UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería del Diseño

DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Práctica 6
Programación Orientada a Objetos en C++

Sistemas Informáticos Industriales (SII 12164)

2019 – 2020

Contenido

Introducción

En esta práctica definiremos clases de objetos en C++ para representar el estado y el comportamiento de los componentes del proyecto de informatización industrial

Objetivos

1. Conocer el paradigma de programación orientada a objetos y descubrir sus ventajas
2. Programar orientado a objetos en C++
3. Definir clases en C++
4. Instanciar objetos
5. Derivar clases mediante el mecanismo de herencia
6. Aprovechar las capacidades de C++ relacionadas con el polimorfismo

Conceptos

POO - Programación Orientada a Objetos

- Se trata de un paradigma de programación muy utilizado
- Ofrece ventajas para la programación de sistemas de tamaño mediano y grande, ya que permite un mayor nivel de abstracción que la programación modular, conseguido éste mediante la ocultación de detalles y la definición de interfaces
- Características Clave:
  - Encapsulamiento - Empaquetar y ocultar lo que no es necesario mostrar
  - Herencia - Aprovechar el diseño previo general, añadir características más específicas
  - Polimorfismo - Obtener comportamiento diferente según el contexto
- Hay lenguajes de programación que soportan la POO y casan bien con el paradigma de diseño de sistemas orientado a objetos DOO. C++ entre ellos

(*) En esta práctica veremos la sintaxis de C++ relacionada con la POO, y en ese recorrido iremos presentando y discutiendo los conceptos del paradigma

POO en C++

Encapsulamiento

POO en C++

Conceptos de Clase y Objeto

Clase

- Define un modelo que puede ser utilizado para construir objetos
- El modelo encapsula ***datos*** y ***funciones*** en una misma estructura
- Los miembros de la estructura permiten representar:
  - El estado del objeto (*datos*)
  - Su comportamiento (*funciones* – que en este contexto son llamadas **métodos**)

Objeto

- Es la instancia de una clase - se construye siguiendo el modelo de la clase
- Pueden instanciarse múltiples objetos de una misma clase:
  - Cada objeto, en cada momento, tiene su propio estado
  - Todos los objetos de una misma clase comparten el mismo tipo de comportamiento

(*) Una analogía con los Tipos y Variables en *C*…

- La Clase sería como un Tipo de Datos
- El Objeto sería como una Variable

Definición de una Clase

Se utiliza la palabra reservada **class**, **struct** o **union**

```
class identificador_clase {

  especificador_de_acceso_1:
    lista_de_miembros_1;
    ...

  especificador_de_acceso_2:
    lista_de_miembros_2;
    ...

  ...
  } lista_de_identificadores_de_objetos;
```

Definición de una Clase – Especificadores de Acceso (1-2)

- Los especificadores de acceso indican qué partes del código tienen derecho a acceder a los miembros de la clase
- Los especificadores pueden ser:
  - **private:**
  - **protected:**
  - **public:**
- Cada uno de estos especificadores puede escribirse tantas veces como sea necesario y en cualquier orden dentro del bloque de definición de la clase **{ … };**
- Cada especificador afecta a los derechos de acceso a los miembros que se declaren a continuación del mismo, y tiene validez hasta que se especifique un derecho de acceso distinto, o se alcance el fin del bloque de definición de la clase **};**

Definición de una Clase – Especificadores de Acceso (2-2)

private:

- Los miembros declarados a continuación de private: son *privados*
- Los miembros *privados* sólo pueden ser accedidos por los otros miembros de la misma clase o por los miembros de las **Clases Amigas**

protected:

- Los miembros declarados a continuación de protected: son *protegidos*
- Los miembros *protegidos* sólo pueden ser accedidos por los otros miembros de la misma clase, por los miembros de las **Clases Amigas**, o por los miembros de las **Clases Derivadas**

public:

- Los miembros declarados a continuación de public: son *públicos*
- Los miembros *públicos* pueden ser accedidos desde cualquier parte del código donde el objeto sea visible

(*) en ciertos entornos de programación pueden encontrarse otros especificadores de acceso, como el especificador **_published** en los componentes *VCL* de *C++Builder*, o el especificador **public slots** en *Qt*, que sin embargo, no son parte del estándar de *C++*

Definición de una Clase – Cabecera Sencilla

- La cabecera más sencilla en una definición de clase utiliza la palabra reservada **class**, **struct** o **union** seguida del identificador de la clase que se está definiendo
- El identificador de la clase debe ser un identificador válido de *C++*
- Dependiendo de la palabra reservada utilizada en la definición de la clase, los derechos de acceso por defecto a los miembros de la clase son diferentes:

  class identificador_de_clase {...
    - Los miembros de la clase son, por defecto, privados

  struct identificador_de_clase {...
    - Los miembros de la clase son, por defecto, públicos

  union identificador_de_clase {...
    - Los miembros de la clase son, por defecto, públicos

Definición de una Clase – con Declaración de Objetos

En la misma sentencia de definición de una clase pueden declararse objetos

```
...
} lista_de_identificadores_de_objetos;
```

lista_de_identificadores_de_objetos es una lista de identificadores válidos en *C++*, opcional, que instancia objetos de la clase al mismo tiempo que se define la clase

Un primer Ejemplo de Clase – CRectangulo

```
class CRectangulo {
   int base, altura;
public:
   void establece_dimensiones(int b, int a);
   int area(void);
} rectangulo;
```

- Define una clase llamada **CRectangulo**
- Simultáneamente, declara un objeto de dicha clase llamado **rectangulo**

- La relación entre **CRectangulo** y **rectangulo** es análoga a la existente entre **int** y **x** en la siguiente declaración:

  int x;

CRectangulo – Miembros de la Clase

Las instancias de **CRectangulo** son representaciones de rectángulos, con sus características, y con los métodos que permiten inspeccionar o modificar dichas características

```
class CRectangulo {
   int base, altura;
public:
   void establece_dimensiones(int b, int a);
   int area(void);
} rectangulo;
```

