Universidad de Los Andes
Facultad de Ingeniería
Escuela de Ingeniería de Sistemas
Departamento de Computación

Programación Orientada por Objeto Utilizando C++

Domingo Hernández H.

Índice

Introducción..... 3
Programación Orientada por Objeto 4
Características que soportan los Lenguajes
Orientados por Objetos 5
Especificación de un programa en C++ 5
Ejecución de un programa compilado en C++ 5
Identificadores en C++ 5
Tipos de Datos Primitivos en C++ 6
Tabla de rangos permitido en C++ 6

Introducción

Debido al auge que ha tenido la metodolgía de diseño de programas orientados por objetos así como los lenguajes que implantan cada uno de los conceptos relacionados con esta nueva forma de pensar y programar, se hace necesario el poseer una guía de apoyo al estudiante que le permita conocer, internalizar e implantar rápidamente todos los constructos que se utilizan en la Orientación por objetos.

En esta guía se revisan tanto los contructos básicos que son estándares para el lenguaje C como para C++ así como, los contructos propios con los que se implantan los conceptos de Clase, Herencia simple, Herencia múltiple, Polímorfismo, y Encadenamiento Dinámico. También se trata como se implantan las clase parametricas en el lenguaje C++.

Los programas que se utilizan como demostraciones a los conceptos estudiados pueden ser compilados tanto en Borland C++ como en C++ para estaciones de trabajo Sun y Silicon Graphics.

1.- Programación Orientada por Objeto (POO)

El diseño orientado por objeto, al igual que otras metodologías orientadas a la información crea una representación del mundo real que posteriormente es transformada a un dominio de soluciones que es el que llamamos programas. A diferencia de otros métodos, el diseño POO da como resultado un diseño que interconexiona objetos que encapsulan tanto una estructura de datos como una colección de métodos u operaciones que son los únicos capaces de alterar la estructura de datos del objeto.

Este encapsulamiento es el paradigma fundamental de la POO. Además esta nueva forma de programar, provee al usuario de un conjunto de herramientas y de paradigmas que se pueden emplear para realizar las siguientes tareas:

1. Modelaje de problemas del mundo real de una manera más sencilla por parte del usuario.

2. Facilidad de interrelación con otros ambientes computacionales.

3. Construcción de componentes de software que pueden ser reusables y que ofrecen la facilidad de ser extensibles (agregar más componentes).

4. Facilidad de modificación e implantación extensiva de los componentes del software sin tener que recodificar el sistema que emplee estos componentes.

1.1- La Orientación por Objeto

Puede ser descrita como la disciplina de modelaje y desarrollo de software que hace fácil la construcción de sistemas complejos por medio de componentes individuales llamados clases de objetos.

El término POO significa que se organizará al software con una colección de objetos discretos que incorporan tanto una estructura de datos como un conjunto de procedimientos que son los únicos capaces de alterar la estructura de dato del objeto.

2.- Características que deben soportar los lenguajes de POO

1. Definición de clases.
2. Manipulación de objetos.
3. Identidad de los objetos.
4. Herencia.
5. Polimorfismo.
6. Encadenamiento Dinámico.
7. Mensajes (envío).

3.- Especificación de un programa simple en C++

```
#include <iostream.h>                // Archivo de cabecera
/* Un programa para presentar una salida estándar por pantalla */
int main ()
{
        int x;
        cout <<  "Introduzca un número entero:";
        cin >> x;
        cout  <<  "El valor que se introdujo fue: " << x << "\n";
}
```

3.1- Compilación y ejecución

```
% CC prueba.cc    Aquí también se puede utilizar el compilador GNU C++ g++
% a.out
Introduzca un número entero:  6
El valor que se introdujo fue: 6

%CC prueba.cc -o prueba
%prueba
Introduzca un número entero:  90
El valor que se introdujo fue: 90
```

4.- Identificadores en C++

Comienzan con un carácter alfabético o underscore.
Pude utilizar otros caracteres alfanuméricos.
Son de longitud arbitrario (en teoría !!)
No pueden ser palabras reservadas.
Son sensitivo a las letras mayúsculas y minúsculas.

5.- Tipo de datos primitivos

int (Usualmente 16 o 32 bit)
short (entero corto Usualmente 16 bit)
long (Entero largo Usualmente 32 bit)
unsigned int (Entero no negativo Usualmente 16 o 32 bit)
unsigned short (No negativo short integer usualmente 16 bit)
unsigned long (No negativo long integer usualmente 32 bit)
char (Carácter usualmente 8 bit entre -127 y 127)
unsigned char (Carácter 8bit entre 0 a 255)
float (Número real punto flotante)
double (Doble precisión número real punto flotante)

6.- Tabla de rangos permitidos para los tipos de datos básicos

Tipo	Descripción	Tamaño en bits	Valores Límites
int	Entero	16	-32768 a +32767
short	Entero corto	16	-32768 a +32767
long	Entero de doble precisión	32	$-2*10^{9}$ a $+2*10^{9}$
char	carácter	8	
float	real	32	-10^{37} a $+10^{38}$
double	real de doble precisión	64	-10^{307} a $+10^{308}$

7.- Formas de asignación de valores a variables

y = x % 2; Le asigna a y el resto de la divisón entera de x entre 2
x = x+2; Le incrementa 2 a la variable x.
x+=2; Otra forma de incrementarle 2 a la variable x.
x*=2; Multiplica por 2 el contenido de la variable x
x++ // Evalúa el valor de la variable y luego incrementa su valor en uno Post-incremento
++x; // Incrementa el valor de la variable y después evalua Pre-ncremento
a=x++; // copia el valor de x en a y luego incrementa el valor de x en uno
a=++x; //incrementa el valor de x y luego asigna el valor de x en a
a--; // post -Decremento
--a // pre-Decremento

Advertencia: Se debe tener cuidado con confundir el operador relacional de igualdad (==) con el operador de asignación (=).

La siguiente expresión no generará un error de compilación
if (x=6) if(x==4)

8.- Operaciones Lógicas

> >= < <= Comprobaciones de mayor y menor que

== != Comprobaciones de igualdad y desigualdad

&& Y Lógico
|| O Lógicos.

! Negación.

El operador ! invierte una condición

9.- El lenguaje C++ depende fuertemente del tipo de dato (Strong Typing)

Esto significa lo siguiente:

* Cada valor utilizado en el programa pertenece a un tipo específico.
* Los valores deben ser utilizados de la manera apropiada con respecto al tipo.
* Valores de distintos tipos no deben ser mezclados en las expresiones.

Conversión explícita de tipos

```
int y =5;
double p = (double) y;    // provoca la conversión de un entero a un flotante
```

Conversión Implícita

La conversión implícita hace uso de reglas de coerción aplicadas por el compilador con la finalidad de generar tipos consistentes

```
int y=4;
```

```
double p = y+3.5;          // el compilador convierte el valor en y a double.
```

Notas:

* Evite las mezclas de enteros sin signos con enteros con signos.
* los caracteres son tratados como un tipo entero. por ejemplo

```
int x='C';
```
se le asigna el valor del código ASCII de 'C' (67) a x.

10.- Constructos estructurados

Estructuras condicionales

```
if (Condición)                    // Si la condición es != de cero entonces entra..
{
        sentencia;
        .................
        sentencia;
}
else
{
        sentencia;
        .............
        sentencia;
}
```

* Si usted tiene una sola sentencia en el cuerpo de la estructura if no necesita colocar las llaves que abren y cierran.
* Las estructuras de decisión anidadas pueden ser identadas al mismo nivel

```
        if (expresión1)
            ........
        else    if (expresión2)
                ......
        else
            ........
```

Estructuras de selección Múltiple

```
switch (Expresión entera) {
        case 1:
        .........;
        break;
        case 2:
        .........;
        break;
        default:
        .........;
        break;
}
```

* Si no se coloca el break en la selección múltiple puede ocasionar que se ejecute la siguiente sección case o que no se salga de la estructura de selección múltiple.

* Si la sección default no es incluida, no se hará nada en caso en el que ninguna de las sentencias case concuerden.

Estructuras de repetición
Repita mientras (La condición es evaluada al comenzar)

```
while (condición)                // si la condición es distinta de cero entra...
{
        sentencia1;
        ..............
        sentencia n;
}
```

Repita hasta (La condición es evaluada al final)

```
do {
        sentencia 1;
        ...............
        sentencia n;
} while (condición);             // Si la condición es distinta de cero sigue en el lazo
```

Condición evaluada en el medio

```
while (1) {  // lazo potencialmente infinito
        .................
        if (condición)
           break;                    // Sale del lazo
        .................
}
```

Repita Para (Lazo controlado por un contador)

```
for (i=0; i < 10; i++) {
        sentencia1;
        .............
        sentencia n;
}
```

Lazo infinito

```
for  ( ; ; ) {          // Se pueden omitir los argumentos
   .............
   if (Condición)
        break;

}
```

Notas:
*La sentencia for en C++ es más general que en pascal.

for (Inicialización de las variables de control; prueba de la condición; incremento de la variable de control)

Ejemplo: for (int i=0; i< 30; i *=2) es buena práctica declarar las variables de for dentro del mismo costructo.

* Pueden utilizarse dos variables de control dentro del lazo for. Por ejemplo:

for (int i=0, j=10; j*i < 23; j--, i++)

11.- Funciones en C++

```
double f (int x, double & y) {
        // se definen las variables locales
        .....;
        return una expresión;
}


// uso de la función f

p=f(cuenta,max);
```

Notas:

* x, y son definidas como los parámetros formales mientras que cuenta y max son los parámetros actuales.

* En c y c++ es común el pase de parámetro por valor como es el caso de x en el ejemplo anterior. Este tipo de parámetros se les conoce como parámetros estrictamente de entrada.
* El & utilizado como prefijo de la variable indica que el parámetro es pasado por referencia. Esto corresponde al parámetro var del Pascal. En ambos lenguajes la dirección del parámetro actual es pasado a la función tal que el valor del parámetro actual puede ser actualizado por algún cambio hecho al parámetro formal y dentro de la función f . Un parámetro de referencia es utilizado normalmente como parámetro de salida o parámetro de entrada salida.

* Los parámetros de entrada pueden también ser pasados como parámetros de referencia, con el fin de evitar el gasto de las copias excesivas. Con la finalidad de evitar algún cambio imprevisto en los parámetros de entrada utilizados por referencia se debe utilizar la palabra reservada **const.** Por ejemplo

```
        long g(int x, const double &y) {
                ...............;
        }
```

* No existen procedimientos en c y C++ pero el mismo efecto se puede ser obtenido por las funciones devolviendo un vacio (void).

