



UNIVERSIDAD DE COSTA RICA

PROGRAMACIÓN ORIENTADA A OBJETOS EN C++

Prof. Marlon Brenes y Prof. Federico Muñoz
Escuela de Física, Universidad de Costa Rica

Introducción

- La programación orientada a objetos (OOP) es un modelo de programación computacional que organiza el diseño de software alrededor de los **datos**, en lugar de funciones y lógica funcional
 - ↳ • Un objeto puede ser definido como un **campo de datos** que contiene ciertos **atributos** y **comportamiento**
- El enfoque en OOP se refiere a la manipulación directa de estos objetos, en lugar de enfocarse en la parte lógica
 - ↳ • Este modelo se utiliza para proyectos grandes y complejos, que requieren mantenimiento continuo
 - e.g., programas para manufactura y diseño, aplicaciones para dispositivos móviles, proyectos científicos abiertos a la comunidad, etc.

Introducción

- Una ventaja de la OOP es su beneficio con respecto a proyectos colaborativos
 - ↳ • El proyecto se puede dividir en grupos con enfoque en unidades e interoperabilidad
 - Cada unidad se encapsula en **clases**
- Otras ventajas incluyen
 - ↳ • Reusabilidad
 - Escalabilidad
 - Eficiencia

Introducción

- El primer ejercicio en el diseño de una aplicación bajo el modelo OOP is coleccionar todos los objetos que se deben manipular e identificar como se conectan el uno con el otro
 - ↳ • Dicho procedimiento se conoce como **modelado de datos**
 - e.g., un objeto puede ser una persona.
 - Dicha persona tiene **atributos**, tales como nombre, tamaño, peso...
 - La persona opera mediante **métodos**, tales como caminar, moverse...
- Una vez identificado un objeto, se clasifica dentro de una **clase**
 - ↳ • Una clase es una prescripción - una firma que identifica como un objeto puede ser creado, copiado y manipulado

Estructura en OOP

- **Clases:** es un tipo de datos definido por el usuario y actúa como una prescripción para la creación de objetos que contiene **atributos** y **métodos**
- **Objetos:** son instancias de una clase, contruidos con datos específicamente definidos. Una clase es la declaración, un objeto existe al momento de la **construcción**
- **Métodos:** son funciones definidas dentro de la clase que describen el comportamiento del objeto. El modelo de programación se basa en la reusabilidad de estos métodos
- **Atributos:** datos que contienen información requerida por el objeto para su clasificación

Principios de la OOP

- **Encapsulación:** La información importante está contenida dentro del objeto. Solamente información selecta se expone. La implementación y estados de un objeto se mantienen de forma privada dentro de la clase. Solo los atributos y métodos **públicos** son accesibles
- **Abstracción:** Los objetos solo revelan mecanismos internos relevantes para el uso de otros objetos. La información interna de operatividad innecesaria se esconde al usuario para prevenir su uso incorrecto
- **Herencia:** Las clases pueden utilizar código de otras clases mediante el establecimiento de relaciones que se establecen entre ellas
- **Polimorfismo:** Los objetos son diseñados para compartir cierto comportamiento y pueden tomar más de una forma. El programa determina cual significado y uso es necesario al momento de crear un objeto. El polimorfismo permite la existencia de objetos de diferentes tipos mediante las mismas interfaces

Beneficios de la OOP

- **Modularidad:** La encapsulación permite que los objetos sean autocontenidos
- **Reutilización:** La herencia permite la reutilización de código existente
- **Productividad:** Se pueden construir nuevos programas mediante bibliotecas y código reutilizable
- **Flexibilidad:** El polimorfismo permite que una sola función se adapte a la clase en la cual está ubicada

OOP en C++

```
1 #include <iostream>
2
3 class Rectangle
4 {
5     public:
6         float width; // Atributos
7         float height;
8         float area(); // Declaración de una función miembro
9 };
10
11 float Rectangle::area() { // Implementación de la función miembro
12
13     return width * height;
14 }
15
16 int main() {
17
18     Rectangle a; // Construcción por defecto
19
20     a.width = 5.0; // Acceso y modifíco atributos
21     a.height = 2.0;
22
23     std::cout << a.area() << std::endl; // Invocación de la función miembro
24
25     return 0;
26 }
```

• rectangle.cpp

Miembros públicos y privados

- Atributos y miembros públicos son visibles fuera de la clase
- Atributos y miembros privados son visibles solamente dentro de la clase

```
1 #include <iostream>
2
3 class MyClass
4 {
5     public:
6         int a;
7     private:
8         int b;
9 };
10
11 int main(){
12
13     MyClass m;
14
15     m.a = 4; // Esto está bien
16     m.b = 3; // Error de compilación: b es privado
17
18     return 0;
19 }
```