- En la clase **CRectangulo** se han definido cuatro miembros:
  - dos ***datos miembro***, de tipo entero (int) – **base** y **altura**
  - dos ***funciones miembro*** – **establece_dimensiones()** y **area()**
- Los ***datos miembro*** permiten representar el estado de los objetos **CRectangulo** – en este caso sus dimensiones
- Las ***funciones miembro*** permiten modificar dicho estado y/o inspeccionarlo – en este caso se pueden modificar sus dimensiones y se puede preguntar por su área

CRectangulo – Derechos de Acceso a los Miembros de la Clase

```
class CRectangulo {
   int base, altura;
public:
   void establece_dimensiones(int b, int a);
   int area(void);
} rectangulo;
```

- Los miembros **base** y **altura** son ***privados***, ya que es la opción por defecto de una clase definida con la palabra reservada **class**, y los derechos de acceso por defecto no se han modificado explícitamente

- Los miembros **establece_dimensiones()** y **area()** sin embargo, son ***públicos***, ya que los derechos de acceso se han modificado explícitamente al escribir el especificador de acceso **public:** antes de la declaración de dichas funciones

- Observad que, de momento, sólo se ha ***declarado*** **establece_dimensiones()** y **area()**, ***sin definir*** todavía dichas funciones

CRectangulo – Referencia a los Miembros de la Clase

- Dentro del ámbito de un objeto, sus miembros, tanto ***datos*** como ***funciones***, se referencian directamente con sus identificadores
- Desde fuera del ámbito del objeto, se utiliza el **Operador Punto .** para referenciar a sus miembros: **`objeto.miembro`**

Ejemplo:

```
rectangulo.establece_dimensiones(3 ,2);
area = rectangulo.area();
```

(*) los miembros privados de **CRectangulo**, **base** y **altura**,
no pueden referenciarse desde fuera del ámbito del propio objeto

Podrían sin embargo referenciarse desde una **Clase Amiga** de **CRectangulo**

CRectangulo – Definición de los Métodos de la Clase

```
class CRectangulo {
   int base, altura;
public:
   void establece_dimensiones(int b, int a);
   int area() {return base*altura;}
};

void CRectangulo::establece_dimensiones(int b, int a) {
   base   = b;
   altura = a;
}
```

CRectangulo – Ejemplo Completo

```
#include <iostream>
using namespace std;
class CRectangulo{
  int base, altura;
public:
  void establece_dimensiones(int b, int a);
  int area() {return base*altura;}};

void CRectangulo::establece_dimensiones(int b, int a){
  base   = b;
  altura = a;}

int main(){
  CRectangulo rectangulo;
  rectangulo.establece_dimensiones(3, 2);
  cout << "área = " << rectangulo.area();
  return 0;}
```

área = 6

Métodos Inline, Operador Ámbito ::

- La definición del método **area()** se ha realizado dentro del bloque de definición de la clase **CRectangulo**
- **area()** accede a los miembros privados **base** y **altura** de **CRectangulo** sencillamente con sus identificadores, ya que la definición de la función **area()** se realiza en el ámbito de la clase **CRectangulo**

- La definición del método **establece_dimensiones()** sin embargo, se ha realizado fuera del bloque de definición de la clase **CRectangulo**. Para indicar que esta función también forma parte del ámbito de la clase **CRectangulo**, y no del ámbito de funciones globales, se utiliza el **Operador Ámbito ::**

- El comportamiento de un método es el mismo, aunque se defina dentro del bloque de definición de la clase o fuera de éste
- Sin embargo, un método definido dentro del bloque de definición de la clase se considera como una función **inline**. Las funciones **inline no se invocan** como las funciones regulares (no-inline) de *C++*, sino que **se copian** como bloques de sentencias en el punto de llamada

Declaraciones de Múltiples Objetos de la misma Clase

```
#include <iostream>
using namespace std;
class CRectangulo{
   int base, altura;
public:
   void establece_dimensiones(int b, int a);
   int area() {return base*altura;}};

void CRectangulo::establece_dimensiones(int b, int a){
   base   = b;
   altura = a;}

int main(){
   CRectangulo rectangulo, rectangulo2;
   rectangulo.establece_dimensiones(3, 2);
   rectangulo2.establece_dimensiones(5, 4);
   cout << "área = " << rectangulo.area() << endl;
   cout << "área2 = " << rectangulo2.area();          return 0;}
```

área = 6

área2 = 20

POO - Ventajas de las Clases y los Objetos

- Aunque los datos y los métodos se definen a nivel de la clase, cada objeto tiene sus propios datos que representan su estado particular, y los métodos se invocan para un objeto concreto, con lo que no necesitan definir muchos parámetros, pues en general, operan sobre los datos del objeto para el que se invoca el método

- Los datos pueden protegerse, convirtiéndolos en protegidos o privados, de forma que sólo puedan accederse a través de métodos de acceso, que potencialmente pueden tener mayor control sobre el uso del dato que si éste se accediese de forma directa

- Conceptualmente es más intuitivo encapsular los datos y los comportamientos juntos que tener conjuntos de datos y conjuntos de funciones aparentemente desligados

Constructores y Destructores

Los objetos generalmente necesitan inicializar su estado (sus miembros de datos)

Ejemplo:

Si se declarase un objeto **rectangulo** de la clase **CRectangulo**, y antes de invocar **rectangulo.establece_dimensiones()** se invocase **rectangulo.area()**, el valor de área devuelto sería un valor indeterminado

Constructor

- Un constructor es un método especial que se invoca automáticamente cuando se declara el objeto y puede utilizarse para inicializar su estado. También puede utilizarse para reservar espacio de almacenamiento dinámico si el objeto lo requiriese
- El método constructor tiene el mismo identificador que su clase y es una función sin tipo; ni siquiera hay que declararlo como tipo **void**
- Un constructor no se puede invocar explícitamente como el resto de métodos; sólo se invoca una vez durante la declaración (construcción) del objeto

CRectangulo – Constructor

```
#include <iostream>

using namespace std;


class CRectangulo{

  int base, altura;

public:

  CRectangulo(int b, int a);

  int area() {return base*altura;}};


CRectangulo::CRectangulo(int a, int b){

  base   = b;

  altura = a;}


int main(){

  CRectangulo rectangulo(3, 2), rectangulo2(5, 4);

  cout << "área = " << rectangulo.area() << endl;

  cout << "área2 = " << rectangulo2.area();

  return 0;}
```