Por ejemplo:

```
void f1(int x, double &y){
        .................;
}
```

* Similarmente funciones sin parámetros pueden ser definidas:

```
int f2(void) {
.............;
}
```

* Las funciones pueden ser utilizadas antes de que ellas sean definidas, esto se hace por medio de la definición preliminar de un prototipo.
Por ejemplo:

```
//
//Declaración de prototipos
//
double f(int x, double &y);

int main() {
        //
        // Usando la función f
        //
        p = f(3,Max);
        ..........
}

//
//      Definición de la función f como se presentó al comienzo del archivo
//
double f(int x, double &y) {
        .............;
}
```

* Así como en Pascal existe un costo de ejecución para las llamadas de cada función. Sin embargo, en C++ este costo puede ser evitado para funciones pequeñas por medio del uso del constructo **inline** . El cual permite al compilador reemplazar la llamada de la función en el programa fuente por el código que la ejecuta. Esto no debe alterar la ejecución del programa pero puede acelerar la ejecución de este. (Este constructo será utilizado más adelante cuando se estudien las clases.

```
inline void f1(int x)
.............
```

* Las funciones pueden ser definidas con parámetros por defectos. Usted puede definir funciones que se pueden llamar con menos de los argumentos que se le hayan definido. Los valores de los argumentos que usted no proveea se le asignan los valores por defecto.
Por ejemplo:

```
// Definición de la función F1.
void F1(int x=10, float y=3.0,int z=4)

// Llamando F1 con los parámetros por defecto
p=F1();
// Llamando a F1 con otros parámetros
p=F1(5,2.9)          // Qué valores se asignaron ?
p=F1(6,7.9,23)
```

Los valores por defecto se deben definir en las posiciones más a la derecha de la lista de argumento de la declaración de la función.
La siguiente declaración no es Válida

```
void F1(int x=10, float y, int z)          // no es válida porque el
```

compilador no puede adivinar cuales valores serán suplidos.

12.- Arreglos en C++

```
const int asize =10;
int nums [ asize ];
int i;

for  (i=0; i<asize; i++)  // segmento de código que permite incrementar en
                          //una unida el contenido del vector
{
        cin >> nums [ i ];
        nums[ i ]++;
        cout << nums[ i ] << '\n';
}
```

Notas

* Los arreglos en C++ comienzan desde 0 y no desde 1.

* nums[i]++ encuentra el valor almacenado en nums[i] y luego lo incrementa en uno. Alternativamente nums[i++] encontrará el valor almacenado en nums[i] y luego incrementará i.

* Los arreglos pueden ser inicializados cuando ellos son definidos.

```
int x [ 5]={2,4,6,8,10};          ó
int x [ ]={2,4,6,8,10};
```

13.- Punteros en C++

Los punteros son una variable cuyo valor es la dirección de otra variable. El puntero debe ser dereferenciado para determinar el valor de la variable apuntada por el.

Ejemplos

```
int x, z;
int *y;              // Declara a y como un puntero a un entero.
y = &x;              // Inicializa a y con la dirección en memoria de la variable x.
```

```
int *yy = &x;        // Definición e inicialización combinada.

z = *y;              // De - referenciación de y; ya que y es un puntero a x
                     // el valor de x es asignado a z
int *p, *q;
int a, b;
p=&a;  q=&b;
*p=*q;               // es equivalente a decir a=b;

int a=3;             // un entero
int *b=&a;   // un puntero a un etero
int **c=&b;// un puntero a un puntero a entero

cout <<*b;  // output 3
cout <<**c;// output 3
```

14.- Aritmética de Punteros

La aritmética sobre punteros pueden ser ejecutada usualmente por adición o substracción de enteros.

```
int *p, *q, x;

p=&x;                // Toma la dirección de x.
q=p+1;               // q apunta a un entero más allá de p en la memoria.


double *r, *s, y;

r = &y;              // Toma la dirección de y.
s= r -2              // s apunta a 2 unidades de memoria tipo double antes de s.
```

Nota: Lo anteriormente expresado parece no ser lo suficientemente útil ya que si se consulta el contenido (de-referencia) los punteros q ó r, no necesariamente encontraremos la data que estamos buscando. Esto es útil cuando se trata con arreglos ya que en C y C++ los nombres de los arreglos son efectivamente un puntero a su primer componente. Así si se tiene el siguiente arreglo

```
int x[ 5]={3,6,9,12,15};
```

Entonces x se puede interpretar como &x[0] (Esto significa encontrar la dirección de x[0]. Así que x+1 puede ser interpretado como &x[1] y x+2 es interpretado como &x[2].

Ejemplos:

```
int f=*(x+1);              // asigna x[ 1] a f
*(x+1)=*(x+2);             // es decir x[ 1]=x[ 2]
(*(x+1))++;          // incrementa en una unidad al contenido de x[ 1]
```

Para muchas cosas los punteros y arreglos son intercambiables. Sin embargo, no son equivalentes. Considere el copiar los elementos de un arreglo en otro.

```
int x[ 6]={1,2,3,4,5,5};
int y[ 6];
for (int i =0; i<6; i++)
        y[ i]=x[ y];          // esto es OK

for (int i =0; i<6; i++)
        *(y+i)=*(x+i);              // esto es OK


for (int i =0; i<6; i++)
        *y++=*x++;              // Esto no es correcto debido a que estamos tratando de incrementar x e y                    // pero estos son las bases del arreglo que son fijas por lo tanto no pueden
                                // incrementarse, aquí se generará un error a tiempo de compilación.

int *p, *q;
for (int i =0; i<6; i++)
        *p++=*q;          // Esto es ok  ya que p y q son simples punteros.
```

15.- Arreglos como parámetro

Cuando un arreglo es pasado como parámetro actualmente se esta pasando el puntero a la base del arreglo. Por lo tanto, los arreglos son pasados efectivamente como parámetros por referencia.
Ejemplos

```
int F1 (int tamaño, int data [ ])

int F1(int tamaño, int *data )          // son equivalentes
```

si deseamos prevenir cualquier intento de cambiar los valores de un arreglo debemos utilizar la palabra reservada const.

```
int F1(int tamaño, const int data [ ])
```

16.- **cadenas**

Las cadenas en c++ son almacenadas como arreglos de caracteres que terminan en nulo, es decir con el caracter '\0' al final.

```
char cadena1 [ ] = {'H','O','L','A','\0'};
char cadena2 [ ] = "HOLA";
```

Las dos declaraciones anteriores son idénticas.
Cuando se introduce una cadena vía teclado, se debe recordar que cualquier espacio en blanco que se introduzca es reconocido como el final de la cadena. Sin embargo, ninguna acción es llevada a cabo para procesar la cadena hasta que el carácter de nueva línea es introducido.

```
char x [ 100];
for (int i=0;i<4;i++) {
    cin >> x;
    cout << x << '\n'
```

```
}
```

17.- Comandos en línea de argumento

C++ y c permiten que algunos argumentos sean introducido desde línea de comando cuando un programa es llamado. Esos parámetros son definidos como parámetros al programa principal **main** y son pasados como un vector de cadenas. Por convención este arreglo es conocido como **argv**: este es un tipo de cadena de caracteres. Cuando los comando de líneas son utilizados, el número de argumentos en líneas también debe ser incluido como parámetro en el programa principal main, este parámetro tradicionalmente se llama argc. De esta manera la declaración del programa principal con uso de argumentos en línea es:

```
int main(int argc, char *argv [ ])
```

o recordando el significado de los punteros con arreglos

```
int main(int argc, char **argv )
```

Cuando el main es ejecutado argv [0] contiene el nombre del programa ejecutable y argv[1], argv[2] contienen las cadenas suministradas como los comandos, argc contiene un entero más que el número de argumentos en la línea de comando ya que el nombre del programa es también incluida en la cuenta. El programa siguiente muestra un programa con el uso de un único comando en la línea de argumento.

```
#include<iostream.h>
#include<libc.h>   // requerido par utiliza la función atoi ... asccii a entero

int main (int argc, char **)
{
        if (argc !=2)          // chequea que se el número correcto de argumentos
          cerr << "Falto introducir un comando en la línea de argumentos \n"
        else {
           cout << argv[ 0] << "\n";      // imprime el nombre del programa
          cout << argv[ 1] +1 << "\n"; // salida argumento +1
        }
```

```
}
```

Salidas del programa:

```
% CC prueba.cc -o prueba
% prueba
% Falto introducir un argumento en la línea de comando
% prueba 3 5
 % Falto introducir un argumento en la línea de comando
%prueba 3
prueba
4
%
```

Note que el uso de cerr es utilizado para presentar un mensaje de error por defecto en la pantalla del computador.

20.- Tipos de datos definidos por el usuario

Tipos enumerados

```
enum pintas {clubs, diamantes, corazones, espadas}; // Definición de un tipo enumerado
struct cartas {                 // Definición de un tipo estructura
        enum pintas s;
        int juego;
};

// Definición de una variable del tipo cartas

struct cartas carta1;       //      Definición para el compilador c y c++

cartas carta2; /* Estrictamente hablando c++ no necesita de utilizar la palabra
                  reservada struct cuando se define una variable de ese tipo. */

// Acceso a los campos de una estructura
carta1.s=corazones;
carta1.juego=1;

// También se puede acceder a los campos de una estructura a través de un
// puntero
```

```
struct cartas *puterocart = & carta1;
punterocart →s=corazones;
punterocart →juego=1;
```

Notas:
1.- La notación punterocart →s es una abreviación de (*punterocart).s.
2.- En pascal los valores de los tipos enumerados son obtenidos por medio de la función ord pero en c++ los valores de los tipos enumerados pueden ser asignados de manera diferente:

```
enum pintas {clubs=2, diamante=23, corazones =14, espadas =1};
```

20.1- Nuevos nombres de tipos

En c y c++ provee la facilidad de renombramiento de los tipos con la finalidad de introducir una notación más corta y hacer los programas más fáciles de leer.

Suponga que se desea utilizar una variable que refleje la edad de una persona.

```
int mi_edad=33;
```

pero para incrementar la legibilidad de m programa se puede introducir un nuevo tipo de nombre

```
typedef  int edad;
edad mi_edad=33;
```

Esto no posee diferencia al momento de chequear los tipos. Edad es vista meramente como un tipo entero.

Similarmente un tipo especial de booleano puede ser definido

```
typedef enum {falso=0,verdadero=1} bool;
```

El cual puede ser utilizado en funciones que devuelvan verdadero o falso.