- `private_and_public.cpp`

Punteros a objetos

```
1 #include <iostream>
2
3 class Rectangle
4 {
5     public:
6         float width; // Atributos
7         float height;
8         float area(); // Declaración de una función miembro
9 };
10
11 float Rectangle::area() { // Implementación de la función miembro
12
13     return width * height;
14 }
15
16 int main() {
17
18     Rectangle a, *b, *c; // "b" y "c" son punteros a objetos de tipo Rectangle
19
20     a.width = 5.0; // Acceso y modifíco atributos
21     a.height = 2.0;
22
23     c = &a; // "c" apunta a "a"
24
25     b = new Rectangle; // "b" apunta a un nuevo objeto creado en el heap
26                       // sus miembros se accesan con un operador distinto: ->
27     b->width = 2.0;
28     b->height = 4.0;
29
30     std::cout << a.area() << std::endl;
31     std::cout << b->area() << std::endl;
32
33     c->width = 12.0;
34     std::cout << a.area() << std::endl; // Modificar "c" modifica "a"
35
36     delete b; // "b" debe borrado del heap
37
38     return 0;
39 }
```

● rectangle_2.cpp

Referencias a objetos

```
1 #include <iostream>
2
3 class Rectangle
4 {
5     public:
6         float width; // Atributos
7         float height;
8         float area(); // Declaración de una función miembro
9 };
10
11 float Rectangle::area() { // Implementación de la función miembro
12
13     return width * height;
14 }
15
16 int main() {
17
18     Rectangle a;
19
20     Rectangle &c = a; // "c" es una referencia a "a"
21
22     a.width = 5.0; // Acceso y modifíco atributos
23     a.height = 2.0;
24
25     std::cout << a.area() << std::endl;
26
27     c.width = 12.0; // Modificar "c" modifica "a"
28                     // Nótese que usamos el operador "." en lugar de "->"
29     std::cout << a.area() << std::endl; // Modificar "c" modifica "a"
30
31     return 0;
32 }
```

● rectangle_3.cpp

Constructores y destructores de objetos

- El **constructor** es una función que ejecuta los pasos necesarios para **crear** un objeto de **manera correcta**
- El **destructor** es una función que ejecuta los pasos necesarios para **destruir** un objeto de **manera correcta**
- Si no se especifican de manera explícita, son creados por el compilador (default)
 - Dichos métodos ubican las variables miembros en memoria y borran estas variables de memoria
- El **constructor** es llamado cuando un objeto es **declarado**, mientras que el **destructor** es invocado cuando el objeto sale de **scope**
- Para **punteros a objetos**, el **constructor** es invocado al usar el operador **new**, mientras que el **destructor** se llama al momento de usar **delete**
- Es posible diseñar un **constructor personalizado**

Constructores y destructores de objetos

```
1 #include <iostream>
2
3 class Rectangle
4 {
5     public:
6         Rectangle(float a, float b); // Constructor personalizado
7         float width; // Atributos
8         float height;
9         float area(); // Declaración de una función miembro
10
11     private:
12         Rectangle(); // El constructor base puede ser escondido como
13                     // privado, de manera tal que no se pueda construir
14                     // un rectángulo sin dimensiones
15 };
16
17 Rectangle::Rectangle(float a, float b){
18     width = a;
19     height = b;
20 }
21
22 Rectangle::Rectangle() { // El constructor base se deja vacío
23
24 }
25
26 float Rectangle::area() { // Implementación de la función miembro
27
28     return width * height;
29 }
30
31 int main(){
32
33     Rectangle a(5.0, 2.0);
34     std::cout << a.area() << std::endl;
35
36     // Rectangle b; // Esto no compila, el constructor base es privado
37
38     return 0;
39 }
```

- rectangle_4.cpp

Clases que manejan recursos de memoria

- La situación es más complicada cuando nuestras clases hacen llamados a asignar memoria en el heap
 - ↳ • La clase debe ser diseñada de manera tal que el destructor libere la memoria del objeto para evitar *memory leaks*
- Analicemos el ejemplo: • `myvector.cpp`

Constructores copia (*copy constructor*)