área = 6

área2 = 20

CRectangulo – Constructor + Métodos de Acceso

```
#include <iostream>

using namespace std;


class CRectangulo{

  int base, altura;

public:

  CRectangulo(int b, int a);

  void establece_dimensiones(int b, int a);

  int area() {return base*altura;}};


CRectangulo::CRectangulo(int a, int b){

  base   = b;

  altura = a;}


void CRectangulo::establece_dimensiones(int b, int a){

  base   = b;

  altura = a;}
```

Destructores

- Un objeto se destruye cuando el hilo de ejecución sale del ámbito donde el objeto se declaró (construyó)
  - Si el objeto pertenece al ámbito global, se destruye cuando se termina el proceso
  - También se destruye con el operador **delete** si es un objeto creado dinámicamente
- Los objetos pueden necesitar realizar algunas operaciones antes de destruirse

Ejemplo:

Si se declarase un objeto de una clase en la que alguno de los datos miembro fueran declarados sobre la memoria dinámica, antes de destruir el objeto habría que recuperar (desasignar) el espacio de la memoria dinámica que se había reservado para dichos datos miembro

Destructor

- Un destructor es un método especial que se invoca automáticamente inmediatamente antes de destruir el objeto y puede utilizarse para realizar todas aquellas operaciones que el objeto debería realizar antes de destruirse
- El método destructor tiene el mismo identificador que su clase precedido del símbolo **~**. Es una función sin tipo y sin parámetros

CRectangulo con Miembros de Datos Dinámicos – Destructor

```
#include <iostream>

using namespace std;

class CRectangulo{

  int *base, *altura;

public:

  CRectangulo(int b, int a);

  ~CRectangulo();

  int area() {return *base * *altura;}};


CRectangulo::CRectangulo(int a, int b){

  base   = new int;

  altura = new int;

  *base   = b;

  *altura = a;}


CRectangulo::~CRectangulo(){

  delete base;

  delete altura;}
```

Sobrecarga de Constructores

Como otras funciones, los constructores se pueden sobrecargar

Sobrecarga

- Un mismo identificador se utiliza para más de una función
- El compilador determina qué función invocar según el número, orden y tipo de los parámetros que se utilicen en la llamada

CRectangulo – Constructores Sobrecargados (1/2)

```
#include <iostream>

using namespace std;


class CRectangulo{

  int base, altura;

public:

  CRectangulo();

  CRectangulo(int b, int a);

  int area() {return base*altura;}};


CRectangulo::CRectangulo(){

  base   = 3;

  altura = 3;}


CRectangulo::CRectangulo(int a, int b){

  base   = b;

  altura = a;}
```

CRectangulo – Constructores Sobrecargados (2/2)

```
int main()
{
  CRectangulo rectangulo(), rectangulo2(5, 4);
  cout << "área = " << rectangulo.area() << endl;
  cout << "área2 = " << rectangulo2.area();
  return 0;
}
```

```
área = 9
área2 = 20
```

CRectangulo – Constructor por Defecto (1/2)

```
#include <iostream>

using namespace std;


class CRectangulo{

  int base, altura;

public:

  CRectangulo();

  CRectangulo(int b, int a);

  int area() {return base*altura;}};


CRectangulo::CRectangulo(){

  base   = 3;

  altura = 3;}


CRectangulo::CRectangulo(int a, int b){

  base   = b;

  altura = a;}
```

CRectangulo – Constructor por Defecto (2/2)

```
int main()
{
  CRectangulo rectangulo, rectangulo2(5, 4);
  cout << "área = " << rectangulo.area() << endl;
  cout << "área2 = " << rectangulo2.area();
  return 0;
}
```

```
área = 9
área2 = 20
```

Sobre los Constructores por Defecto (1/2)

- Si no se define ningún constructor, C++ define un constructor por defecto que no tiene parámetros, y se pueden declarar objetos sin la notación paréntesis
- Sin embargo, si se define algún constructor, entonces los objetos se deben declarar utilizando alguno de los prototipos de los constructores definidos

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


class CRectangulo{
  int base, altura;
public:
  void establece_dimensiones(int b, int a);
  int area() {return base*altura;}};


void CRectangulo::establece_dimensiones(int b, int a){
  base = b; altura = a;}
```

Sobre los Constructores por Defecto (2/2)

- Si no se define ningún constructor, C++ define un constructor por defecto que no tiene parámetros, y se pueden declarar objetos sin la notación paréntesis
- Sin embargo, si se define algún constructor, entonces los objetos se deben declarar utilizando alguno de los prototipos de los constructores definidos

```
int main()
{
  CRectangulo rectangulo;
  rectangulo.establece_dimensiones(3, 2);
  cout << "área = " << rectangulo.area() << endl;
return 0;
}
```

```
área = 6
```

Constructores por Defecto – Copia y Operador de Asignación

Si *C++* crea un *Constructor por Defecto*, entonces crea también un *Constructor de Copia* y un *Constructor de Copia de Operador de Asignación*

```
int main() {
  CRectangulo rectangulo;
  rectangulo.establece_dimensiones(3, 2);
  CRectangulo rectangulo2(rectangulo); //Copia
  CRectangulo rectangulo3;
  rectangulo3 = rectangulo; //Asignación
  cout << "área = " << rectangulo.area() << endl;
  cout << "área = " << rectangulo2.area() << endl;
  cout << "área = " << rectangulo3.area();
  return 0;}
```

```
área = 6
área = 6
área = 6
```

Punteros a Clases

Puntero a Clase

- Igual que los punteros a cualquier otro tipo
- Los miembros de un objeto accedido a través de un puntero se acceden con el operador **Indirección ->**

Recordar que…

- *x apuntado por x
- &x dirección de x
- x.y miembro y del objeto x
- x->y miembro y del objeto apuntado por x
- (*x).y miembro y del objeto apuntado por x (*como el anterior*)
- x[0] primer objeto apuntado por x
- x[1] segundo objeto apuntado por x
- x[n] (n+1)ésimo objeto apuntado por x

CRectangulo – Punteros a la Clase