22.- Asignación dinámica de memoria

La asignación dinámica de memoria es una técnica de programación muy valiosa. Tamaños apropiados de memorias pueden ser asignados desde la memoria libre disponible en el heap del sistema y retornada a este cuando ya no se necesite. La asignación y devolución de memoria es llevada a cabo directamente por el programador. En c++ se utilizan los operadores unarios **new** y **delete** para lleva a cabo estas tareas.

```
int *p;          //Declara un puntero a entero
p=new int;   // Memoria para un entero es asignada desde el heap del sistema
             // y p apunta hacia ella.
*p=6;
delete p;      // Devuelve la memoria apuntada por p al heap del sistema.
 p=new int;  // p ahora apunta a una nueva localidad de memoria.
```

Se debe notar que estos operadores son sobrecargados ya que estos pueden ser utilizados con cualquier tipo apropiado ya sea estándar o definido por el usuario. Ejemplo

```
double *x=new double;
cartas *carta1=new cartas;
```

El formato general es:

<puntero a un tipo> = new <tipo>;
delete <puntero a un tipo>;

Sin embargo, existe un formato especial para los arreglos

```
int *numero = new int[ 20];
delete [] numeros;
```

Notas:

* Nunca intente acceder a la memoria apuntada por un puntero que fue eliminado con el comando *delete* .

* Nunca borre la memoria direccionada dinámicamente hasta que usted no este seguro de que no la necesite más.

* Siempre elimine la memoria asignada dinámicamente cuando definitivamente finalice el uso de esta.

- Solo puede ser utilizado el operador *delete* con aquellas memorias asignadas con el operador *new*.

24.- Sobrecarga (Polimorfismo por sobrecarga)

En c++ los nombres de las funciones de los usuarios y operadores pueden ser sobrecargados, es decir, el mismo nombre puede ser dado a dos o más funciones u operadores. De igual manera los nombre de los operadores estándares del c++ pueden ser sobrecargados.
Ejemplo:

// Ejemplo que muestra la sobrecarga de funciones con dos tipos distintos de datos definidos por //el usuario.

```
#include<iostream.h>
struct estudiante{
        char Apellido[30];
        int edad;
};

struct profesor{
        char Apellido[30];
        char iniciales[4];
};

void asignacion(estudiante *est){
        cin >> est→Apellido;
        cin >> est→edad;
}

void asignacion(profesor *prof){
        cin >> prof→Apellido;
        cin >> prof→iniciales;
}

void presenta(estudiante *est){
        cout << est→Apellido << " ";
        cout << est→edad << '\n';
}

void presenta(profesor *prof){
        cout << prof→Apellido << " ";
        cout << prof→iniciales << '\n';
}

int main() {
        estudiante *estud=new estudiante;    // Asignación dinámica de memoria
        profesor *profe=new profesor;        // para estud y profe
        asignacion(estud);        // El compilador puede resolver la ambiguedad por tipo
        asignacion(profe);
        presenta(estud);          // El compilador puede resolver la ambiguedad por tipo
```

```
        presenta(profe);
        delete estud;           // Libera memoria.
        delete profe;
}
```

25.- Otra forma de realizar este problema es por sobrecarga de los operadores CIN y COUT

```
// Ejemplo que muestra la sobrecarga de los operadores cin y cout.

#include<iostream.h>
struct estudiante{
        char Apellido[30];
        int edad;
};

struct profesor{
        char Apellido[30];
        char iniciales[4];
};

istream& operator >>(istream &entrada estudiante *est){
        entrada >> est→Apellido;
        entrada >> est→edad;
        return entrada;
}

istream& operator >>(istream &entrada profesor *profe){
        entrada >> profe→Apellido;
        entrada >> profe→iniciales;
        return entrada;
}

ostream& operator <<(ostream &salida estudiante *est){
        salida << "El apellido del estudiante es: " << est→Apellido '\n';
        salida << "La edad del estudiante es: " << est→edad;
        return salida;
}

ostream& operator <<(ostream &salida profesor *prof){
        salida << "El apellido del profesor  es: " << prof→Apellido '\n';
        salida << "Las iniciales del profesor son : " << prof→edad;
        return salida;
}

int main() {
```

```
        estudiante *estud=new estudiante;    // Asignación dinámica de memoria
        profesor *profe=new profesor;        // para estud y profe
        cin >> estud;
        cin >> profe;
        cout << estud;
        cout << profe;
        delete estud;              // Libera memoria.
        delete profe;
}
```

26.- Operadores Input output.

>> y >> son operadores de entrada salida que son sobrecargados tanto para tipos estándares del lenguaje como para tipos definidos por el usuario como se vio en el ejemplo anterior.
Por ejemplo el prototipo para la sobrecarga de los tipos estándares es como se presenta a continuación:

```
ostream& operator << (int i);
ostream& operator << (float t);
ostream& operator << (char *s);
```

Note que >> o << devuelven la referencia a un stream. Se puede pensar que << como una salida a un stream especificado por ejemplo

```
cout << "asdf" << 22  << "casa" <<33;
```

27.- Tipos de datos Abstractos y su implantación.

ABSTRACCIÓN DE DATOS (Ocultamiento de Información)

Abstracción: idea general que se concentra sobre las cualidades esenciales de algún objeto del mundo real más que sobre la realización concreta del mismo.

La abstracción consiste en enfocar los aspectos esenciales inherentes a una entidad e ignorar las propiedades accidentales.

La abstracción como proceso: consiste en separar las propiedades esenciales de un objeto, sistema, fenómeno o problema y/o omitir las propiedades no esenciales.

La abstracción como producto: es una descripción o especificación de un sistema en el que se enfatizan algunos detalles o propiedades y se suprimen otros. Es importante destacar que las propiedades esenciales de un sistemas son tratadas como un todo.

Tipo de datos abstractos

Una herramienta útil para especificar las propiedades lógicas de los tipos de datos abstractos (TDA), el cual es fundamentalmente una colección de valores y un conjunto de operaciones sobre esos valores. La colección de valores y las operaciones sobre esos valores forman una construcción matemática que pueden implementarse utilizando una estructura de dato particular ya sea de hardware o de software.

Definición: "Si se posee un tipo de dato llamado T que se define como una clase de valores y una colección de operaciones sobre esos valores, y si las propiedades de esas operaciones son especificadas solamente con axiomas, entonces T es un TDA".

Un TDA al momento de la implantación, debe de cumplir con las especificaciones algebraicas de sus operaciones (SINTAXIS Y SEMÁNTICA).

La correcta especificación (según Gutag) de un TDA debe ser:

A.-Especificación Sintáctica:
Aquí se tratan las funciones u operaciones que actúan sobre las instancias de T, definiéndose los nombres, dominios y rangos de dichas funciones. Estas operaciones pueden clasificarse de la siguiente manera:

1. **Operación Constructor**: esta operación produce una nueva instancia para el tipo de dato abstracto, proveyendo al usuario de una capacidad de para generar dinámicamente instancias de nuevos objetos y asignarles valores por defecto a las propiedades (atributos) del TDA. Aparta memoria principal**.**

2. **Operación Destructor**: elimina aquellas instancias del tipo de dato T que el usuario tiene en desuso. Libera memoria principal**.**

3. **Operación de Acceso**: permite al usuario obtener elementos que sólo son propiedades del tipo de dato del sistema.

4. **Operación de Transformación**: producen nuevos elementos del tipo de dato abstracto, partiendo del elemento ya existente y posiblemente de otros argumentos.

Se observa el efecto que tiene cada una de las operaciones especificadas sobre el resto de las operaciones del TDA. Para esta especificación se realiza la siguiente tabla:

Nombre de la operación	Dominio de los argumentos	Rango de los resultados	Tipo de operación
nombre de cada una de las operaciones que actúan sobre el TDA	explícito	lo que se desea obtener o de lo contrario un mensaje de error	Constructor, Destructor, Transformación o Acceso.

B.- Especificación Semántica:

Se compone de un conjunto de axiomas en forma de ecuaciones las cuales indican como operan cada una de las operaciones del tipo de dato abstracto al ser especificadas sobre las operaciones restantes del mismo tipo de dato abstracto.
Operaciones válidas sobre la estructura de datos del TDA, y cuales son los resultados que cada una de las operaciones regresa una vez que se haya ejecutado.

Nombre de la operación	Dominio del argumento	Rango del dato
nombre de la operación aplicada al TDA	la base de la aplicación del la función	rango del resultado obtenido de la operación

Cualquier estructura de datos conocida se puede representar como un TDA (vectores, imaginarios, colas, pilas, reales, etc.).
Ejemplo:

Propiedades de los TDA

- Encapsulación: un TDA encapsula ciertos tipos de datos y operaciones con el objetivo de localizar en un punto determinado de su programa la especificación del TDA.
- Generalización: un programador es libre de definir sus propios tipos de datos y sus propias operaciones con el objetivo de aplicarlos a operandos que no necesariamente tiene que ser de un tipo fundamental.

28.- Constructos de la Orientación por objeto

1. Definición de clases.
2. Objetos.
3. Identidad de los objetos.
4. Herencia.
5. Polimorfismo.
6. Encadenamiento Dinámico.
7. Mensajes (envío).

