

C++

Paul Vallejos  
EL710

**Basado en:**

C++: The Complete Reference, Third Edition

Autor: Herbert Schildt

Editorial: Osborne McGraw-Hill

ISBN: 0078824761



Introducción a

# **PROGRAMACIÓN ORIENTADA A OBJETOS**

# [ Programación orientada a objetos ]

---

- Un programa puede estar organizado en torno al código o en torno a los datos.
- En programación orientada a objetos se organiza en torno a los datos.
- Se definen los datos y las rutinas que pueden actuar sobre esos datos.

# [ Encapsulación, polimorfismo, y herencia. ]

---

## ■ Encapsulación

- Se junta el código y la información que manipula.
- Se mantiene a salvo la información de interferencia externa y de mal uso.
- En un objeto la información puede ser privada o publica.

# [ Encapsulación, polimorfismo, y herencia. ]

- Privada: sólo accesible por otra parte del objeto.
- Pública: Se puede acceder desde cualquier parte. Típicamente es la Interfaz.

## ■ Polimorfismo:

- Una interfaz, muchos métodos.
- Una interfaz para controlar acceso a una clase general de acciones.

# [ Encapsulación, polimorfismo, y herencia. ]

---

- Herencia:

- Proceso a través del cual un objeto puede adquirir las características de otro.
- Cuadrado->Paralelepípedo->Figuras geométricas.

[

]

**CLASSES**

# [ Declaración de Clases ]

- Las clases se declaran con la palabra `class`.
- La declaración de una clase define un nuevo tipo que une datos y código.
- El nuevo tipo se usa para declarar objetos de dicha clase.
- La clase es una abstracción, mientras que un objeto tiene una existencia física. Un objeto es una instancia de una clase.



# [ Declaración de Clases ]

## ■ Declaración general

```
class class-name
{
    private data and functions
    access-specifier:
        data and functions
    access-specifier:
        data and functions
    // ...
    access-specifier:
        data and functions
} object-list;
```

- Por defecto los miembros son privados.

# [ Declaración de Clases ]

- Los especificadores de clases pueden ser:
  - public:
    - Los miembros públicos pueden ser accedidos desde cualquier parte.
  - private:
    - Los miembros privados sólo pueden ser accedidos sólo por otros miembros de la clase.

# [ Declaración de Clases ]

- protected:
  - Los miembros protected pueden ser accedidos por otros miembros de la clase, o por miembros de una clase hijo.

# [ Declaración de Clases ]

```
#define SIZE 100
// This creates the class stack.
class stack
{
    int stck[SIZE];
    int tos;
public:
    void init();
    void push(int i);
    int pop();
};
```

# [ Declaración Clases ]

## ■ Variables miembro:

- `int stck[SIZE];`
- `int tos;`

## ■ Funciones miembro:

- `void init();`
- `void push(int i);`
- `int pop();`

## ■ Los objetos se declaran usando el nombre de la clase.

```
stack mystack;
```

# Definición de funciones miembro

- Al definir una función miembro se debe indicar a que clase pertenece usando el operador ::

```
void stack::push(int i)
{
    if(tos==SIZE)
    {
        cout << "Stack is full.\n";
        return;
    }
    stck[tos] = i;
    tos++;
}
```

# [ Uso de funciones miembro ]

- Para utilizar una función o variable miembro desde un código que no pertenece a la clase se debe usar en conjunto con una instancia de la clase y con el operador .

```
stack stack1, stack2;  
stack1.init();
```

# [ Uso de funciones miembro ]

- El operador -> es equivalente a (\*ptr).

```
stack stack1;  
stack *stackPtr=& stack1;  
  
(*stackPtr).init(); //para acceder a un miembro  
stackPtr->init(); //forma equivalente
```



# [Puntero this]

- Las funciones miembro poseen un puntero llamado this.
- El puntero this apunta al objeto que está ejecutando la función miembro.

```
int stack::pop() {  
    if(this->tos==0)  
        return 0;  
    tos--;  
    return this->stck[tos];  
}
```

[

]

**SCOPE 2.0**

# [ Scope 2.0 ]

- Las variables locales se pueden definir en cualquier parte del código, no necesariamente al comienzo.
- Variables locales.
  - Se crean cuando se comienza su bloque.
  - Se destruyen cuando termina su bloque.

# [ Scope 2.0 ]

---

- Variables globales o estáticas.
  - Se crean al comienzo del programa.
  - Se destruyen al finalizar el programa.