```
int main(){
    CRectangulo a, *b, *c;
    CRectangulo *d = new CRectangulo[2];
    b = new CRectangulo;
    c = &a;
a.establece_dimensiones(1, 2);
b->establece_dimensiones(3, 4);
d->establece_dimensiones(5, 6);
d[1].establece_dimensiones(7, 8);
    cout << "área a = " << a.area() << endl;
    cout << "área *b = " << b->area() << endl;
    cout << "área *c = " << c->area() << endl;
    cout << "área d[0] = " << d[0].area() << endl;
    cout << "área d[1] = " << d[1].area();
  delete[] d;
  delete b;
  return 0;}
```

```
área a = 2
área *b = 12
área *c = 2
área d[0] = 30
área d[1] = 56
```

Clases definidas como struct o union

En *C++*, tanto **class**, **struct** como **union** pueden definir clases de objetos, encapsulando juntos *datos* y *métodos*

- **class** y **struct** son equivalentes, a excepción del especificador de acceso por defecto:
  - class – privado
  - struct – publico
- **union** encapsula sólo uno de los datos de la unión junto con los métodos. Su especificador de acceso por defecto es público. No puede tener clases base ni funciones virtuales

union

- Una unión es un tipo de datos derivado; como una estructura
- Los miembros de una unión comparten espacio de almacenamiento
- El tamaño de la unión en memoria es suficiente para contener el miembro más grande
- En momentos diferentes el miembro de la unión que se utiliza es uno u otro

Ejemplo

```
union Numero{
  int entero;
  float real;}num;

num.entero = 10;
cout << num.entero << endl;

num.real = 3.14;
cout << num.real;
```

```
10
3.14
```

Sobrecarga de Operadores

- Se pueden definir operadores que operan sobre clases
- Se dice que se sobrecargan operadores porque se utilizan los mismos símbolos de los operadores estándar de C++

Definición

- Para sobrecargar operadores y utilizarlos en clases, se definen ***funciones operador***
- Una función operador es una función normal, pero su identificador se define con la palabra reservada **operator** seguida del ***símbolo del operador***

```
tipo operator simbolo (parametros)
{
// ...
}
```

Operadores Sobrecargables

- + - * / % ++ --
- ~ & | ^ << >>
- == != <= >= < >
- ! && ||

- =
- += -= *= /= %=
- &= |= ^= <<= >>=

- [] () , ->* ->

- new new[] delete delete[]

Ejemplo Operador Sobrecargado – CVector + (1/3)

```
#include <iostream>
using namespace std;
class CVector{
public:
  int x, y;
  CVector(){x = 0; y = 0;}; //Constructor sin parámetros. Observar que es necesario para definir c en main
  CVector(int a, int b);
  CVector operator+(CVector);};

CVector::CVector(int a, int b){
  x = a;
  y = b;}

CVector CVector::operator+ (CVector v){
  CVector temporal;
  temporal.x = x + v.x;
  temporal.y = y + v.y;
  return temporal;}
```

Ejemplo Operador Sobrecargado – CVector + (2/3)

```
int main()
{
  CVector a(3, 1);
  CVector b(1, 2);
  CVector c; //Constructor sin parámetros que hemos definido
  c = a + b;
  cout << "(" << c.x << "," << c.y << ")";
  return 0;
}
```

```
(4,3)
```

Ejemplo Operador Sobrecargado – CVector + (3/3)

Se puede utilizar indistintamente la forma con el operador o la forma con la función:

```
int main()
{
  CVector a(3, 1);
  CVector b(1, 2);
  CVector c;

//  c = a + b;
  c = a.operator+(b);

  cout << "(" << c.x << "," << c.y << ")";
  return 0;
}
```

```
(4,3)
```

this

Palabra reservada **this**

Representa un puntero al objeto al que pertenece la función miembro donde aparece **this**

this – Ejemplo de Uso (1/2)

Cómo asegurarse de que un parámetro puntero que se le pasa a un método es realmente un puntero al objeto al que pertenece el método

```
#include <iostream>
using namespace std;
struct UnaClase{
  bool SoyYo(UnaClase& parametro);};

bool UnaClase::SoyYo(UnaClase& parametro){
  if (&parametro == this) return true;
  else return false;}

int main(){
  UnaClase a;
  UnaClase *b = &a;
  if(b->SoyYo(a)) cout << "Sí, &a es b";
  return 0;}
```

```
Sí, &a es b
```

this – Ejemplo de Uso (2/2)

Retornar **this** en miembros función **operator=** que devuelven objetos por referencia, evitando tener que usar objetos temporales

```
CVector& CVector::operator=(const CVector& v)
{
  x = v.x;
  y = v.y;
  return *this;
}
```

Miembros Estáticos

- Las clases pueden contener datos y funciones miembro estáticos

- Los **datos miembro estáticos**:
  - Se conocen como variables de clase, ya que sólo hay un valor para dicha variable que es compartido por todos los objetos de la clase
  - Tienen las mismas propiedades que las variables globales, pero están definidas en el ámbito de una clase
  - Se declaran en la definición de la clase, pero se inicializan explícitamente fuera de dicha definición, sólo una vez, utilizando el operador ámbito **::**
  - Se pueden referenciar para un objeto, o para la clase:
    - Objeto.Miembro_Estatico
    - Clase::Miembro_Estatico

- Las **funciones miembro estáticas**:
  - Se pueden referenciar para un objeto, o para la clase
  - Pueden acceder **sólo** a datos miembro estáticos de la clase

Miembros Estáticos – Ejemplo – Una clase con un contador de objetos instanciados

```
#include <iostream>
using namespace std;
class UnaClase{
public:
  static int NumeroObjetos;
  UnaClase(){ NumeroObjetos++; };
 ~UnaClase(){ NumeroObjetos--; };
};
int UnaClase::NumeroObjetos = 0;

int main(){
  UnaClase a;
  UnaClase b[5];
  UnaClase *c = new UnaClase;
  cout << a.NumeroObjetos << " ";
  delete c;
  cout << UnaClase::NumeroObjetos;
  return 0;}
```

