**Instituto Superior de Formación Técnica Nº 151 logo151-trans.gif  
Carrera: Analista de Sistemas  
1 Año. Algoritmos y Estructuras de Datos I**

|  |  |
| --- | --- |
| **Trabajo Práctico Nº 10.2** | **Unidad 10.2** |
| **Modalidad:** Semi-Presencial | **Estratégica Didáctica:** Trabajo Individual |
| **Metodología de Desarrollo:** acordar | **Metodología de Corrección:** acordar docente |
| **Carácter de Trabajo:** Obligatorio – con Nota | **Fecha Entrega:** A confirmar por el Docente. |

**Paradigma Orientado a Objetos – Programación Orientada a Objetos**

**Marco Teórico:**

1. Dar un ejemplo de Creación de una clase y de instanciar un Objeto
2. Explicar los alcances de Visibilidad “*public, private o protected”*
3. ¿Qué se entiende por función Miembro de una clase?
4. ¿Qué diferencia hay entre implementar la función miembro dentro y fuera de la clase? Dar un ejemplo en código.
5. ¿Qué se entiende por constructor y que es el constructor por defecto?
6. ¿Para qué sirven las listas de Inicialización?
7. ¿Cuándo un Atributo es privado como le asigno valores y como puedo ver su estado?
8. ¿Qué se entiende por Atributo Estático? Dar un ejemplo.
9. ¿Qué se entiende por función miembro Estática? Dar un ejemplo.
10. ¿Qué utilidad tiene un Destructor? ¿Qué ocurre si no lo Escribo?
11. ¿Qué debería poner dentro de Un Destructor? Dar un ejemplo.
12. Explicar la cita de autor “*La mente humana clasifica los conceptos de acuerdo a dos dimensiones: Variedad y Pertenencia*”
13. Explicar el Concepto de “Generalización y Especialización” en Herencia.
14. ¿Qué es el constructor de Copia? ¿Para qué sirve?
15. Describir el concepto de “Jerarquía de Clases”
16. Explicar la diferencia entre “early binding” y “late binding”
17. Dar un ejemplo de código del punto anterior.
18. ¿Qué resuelven o permiten las Funciones Virtuales?
19. ¿Qué relación tienen los puntos anteriores con el “Principio de sustitución de Liskov”?
20. ¿A qué referencia hace el autor cuando dice “*Existe polimorfismo cuando interactúan la herencia y el enlace dinámico*”?

**Marco Practico:**

**Tener en Cuenta:**  
. Modularizar el Programa.  
. Proteger contra Inclusiones Múltiples.  
. Aplicar Espacios de Nombres.  
. Aplica apropiadamente los conceptos de abstracción, encapsulación y ocultamiento de información.  
. Realiza una apropiada distribución de responsabilidades entre las entidades del espacio de la solución.   
. Desarrolla para reusar.  
. Reúsa apropiadamente las entidades desarrolladas en el espacio curricular.  
. Demuestra un uso apropiado de la sintaxis y semántica del lenguaje de programación C++.

**Desarrollar un Programa** que:

1. *Desarrollar una “Solución”* que represente la siguiente estructura jerárquica.
   1. *Que realice la Herencia solicitada*
   2. *Que separe “Interfaz” de “Implementación” en código, modularizándolo.*
   3. *Que permita por menú elegir en tiempo de ejecución el “Tipo” de Objeto.*
   4. *Que permita cargar 5 objetos almacenados en un “Array” de figuras pudiendo ser de cualquier tipo derivado.*
   5. *Que Implemente métodos Virtuales la clase “Base”*
   6. *Que permita llamar de forma “Genérica” los métodos sin importar el tipo de objeto del que se trate.*
   7. *Que se Implemente con Punteros del tipo “Figura” apuntando a las clases derivadas*
   8. *Que recorra el Array ejecutando el método “Dibujar” usando Polimorfismo (virtual)*

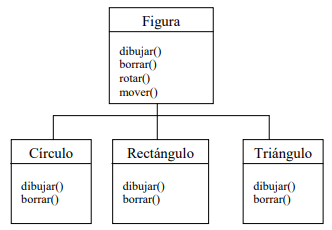


Diagrama de Clase resumido de la solución.

Lic. Oemig José Luis.

1. Dar un ejemplo de Creación de una clase y de instanciar un Objeto

class Alumno

{

public:

Alumno(){}

SetNombre(string nom)

{

nombre = nom;

}

string GetNombre()

{

return nombre;

}

.

.

.

private:

string nombre;

int DNI;

float nota;

};

Int main()

{

Alumno alumno;

return 0;

}

alumno es un objeto (una instancia) de la clase Alumno.