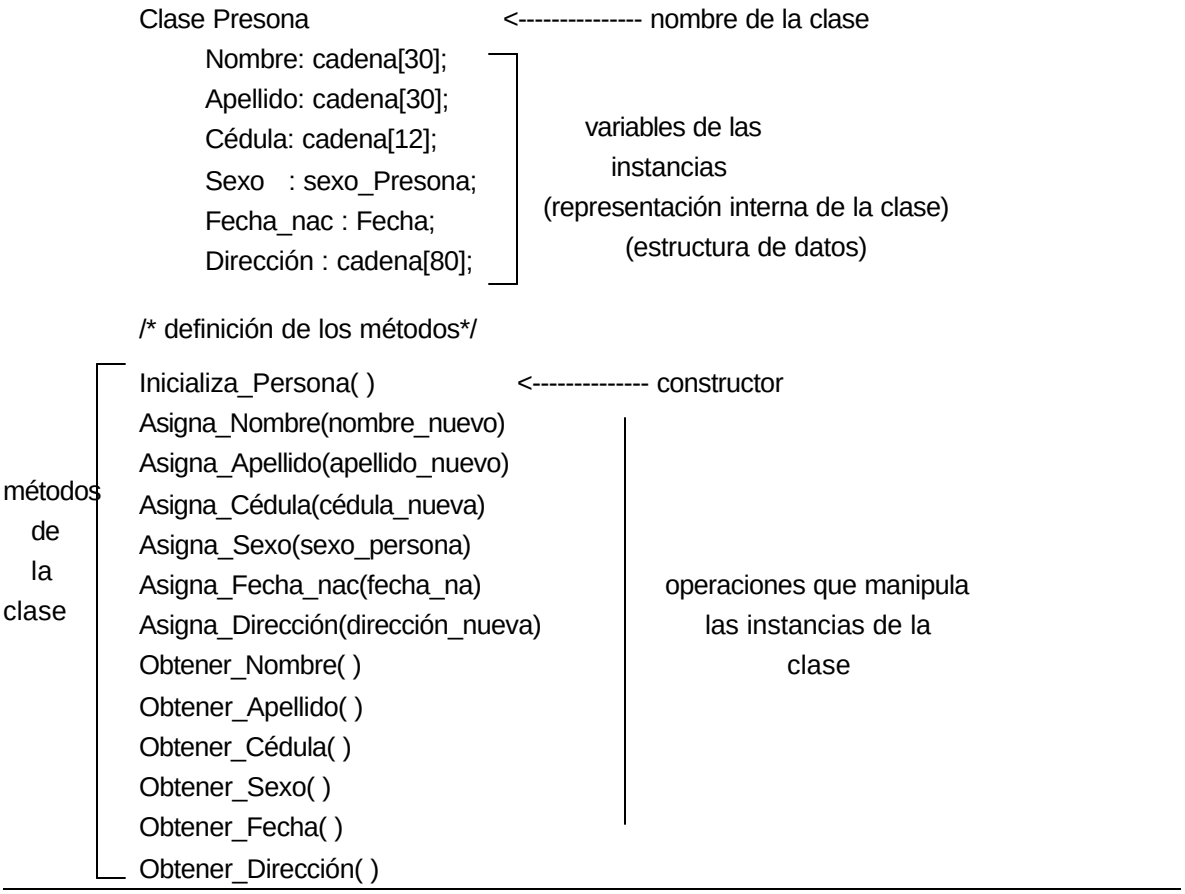
Clase: Un mecanismo para llevar a cabo la abstracción de datos. Es constructo comúnmente utilizado para definir un TDA en los lenguajes POO.

Uno de los primeros usos de las clases fue en el SIMULA 67. En este lenguaje las clases son empleadas para declarar un conjunto de objetos que poseen una estructura de datos y un comportamiento similar. En primera instancia este lenguaje utilizó a las clases como plantillas para crear nuevos tipos de datos abstractos que no poseía el lenguaje SIMULA 67.

Normalmente la definición de una clase incluye los siguientes puntos:

- nombre de la clase.
- las operaciones que manipula las instancias de la clase.
- una representación interna de la clase (estructura de datos)
- implantación de los métodos de la clase.

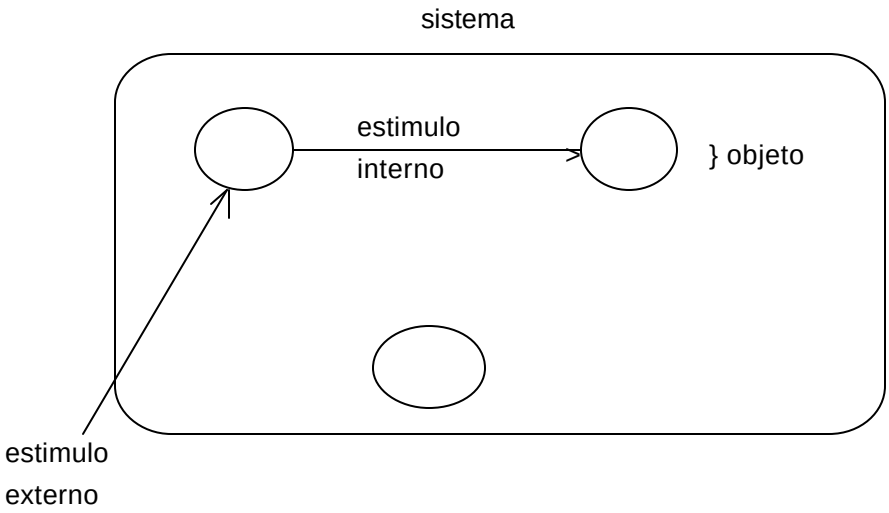
Estudiaremos un ejemplo: implantación de la clase Persona.



Implantación de los métodos de la clase

```
función Inicializa_Persona( )
        Nombre = ' ' ;
        Apellido = ' ' ;
        Cédula  =  ' ' ;
        Sexo = F;
        Fecha_Nac.Día = ' ';
        Fecha_Nac.Mes = ' ';
        Fecha_Nac.Año = ' ';
        Dirección = ' ';
fin
```

OBJETO: es una entidad física o abstracta que posee un comportamiento ante ciertos estímulos, que pueden ser tantos externos como de otro objeto específico que se muestra en el sistema:



Identidad del Objeto

Existe la necesidad de identificar los objetos de cada clase. Cada objeto posee su propia identidad que lo distingue de los demás objetos. En otras palabra, dos objetos distintos no son iguales aunque todos los valores de sus atributos sean iguales.

La identidad del objeto significa que los objetos son distinguibles por su existencia inherente y no por las propiedades descriptivas que ellos poseen. Por ejemplo dos manzanas con el mismo color, forma y textura son aún dos manzanas individuales.

La técnica más comúnmente empleada para identificar objetos en un lenguaje de programación es el nombre del objeto definido por el usuario.

nota: los conceptos serán orientados al lenguaje.

Instancias de una clase

Es una ocurrencia y esta representada por un objeto. Los valores de las variables contenidas en la representación interna de la clase, pertenecen a los objetos individuales. Esto quiere decir que cada objeto puede tomar valores distintos para cada una de las variables de la representación interna de la clase. Por ejemplo sea P1 una instancia de la clase Persona:

Persona P1,P2,P3; /* es como cuando se declaraban constantes: const : a,b,c; */

los valores para la instancia P1 pueden ser:

```
Nombre = 'Ricardo';
Apellido = 'Salas';
Cédula = 'V-9325336';
Sexo = 'M';
Fecha_Día = '19';
Fecha_Mes = '8';
Fecha_Año = '66';
Dirección = 'Mérida';
```

Implantación en C++ de una clase

Se utiliza el constructo "class".

```
class          Nombre de la clase  {
      cuerpo de la clase
};
```

Como ejemplo utilizaremos la especificación de la clase Persona (partiendo de la implantación de dicha clase):

```
class  Persona  {
   char        nombre[31], apellido[31], cédula[13];
   sexo_per    sexo;
   fecha       fecha_nav;
   char        dirección[81];

 /* definición de las funciones Miembros */

   public:     /* palabra reservada, de aquí en adelante todo es público */
      persona( );
      void asigna_nombre(char  nombre_nuevo[31]);
      void asigna_apellido(char  apellido_nuevo[31]);
      void asigna_cédula(char  cédula_nueva[13]);
      void asigna_sexo(char  sexo_persona);
      void asigna_fecha_nac(unsigned char  fecha_na);
      void asigna_dirección(char  dirección_nueva[80]);
      obtener_nombre( );
```

```
        obtener_cédula( );
        obtener_apellido( );
        obtener_sexo( );
        obtener_fecha( );
        obtener_dirección( );
    };
```

Implantación de clases en C++

Implantación de las funciones de una clase:

- Funciones Miembro.
- Funciones Amiga.

Implantación de las funciones que pertenecen a una clase

Funciones Miembro:

Se pueden implementar de la siguiente manera:

1. Declaración de la función miembro dentro de la clase e implantación de la función miembro fuera de la clase.
2. Declaración e implantación de las funciones Miembro dentro de la clase.

Explicación:

Para 1.

```
class    persona {
    /********/
};
```

Implantación de la función Miembro

Sintaxis para la implantación de una función Miembro es la siguiente:

< tipo de la función > <nombre de la clase> :: nombre de la función( ) {

void, char, .., cualquier
tipo estructurado

cuerpo de la función (instrucciones a ejecutarse)

};

Por ejemplo:

```
    void  persona::asigna_nombre(char nombre_nuevo[31]) {
        strcpy(nombre, nombre_nuevo);
    };
```

```
void  persona::asigna_sexo(unsigned char sexo_persona) {
   if  (!strcmp(sexo_persona, "F")
         sexo = F;
   else
         if  (!strcmp(sexo_persona,"M")
               sexo = M;
}
```

nota: las funciones miembros tienen sus propios parámetros locales.

Para 2.
Aquí tenemos los siguientes casos:
a)

```
class  persona {

         /* Miembros datos */

/* especificación de las funciones miembros*/
persona( );

void  asigna_nombre(char nombre_nuevo[31]) {
   strcpy(nombre, nombre_nuevo);
};

void  asigna_sexo(unsigned char sexo_persona) {
   if  (!strcmp(sexo_persona, "F")
         sexo = F;
   else
         if  (!strcmp(sexo_persona,"M")
               sexo = M;
};

/* así sucesivamente hasta englobar todos los procedimientos */
}; fin
```

Ventajas:

- cada vez que se llama una función, se crea una sobrecarga en el punto del llamado. Si la función se llama muchas veces, pueden aumentar las sobrecargas. El método presente disminuye dichas sobrecargas.

- este método óptimiza el tiempo de proceso.

Desventajas

- las funciones Miembro no deben tener más de tres (3) instrucciones.

b)
```
class  persona  {
        /*Miembros datos*/
/*especificación de las funciones Miembro*/
void  asigna_nombre(char nombre_nuevo[31]);
void  asigna_sexo(uinsigned char sexo-persona);
void  asigna_apellido(char apellido_nuevo[31]);

inline   void  persona::asigna_nombre(char nombre_nuevo[31]) {
              strcpy(nombre, nombre_nuevo);
         };

inline   void  persona::asigna_sexo(unsigned char sexo_persona) {
         if  (!strcmp(sexo_persona, "F")
               sexo = F;
         else
               if  (!strcmp(sexo_persona,"M")
                     sexo = M;
         };
/* y así sucesivamente*/
```

Funciones Amigas:

Normalmente el acceso a los miembros privados de una clase es restringido sólo a las funciones Miembro de la clase. Ocasionalmente puede ser necesario dar acceso a los datos de una clase a funciones que no pertenecen a dicha clase, estas funciones que no pertenecen a esa clase son declaradas como funciones Amiga.

Por ejemplo:

```
class  cadena {
      .
      .
      /* funciones Miembro */
      .
      .
```

```
    cadena   concatenar(char  cad1[10], char  cad2[10]);
    .
    .
    };

    cadena::concatenar(.....) {
    .
    .
    };

    class  persona {
    .
    .
    /*funciones Miembro*/
    friend   concatenar(.....);
    .
    .
    };
```

nota: las funciones Miembro son las únicas capaces de cambiar la estructura de datos de la misma clase.