```
7  6
```

Funciones Amigas

- Las **funciones amigas** de una clase son funciones externas a la definición de la clase que tienen permiso de acceso a los miembros de dicha clase, independientemente de que sean miembros privados o protegidos

- Se declaran en la definición de la clase con la palabra reservada **friend**

(*) Las funciones amigas se utilizan por conveniencia, para facilitar las operaciones entre clases, sin embargo, no casan completamente con el paradigma de la POO

Para seguir mejor el enfoque de la POO, deberían utilizarse métodos miembro, antes que funciones amigas, siempre que sea posible

Funciones Amigas – Ejemplo (1/2)

```
#include <iostream>

using namespace std;


class CRectangulo{

  int base, altura;

public:

  void establece_dimensiones(int b, int a);

  int area() {return base*altura;}

  friend CRectangulo doble_tamano(CRectangulo r);};


void CRectangulo::establece_dimensiones(int b, int a){

  base = b; altura = a;}


CRectangulo doble_tamano(CRectangulo r){

  CRectangulo rectangulo;

  rectangulo.base   = 2 * r.base;

  rectangulo.altura = 2 * r.altura;

  return rectangulo;}
```

Funciones Amigas – Ejemplo (2/2)

```
int main()
{
 CRectangulo ra, rb;
  ra.establece_dimensiones(2, 3);
  rb = doble_tamano(ra);
  cout << rb.area();
  return 0;
}
```

```
24
```

- **doble_tamano()** es una función amiga de la clase **CRectangulo**, y por tanto tiene acceso a los miembros privados **base** y **altura** de los objetos **CRectangulo**
- **doble_tamano()** se ha declarado en la definición de **CRectangulo** como **friend**
- En la definición de **doble_tamano()** no se ha utilizado el **ámbito CRectangulo**, ya que no es una función miembro de la clase. Tampoco se ha utilizado cuando se ha invocado a la función desde **main()**

Clases Amigas

- Las clases amigas de una clase son clases que tienen permiso de acceso a los miembros de esta, independientemente de que sean miembros privados o protegidos
- Se declaran en la definición de la clase con las palabras reservadas **friend class**

(*) La amistad entre clases **NO** tiene por qué ser **recíproca**, salvo que se establezca de forma explícita

Las amistades entre clases **TAMPOCO** se transmiten de forma **transitiva**, salvo que se establezcan de forma explícita

Clases Amigas – Ejemplo (1/2)

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

class CCuadrado;

class CRectangulo{

  int base, altura;

public:

  void establece_dimensiones(int b, int a);

  int area() {return base*altura;}

  void convertir_en_cuadrado(CCuadrado c);};

void CRectangulo::establece_dimensiones(int b, int a){

  base = b; altura = a;}


class CCuadrado{

  int lado;

public:

  void establece_dimension(int l) {lado = l;}

  friend class CRectangulo;};


void CRectangulo::convertir_en_cuadrado(CCuadrado c){

  base = altura = c.lado;}
```

Clases Amigas – Ejemplo (2/2)

```
int main(){
  CCuadrado   c;
  CRectangulo r;
  c.establece_dimension(4);
  r.convertir_en_cuadrado(c);
  cout << r.area();
  return 0;}
```

```
16
```

CRectangulo se ha declarado como amiga de **CCuadrado**, con lo que puede acceder al miembro privado **lado** de ésta

(*) Observad la **declaración parcial** de **CCuadrado** previa a la definición de **CRectangulo**, y su posterior definición completa. Es necesaria para poder declarar el parámetro del método **convertir_en_cuadrado()** de la clase **CRectangulo**

Herencia

POO en C++

Herencia

- Definición de clases a partir de otras clases predefinidas

- Dada una **Clase Base**, la **Clase Derivada** de la clase base, hereda los miembros de la clase base y añade otros miembros propios que son más específicos

- Se puede crear una jerarquía de clases de objetos

Ejemplo

- Clase Base: **CPoligono**
- Clases Derivadas: **CRectangulo** y **CTriangulo**

- Propiedades Comunes a todos los polígonos: **base** y **altura**
- Propiedades Específicas de cada clase de polígono: **area**

Herencia – Ejemplo CPoligono

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

class CPoligono{
protected:
   int base, altura;
public:
   void establece_valores(int b, int a){
      base = b; altura = a;}
};
```

```
class CRectangulo: public CPoligono{
public:
   int area(){
      return base * altura;}
};

class CTriangulo: public CPoligono{
public:
   int area(){
      return base * altura / 2;}
};
```

Herencia – Ejemplo CPoligono

```
int main(){
    CRectangulo rectangulo;
    CTriangulo  triangulo;
    rectangulo.establece_valores(4, 5);
    triangulo .establece_valores(4, 5);
    cout << rectangulo.area() << endl;
    cout << triangulo.area();
    return 0;
}
```

```
20
10
```

Herencia - Especificadores de Acceso

- Desde el punto de vista de la accesibilidad a los miembros de una clase desde el ámbito externo a la misma, los especificadores de acceso **protected** y **private** son similares. En ambos casos, los miembros son inaccesibles
- Sin embargo, estos especificadores de acceso son diferentes para las clases derivadas. Una clase derivada puede acceder a los miembros protegidos de la clase base, pero no a los miembros privados, que sólo pueden ser accedidos en el ámbito de la clase base

Acceso desde…	public	protected	private
miembros de la misma clase	sí	sí	sí
miembros de clases derivadas	sí	sí	no
no miembros	sí	no	no

En el ejemplo, **CTriangulo** y **CRectangulo** (**clases derivadas**), necesitan tener acceso a los miembros **base** y **altura** de la clase **CPoligono** (**clase base**). Por ese motivo, se declaran en el ámbito protegido. De esa forma, las clases derivadas tienen acceso a **base** y **altura**, a la vez que éstos siguen permaneciendo inaccesibles desde el ámbito externo

Herencia – Ejemplo CPoligono

