



Clases

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 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:

```
class MiClase{  
    //miembros  
}
```

Las miembros en una clase se definen 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 y 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, de manera predeterminada ninguna de las dos retorna nada, éstas son:

```
MiClase( parametros );  
~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** esta 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.hh*.

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

class MiFigura {
protected:
    string nombre;
public:
    MiFigura(string miNombre);
    virtual ~MiFigura(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;
    }
}; //NO OLVIDAR ESTE PUNTO Y COMA!!!!!!!!!!!!!!
```

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

```
#include"figura.hh"
MiFigura::MiFigura(string miNombre){
    this->nombre = miNombre; //this es el equivalente a self en Python
    cout << "Ejecutando el Constructor de MiFigura\n\n" << endl;
}
MiFigura::~MiFigura(void){
    cout << "El Destructor de " << nombre << " dice adios" << endl;
}
bool MiFigura::dibujar(){
    cout<< "dibujando " << nombre << endl;
    return true;
}
bool MiFigura::mover(){
    cout<< "moviendo " << nombre << endl;
    return true;
}
bool MiFigura::borrar(){
    cout<< "borrando " << nombre << endl;
    return true;
}
```



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

circulo.hh

```
#include "figura.hh"
class MiCirculo : public MiFigura{ //Esto es herencia
public:
    MiCirculo(void);
    bool area(void);
    bool perimetro(void);
};
```

circulo.cpp

```
#include "circulo.hh"
MiCirculo::MiCirculo(void) : MiFigura("circulo"){
    //Se llama el constructor de MiFigura
    cout << "Ejecutando el Constructor de MiCirculo\n\n" << endl;
}
bool MiCirculo::area(void){
    cout<< "el area de un " << nombre <<
        " se calcula como pi*r^2" << endl;
    return true;
}
bool MiCirculo::perimetro(void){
    cout<< "el perimetro de un " << nombre <<
        " se calcula como 2*pi*r" << endl;
    return true;
}
```

- Copie en su computadora las clases anteriores, asegúrese que todo compila bien, para ello ejecute: g++ -Wall -c -pedantic [__miCodigo.cpp__](#).

Ahora es necesario crear un programa para utilizar las clases definidas, como primer programa tome el siguiente ejemplo (*principal.cpp*):

```
#include "circulo.hh"
int main(){
    MiCirculo mi_circulo; //Se crea un objeto
    MiFigura *mi_figura = &mi_circulo; //Se define un puntero a un objeto
    cout << "***** Aqui se maneja todo como un puntero base
    *****" << endl;

    mi_figura->mover();
    mi_figura->borrar();
    mi_figura->area();
    cout << "\n***** Aqui 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. ¿Porqué `mi_figura` puede usarse para direccionar `mi_circulo` si son de tipos diferentes?
- 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 enlazador (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.

- Cree las clases *MiCuadrado* y *MiTriangulo*. La clase *MiTriangulo* 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`.
- ¿encuentra algún problema?

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 lo que está entre `#ifndef __DefinicionXYZ__` y `#endif`, no se ejecuta si se definió `__DefinicionXYZ__`.

- Recompile y ejecute el principal.
- Cree un puntero tipo figura y utilícelo para direccionar un objeto tipo triangulo, ejecute el método `girarVertical`. ¿Cómo solucionaría el problema?, Soluciónelo.

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 Pruébalo, observe la ayuda del comando `make` para entender el ejemplo dado, ahora para compilar sólo digite `make` (Recuerde tener cuidado con el espaciado o el tabulado en el archivo makefile!!!).



```
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.hh circulo.hh circulo.cpp
    $(CC) $(CFLAGS) circulo.cpp
cuadrado.o : figura.h cuadrado.h cuadrado.cpp
    $(CC) $(CFLAGS) cuadrado.cpp
figura.o : figura.hh figura.cpp
    $(CC) $(CFLAGS) figura.cpp
clean:
    rm -f *.o $(TARGET)
```

En 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 la de poder asignar memoria en forma dinámica, de esta forma se pueden crear objeto en tiempo de ejecución, cuando sea necesario. Para 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 MiTriangulo;`

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

MiPila.hh:

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

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.



UNIVERSIDAD DE COSTA RICA
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA ELÉCTRICA
II CICLO 2013
**ESTRUCTURAS ABSTRACTAS DE DATOS Y
ALGORITMOS PARA INGENIERÍA IE-0217**
LABORATORIO 2: Clases en C++



Al terminar pruebe su implementación con el siguiente programa principal (principal.cpp).

```
#include "pila.hh"
int main (void){
    int temp;
    MiPila 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;
    }
}
```