# [ Scope 2.0 ]

```
#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
int valorInicial = 0;
class stack {
    int stck[SIZE];
    int tos;
    int test;
public:
    void init();
    void push(int i);
    int pop();
};
void stack::init()
{
    test = valorInicial;
    tos = 0;
}
void stack::push(int i)
{
    if(tos==SIZE) {
        return;
    }
    stck[tos] = i;
    tos++;
}
```

```
int stack::pop()
{
    if(tos==0) {
        return 0;
    }
    tos--;
    return stck[tos];
}
int value1 = 0;
int main()
{
    int value2 = 2;
    stack stack1;
    stack1.init();
    valorInicial = 1;
    stack stack2;
    stack2.init();
    stack1.push(value1);
    stack2.push(value2);
    ++value1;
    ++value2;
    stack1.push(value1);
    stack2.push(value2);
    return 0;
}
```

# [ Scope 2.0 ]

- Cuando una variable miembro es declarada estática, significa que existirá solo una variable que es compartida por todos los objetos de la clase.
- La declaración de una variable estática no la define. Se debe hacer una definición global fuera de la clase.

# [ Scope 2.0 ]

- Las variables miembro no estáticas no pueden tener un inicializador.

```
class shared
{
    static int a;
    int b;
public:
    void set(int i, int j) {a=i; b=j;}
    void show();
} ;

int shared::a; // define a
```

# [ Scope 2.0 ]

- Los miembros estáticos pueden ser referenciados de forma independiente de un objeto usando el operador ::

```
class shared {  
public:  
    static int a;  
} ;  
int shared::a; // define a  
int main() {  
    shared::a = 99;  
    return 0;  
}
```



# [ Scope 2.0 ]

---

- Las funciones miembro estáticas sólo pueden acceder variables miembro estáticas.
- Las funciones miembro estáticas no tienen puntero this.

# [ Scope 2.0 ]

## ■ Operator ::

```
int i; // global i
void f()
{
    int i; // local i
    i = 10; // uses local i

    ::i = 15; // now refers to global i
}
```

# [ Referencias ]

---

- Referencias son como punteros, pero se utilizan como datos.
- Una referencia se declara utilizando el operador &.
- Sirven para evitar copiar memoria al invocar una función.
- También se pueden usar para pasar punteros.



# [ Referencias

```
// Use a reference parameter.
#include <iostream>
using namespace std;
void neg(int &i);

int main(){
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(x);
    cout << x << "\n";
    return 0;
}

void neg(int &i){
    i = -i;
}
```

# [Referencias]

```
#include <iostream>
using namespace std;
class cl {
    int id;
public:
    int i;
    cl(int i);
    ~cl();
    void neg(cl &o) { o.i = -o.i; }
};

cl::cl(int num){
    cout << "Constructing " << num << "\n";
    id = num;
}

cl::~~cl() {
    cout << "Destructing " << id << "\n";
}
```

```
int main()
{
    cl o(1);
    o.i = 10;
    o.neg(o);
    cout << o.i << "\n";
    return 0;
}
```

## Output

```
Constructing 1
-10
Destructing 1
```


# [ Referencias ]

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

char &replace(int i); // return a reference
char s[80] = "Hello There";

int main()
{
    replace(5) = 'X'; // assign X to space after Hello
    cout << s;
    return 0;
}

char &replace(int i)
{
    return s[i];
}
```

A horizontal line spans the width of the slide. On the left side, a large black opening square bracket '[' is positioned above the line. On the right side, a large yellow closing square bracket ']' is positioned above the line.

**CONSTRUCCIÓN,  
DESTRUCCIÓN Y COPIA**

# [Constructores]

- La necesidad de inicializar los objetos da lugar a los constructores.
- Son funciones miembro que se llaman igual que la clase.
- No retornan nada.
- Se invocan automáticamente cuando un objeto se crea (en la definición).





# [Constructores

```
class stack
{
    int stck[SIZE];
    int tos;
public:
    stack(); // constructor
    void push(int i);
    int pop();
};
```

```
// stack's constructor function
stack::stack()
{
    tos = 0;
    cout << "Stack Initialized\n";
}
```

# [Constructores]

- Se pueden declarar constructores con parámetros.
- El caso en que tiene exactamente 1 parámetro es especial, pues se puede usar como inicialización.

# [Constructores]

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

class myclass {
    int a, b;
public:
    myclass(int i, int j) {a=i; b=j;}
    void show() {cout << a << " " << b;}
};

int main(){
    myclass ob(3, 5);
    ob.show();
    return 0;
}
```

