Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE-0217 Estructuras Abstractas de Datos y Algoritmos para Ingeniería
Prof: Roberto Rodríguez Rodríguez

Laboratorio #2: Introducción a C++

Problema Propuesto

Una clase es la definición de un tipo de objetos. De esta manera, una clase "Empleado" representaría todos los empleados de una empresa, mientras que un objeto de esa clase (también denominado instancia) representaría a uno de esos empleados en particular.

Un objeto es una entidad que tiene unos atributos particulares (datos) y unas formas de operar sobre ellos (los métodos o funciones miembro). Es decir, un objeto incluye, por una parte una serie de operaciones que definen su comportamiento, y una serie de variables, manipuladas por esas funciones, que definen su estado. Por ejemplo, una ventana en una interfaz gráfica contendrá operaciones como "maximizar" y variables como "ancho" y "alto" de la ventana.

Un método (función miembro) se implementa dentro de un objeto y determina como tiene que actuar el objeto cuando se produce el mensaje asociado. La estructura más interna de un objeto está oculta, de tal manera que la única conexión con el exterior son los mensajes.

Para definir una nueva clase en C++ se utiliza la palabra ***class***, después sigue el nombre que se desea dar a la clase, ejemplo

```
class C_mi_clase {
   ...
}
```

La definiciones de una clase se realizan bajo tres premisas, ***public, private y protected***.

Public: Lo que se declara bajo esta premisa es accesible a todo el mundo, por lo general se desea declarar *public sólo métodos nunca miembros.*

Private: Lo que se declara bajo esta premisa no es accesible a nadie, ni siquiera a las clases heredadas, sólo a las funciones amigas, a los miembros y métodos de la clase, por lo general los miembros se declaran *private* y se hacen métodos para acceder estos miembros.

Protected: Cuando se desea que las clases heredadas tengan acceso a algunos métodos y miembros pero que los demás usuarios no, se declaran como *protected*.

Toda clase posee dos funciones especiales, estas se llaman igual que la clase, por defecto ninguna de las dos retorna nada, estás son:

```
C_MiClase(parámetros);
~C_MiClase(void);
```

La primera función es el constructor, esta puede recibir o no parámetros, lo que se hace en esta función es inicializar todos los miembros, que necesiten ser inicializados, asignar memoria dinámica o llamar otros constructores; el constructor se llama, automáticamente, al instanciar un objeto.

La segunda función es el destructor está función se llama cuando el objeto sale del ámbito (*scope*), en esta función entre otras cosas se libera toda la memoria que el objeto utilizó; esta función no recibe ni regresa nada.

Ahora se escribe el prototipo para la clase Figura, esta declaración debe escribirse en un archivo figura.h.

```
/*****************************************************************
figura.h
*****************************************************************/
#include<string>
#include<iostream>
using namespace std;

class C_figura {
    protected:
          string nombre;
    public:
            C_figura(string mi_Nombre);
            virtual ~C_figura(void);
            bool dibujar(void);
            bool mover(void);
            bool borrar(void);
             virtual bool area(void)
             {
                  cout << "Esto calculara el area de la figura" <<
endl;
                  return true;
             }
             virtual bool perimetro(void)
             {
                  cout << "Esto calculara el perimetro de la figura"
<< endl;
                  return true;
             }
}; //OJO ESTE PUNTO Y COMA SIEMPRE SE OLVIDA
```

La implementación de esta clase se hace en un archivo que se nombrará figura.cpp

```
/*****************************************************************
figura.cpp
*****************************************************************/
```

```
#include"figura.h"

C_figura::C_figura(string mi_Nombre)
{
     this->nombre=mi_Nombre;
}

C_figura::~C_figura(void)
{
     cout<< nombre << " dice adiós" << endl;
}

bool C_figura::dibujar()
{
     cout<< "dibujando " << nombre << endl;
     return true;
}

bool C_figura::mover()
{
     cout<< "moviendo " << nombre << endl;
     return true;
}

bool C_figura::borrar()
{
     cout<< "borrando " << nombre << endl;
     return true;
}
```

