

## FUNCTORES

- DEFINICIÓN

En un concepto básico los funtores son clases que se usan como funciones, y ¿cómo se puede lograr esto? Debido a que dentro de la función se sobrecarga el operador “( )” con este mecanismo podemos hacer que al usar ( ) como un método pueda recibir el número de parámetros que deseemos 0, 1, 2, 3, etc., además al ser una clase podemos usar las variables privadas declaradas dentro, también podemos hacer uso del polimorfismo y herencia, estas cosas pueden ser útiles para funcionalidades que se explicarán más adelante. Este es un ejemplo básico de cómo funciona un functor:

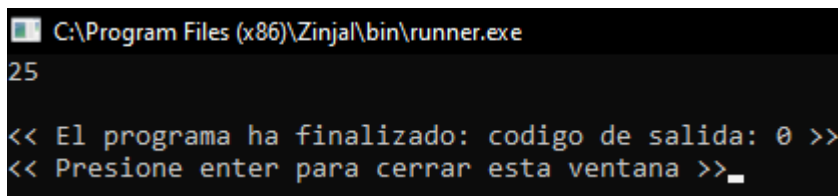
```
//Functor EJ01.cpp
#include <iostream>
using namespace std;

//Creamos la función suma
class suma {
public:
    //Recargamos el operador ( ) para que reciba 2 enteros
    //y devuelva un entero
    int operator()(int s1, int s2) {
        //Retornamos la suma de los valores
        return s1 + s2;
    }
};

int main() {
    //Creamos las variables
    int v1 = 10, v2 = 15;

    //Creamos el objeto de tipo suma que usaremos
    suma sum_var;

    //Usamos el operador ( ) sobrecargado de la función
    cout << sum_var(v1,v2);
    return 0;
}
```



- UTILIDAD

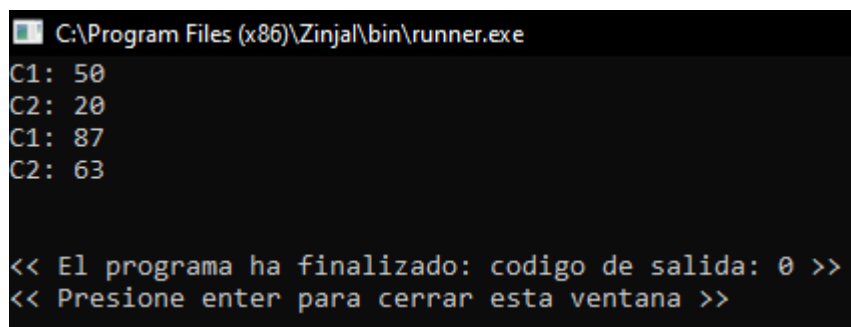
Algunos usos de los funtores es que se pueden usar para guardar variables locales sin la necesidad de crear estas, genera un código más sustentable y la variable se guarda de manera más segura, sin que accidentalmente otra parte del código termine usándola, se puede poner de ejemplo un objeto contador que guarda el “contador” como variable privada:

```
//Functor EJ02.cpp
#include <iostream>
using namespace std;

class contador {
private:
    int cuenta;
public:
    contador() { cuenta = 0; }
public:
    int operator()(int aum) {
        return (cuenta += aum);
    }
};

int main() {
    //Usaremos 2 contadores
    contador c1, c2;
    cout << "C1: " << c1(50) << endl;
    cout << "C2: " << c2(20) << endl;
    cout << "C1: " << c1(37) << endl;
    cout << "C2: " << c2(43) << endl;
    return 0;
}
```

Cada contador tiene un valor distinto al ser objetos diferentes:



```
C:\Program Files (x86)\Zinjal\bin\runner.exe
C1: 50
C2: 20
C1: 87
C2: 63

<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>
```

Otra utilidad que tienen los funtores es que se puede usar polimorfismo y herencia con ellos debido a que son objetos de clase.

```
//Functor EJ03.cpp
#include <iostream>
using namespace std;

class compara {
public:
    virtual ~compara() {};
    virtual int operator()(const int n1, const int n2) const = 0;
    virtual int operator()(const int n1, const int n2, const int n3) const = 0;
};

class mayor : public compara {
public:
    int operator()(const int n1, const int n2) const override {
        if (n1 > n2)
            return n1;
        else
            return n2;
    }
    int operator()(const int n1, const int n2, const int n3) const override {
        if (n1 >= n2 && n1 >= n3)
            return n1;
        else if (n2 >= n1 && n2 >= n3)
            return n2;
        else if (n3 >= n1 && n3 >= n2)
            return n3;
    }
};
```