1. Explicar los alcances de Visibilidad “*public, private o protected”*

Para implementar los conceptos de Encapsulación, ocultamiento de información y visibilidad, C++ provee de distintos niveles de acceso a las clases:

**Public**: significa que las definiciones posteriores están disponibles para cualquiera.

**Private**: significa que nadie puede acceder a estas definiciones (salvo el creador). La única forma de que un programador cliente pueda acceder a los datos declarados private es a través de métodos definidos en public.

**Protected**: actúa como private, con la excepción de que las clases derivadas tienen acceso a miembros protegidos, pero no a los privados.

1. ¿Qué se entiende por función Miembro de una clase?

Un método o función miembro de una clase es una función que está definida dentro de una clase y que actúa sobre los datos de ésta.

1. ¿Qué diferencia hay entre implementar la función miembro dentro y fuera de la clase? Dar un ejemplo en código.

En rigor de verdad no habría diferencia alguna en cuanto al funcionamiento que observaría el usuario. Ocurre que esto no es lo deseable en cuanto a diseño: venimos haciendo hincapié en separar interfaz de implementación, de manera tal que la implementación quede oculta a los ojos del programador que es usuario de la clase. Por ende, preferimos declarar la clase con los prototipos de sus funciones miembro en un archivo de cabecera (.h) e implementar todos los métodos en el archivo de implementación (.cpp).

**Ejemplos**: se adjuntan los archivos: Ej01\_v0\_TP10.cpp (todo junto) y Ej01\_v1\_TP10.rar (separado)

1. ¿Qué se entiende por constructor y qué es el constructor por defecto?

Un *constructor* es una función miembro que se ejecuta automáticamente cuando se crea un  
objeto de una clase. Sirve para inicializar los miembros de la clase.  
El constructor tiene el mismo nombre que la clase. Cuando se define no se puede especificar un valor de retorno, nunca devuelve un valor. Sin embargo, puede tomar cualquier número de argumentos.

Un constructor que no tiene parámetros, o que se puede llamar sin argumentos, se llama *constructor por defecto*. Un constructor por defecto normalmente inicializa los miembros dato de la clase con valores por defecto.

C++ crea automáticamente un constructor por defecto cuando no existen otros constructores.

1. ¿Para qué sirven las listas de Inicialización?

Cuando se crea una clase, existen distinta clase de métodos públicos entre los que se encuentran aquellos dedicados a pasarle valores a los datos privados (Setters) y aquellos que le devuelven al programador usuario los valores de dichos datos (Getters). Esto sugiere que, una Precondición para el uso de un getter es, justamente, que se haya ejecutado antes el setter correspondiente.

Si esto no ocurre, se rompe el contrato entre el cliente y la implementación de la función. Como las clases casi siempre encapsulan datos, el creador de una clase no debe depender del usuario para inicializar los datos. Si el usuario olvida hacerlo podrían ocurrir resultados desagradables.  
Es por ello que C++ proporciona el *constructor por defecto*: para garantizar la inicialización de un objeto de clase.

Al igual que se puede sobrecargar un método de una clase, también se puede sobrecargar el  
constructor de una clase.

Por ende hay al menos dos maneras de inicializar los miembros privados: por defecto (vía el constructor por defecto) o a través de un constructor.

Esto último puede realizarse de dos maneras:

* Mediante asignación de parámetros por defecto.
* Mediante una lista de inicialización.

Veamos un ejemplo de estos dos últimos casos:

Dada la clase Caja:

**Asignación de parámetros por defecto:**

class Caja {

public:

Caja ( double lar, double an, double al = 0.8)

{

largo = lar;

ancho = an;

alto = al;

}

double Volumen()

private:

double largo;

double ancho;

double alto;

};

**Asignación mediante lista de inicialización:**

El bloque correspondiente al constructor puede cambiarse por:

Caja ( double lar, double an, double al = 0.8) : largo ( lar ) , ancho ( an ) , alto ( al ) { }

1. Cuando un Atributo es privado ¿cómo le asigno valores y cómo puedo ver su estado?

En general en las clases, los datos se declaran en la parte privada, y éstos se gestionan a través de métodos públicos. Están aquellos dedicados a asignarle valores a los datos privados (Setters) y aquellos que le devuelven al programador usuario los valores de dichos datos (Getters).

1. ¿Qué se entiende por Atributo Estático? Dar un ejemplo.

Los datos y las funciones de una clase pueden ser declarados ***static*** (estáticos). Cuando un campo miembro se declara como estático, éste se crea una única vez y es compartido por todos los objetos de la clase. Cada objeto posee su propia copia de cada uno de los campos miembro ordinarios, pero existe sólo una instancia de los campos miembro estáticos, independientemente de la cantidad de objetos de la clase que sean creados [4]. Un uso común de un dato miembro estático es para contar cuántos objetos de una clase existen*.*

1. ¿Qué se entiende por función miembro Estática? Dar un ejemplo.