Cómo invocar un método o función sobre un objeto?
R)por ejemplo:

```
    persona  P1;
       nombre del objeto.nombre del método;
       P1.asigna_nombre("domingo");
```

Las funciones miembro se invocan por medio de la notación "**.**" , por ejemplo:

P1.asigna_nombre("Ana")

Esta sentencia significa que se va a invocar la función asigna_nombre de la clase persona para que se ejecute sobre el objeto P1; es decir que P1 es un argumento explícito para la función asigna_nombre.

los miembros datos de una clase son generalmente considerados datos privados de la clase, de tal manera que se pueden garantizar que el usuario de

la clase no pueda operar con estos datos de una forma directa, esto quiere decir que si el usuario intenta hacer lo siguiente:

```
P1.nombre="domingo";
```

es inválido por que no se puede acceder a los miembros datos de una clase por medio de otras funciones que no estén declaradas dentro de la clase.

Es importante resaltar que una vez que se ha definido de esta forma una clase su nombre denota un nuevo tipo de dato, que nos permite crear instancias (objetos) que son de ese tipo.

Control del acceso a los miembros de una clase

Hay que recordar que los miembros de una clase (dato y función) son miembros que poseen por defecto un acceso privado, esto quiere decir que sólo las funciones miembro de la clase son las únicas capaces de alterar la estructura de datos de la clase. Sin embargo existe ocasiones en la que los miembros de una clase pueden ser requeridos por algunas otras clases en el momento de una aplicación particular. Para lograr este objetivo el lenguaje C++ provee de un conjunto de palabra reservadas que son capaces de alterar el permiso de acceso a los miembros de una clase. Estas palabras reservadas son las siguientes:

private: los miembros de una clase que se encuentren a continuación de esta palabra reservada, pueden ser accedido únicamente por las funciones miembros declaradas dentro de la misma clase.

protected: los miembros de una clase que se encuentren a continuación de esta palabra reservada, pueden ser accedidos por las funciones miembros que se encuentran dentro de la clase y por las funciones miembro de las clases derivadas.

public: los miembros de una clase que se encuentren a continuación de esta palabra reservada, pueden ser accedidos por cualquier función que se encuentre dentro del alcance de la clase.

Por ejemplo:

```
class  persona  {
       char  nombre[31]; apellido[31];cédula[13];
        public:
          sexo_persona      sexo;
        private:
          fecha    fecha_nacimiento;
        protected:
          char     dirección[81];
        public:
          .
          .
        private:
          .
          .
};
```

Constructores:

Es una función Miembro de una clase que tiene como propósito especificar como un nuevo objeto de una clase será creado, es decir, reservar memoria e inicializar al objeto. La definición del Constructor puede incluir código de programación con instrucciones específicas para reservar memoria, asignar valores, conversión de tipo y cualquier otra línea de instrucción que le pueda ser útil al constructor.

Los Constructores pueden ser invocados tanto explícitamente como implícitamente. El compilador del C++ llama automáticamente al constructor apropiado en el momento en que éste sea invocado.

Sintaxis para la implementación de un Constructor

```
Nombre de la clase::nombre de la clase (parámetros) {
                    cuerpo del Destructor;
};
```

nota: los Constructores en C++ llevan en mismo nombre de la clase.

Por ejemplo:

```
persona::persona(char nombre[31], char apellido[31],..., char cédula[13],...) {
        strcpy(nombre,nombre1);
        strcpy(apellido,apellido1);
    };
```

¿Cómo se instancia una clase persona?
R)en la declaración del tipo persona tendríamos:

```
persona    P1, P2  ("Pedro", "Camejo", "V-7585673", ....);
```

otra forma de crear instancias de la clase es utilizar el operador **new**:

```
persona   *P3 = new persona("ricardo", "salas", "V-9325336",....);
persona   *P2 = new persona( )     ------> significa que no hay parámetros
```

P2 y P3 son punteros hacia un objeto.

nota: el Constructor no debe llevar un tipo.

Destructores:
Deshabita y limpia la memoria ocupada por un objeto, destruye un objeto que previamente fue creado por un Constructor. Así como en el caso de los Constructores, los Destructores pueden ser llamados explícitamente utilizando el operador **delete** del C++ o implícitamente cuando el objeto queda fuera del alcance de la clase. Si no se define un Destructor para una clase dada, el C++ genera un Destructor por defecto para la clase.

Si se crea un objeto en forma dinámica utilizando el operador **new**, se debe de tener presente que es responsabilidad del programador liberar la memoria del objeto ya que el C++ no posee la posibilidad de conocer cuando un objeto no es necesitado en su aplicación.

Sintaxis para la implantación de un Destructor

```
nombre de la clase::~nombre de la clase(parámetros) {
                cuerpo del Destructor;
```

```
};
```

Especificación del Destructor dentro de la clase

```
class  persona
        char  nombre[31], apellido[31],..... ;
/*especificación de las funciones miembro*/
persona( );    <------- es como un Constructor
~persona( );  <------- es como un Destructor
.
.
};
```

Por ejemplo:

```
persona  *P3 = new persona(".....", ".....", ......)
delete P3;
```

nota: si hago x = malloc(50), estoy asignando memoria a x, por lo tanto es mi responsabilidad liberar esta memoria, y lo hago de la forma:

free x;

Valores por defecto en los parámetros de una función
ejemplo:

```
int   x = 0;
int   suma(int  x=0, int y=0)  {
        return (x+y);
};
```

Definiciones válidas

```
int  suma(int  x=0, int  y=0);
int  suma(int  x, int  y=0);
```

Definiciones inválidas

```
int  suma(int  x=0, int  y);
```

nota: para definiciones válidas, los parámetros con valores por defecto se colocan al final, después de los parámetros que no tienen valores por defecto.

ejemplo:
int suma(int x, int y, int z, int m=0, int n=0);
Los valores por defecto en los parámetros de una función no son solamente válidos para las funciones globales, si no también pueden ser utilizados en la implantación de las funciones miembro de la clase.
por ejemplo:

```
class punto {
        int x,y;
   public:
        punto (int x=0, int y=0);   <------ es como un Constructor
        ~punto( );
};
```

Puedo declarar instancias de la clase punto de la siguiente forma:

```
punto P1,P2 (20,15), *P3 = new punto(5); *P;
```

nota: punto(5) se entiende que x=5 y y=0. El puntero *P es instanciado con la sentencia P=new punto(5,6);

Ejemplo de especificación de clase

```
#include <stdlib.h>
#include <fstream.h>
class punto1 {
      double x;
      double y;
public:
      punto1(double xi=0, double yi=0) : X(xi), Y(yi) {}
      double X()    {return x;}
      double Y()    {return y;}
      friend istream & operator >> (istream& in, punto1 &punto){
         in >> punto.x  >> punto.y
        return in
      }
      friend ostream & operador << (ostream & out, const punto1 &punto){
         out << '(' << punto.x  << punto.y << ')';
         return out
      }
```

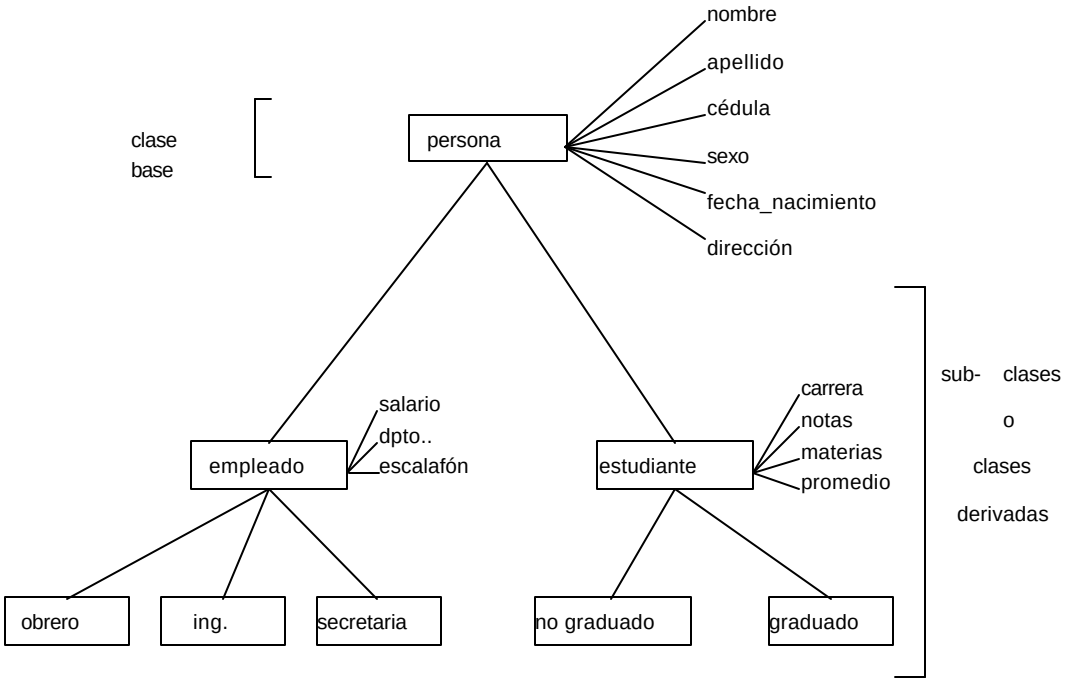
```
};

class punto2 {
        double x;
        double y;
        double z;
public:
        friend istream & operator >> (istream& in, punto2 &punto){
           in >> punto.x  >> punto.y >> punto.z;
          return in
        }
        friend ostream & operador << (ostream & out, const punto1 &punto){
           out << '(' << punto.x  << punto.y << punto.z  <<')';
           return out
        }
        friend punto2 operator + (const punto2 &p1, const punto2 &p2){
           punto2 p(p1.x +p2.x, p1.y+p2.y, p1.z+p2.z);
           return p;
        }
        friend punto2 operator + (const punto2 &p1, double c){
           punto2 p(p1.x +c, p1.y+c, p1.z+c);
           return p;
        }
};
int main () {
        punto1 p1;
        cout << p1 << '\n';
        punto2 p2;
        cout << "introduzca 3 coordenadas: "
        cin >> p2;
        cout << p2 << endl;
        punto2 ps = p2, p3(1.1,2.2,3.3);
        cout << ps << endl;
        cout << ps + p3 << '\n' << p3 +0.9 << endl;
}
```

Herencia

La orientación por objeto intenta modelar las aplicaciones del mundo real tan cerca de el como sea posible. También intenta incrementar la reusabilidad y la

extensibilidad del software. Todas estas capacidades son provistas por un concepto poderoso que es llamado la Herencia, a través del cual el diseñador puede construir una jerarquía de clases a partir de una clase base. Las clases generadas a partir de las clases base se llaman clases derivadas. Estas nuevas clases pueden generar tanto comportamiento (operaciones, métodos o funciones miembros) como la representación interna de la clase (variables de instancias, datos miembros) de la clase base existente. La herencia de representación conduce a compartir la estructura de dato entre la clase base y la clase derivada.