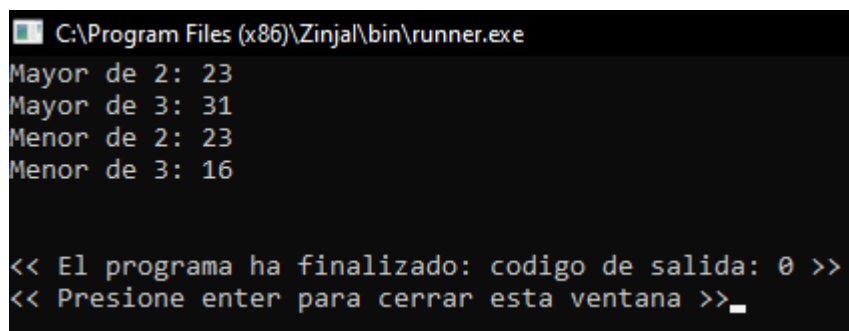
```

        return n2;
    else
        return n3;
    }
};

class menor : public compara {
public:
    int operator()(const int n1, const int n2) const override {
        if (n1 < n2)
            return n1;
        else
            return n2;
    }
    int operator()(const int n1, const int n2, const int n3) const override {
        if (n1 <= n2 && n1 <= n3)
            return n1;
        else if (n2 <= n1 && n2 <= n3)
            return n2;
        else
            return n3;
    }
};

int main() {
    int n1 = 23, n2 = 16, n3 = 31;
    compara *Comp;
    Comp = new mayor;
    cout << "Mayor de 2: " << (*Comp)(n1,n2) << endl;
    cout << "Mayor de 3: " << (*Comp)(n1,n2,n3) << endl;
    delete Comp;
    Comp = new menor;
    cout << "Menor de 2: " << (*Comp)(n1,n3) << endl;
    cout << "Menor de 3: " << (*Comp)(n1,n2,n3) << endl;
    delete Comp;
    return 0;
}

```



```

C:\Program Files (x86)\Zinjal\bin\runner.exe
Mayor de 2: 23
Mayor de 3: 31
Menor de 2: 23
Menor de 3: 16

<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>_

```

Hay que aclarar que los funtores son muy parecidos a los punteros de función y para incluirlos dentro de una función como parámetro la estructura es la misma, sin embargo, es más fácil crear un template que admite ambos mecanismos.

#### //Functor EJ04.cpp

```

#include <iostream>
using namespace std;

class clasesuma {
public:
    int operator()(const int n1, const int n2) {
        return n1 + n2;
    }
};

```

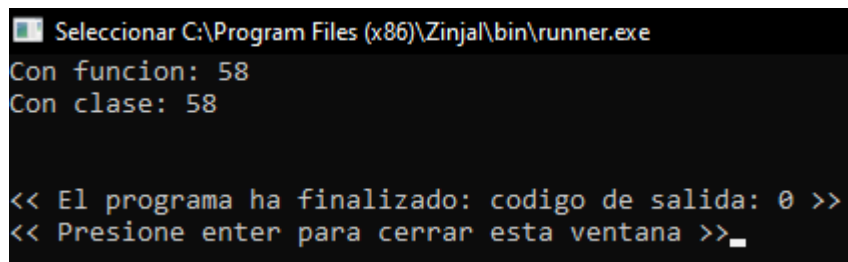
```

int funcsuma (const int n1, const int n2) {
    return n1 + n2;
}

template <typename OperRealz>
int matenum (int n1, int n2, OperRealz opm) {
    return opm(n1,n2);
}

int main() {
    int n1 = 15, n2 = 43;
    clasesuma sum;
    cout << "Con funcion: " << matenum(n1,n2,funcsuma) << endl;
    cout << "Con clase: " << matenum(n1,n2,sum) << endl;
    return 0;
}

```



```

Seleccionar C:\Program Files (x86)\Zinjal\bin\runner.exe
Con funcion: 58
Con clase: 58

<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>_

```

La principal diferencia entre funtores y punteros a funciones radica en que los punteros a funciones no pueden incluir otro puntero a función dentro de sus parámetros, mientras que los funtores al ser una clase si pueden, aunque esta aplicación es poco usada en algunos casos puede servir mucho.

