exactamente  
lo mismo



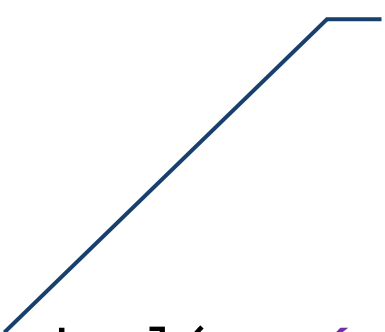
# Sintáxis

Una expresión lambda se define de la siguiente manera:

```
[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}
```

# Sintaxis

Una expresión lambda se define de la siguiente manera:



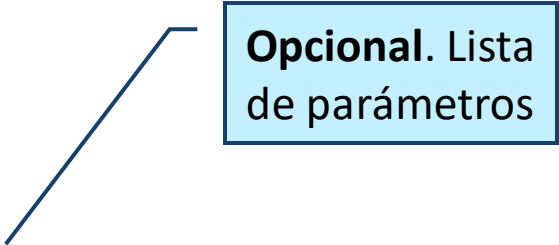
Lista. Detalla las variables  
externas que puede acceder el  
cuerpo de la función

`[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}`

# Sintaxis

Una expresión lambda se define de la siguiente manera:

[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}



**Opcional.** Lista de parámetros

# Sintáxis

Una expresión lambda se define de la siguiente manera:

**Opcional.** Permite al cuerpo modificar las variables capturadas por valor.

[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}

# Sintaxis

Una expresión lambda se define de la siguiente manera:

**Opcional.** Indica que la expresión no producirá ninguna excepción.

[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}

# Sintáxis

Una expresión lambda se define de la siguiente manera:

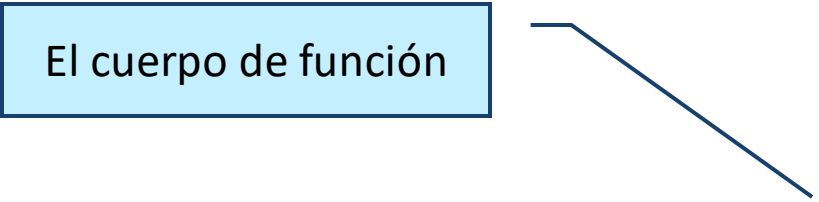
**Opcional.** Tipo de retorno.  
Si se omite, se considera  
como **void**, a no ser que  
se utilice el iniciador

`[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}`

# Sintaxis

Una expresión lambda se define de la siguiente manera:

El cuerpo de función



```
[iniciador](parámetros) mutable noexcept -> tipo {cuerpo}
```

# Sintáxis: El *iniciador*

El iniciador también es llamado cláusula por la característica de captura las variables en el cuerpo de la función. Entre los modos de captura tenemos:

- `[]`: Ninguna variable externa es capturada
- `[lista]`: Solo las variables en lista son capturadas
- `[&]`: Todas las variables externas del cuerpo se capturan por referencia.
- `[=]`: Todas las variables externas del cuerpo se capturan por valor.



# Ejemplo 5

```
struct C{
    int num = 0;
    void metodo(int x){
        auto f = [=]{this->num = x;};
        f();
    }
};

int main(){
    C c;
    cout << "this->num = " <<c.num << endl;
    c.metodo(5);
    cout << "this->num = " <<c.num << endl;

    return 0;
}
```

# Ejemplo 5

```
struct C{  
    int num = 0;  
    void metodo(int x){  
        auto f = [=]{this->num = x;};  
        f();  
    }  
};
```

Estamos capturando  
*this* y *x*.

```
int main(){  
    C c;  
    cout << "this->num = " << c.num << endl;  
    c.metodo(5);  
    cout << "this->num = " << c.num << endl;  
  
    return 0;  
}
```

Librería <algorithm>

---

# Librería Algorithm

Los objetos llamables en ocasiones puede ser utilizados en funciones predefinidas en la librería `algorithm`. Algunas de estas herramientas son:

- `for_each`
- `generate`
- `sort`
- `find`
- `transform`
- `shuffle`

## Ejemplo 6

Utilizamos  
una expresión lambda  
para generar un  
vector con números  
aleatorios entre 0 y 9

```
int main(){  
    vector<int> v(5);  
  
    generate(v.begin(),v.end(),[](){return rand()%10;});  
    for_each(v.begin(),v.end(),[](int x){cout<< x <<" ";});  
    cout << "\n-----\n";  
}
```

## Ejemplo 6

```
int main(){
    vector<int> v(5);

    int n=1;
    generate(v.begin(), v.end(), [&n]() {return n++;});
    for_each(v.begin(), v.end(), [](int x){cout << x << " ";});
    cout << "\n-----\n";
}
```

Utilizamos  
una expresión lambda  
para generar un  
vector con números  
desde 1 hasta el 5

## Ejemplo 6

Utilizamos  
una expresión lambda  
para generar un  
vector con potencias  
de dos

```
int main(){
    vector<int> v(5);

    generate(v.begin(), v.end(), [i=1]()mutable{return i*=2;});
    for_each(v.begin(), v.end(), [](int x){cout << x << " ";});
    cout << "\n-----\n";

}
```

# Ejemplo 7

```
int main(){
    vector<string> v = {"Hola", "Mundo", "en", "C++"};

    for_each(v.begin(), v.end(), [](string s){cout << s << ' ';});
    cout << "\n-----\n";

    sort(v.begin(), v.end(),
        [](string s1, string s2){return s1.size() < s2.size();});

    for_each(v.begin(), v.end(), [](string s){cout << s << ' ';});
    cout << "\n-----\n";

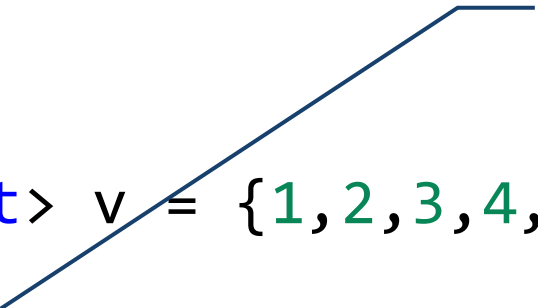
}
```

Utilizamos una expresión lambda para ordenar (con la función sort) un vector en base al tamaño del string.



## Ejemplo 8

```
int fibonacci(int n){  
    if(n<=2) return 1;  
    return fibonacci(n-1) + fibonacci(n-2);  
}  
  
int main(){  
    vector<int> v = {1,2,3,4,5,6,7,8,9,10};  
  
    transform(v.begin(), v.end(), v.begin(), fibonacci);  
    for_each(v.begin(), v.end(), [](int x){cout << x << " ";});  
    cout << "\n-----\n";  
}
```



Utilizamos transform para convertir cada elemento en un término de la serie de fibonacci