A continuación se realizará el prototipo y la implementación de la clase circulo.

```
/*************************************************************
circulo.h
*************************************************************/
#include"figura.h"
class  C_circulo : public C_figura{ //Así se hace la herencia
    public:
             C_circulo(void);
             bool area(void);
             bool perimetro(void);
};


/*************************************************************
circulo.cpp
*************************************************************/
#include"circulo.h"
```

```
C_circulo::C_circulo(void) : C_figura("circulo")    //Se llama el
constructor de figura
{

}

bool C_circulo::area(void)
{
     cout<< "el área de un " << nombre <<
             " se calcula como pi*r^2" << endl;
     return true;
}

bool C_circulo::perimetro(void)
{
     cout<< "el perimetro de un "<< nombre <<
             " se calcula como 2*pi*r" << endl;
     return true;
}
```

1) Copie en su computadora las clases anteriores, asegúrese que todo compila bien, para ello ejecute g++ -Wall –c –pedantic **mi_codigo.cpp.**

Ahora es necesario crear un programa para utilizar las clases definidas, como primer programa tome el siguiente ejemplo:

```
/*************************************************************
principal.cpp
*************************************************************/
#include"circulo.h"
int main()
{
     C_circulo mi_circulo; //Se crea un objeto
     C_figura *mi_figura=&mi_circulo; //Se define un puntero a un
objeto

     cout << "**************** Aquí se maneja todo como un puntero
base ****************"
                    << endl;
     mi_figura->mover();
     mi_figura->borrar();
     mi_figura->area();

     cout << "\n**************** Aquí se maneja directamente el
objeto ****************"
              << endl;
     mi_circulo.mover();
```

```
    mi_circulo.borrar();
    mi_circulo.area();
}
```

Compile el programa anterior y asegúrese que funcione; para compilarlo ejecute
g++ -Wall –o principal –pedantic principal.cpp figura.o circulo.o

Antes de continuar cerciórese que entiende la diferencia entre un objeto y un puntero a un objeto. ¿Por qué mi_figura puede usarse para direccionar mi_circulo si son de tipos diferentes?

2) Modifique los archivos circulo.h y circulo.cpp, de forma que se reescriba el método mover, recompile circulo.cpp y cree de nuevo el ejecutable principal.

Ejecute el programa, ¿nota algo diferente?, ¿Qué pasa cuando se llama el método mover directamente y con el puntero de tipo figura?

Edite el archivo figura.h e incluya la palabra **virtual** antes de la declaración de la función mover, recompile todo y ejecute de nuevo el programa, ¿qué pasó ahora?

Lo que se sucede es que cuando el "linker" une la tabla de métodos, los métodos que no son declarados como virtual, se asignan al método del tipo de variable que se está manejando, en este caso el tipo era figura de ahí que se asignó el método mover de figura.

Cuando se utiliza la directiva virtual, al compilador se le dice: el siguiente método puede ser definido en otro punto y en otro momento, haga la unión hasta el tiempo de ejecución no en tiempo de compilación. De ahí que cuando se tienen métodos virtuales la asignación final se realiza en tiempo de ejecución, antes de asignarlo se hace una búsqueda de cual es la verdadera clase que se está ejecutando y se asigna ese método. Esto se denomina polimorfismo.

3) Realice las clases cuadrado y triángulo, son similares a círculo, la clase triángulo debe incluir dos nuevos métodos, *girarHorizontal y girarVertical*. Compile estas clases, realice un programa prueba en *principal.cpp*, donde llame estas clases. Compile *principal.cpp*.

El problema que se da es que en las tres clases se trata de definir figura, para corregir este problema al inicio del archivo *figura.h,* después de los *includes* copie las siguientes líneas:

```
#ifndef CLASE_FIGURA
#define CLASE_FIGURA
```

Al final de la declaración de la clase escriba

```
#endif
```

Estas sentencias se utilizan para evitar la redefinición de clases. Básicamente los que está entre *#indef LoQueSea* y *#endif*, no se ejecuta si se definió LoQueSea.

Ahora compile y corra *principal*.

Cree un puntero tipo *figura* y utilícelo para direccionar un objeto tipo *triangulo*, ejecute el método

girarVertical.

¿Cómo solucionaría el problema?, Soluciónelo.

4) Hasta este momento se han compilado a pie, como vera esto es largo, tedioso y aburrido, sino lo considera así imagínese un proyecto pequeño que puede incluir 20 archivos fuente, habría que compilar cada fuente a pie, y después unir todo y cada vez que se cambia algo se debe recompilar ese algo, y unir todo.