## SMART POINTERS

- **DEFINICION**

Los Smart Pointers se crearon para solucionar los problemas de los punteros normales, como sería la fuga de memoria indefinida al olvidarse de delete dentro de las funciones que creamos, así podemos perder memoria que no se puede reasignar, que en programas pequeños no afecta casi nada, pero en programa grandes... cuando se trabaja con muchos datos puede llegar incluso a provocar la caída del programa.

Los Smart Pointers son un mecanismo de C++ cuya principal función es que cuando destruye el objeto también libera la casilla de memoria, esto dentro de la librería memory, pero ¿cómo lo logran? Al igual que los funtores, los Smart Pointers sobrecargan los operadores \* y ->. El destructor se invoca automáticamente cuando la variable sale de su ámbito, puede se una función, un bucle, etc. Y al llamar a este la memoria que se le asignó al puntero también se elimina automáticamente.

También trabaja con templates para poder eliminar todo tipo de punteros.

```

//SP_EJ01.cpp
#include <iostream>
using namespace std;

template <class R>
class SmartPointer {
    R* ptr;
public:

```

```

SmartPointer(R* p = NULL) { ptr = p; }
~SmartPointer() {
    delete (ptr);
    cout << "Puntero liberado" << endl;
}

R& operator*() { return *ptr; } //Sobrecarga de *
R* operator->() { return ptr; } //Sobrecarga de ->
};

int main() {
    SmartPointer<int> ptr1(new int());
    SmartPointer<double> ptr2(new double());
    *ptr1 = 20;
    cout << *ptr1 << endl;
    *ptr2 = 3.122412;
    cout << *ptr2 << endl;
    return 0;
}

```

```

C:\Program Files (x86)\Zinjal\bin\runner.exe
20
3.12241
Puntero liberado
Puntero liberado

<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>_

```

- **TIPOS**

### UNIQUE\_PTR

Smart Pointers que apuntan a una única casilla de memoria entonces ¿cuál es su utilidad? Al apuntar solo a una casilla de memoria, solo ese puntero tiene acceso a esa variable, puede servir para hacer alguna conexión de cuentas donde solo el cliente tiene acceso a la cuenta, por otro lado este tipo permite que otro puntero pueda apuntar a esta casilla de memoria solo si el primer puntero se elimina, para esto se puede usar la función `move( puntero1)`. Además en el momento de la declaración del puntero se puede ser `make_unique` en vez de `new`, que prácticamente es lo mismo pero el primero es más seguro debido a que `new` puede fallar si es que el sistema no logra reservar bien la memoria necesaria.

```

//SP_EJ02.cpp
#include <iostream>
#include <memory>
using namespace std;

int main(int argc, char *argv[]) {
    unique_ptr<int> ptr1(new int(40));
    unique_ptr<int> ptr2 = make_unique<int>(50);
    unique_ptr<int> ptr3;

    cout << "Puntero 1 con new: " << *ptr1 << endl;
    cout << "Puntero 2 con make_unique: " << *ptr2 << endl;
    ptr3 = move(ptr2);
    cout << "Puntero 2: ";
    if (ptr2 == nullptr) {

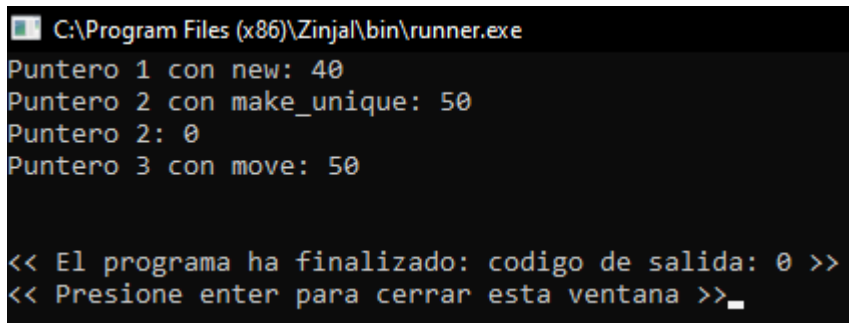
```

```

        cout<< 0 << endl;
    } else {
        cout<< *ptr2 << endl;
    }
    cout << "Puntero 3 con move: " << *ptr3 << endl;

    return 0;
}