```
class CPoligono{
protected:
   int base, altura;
...
```

```
class CRectangulo: public CPoligono{
public:
   int area(){
      return base * altura;}
};

class CTriangulo: public CPoligono{
public:
   int area(){
      return base * altura / 2;}
};
```

Herencia – Especificadores de Acceso de la Clase Derivada

```
class CBase{
public:
    int basePublic;
protected:
    int baseProtected;
private:
    int basePrivate;
};
```

Si no se especifica,

```
class CDerivadaPublic : CBase{…};
```

por defecto se supone que es **private**

```
class CDerivadaPublic: public CBase{
    // basePublic     es público
    // baseProtected  es protegido
    // basePrivate    no es accessible
};
class CDerivadaProtected: protected CBase{
    // basePublic     es protegido
    // baseProtected  es protegido
    // basePrivate    no es accessible
};
class CDerivadaPrivate: private CBase{
    // basePublic     es privado
    // baseProtected  es privado
    // basePrivate    no es accessible
};
```

Herencia – ¿Qué se hereda?

Una clase derivada hereda de su clase base todos sus datos miembro y sus funciones miembro, a excepción de:

- Su constructor y su destructor
- Sus operadores de asignación **=()**
- Sus clases amigas y sus funciones amigas

Herencia – Constructores

- Aunque los constructores y destructores de la clase base no se heredan, el constructor por defecto y el destructor, son llamados de forma implícita cuando se construyen y destruyen objetos de la clase derivada
- Si la clase base no define un constructor por defecto, o si se desea utilizar otro constructor sobrecargado de la clase base cuando se instancien objetos de la clase derivada, se puede declarar el constructor de la clase derivada con la notación:

```
constructor_derivado(parámetros) : constructor_base(parámetros){…};
```

Herencia – Constructores – Ejemplo (1-3)

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


class CBase{
public:
   CBase(){ cout << "CBase sin parámetros\n"; }
   CBase(int p){ cout << "CBase con parámetro int\n"; }
};
```

Herencia – Constructores – Ejemplo (2-3)

```
class CDerivada_1: public CBase{
public:
   CDerivada_1(int p){ cout << "CDerivada_1 con parámetro int\n"; }
};

class CDerivada_2: public CBase{
public:
   CDerivada_2(int p): CBase(p){ cout << "CDerivada_2 con parámetro int\n"; }
};
```

Herencia – Constructores – Ejemplo (3-3)

```
int main(){
   CDerivada_1 derivada_1(0);
   CDerivada_2 derivada_1(0);
   return 0;
}
```

```
CBase sin parámetros
CDerivada_1 con parámetro int
CBase con parámetro int
CDerivada_2 con parámetro int
```

Herencia Múltiple

- Una clase puede heredar de más de una clase base
- Para ello, en su declaración se debe especificar la lista (separada por comas) de sus clases base

Ejemplo:

```
class CDerivada: public CBase_1, public CBase_2{…};
```

Herencia Múltiple – Ejemplo (1-3)

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


class CPoligono{
protected:
   int base, altura;
public:
   void establece_valores(int b, int a){
      base = b; altura = a;}
};
class CSalida{
public:
   void mostrar(int i);
};
```

Herencia Múltiple – Ejemplo (2-3)

```
class CRectangulo: public CPoligono, public CSalida{
public:
   int area(){
      return base * altura;}
};

class CTriangulo: public CPoligono , public CSalida{
public:
   int area(){
      return base * altura / 2;}
};
```

Herencia Múltiple – Ejemplo (3-3)

```
int main(){
   CRectangulo rectangulo;
   CTriangulo  triangulo;
   rectangulo.establece_valores(4, 5);
   triangulo .establece_valores(4, 5);
   rectangulo.mostrar(rectangulo.area());
   triangulo .mostrar(triangulo .area());
   return 0;
}
```

```
20
10
```

Polimorfismo

POO en C++

Polimorfismo – Punteros a la Clase Derivada

- Una característica clave en C++ es que un puntero a una clase derivada es de tipo compatible con un puntero a su clase base
- El **Polimorfismo** en C++ consiste en aprovechar la compatibilidad anterior, y permite sacar todo el partido a la POO

Polimorfismo – Ejemplo (1-3)

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

class CPoligono{
protected: int base, altura;
public: void establece_valores(int b, int a){base = b; altura = a;}};

class CRectangulo: public CPoligono{
public: int area(){return base * altura;}};

class CTriangulo: public CPoligono{
public: int area(){return base * altura / 2;}};
```

Polimorfismo – Ejemplo (2-3)

```
int main(){
   CRectangulo rectangulo;
   CTriangulo  triangulo;
   CPoligono *pPoligono1 = &rectangulo;
   CPoligono *pPoligono2 = &triangulo;
   pPoligono1->establece_valores(4, 5);
   pPoligono2->establece_valores(4, 5);
   cout << rectangulo.area() << endl;
   cout << triangulo.area()  << endl;
   return 0;
}
```

```
20
10
```

Polimorfismo – Ejemplo (3-3)

- En la función ***main()*** del ejemplo, se crean dos punteros ***pPoligono_1*** y ***pPoligono_2*** a la clase ***CPoligono***, y se les asignan las direcciones de los objetos ***rectangulo*** y ***triangulo*** respectivamente
- Las asignaciones anteriores se pueden realizar porque los punteros a ***CRectangulo*** y ***CTriangulo*** son compatibles con los punteros a ***CPoligono***
- La única limitación de usar dichos punteros respecto de usar directamente ***rectangulo*** y ***triangulo***, es que sólo pueden ser utilizados para acceder a los miembros de la clase ***CPoligono*** que son heredados por ***CRectangulo*** y ***CTriangulo***
- Debido a dicha limitación, para acceder a la función ***area()*** se ha tenido que utilizar ***rectangulo*** y ***triangulo*** en lugar de los punteros, ya que ***CPoligono*** no define una función ***area()***

Para solucionar esta limitación, se pueden definir miembros **virtuales**

Polimorfismo – Miembros Virtuales

- Un miembro de una clase se considera **virtual**, si se puede redefinir en las clases derivadas
- Para que un miembro sea virtual se utiliza la palabra reservada **virtual** precediendo a su declaración