# [Constructores]

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

class X {
    int a;
public:
    X(int j) { a = j; }
    int geta() { return a; }
};

int main(){
    X ob = 99; // passes 99 to j
    cout << ob.geta(); // outputs 99
    return 0;
}
```

# [ destructores ]

- Son el complemento de los constructores.
- Se invocan automáticamente cuando se destruye un objeto.
- Sirven para liberar memoria o cerrar archivos.
- No retornan nada.
- Se declaran como los constructores, pero precedidos por un ~.



# [ Destructores

```
class stack
{
    int stck[SIZE];
    int tos;
public:
    stack(); // constructor
    ~stack(); // destructor
    void push(int i);
    int pop();
};
```

```
// stack's destructor function
stack::~~stack()
{
    cout << "Stack Destroyed\n";
}
```

# [ Constructor de copia ]

- Es un constructor especial que recibe una referencia constante.
- Se utiliza cada vez que se necesita crear un objeto que será una copia.

```
classname (const classname &o) {  
    // body of constructor  
}
```



# [ Constructor de copia ]

```
class myclass{
    int i;
public:
    myclass(int v1,int v2): i(v1),j(v2){};
    myclass(const myclass &o){i=o.i; j=o.j};
    int j;
};

void func1(myclass p){
    p.j=0;
};

myclass func2(){
    return myclass(0,0);
};

main(){
    myclass y(1,1);
    myclass x = y; // y explicitly initializing x
    func1(y); // y passed as a parameter
    y = func2(); // y receiving a temporary, return object
}
```



# [ Operadores ]

- Los operadores se pueden sobrecargar.
- Entre otros, se pueden sobrecargar los operadores:
  - =
  - +, -, \*, /
  - ++, --
  - [], etc
- Un caso util es el del operador =.

# [ Operadores ]

```
class myclass{
    int i;
public:
    myclass(int v1,int v2): i(v1),j(v2){};
    myclass(const myclass &o){i=o.i; j=o.j};
    myclass &operator=(const myclass &o) {i=o.i;
    j=o.j};
    int j;
};

main(){
    myclass y(1,1),x(0,0);
    x = y; //invoca al operador =
    myclass z(y); //invoca al constructor de copia
    myclass w = y; //invoca al constructor de copia
}
```



Petición y liberación de

**MEMORIA**

# [Funciones para manejar memoria]

---

- Además de las funciones de C malloc y free, se agregan 4 nuevas funciones:
  - new
  - delete
  - new []
  - delete []

# [ Operador new ]

- Pide la memoria necesaria para el objeto.
- Invoca al constructor.
- Retorna un puntero a la memoria pedida

```
int main() {  
    int *p;  
    p = new int; // allocate space for an int  
    *p = 100;  
    cout << "At " << p << " ";  
    cout << "is the value " << *p << "\n";  
    return 0;  
}
```

# [ Operador new ]

```
class myclass() {  
public:  
    int i;  
    myclass(int j):i(j){};  
};  
  
int main() {  
    myclass *p;  
    p = new myclass(8); // allocate space for a myclass  
    return 0;  
}
```

# [ Operador delete ]

- Libera la memoria pedida con new

```
int main() {  
    int *p;  
    p = new int; // allocate space for an int  
    *p = 100;  
    cout << "At " << p << " ";  
    cout << "is the value " << *p << "\n";  
    delete (p);  
    return 0;  
}
```

# [ Operador new[] ]

- Pide la memoria necesaria para un arreglo de objetos.
- Invoca a todos los constructores vacios.
- Retorna un puntero a la memoria pedida

```
int main() {  
    int *p;  
    p = new int[10]; // allocate space for an int[10]  
    P[2] = 100;  
    cout << "At " << p << " ";  
    cout << "is the value " << p[2] << "\n";  
    return 0;  
}
```



# [ Operador new[] ]

```
class myclass() {  
public:  
    int i;  
    myclass(int j):i(j){};  
};  
  
int main() {  
    myclass *p;  
    p = new myclass[8]; // allocate space for a 8 myclass  
    return 0;  
}
```

# [ Operador delete [] ]

- Libera la memoria pedida con new []

```
int main() {  
    int *p;  
    p = new int[8]; // allocate space for 8 int  
    p[2] = 100;  
    cout << "At " << p << " ";  
    cout << "is the value " << p[2] << "\n";  
    delete [] (p);  
    return 0;  
}
```