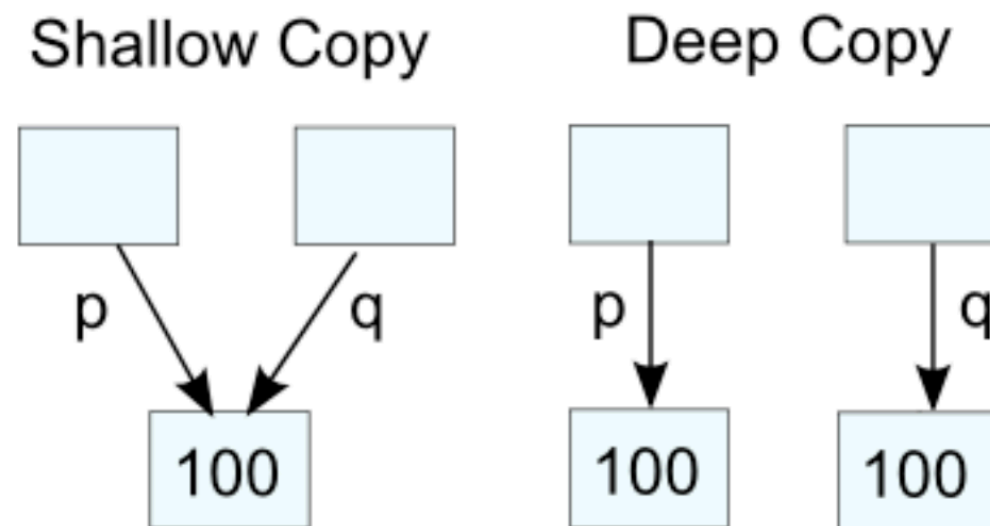
- Un constructor copia es un método (función) que crea un objeto mediante inicialización con otro objeto de la misma clase que ha sido creado previamente. Los constructores copia se utilizan para:
 - ↳ • Inicializar objetos del mismo tipo
 - Copiar un objeto para ser pasado como argumento de una función
 - Copiar un objeto para ser utilizado como valor de retorno en una función
 - Sintaxis: `MyClass(const MyClass &obj)`
- Al igual que el constructor y el destructor, el compilador genera un constructor copia por defecto si uno no está definido en el scope de la clase

Constructores copia (*copy constructor*)

- Analicemos los ejemplos:
 - `rectangle_5.cpp`
 - `rectangle_6.cpp`

Copias profundas vs copias superficiales (*deep and shallow copies*)

- Si la clase maneja un recurso mediante uno o más punteros, el constructor copia definido por el compilador (*default copy constructor*) no duplica la información a la cual el puntero apunta, solamente copia el puntero
 - A esto se le conoce como copia superficial (*shallow copy*)
- Una copia profunda (*deep copy*) se refiere a una completa duplicación, no solamente del puntero pero también a los datos a los cuales dicho puntero apunta



Copias profundas vs copias superficiales (*deep and shallow copies*)

- Analicemos los ejemplos:
 - `myvector_shallow.cpp`
 - `myvector_deep.cpp`

El puntero `this`

- Los objetos tienen implementados internamente un puntero que contiene la dirección de memoria del objeto como tal: es un puntero a la referencia del objeto (cuidado)
- Cada llamado a un miembro o a un atributo de la clase dentro del objeto como tal, son llamados equivalentes a realizar la operación con el puntero `this->` antepuesto
- Es muy útil, por ejemplo, para cuando necesitamos retornar un puntero o referencia al objeto en una función miembro
- Ejemplo: operador de asignación

Constructor copia (copy constructor) vs operador de asignación (assignment operator)

- Un constructor copia se utiliza para inicializar un objeto que está previamente sin inicializar, utilizando los datos de algún otro objeto del mismo tipo
- El operador de asignación se utiliza para reemplazar los datos de un objeto previamente inicializado con los datos de algún otro objeto
- Analicemos el ejemplo: • `rectangle_copy_vs_assignment.cpp`

La regla de tres

- Si una clase define uno o más de los siguientes miembros, MUY probablemente deba definir todos los tres:
 - └─• Destructor
 - Constructor copia
 - Operador de asignación

Archivos *header* (cabeceros) y *source* (fuente)

- Esta estructura permite ordenar el código de manera concisa para mantener modularidad
 - └─• La estructura se basa en crear un archivo header (usualmente con terminación `.hpp`) en el cual yace la declaración de la clase
 - Este archivo va acompañado de un archivo source (usualmente con terminación `.cpp`) donde yace la implementación de la clase
- Analicemos el ejemplo:
 - `header/*`
- Para compilar:
 - `g++ -c myvector.cpp`
 - `g++ -c main.cpp`
 - `g++ myvector.o main.o -o myvector.x`

Herencia (*inheritance*)

- Herencia es la construcción lógica en OOP en la cual una nueva clase es definida con base en la declaración de otra clase
 - ↳
 - La lógica que se implementa es la lógica “es un”
 - Un perro “es un” animal, una casa “es un” edificio...
- La declaración de un miembro protegido (`protected`) permite establecer miembros y métodos que son visibles solamente dentro de la clase y sus clases derivadas, mientras que siguen siendo invisibles fuera de la clase
- Analicemos el ejemplo: • `inheritance/*`

Herencia (*inheritance*)

- Si no se especifica, el constructor de una clase derivada invoca el constructor por defecto de su pariente
- El destructor de una clase derivada siempre invoca el destructor de su pariente
- Los constructores se invocan en el orden: pariente \rightarrow derivada
- Los destructores se invocan en el orden: derivada \rightarrow pariente