```



```

C:\Program Files (x86)\Zinjal\bin\runner.exe
Puntero 1 con new: 40
Puntero 2 con make_unique: 50
Puntero 2: 0
Puntero 3 con move: 50

<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>_

```

## SHARED\_PTR

A diferencia del anterior tipo `shared_ptr` permite que varias instancias apunten a la misma casilla de memoria, esto se puede lograr solo igualando los punteros después de declararlos, al igual que el método `make_unique` también existe el método `make_shared` en reemplazo al método `new`, también existen diferentes métodos como `use_count()` que te devuelve la cantidad de instancias que están apuntando a la misma casilla de memoria, pero ¿cuál es la utilidad de este tipo de puntero? Esta radica en que al ser varios punteros que apuntan a un mismo dato, la casilla de memoria solo se liberará cuando todos los punteros salgan de sus ámbitos, se puede usar cuando se desee compartir nuestros datos al igual que lo haríamos con punteros normales. En el siguiente ejemplo se puede ver cómo en la función existen 4 punteros que apuntan a la misma casilla de memoria, el declarado en el parámetro y el creado dentro.

```

//SP_EJ03.cpp
#include <iostream>
#include <memory>
using namespace std;

void instancia (shared_ptr<string> ptr3) {
    shared_ptr<string> ptr4;
    ptr4 = ptr3;
    cout << "Punteros en la función: " << ptr4.use_count() << endl;
}

int main() {
    shared_ptr<string> ptr1 = make_shared<string>("SESION");
    shared_ptr<string> ptr2;
    ptr2 = ptr1;

    instancia(ptr2);
    cout << "Punteros después de la función: " << ptr2.use_count() << endl;
    return 0;
}

```

```
C:\Program Files (x86)\Zinjal\bin\runner.exe
Punteros en la función: 4
Punteros después de la función: 2

<< El programa ha finalizado: código de salida: 0 >>
<< Presione enter para cerrar esta ventana >>
```

## WEAK\_PTR

Son un tipo de puntero inteligente débil que podría decirse es un `shared_ptr`, solo que este no aumenta el número de referencias que existen para la casilla de memoria, no implementa el método `use_count()`, además de eso no puede leer, modificar o acceder al objeto referenciado. Solo si se convierte en `shared_ptr`, con el método `lock()` o `expired()` puede hacerlo, pero entonces ¿Para qué sirve? `weak_ptr` sirve como una manera de sensor, porque al llamar a los métodos anteriores podemos saber si la casilla de memoria a la que apunta el `shared_ptr` ha sido desasignada o no.

//SP\_EJ04.cpp

```
#include <iostream>
#include <memory>
using namespace std;

weak_ptr<int> instancia () {
    shared_ptr<int> ptr = make_shared<int>(1234);
    weak_ptr<int> ptr2 = ptr;

    if ( ptr2.expired() == true ) {
        cout << "Ptr func: EL OBJETO REFERENCIADO YA HA SIDO ELIMINADO\n";
    } else {
        cout << "Ptr func: EL OBJETO REFERENCIADO ESTA ACTIVO\n";
    }


    return ptr2;
}

int main(int argc, char *argv[]) {
    shared_ptr<int> ptr = make_shared<int>(123);
    weak_ptr<int> ptr2 = ptr;

    if ( ptr2.expired() == true ) {
        cout << "Ptr main: EL OBJETO REFERENCIADO YA HA SIDO ELIMINADO\n";
    } else {
        cout << "Ptr main: EL OBJETO REFERENCIADO ESTA ACTIVO\n";
    }

    if ( instancia().expired() == true ) {
        cout << "Ptr func: EL OBJETO REFERENCIADO YA HA SIDO ELIMINADO\n";
    } else {
        cout << "Ptr func: EL OBJETO REFERENCIADO ESTA ACTIVO\n";
    }

    return 0;
}
```

 C:\Program Files (x86)\Zinjal\bin\runner.exe

Ptr main: EL OBJETO REFERENCIADO ESTA ACTIVO

Ptr func: EL OBJETO REFERENCIADO ESTA ACTIVO

Ptr func: EL OBJETO REFERENCIADO YA HA SIDO ELIMINADO

<< El programa ha finalizado: codigo de salida: 0 >>

<< Presione enter para cerrar esta ventana >>