Una clase derivada puede ser una clase base de las que le derivan.

Como implantar en C++ una clase derivada
Sintaxis:

class <nombre de la clase derivada>:<modificador de accesos>nombre clase base{

definición de los datos Miembros;
definición de las funciones Miembros;

};

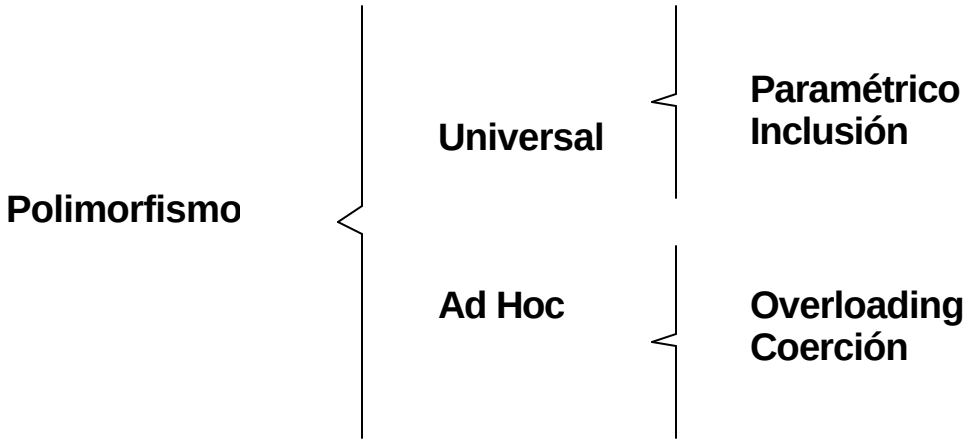
Modificador de Acceso: es utilizado para modificar el acceso a los miembros que son heredados por las clases derivadas. Este modificador de acceso

puede ser opcional y sólo puede manipular dos valores que pueden ser la palabra reservada **public** o **private**. Se debe recordar que los miembros de una clase son privados por defecto y que este permiso de accesibilidad puede ser alterado por medio del uso de las palabras reservadas **public**, **protected** y **private**.

Polimorfismo

Polimorfismo significa que la misma operación puede comportarse diferentemente sobre distintas clases. Por ejemplo, la operación "mover" ejemplo puede comportarse diferentemente sobre una clase llamada Ventana y una clase llamada Piezas_ajedrez. Una operación es una acción o transformación que un sujeto ejecuta sobre un objeto, por ejemplo, justifica a la derecha, desplegar, mover. Una implantación específica de un operación para una cierta clase es llamada método. Un operador orientado por objeto o polimórfico puede tener más de un método implantado.

Clasificación de Polimorfismo



Polimorfismo Paramétrico: Se obtiene cuando una función trabaja uniformemente sobre un rango de tipos; esos tipos normalmente exhiben una estructura común y puede comportarse de manera distinta para cada tipo.

Polimorfismo de Inclusión: Es un polimorfismo utilizado por modelos de subtipos y herencia. En este tipo de polimorfismo un objeto puede pertenecer a clases diferentes que no necesariamente son disjuntas.

El polimorfismo paramétrico y de inclusión están clasificados en una categoría mayor llamada polimorfismo universal, el cual contrasta con el polimorfismo no universal o el polimorfismo ad hoc.

Polimorfismo por Overloading: En este caso el mismo nombre de la variable se utiliza para denotar diferentes funciones, y el contexto se utiliza para decidir cual función se debería ejecutar para una invocación particular del nombre.

Puede imaginarse que para el procesamiento de un programa eliminemos el overloading por asignación de nombres distintos a las funciones diferentes, en este caso tendríamos programas con muchos nombres de funciones. El overloading es justamente una abreviación sintáctica conveniente, que permite poseer diferentes funciones con un mismo nombre.

Polimorfismo por Coerción: Es una operación semántica que convierte argumentos a los tipos esperado por una función, en una situación que de otra forma resultaría en un tipo de error. La coerción puede estar dada

estáticamente, insertándose automáticamente entre argumentos y funciones a tiempo de compilación o pueden tener que determinarse dinámicamente, con pruebas a tiempos de ejecución sobre los argumentos.

28.- Herencia Jerárquica

Un poderoso concepto que caracteriza a los lenguajes de orientación por objeto es la herencia. La herencia consiste en el compartimiento de atributos y métodos entre clases basándose en una relación jerárquica. Una clase puede definirse ampliamente y redefinirse sucesivamente en subclases más refinadas. Cada subclase que se incorpora, hereda todas las propiedades de su superclase y adiciona sus propias y únicas propiedades. Las propiedades de la superclase no necesariamente deben de repetirse en cada subclase. La habilidad para sacar factor común de las propiedades de una clase dentro de una superclase común y la herencia de propiedades desde la superclase pueden ayudar enormemente en el diseño.

Por ejemplo, si tenemos una compañía que está dividida en dos subclase llamadas Compañía con fines de lucro y organizaciones sin fines de lucro. Ambos tipos de organizaciones contienen información almacenadas como: nombre de la compañía, dirección de la compañía y personal. La subclase Organización sín fines de lucro tiene una información especializada como: garantías gubernamentales, apoyo comercial, enlaces internacionales,

etc. Símilarmente, la compañía con fines de lucro tiene una especialización tal como: deducción de impuesto. Por lo tanto, para capturar las operaciones comunes o comportamiento construiremos una clase genérica llamada Compañía que posee dos clases, compañías comerciales y organización sín fines de lucro. En la figura 21 se observa lo explicado.

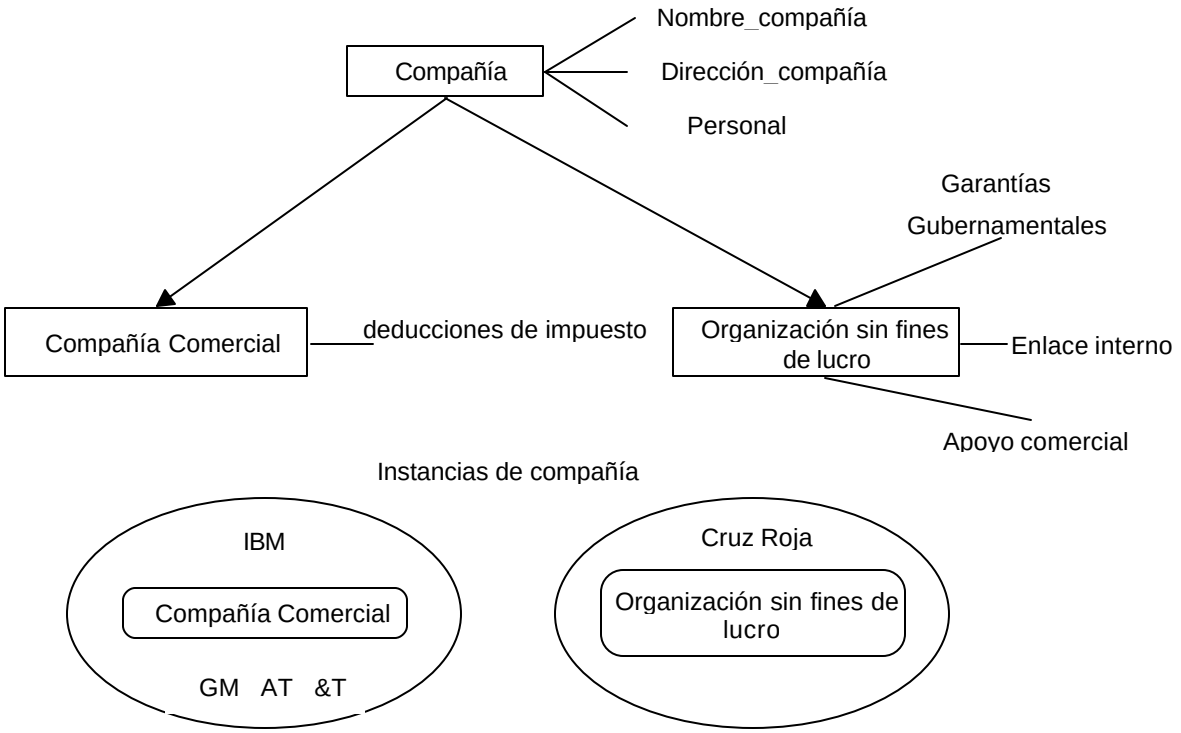


Fig. 21 Herencia Jerárquica

Compañía comercial y Organización sin fines de lucro son subclase de la clase Compañía. La clase Compañía es una superclase de compañía comercial y Organización sín fines de lucro. La subclase y las superclases tienen una relación transitiva, esto significa lo siguiente: si "X" es una subclase y además es superclase de "Y" , y "Y" es una subclase y además superclase de "Z" entonces "X" también es una subclase y además superclase de "Z". Por ejemplo si en el gráfico anterior se anexa una subclase llamada compañía de semiconductores a compañía comercial, entonces por transitividad Compañías de semiconductores es una subclase de Compañía. Esto significa que compañía de semiconductores hereda el compor-tamiento y forma de representación de Compañía a través de Compañía comercial. La figura 22 expresa gráficamente lo anterior.

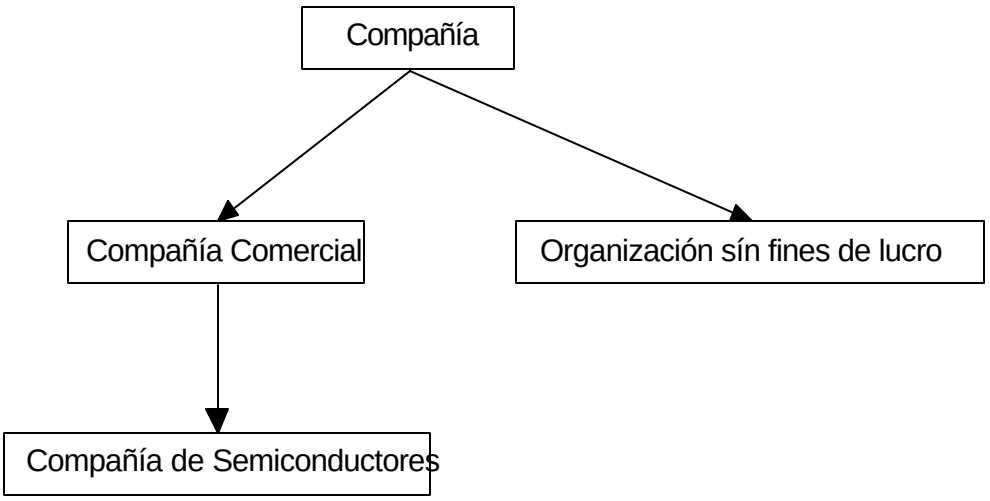


Fig. 22 Herencia estructural y de comportamineto.