Cuando se declara como estática una función, se la hace independiente de cualquier objeto de la clase. La ventaja es que existe y puede ser invocada antes de que se instancie cualquier objeto. Estas funciones, al igual que los campos miembro estáticos, pueden llamarse desde un objeto instanciado o a partir del nombre de la clase : NombreClase::metodo(). No es recomendable llamar con un objeto de la clase debido a que son métodos dependientes de la clase y no de los objetos.

Los métodos definidos como *static* no tienen asignado la referencia *this*, por eso sólo  
pueden acceder a miembros *static* de la clase. Es un error que una función miembro *static*  
acceda a miembros de la clase no *static*.

**Ejemplo:** la clase SumaSerie define tres variables static y un método static que calcula la suma cada vez que se llama.

class SumaSerie  
{  
private:  
 static long n;  
 static long m;  
public:  
 static long suma()

{  
 m += n;  
 n = m - n;  
 return m;  
 }  
};

long SumaSerie::n = 0;  
long SumaSerie::m = 1;

1. ¿Qué utilidad tiene un Destructor? ¿Qué ocurre si no lo Escribo?

Un ***destructor*** es una función miembro que destruye un objeto cuando ya no se lo necesita o cuando queda fuera de alcance. Tiene el mismo nombre que la clase pero precedido con tilde (~). No retorna ningún valor y tampoco toma ningún parámetro, por lo que sólo puede existir un único ***destructor*** por clase

Si no se define un ***destructor*** para una clase, el compilador siempre genera un ***destructor*** por defecto, pero éste no elimina objetos que han alocado memoria con el operador new. Por eso se debe definir el ***destructor*** con el correspondiente operador ***delete***.

1. ¿Qué debería poner dentro de Un Destructor? Dar un ejemplo.

El destructor es necesario implementarlo cuando el objeto contenga memoria reserva dinámicamente

**Ejemplo:** Se declara la clase Equipo con dos atributos de tipo puntero, un constructor con valores por defecto y el destructor.

El constructor define un array de n objetos Jugador con el operador new. El destructor  
libera la memoria reservada.

class Equipo  
{  
private:  
 Jugador\* jg;  
 int numJug;  
 int actual;  
public:  
 Equipo(int n = 12)  
 {  
 jg = new Jugador[n];  
 numJug = n; actual = 0;  
 }  
 ~Equipo() // destructor  
 {  
 if (jg != 0) delete [] jg;  
 }  
};

1. Explicar la cita de autor “*La mente humana clasifica los conceptos de acuerdo a dos dimensiones: Variedad y Pertenencia*”

Justamente: un concepto no existe en forma aislada, sino que coexiste con otros próximos a él. Y gran parte de su fuerza radica en las relaciones entre ellos. La **variedad** hace referencia a la relación es-un, por ejemplo: un taxi es un tipo (una variedad de) de auto. La **pertenencia** hace referencia a la relación tiene-un/a, por ejemplo: una rueda pertenece a (es parte de un) auto.

1. Explicar el Concepto de “Generalización y Especialización” en Herencia.

La **herencia** implica una relación de **generalización/especialización** en la que, una clase derivada especializa el comportamiento o estructura más general de sus clases bases. Realmente esta es la piedra fundamental de la **herencia**: si B no es un tipo de A, entonces B no debería heredar de A. Las clases bases representan abstracciones **generalizadas** y, las clases derivadas representan **especializaciones** en las que, los datos y funciones miembros de la clase base, sufren añadidos, modificaciones o incluso ocultaciones.

1. ¿Qué es el constructor de Copia? ¿Para qué sirve?

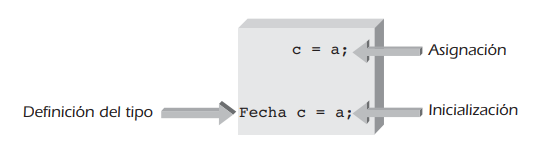
Aunque la asignación luce similar a la inicialización, vale la pena señalar que son dos operaciones diferentes por completo. En C++ ocurre una inicialización cada vez que se crea un objeto nuevo. En una asignación no se crea un objeto nuevo, tan sólo se cambia el valor de un objeto existente.