Algunas personas pensando en eso crearon los archivos *makefile*, estos archivos sirven para tener un medio que lleve el orden de compilación, en donde se recompilen sólo las fuentes que cambien, y las dependencias se manejen. A continuación se da un ejemplo de archivo *makefile* para el proyecto anterior, cópielo y pruébelo, observe la ayuda del comando make para entender el ejemplo dado, ahora para compilar sólo digite *make*.

```
OBJS = principal.cpp figura.o circulo.o cuadrado.o
CC = g++
DEBUG = -g
CFLAGS = -Wall  -c  $(DEBUG)  -pedantic
LFLAGS = -Wall   $(DEBUG)   -pedantic
TARGET = principal

$(TARGET) : $(OBJS)
     $(CC)  $(LFLAGS)   $(OBJS)  -o  $(TARGET)

circulo.o : figura.h circulo.h circulo.cpp
     $(CC)   $(CFLAGS)   circulo.cpp

cuadrado.o : figura.h cuadrado.h cuadrado.cpp
     $(CC) $(CFLAGS) cuadrado.cpp

figura.o : figura.h figura.cpp
     $(CC)   $(CFLAGS)   figura.cpp

clean:
     \rm  -f  *.o  $(TARGET)
```

Note usted tiene que manejar las dependencias. A lo largo del curso salvo que se diga otra cosa TODOS los programas deberán compilarse con las banderas anteriores.

5) En el ejemplo anterior cuando se deseaba asignar un objeto a un puntero, primero se creaba explícitamente el objeto y después se asignaba, una de las capacidades más poderosas de un lenguaje es poder asignar memoria en forma dinámica, de esta forma se puede crear el objeto en tiempo de ejecución, cuando sea necesario. Para el manejo de memoria dinámica en C++ se usan las instrucciones *new* y *delete*. Por ejemplo, si se tiene un puntero tipo *figura* para asignar un objeto tipo *triangulo,* se hace: mi_puntero = new C_triangulo;.

A continuación se va a hacer una estructura de datos muy conocida, una pila. Se le dará la definición de la clase y cada uno hará la implementación.

```
/
*****************************************************************
*
C_pila.h
*****************************************************************
*/
#include<string>
#include<iostream>
using namespace std;

#ifndef CLASEPILA
#define CLASEPILA
//La directiva struct se usaba en C para crear nuevos tipos.
typedef struct S_celda{
   int dato;
   S_celda *proximo;
}T_celda;

class C_pila {
private:
    T_celda  *primer_elemento;
public:
   C_pila(void);
   ~C_pila(void);
   bool push(int);
   int pull(void);
};
#endif
```

Cosas importantes:

- En el constructor se deberá hacer que primer elemento apunte a **NULL.**
- En las funciones *push* y *pull* se deberán crear punteros auxiliares, para que contengan la pila antes de mover el puntero *primer_elemento*.
- En la función *pull,* note que debe eliminar la celda, antes de terminar, tenga cuidado de no perder el puntero *primer_elemento*.

Al terminar pruebe su implementación con el siguiente programa principal.

```
/
*****************************************************************
*
principal.cpp
*****************************************************************
*/
#include "pila.h"
```

```
int main (void)
{
  int temp;
  C_pila mi_pila;

  for (int i=100; i<110; ++i)
  {
   mi_pila.push(i);
  }

  for (int i=0; i<10; ++i)
  {
     cout << i <<"-) " << mi_pila.pull() << endl;
  }
}
```

6) Modifique la pila de forma que tenga un padre de la forma:

```
/
*****************************************************************
*
C_base.h
*****************************************************************
*/
#include<string>
#include<iostream>
using namespace std;

#ifndef CLASEBASE
#define CLASEBASE

typedef struct S_celda{
   int dato;
   S_celda *proximo;
}T_celda;

class C_base {
protected:
    T_celda  *primer_elemento;
public:
   C_base(void);
   virtual ~C_base(void);
   virtual bool agregar(int);
   virtual int quitar(void);
};
#endif
```

De esta forma la clase pila heredará de la base, la nueva definición de la clase pila será:

```
/
*****************************************************************
*
C_pila.h
*****************************************************************
*/
#include"base.h"

#ifndef CLASEPILA
#define CLASEPILA

class C_pila : public C_base{
public:
   C_pila(void);
   ~C_pila(void);
   bool push(int);
   int pull(void);
   bool agregar(int);
   int quitar(void);
};
#endif
```

Note que *agregar* y *quitar* llamarán a push y pull.

7) Al igual que pila, ahora cree una cola que herede de base, recuerde una pila utiliza un algoritmo LIFO una cola es FIFO.