Polimorfismo – Miembros Virtuales – Ejemplo (1-3)

```
#include <iostream> using namespace std;
class CPoligono{
protected: int base, altura;
public:
   void establece_valores(int b, int a){base = b; altura = a;}
   virtual int area(){ return (0); }};

class CRectangulo: public CPoligono{
public: int area(){return base * altura;}};

class CTriangulo: public CPoligono{
public: int area(){return base * altura / 2;}};
```

Polimorfismo – Miembros Virtuales – Ejemplo (2-3)

```
int main(){
   CRectangulo rectangulo;
   CTriangulo  triangulo;
   CPoligono   poligono;
   CPoligono *pPoligono1 = &rectangulo;
   CPoligono *pPoligono2 = &triangulo;
   CPoligono *pPoligono3 = &poligono;
   pPoligono1->establece_valores(4,5);
   pPoligono2->establece_valores(4,5);
   pPoligono3->establece_valores(4,5);
   cout << pPoligono1->area() << endl;
   cout << pPoligono2->area() << endl;
   cout << pPoligono3->area() << endl;
   return 0;
}
```

```
20
10
0
```

Polimorfismo – Miembros Virtuales – Ejemplo (3-3)

- En este ejemplo, las tres clases ***CPoligono***, ***CRectangulo*** y ***CTriangulo*** tienen todas los mismos miembros: ***base***, ***altura***, ***establece_valores()*** y ***area()***
- La función miembro ***area()*** se ha declarado como ***virtual*** en la clase base

  Haced una prueba: eliminad la palabra ***virtual*** y ejecutad de nuevo el programa, y veréis que siempre obtenéis 0 como resultado (en los tres casos). Esto es debido a que ***area()*** se invoca a través de punteros a ***CPoligono***
- Así, la palabra ***virtual*** permite que un miembro de una clase derivada (cuyo nombre coincide con otro de su clase base) se pueda invocar utilizando un puntero a la clase base, pero que en realidad está apuntando a un objeto de la clase derivada
- Las clases que declaran o heredan funciones virtuales son llamadas ***clases polimórficas***
- Observad que a pesar de su virtualidad, en el ejemplo se ha podido declarar un objeto de la clase ***CPoligono*** e invocar su función ***area()***, la cual siempre retorna 0

Polimorfismo – Clases Abstractas

- Una clase abstracta es aquella en la que se declara al menos una función virtual que no se ha definido, pero que normalmente se definirá en sus clases derivadas
- Se expresan con la notación **=0** siguiendo la declaración de la función virtual, en lugar de facilitar su definición. Por ejemplo: `virtual int area()=0;`
- La diferencia entre una clase abstracta y una clase polimórfica regular es que no se pueden declarar objetos de la clase abstracta. Lo cual puede hacer pensar que no son útiles. Sin embargo, sí se pueden crear punteros a clases abstractas, y dichos punteros pueden apuntar a objetos de las clases derivadas

Polimorfismo – Clases Abstractas – Ejemplo (1-2)

```
#include <iostream> using namespace std;
class CPoligono{
protected: int base, altura;
public:
   void establece_valores(int b, int a){base = b; altura = a;}
   virtual int area()=0;};

class CRectangulo: public CPoligono{
public: int area(){return base * altura;}};

class CTriangulo: public CPoligono{
public: int area(){return base * altura / 2;}};
```

Polimorfismo – Clases Abstractas – Ejemplo (2-2)

```
int main(){
   CRectangulo rectangulo;
   CTriangulo  triangulo;
   CPoligono *pPoligono1 = &rectangulo;
   CPoligono *pPoligono2 = &triangulo;
   pPoligono1->establece_valores(4,5);
   pPoligono2->establece_valores(4,5);
   cout << pPoligono1->area() << endl;
   cout << pPoligono2->area() << endl;
   return 0;
}
```

```
20
10
```

Polimorfismo – Utilidad de las Clases Abstractas

- Observad que en el ejemplo anterior, nos hemos podido referir a objetos de clases distintas (**CRectangulo**, **CTriangulo**) utilizando punteros de un mismo tipo (puntero a **CPoligono**)
- La característica anterior es muy útil
- Considérese por ejemplo, la utilidad de poder definir una función en la clase base **CPoligono** que muestre el valor del área de cualquier polígono. Incluso sin tener que definir una función ***area()*** en la propia clase ***CPoligono***. Simplemente definiendo ***area()*** como virtual, y permitiendo que cada clase derivada (***CRectangulo***, ***CTriangulo***) implementen sus propias funciones ***area()***

Polimorfismo – Utilidad de las Clases Abstractas – Ejemplo (1-2)

```
#include <iostream> using namespace std;
class CPoligono{
protected: int base, altura;
public:
   void establece_valores(int b, int a){base = b; altura = a;}
   virtual int area()=0;
   void muestra_area(void){ cout << this->area() << endl; }};

class CRectangulo: public CPoligono{
public: int area(){return base * altura;}};

class CTriangulo: public CPoligono{
public: int area(){return base * altura / 2;}};
```

Polimorfismo – Utilidad de las Clases Abstractas – Ejemplo (2-2)

```
int main(){
   CRectangulo rectangulo;
   CTriangulo  triangulo;
   CPoligono *pPoligono1 = &rectangulo;
   CPoligono *pPoligono2 = &triangulo;
   pPoligono1->establece_valores(4,5);
   pPoligono2->establece_valores(4,5);
   pPoligono1->muestra_area();
   pPoligono2->muestra_area();
   return 0;
}
```

```
20
10
```

Polimorfismo – Utilidad de las Clases Abstractas

- Los miembros virtuales y las clases abstractas permiten aplicar toda la potencia de la programación orientada a objetos en C++, y abordar de este modo proyectos de gran tamaño
- Los ejemplos presentados aquí han sido ejemplos muy simples
- Estas características, sin embargo, podrían aplicarse a vectores de objetos y a objetos reservados dinámicamente en la memoria. Consideremos un último ejemplo…