Tengamos en mente la clase:

// declaración de clase  
class Fecha  
{  
 private:  
 int mes;  
 int dia;  
 int anio;

public:  
 Fecha(int = 7, int = 4, int = 2005); // constructor  
 void operator=(Fecha&); // define la asignación de una fecha  
 void mostrarFecha(); // función miembro para desplegar una fecha  
};

// sección de implementación  
Fecha::Fecha(int mm, int dd, int aaaa)  
{  
 mes = mm;  
 dia = dd;  
 anio = aaaa;  
}



Un tipo de inicialización que se parece mucho a la asignación ocurre en C++ cuando un  
objeto es inicializado usando otro objeto de la misma clase. Por ejemplo, en la declaración

Fecha b = a;

o su forma equivalente por completo

Fecha b(a);

El objeto b es inicializado con un objeto a declarado previamente. El constructor que ejecuta este tipo de inicialización se llama ***constructor de copia*** y si no declara uno el compilador elaborará uno para usted. El ***constructor de copia por omisión*** hace una copia miembro por miembro entre objetos. Por tanto, para la declaración Fecha b = a; el constructor de copia por omisión establece los valores de mes, día y año de b con las contrapartes respectivas en a. Como con los operadores de asignación por omisión, los constructores de copia por omisión funcionan bien a menos que la clase contenga apuntadores a miembros de datos. En las clases que poseen punteros o arreglos como campos miembro, el constructor de copia por defecto no funciona como se espera, pues los punteros se copian y los punteros del nuevo objeto quedan apuntando a los miembros del primero. En este caso, el programador debe proveer un constructor de copia. Cuando se implementa un constructor de copia, el parámetro debe ser una referencia constante:

*Caja(const Caja& unaCaja);*

1. Describir el concepto de “Jerarquía de Clases”

La herencia es una relación entre clases donde una clase comparte la estructura o comportamiento, definida en una (herencia simple) o más clases (herencia múltiple). Así, estas clases se pueden organizar en estructuras jerárquicas: unas en forma arborescente (herencia simple) y otras en forma de grafos (herencia múltiple).



1. Explicar la diferencia entre “early binding” y “late binding”

***Ligadura*** (binding) representa generalmente, una conexión entre una entidad y sus propiedades.

La ligadura se clasifica según sea el tiempo o momento de la ligadura: ***estática*** (early binding) y ***dinámica*** (late binding).  
Ligadura ***estática*** se produce durante la *compilación*, mientras que la ligadura ***dinámica*** ocurre durante la *ejecución*.

La ventaja de la ligadura dinámica está en que, cuando se tiene una jerarquía de clases, las clases derivadas pueden necesitar redefinir un determinado método. Ahora: cuando un objeto de la clase hija llama a dicho método y la ligadura es estática, el compilador llama al método de la clase base. La ligadura dinámica, en cambio, hace que dicho llamado se efectúe en tiempo de ejecución, permitiendo así vincular al objeto con el método deseado.

1. Dar un ejemplo de código del punto anterior.

void Caja :: mostrarVolumen ( void ) const  
{  
cout << endl  
<< "El volumen de la caja es: " << volumen ( ) ;  
}  
#include <iostream>  
#include "CajaBotellas.h"  
using std :: cout ;  
using std :: endl ;  
int main ( )  
{  
Caja caja1 ( 4.0 , 3.0 , 2.0 ) ;  
CajaBotellas cajab1 ( 4.0 , 3.0 , 2.0 ) ;  
Caja \* cajap = 0 ; //puntero nulo a la clase base Caja  
caja1. mostrarVolumen ( ) ;  
cajab1. mostrarVolumen ( ) ; //sino es virtual se llama volumen de la clase Caja  
cajap = & caja1 ; //puntero a objeto de la clase base  
cajap - > mostrarVolumen ( ) ;  
cajap = & cajab1 ; //puntero a objeto de la clase derivada  
cajap - > mostrarVolumen ( ) ;  
cout << endl ;  
return 0 ;  
}

1. ¿Que resuelven o permiten las Funciones Virtuales?

Una función virtual es una función en una clase base que se declara usando la palabra reservada **virtual**. Si existe, en alguna clase derivada otra definición para dicha función declarada como virtual en la clase base, le indica al compilador que no debe usar enlace estático para la misma, se determinará en tiempo de ejecución qué versión se invoca. Esto permite, como se comentó en el punto 16, que cada objeto se vincule con métodos específicos para su correcto funcionamiento.

1. ¿Qué relación tienen los puntos anteriores con el “Principio de sustitución de Liskov”?

El Principio de sustitución de Liskov dice que “Las clases derivadas deben ser utilizables a través de la interfaz de la clase base, sin necesidad de que el usuario conozca la diferencia”. Justamente: esta sustitución puede hacerse en esos términos cuando se hace uso del late binding, dado que sólo así estaremos seguros de que, al utilizar objetos de las clases derivadas utilizando la interfaz de la clase base, se llamen a los métodos apropiados para dichos objetos.

1. ¿A qué referencia hace el autor cuando dice “*Existe polimorfismo cuando interactúan la herencia y el enlace dinámico*”?

Justamente por lo comentado en el punto anterior: cuando entra en juego la herencia, lo que hace potente y sintética a esta forma de programar es el hecho de que ahora todos los objetos de un tipo particular pueden recibir los mismos mensajes y el enlace dinámico se encarga de que el mensaje le llegue al método correcto.