Polimorfismo – Objetos Dinámicos – Ejemplo

```
#include <iostream> using namespace std;
class CPoligono{…} …
```

```
int main(){
   CPoligono *pPoligono1 = new CRectangulo;
   CPoligono *pPoligono2 = new CTriangulo;
   pPoligono1->establece_valores(4,5);
   pPoligono2->establece_valores(4,5);
   pPoligono1->muestra_area();
   pPoligono2->muestra_area();
   delete pPoligono_1;
   delete pPoligono_2;
   return 0;
}
```

```
20
10
```

Observad que los punteros son a la clase ***CPoligono***,
pero se ha reservado memoria
con el operador ***new***
para objetos ***CRectangulo*** y ***CTriangulo***

Actividades

Actividad 01 – Definición de una Clase

En una nueva aplicación de ventanas, definid una clase CPoligono siguiendo las siguientes indicaciones:

- Cread una nueva unidad “CPoligono.h/cpp” para esta clase
- Una propiedad de CPoligono debe ser un vector **lados**, de dimensión dinámica, que defina las longitudes de los lados del polígono
- En el **constructor** de CPoligono se debe indicar el número de lados deseado
- Un método **NumeroLados()** debe devolver el número de lados
- Un método **EstablecerLongitud(lado, longitud)** debe permitir establecer la longitud de uno de los lados del polígono
- Un método **BienDefinido()** debe devolver **true** si todos los lados del polígono han sido definidos
- Un método **Perimetro()** debe devolver el perímetro del polígono

Actividad 02 – Instanciación de Objetos

Sobre la ventana MainWindow, definid una interfaz de usuario que permita:

- Solicitar la construcción de un nuevo polígono, al presionar un **pushButton**, habiendo indicado el número de lados deseados en un **lineEdit**
- Definir la longitud de cada lado del polígono, al presionar un **pushButton**, habiendo indicado el identificador del lado y la longitud del mismo en dos **lineEdit**
- Mostrar las propiedades del polígono, al presionar un **pushButton**, visualizando las siguientes propiedades sobre un **plainTextEdit**
  - Número de lados del polígono
  - Lista de las longitudes de cada lado
  - Perímetro del polígono

Actividad 03 – Herencia y Polimorfismo

1. Derivad de CPoligono, dos clases **CRectangulo** y **CTriangulo**, para representar rectángulos y triángulos respectivamente, implementando constructores que permitan definir las longitudes de los lados. Cread dos nuevas unidades “CRectangulo.h/cpp” y “CTriangulo.h/cpp” para estas clases
2. Modificad la interfaz de usuario para poder indicar qué tipo de polígono (rectángulo o triángulo) se crea, y definir las longitudes de sus lados

Actividad del Proyecto

Actividad del Proyecto – Modelado Orientado a Objetos (1-2)

Objetivo

Aprender a analizar un problema de informatización industrial y modelarlo siguiendo el paradigma de orientación a objetos

Motivación

Antes de comenzar a programar la aplicación de control y definir sus variables y funciones, es conveniente realizar un análisis previo del problema, para poder identificar los objetos del mundo real que luego describiremos como objetos virtuales en el programa (como clases y objetos en C++)

Especificación

1. Analizad el problema que estéis considerando en el proyecto
2. Identificad los objetos de la instalación susceptibles de ser modelados como objetos C++ en la aplicación. Comenzad por los objetos físicos relacionados con el problema
3. Identificad las propiedades de los objetos reales, que se corresponderán con los datos miembro de las clases C++ a definir
4. Identificad las operaciones realizables sobre los objetos reales, las cuales se corresponderán con los métodos de las clases C++
5. Extended el diseño a objetos lógicos (no físicos) que también serán componentes necesarios de la aplicación

Actividad del Proyecto – Modelado Orientado a Objetos (2-2)

Sugerencia

- Centraros en sólo unos pocos objetos de la instalación, los que pienses que son más significativos, más que en tener todo un sistema completo
- Intentad identificar objetos que os permitan practicar su modelado utilizando tanto una descripción composicional (algunos miembros de la clase son objetos de otras clases) como una descripción por herencia (la clase es una clase derivada de otra clase base)
- Realizad diagramas de clases, aunque sean a mano alzada, para incorporarlos a la memoria. Hay herramientas para representar distintos tipos de diagramas relacionados con la programación. Considerad por ejemplo **Umbrello** (https://umbrello.kde.org/), o investigad otras herramientas con funcionalidad similar

Actividad del Proyecto – Programación Orientada a Objetos (1-2)

Objetivo

Aprender a utilizar clases y objetos C++ para programar una aplicación orientada a objetos

Especificación

1. Partid del modelo de objetos definido en la Actividad 1. Utilizadlo como una especificación de software
2. Definid las clases de objetos del modelo siguiendo la sintaxis de C++
   - Declarad las clases con sus datos y métodos miembro en archivos de cabecera ***.h**
   - Definid los métodos de dichas clases en archivos de implementación ***.cpp**
3. Programad una aplicación (por simplicidad, puede ser de momento una aplicación de consola) para declarar instancias de las clases definidas, y para poner a prueba los métodos implementados

Actividad del Proyecto – Programación Orientada a Objetos (2-2)

Sugerencia

- Utilizad identificadores siguiendo un enfoque sistemático, para que al leerlos sea más fácil predecir el tipo de los objetos o la funcionalidad de los métodos que están identificando
- Seguid un estilo coherente de escritura, con sangrados, separaciones…
- Comentad suficientemente el código
- Opcionalmente podéis utilizar una herramienta de documentación automática. Considerad por ejemplo **doxygen** (http://www.doxygen.org/), o investigad otras herramientas con funcionalidad similar

Criterios de Evaluación

Criterios de Evaluación

Práctica 6 – hasta 10 puntos

- Actividad 01 – 3.5ptos
- Actividad 02 – 3.5ptos
- Actividad 03 – 3ptos

UNIVERSITAT POLITÈCNICA DE VALÈNCIA
Escuela Técnica Superior de Ingeniería del Diseño
DISCA
DEPARTAMENTO DE INFORMÁTICA DE SISTEMAS Y COMPUTADORES

Práctica 6
Programación Orientada a Objetos en C++

Sistemas Informáticos Industriales (SII 12164)

2018 – 2019