Como se observa en el la figura 22 existe dos aspectos importantes de herencia:

- Herencia Estructural.
- Herencia de Comportamiento (herencia de métodos).

Herencia Estructural: Significa que las instancias de una clase tal como Organizaciones sín fines de lucro, la cual es una subclase de compañía, heredará las instancias variables de Compañía tales como: nombre, dirección, etc.

Herencia de Comportamiento: La clase Compañía tiene métodos tales como: Adicionar subsidiarias, Evaluación de presupuesto, etc.; las cuales son heredadas por las subclases Compañía comercial y Organización sin fines de lucro. Esto significa que se puede mandar un mensaje al método Adicional subsidiarias para una instancia llamada NPO de Organización sin fines de lucro y ejecutar el método Adicional subsidiaria en Compañía con NPO como objeto destino.

De lo anterior se observa que una clase define la estructura y el comportamiento de una colección de objetos.

El comportamiento se especifica en el método asociado con las instancias de la clase, recordando que los métodos son operaciones que pueden recuperar o actualizar el estado de un objeto. El estado de un objeto está almacenado en las variables de instancia.

En la herencia jerárquica un método definido por una clase es heredado por una subclase de ésta, por lo tanto la herencia de métodos es parte de

la interfaz de manipulación de instancia de una subclase. Por ejemplo si se tiene una clase Ventana con la subclase Ventana con borde y Ventanas de texto. Los métodos tales como mover_ventana cambia tamaño_ventana y rotar_ventana puede invocarse sobre las instancias de Ventana de texto y Ventana con bordes. La subclase Ventana sin texto posee métodos más especializados para editar y modificar el tipo de caracteres y tamaño de caracteres (editar, font, size). Esos métodos no están definidos sobre las instancias Ventana. La subclase ventana con borde pueden también tener sus propios métodos (editar, font, etc), que no tienen ninguna relación con los otros métodos con los mismos nombres en otras clases.

Otro ejemplo de herencia se observa en la figura 23.

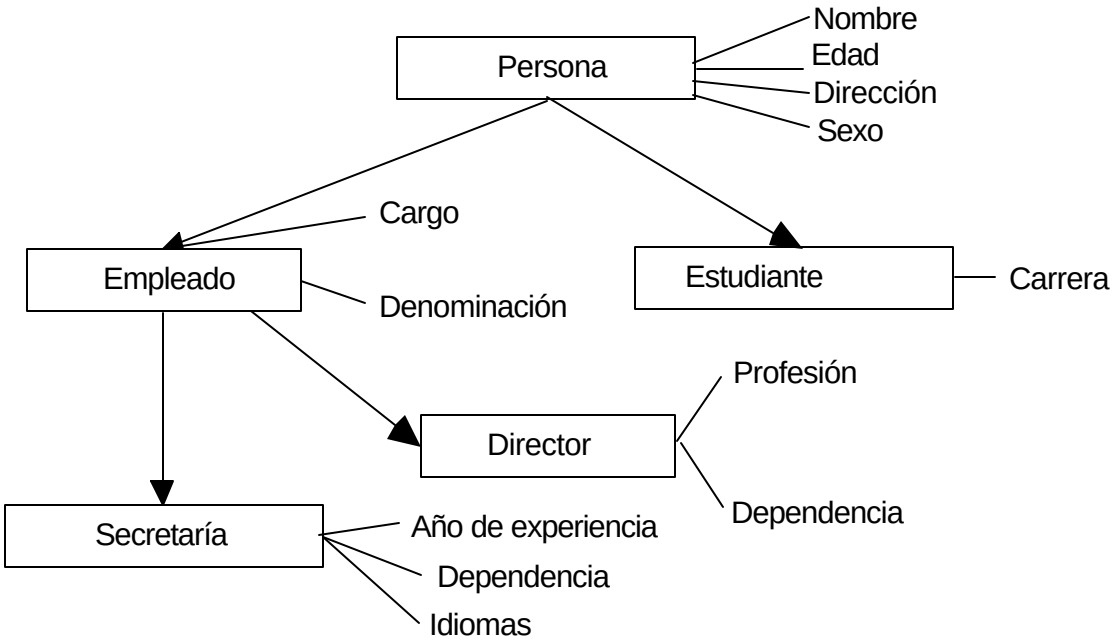


Fig. 23 Ejemplo de herencia.

Herencia Múltiple: La herencia múltiple permite a una clase tener más de una superclase y heredar todos los rasgos de todos sus padres. Esto permite mezclar información de dos o más fuentes. Es una forma más complicada de generalización que de simple herencia, la cual restringe las clases a una jerarquía de árbol. La ventaja de la herencia múltiple es que

aumenta el poder en una clase específica o incrementa la posibilidad de reutilizarse. La desventaja es que se pierde toda simplicidad tanto conceptual como de implantación.

Definición de Herencia Múltiple: Una clase puede heredar rasgos de más de una superclase. Una clase con más de una superclase es llamada clase junta. Un rasgo de una clase ancestro que se encuentra más de una vez a lo largo de una ruta solo se hereda una vez. Los conflictos entre definiciones paralelas de clases crean ambiguedades que son resueltas en la implantación. En la práctica, tales conflictos deben evitarse o resolverse explícitamente.
En la figura 24 se explica la herencia múltiple con un ejemplo:

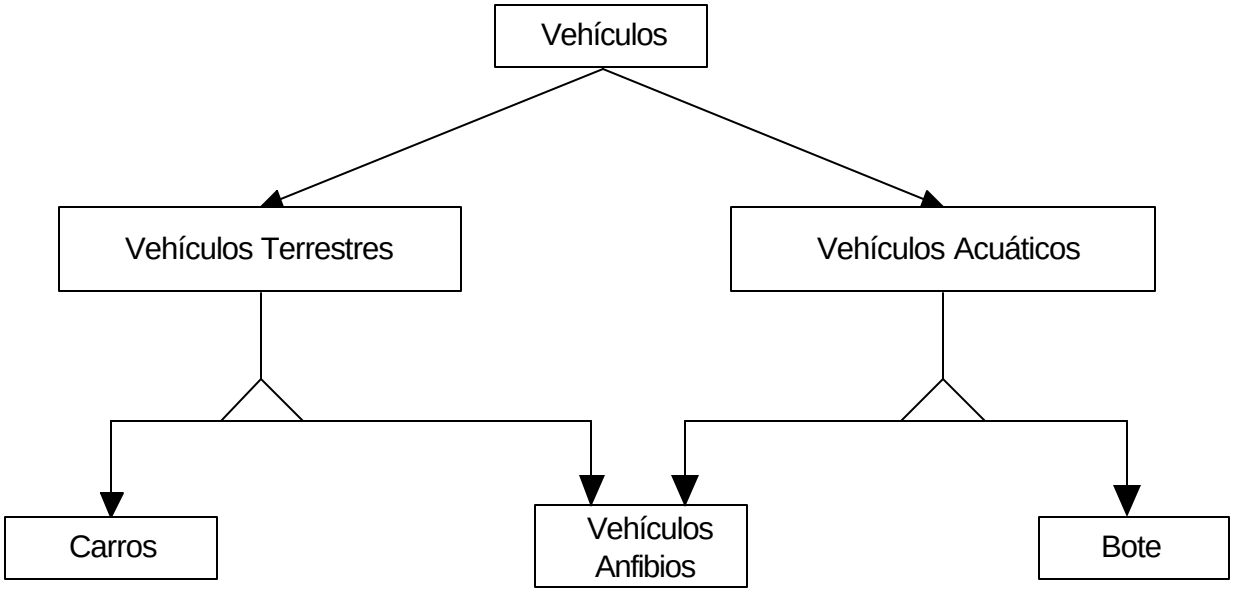


Fig. 24 Herencia múltiple para clases juntas.

En la figura 24 Vehículo_anfibio es tanto un Vehículo terrestre como un Vehículo acuático. Por lo tanto Vehículo_anfibio es una clase junta. A este tipo de herencia se le conoce como herencia multiple por el sobrelapamiento de clase, esto quiere decir que una subclase hereda tanto la estructura de datos como el comportamiento de dos o más superclases.

Si una clase puede refinarse en algunas dimensiones distintas e independientes, entonces debe utilizarse generalizaciones múltiples. Recuerdese que el contenido del método objeto es conducido por esas relevancias para una solución eficiente a una aplicación.

En la figura 25 se muestra un ejemplo de multiherencia para clases disjuntas:

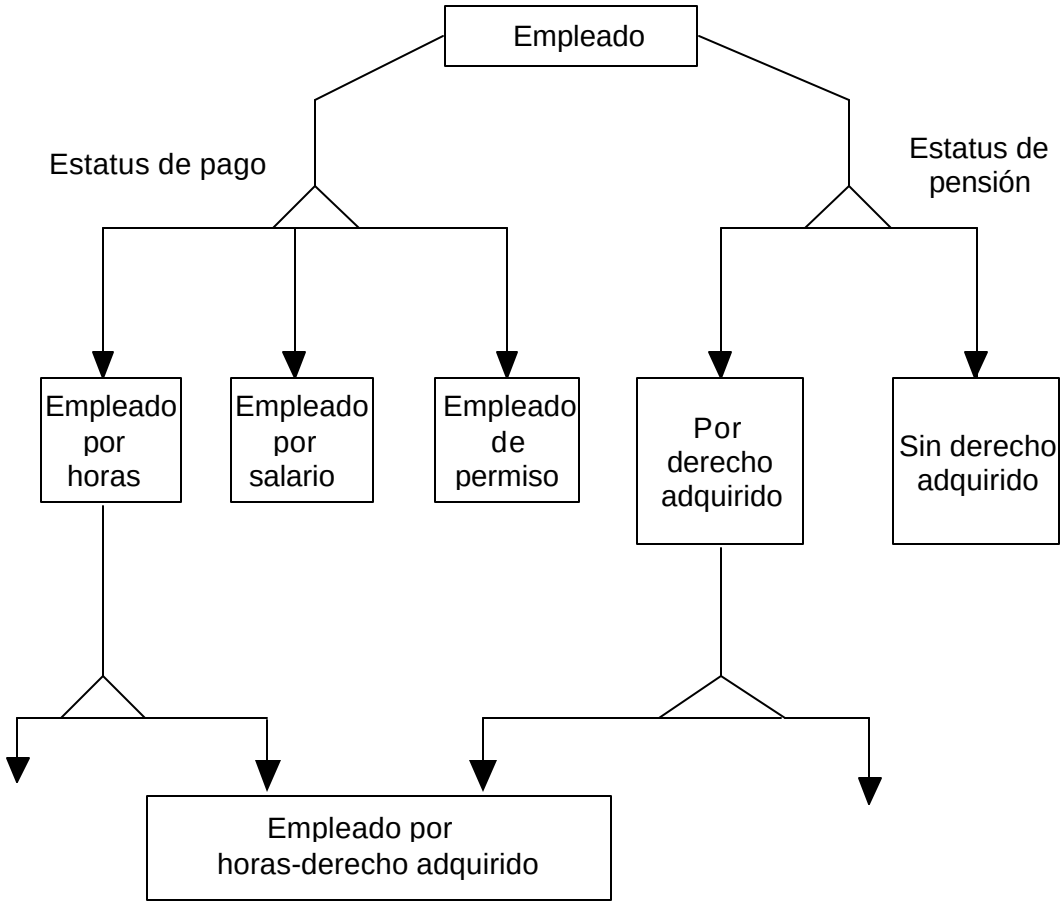


fig. 25 Herencia múltiple para clases disjuntas.

En la figura 25, la clase Empleado es independiente de las especializaciones, estatus de pago y estatus de pensión, por esta razón se muestra con dos generalizaciones separadas.

Las subclases de una generalizaciones pueden o no ser disjuntas. Por ejemplo, en la figura 24 Vehículo de tierra y Vehículo de agua se superponen porque algunos vehículos pueden viajar sobre agua y tierra, como es el caso de los vehículos anfibios de la fuerza naval. Por otro lado se observa de la figura 25 que Empleado por horas, Empleado por salario, Empleado por permiso son disjuntos.

30. -Encadenamiento Dinámico:

Una de las ventajas que promueve el estilo de programación orientada por objeto es la característica del encadenamiento dinámico, también llamado encadenamiento tardío. En efecto, no se tendrían sistemas orientados por objeto sin esa poderosa capacidad.

Simplemente, la declaración encadenamiento dinámico significa que el sistema encadenará una rutina a un selector para un método particular que está implantado sobre un objeto clase. La capacidad del encadenamiento a tiempo de ejecución es necesario por:

1.- El mismo mensaje o nombre del método (es decir, selector) puede utilizarse por diferentes clases (overloading)
2.- Una variable objeto de una clase puede ser no conocida hasta el tiempo de corrida (por ejemplo cuando el lenguaje es menos fuerte en tipos y las variables pueden ser objetos de diferentes tipos en un mismo programa).

Un ejemplo prototipo para ilustrar la ventaja del encadenamiento dinámico es la escritura de un mensaje que se aplica a todos los elementos de una colección heterogénea de objetos. Asumiremos tener una pila que pueda contener algunas especies de objetos y estamos interesados en imprimir toda la pila. Por lo tanto, asumiremos que un método de impresión con una particular implantación se asocia con cada clase (de aquí esos métodos de impresión son distintos y sin relación alguna). Asumiremos que la pila es ST y se implanta como un arreglo de 1 hasta el tope de objetos, el pseudocódigo para imprimir todos los elementos de la pila es for P:=1 to tope do print(ST [i]), de aquí que cada objeto ST [i] será ejecutado con su apropiado método de impresión, dependiendo de la clase a la cual el pertenece; esto se ilustra en la figura 26.

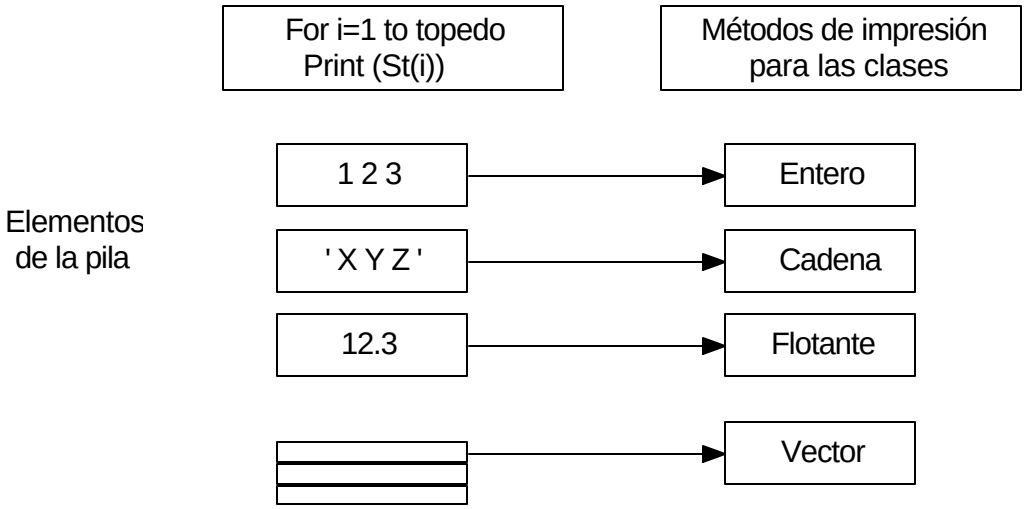


Fig. 26 Encadenamiento Dinámico a tiempo de ejecución.

El puntero al objeto es el que decide que pieza de código se ejecuta para imprimir el mensaje.

28.- Stream de entrada salida.

Un stream puede ser considerado como una secuencia de caracteres que potencialmente puede ser "infinita". Los elementos para entrada son removidos de la cabeza de un input stream y los elementos de salida son adicionados al final de un outpu stream.

29.- Ejemplos de programas codificados en C++ empleando todo lo estudiado en esta guía.

```
/*****************************************************************/
/*  Persona.h                                                    */
/*****************************************************************/
#include<string.h>
class persona {
// Mienbros Datos o Estructura de datos de la clase
// public:
        char nombre[15],apellido[15],cedula[15];

        // Miembros Funci¢n o M,todos de la clase
        public:
        persona();      // Constructor de la clase
   //   ~persona();     // Destructor de la clase
        virtual void presenta_valores();
        void asigna_nombre(char *nombre_nuevo);
        void asigna_apellido(char *apellido_nuevo);
        void asigna_cedula(char *cedula_nueva);
        void cumpleano();
```

```
        }; // Observe que se debe finalizar con punto y coma.

class empleado : public persona {
        float salario;
        char departamento[15],escalafon[15];
        public:
        empleado();
  //    ~empleado();
        void asigna_salario(float &sal);
        void asigna_dept(char *dept);
        void asigna_esc(char *esc);
        void presenta_valores();
        };
class estudiante : public persona {
        char universidad[26],carrera[26];
        float nota_promedio;
        public:
        estudiante();
   //   ~estudiante();
        void asigna_univer(char * univer);
        void asigna_carrera(char *carr);
        void asigna_notas(float nota);
        };
class beca_trab : public empleado,estudiante {
        float horas_trab;
        public:
        beca_trab();
   //   ~beca_trab();
        void asigna_horas(float horas);
        };



/*****************************************************************/
/*  Persona.cc                                                   */
/*****************************************************************/
# include<string.h>
# include<stdio.h>
# include "persona.h"       // incluye a persona.h declarado en la página anterior

// Implantación de los Miembros Función o M,todos de las clases
// que se emplear n en el curso.

// Implantación de las Funciones miembros de la Clase persona.

        persona::persona(){
        // Implantaci¢n del Constructor de la clase persona
                char *x="xxxxxxx";
                strcpy(nombre,x);
                strcpy(apellido,x);
                strcpy(cedula,x);
           }
```

```
void persona::presenta_valores(){
        printf("Los datos de la persona son:\n");
        printf("El Nombre: %s \n",nombre);
        printf("El Apellido: %s \n",apellido);
        printf("La C,dula: %s \n",cedula);
    }

void persona::asigna_nombre(char *nombre_nuevo) {
    // Funci¢n en linea
    strcpy(nombre,nombre_nuevo);
    } ;

void persona::asigna_apellido(char *apellido_nuevo){
    strcpy(apellido,apellido_nuevo);
   }

void persona::asigna_cedula(char *cedula_nueva) {
        strcpy(cedula,cedula_nueva);
   }

void persona::cumpleano(){
        presenta_valores();
        printf("Feliz cumplea¤o");
   }

// Implantación de los métodos de la clase Empleado

empleado::empleado() : persona() {
        char *x="xxxxxx";
        // Implantaci¢n del Constructor de la clase empleado
        salario=0.0;
        strcpy(departamento,x);
        strcpy(escalafon,x);
  }

//      empleado::~empleado();

void empleado::asigna_salario(float &sal){
        salario=sal;
  }

void empleado::asigna_dept(char *dept){
     strcpy(departamento,dept);
  }

void empleado::asigna_esc(char *esc){
     strcpy(escalafon,esc);
  }

void empleado::presenta_valores(){
     printf("El salario: %f \n",salario);
     printf("El departamento: %s \n",departamento);
     printf("El escalaf¢n: %s \n",escalafon);
   }
```

```
// Implantación de los métodos de la clase Estudiante

        estudiante::estudiante() : persona(){
                // Implantaci¢n del Constructor de la clase estudiante
                strcpy(universidad,"Universidad de Los Andes");
                strcpy(carrera,"Ingenier¡a de Sistemas");
                nota_promedio=14.0;
          }
  //    estudiante::~estudiante();

        void estudiante::asigna_univer(char *univer){
                strcpy(universidad,univer);
          }

        void estudiante::asigna_carrera(char *carr){
                strcpy(carrera,carr);
          }

        void estudiante::asigna_notas(float nota){
                nota_promedio=nota;
          }

// Implantación de los métodos de la clase derivada Beca trabajo

        beca_trab::beca_trab():empleado(),estudiante(){
                // Implantaci¢n del Constructor de la clase beca trabajo
                horas_trab=0.0;
            }

    //  beca_trab::~beca_trab();

        void beca_trab::asigna_horas(float horas){
                horas_trab=horas;
        }

/*****************************************************************/
/* creación de un objeto de la clase persona                     */
/*****************************************************************/
# include<string.h>
# include<stdio.h>
# include "persona.h"
void main()
{
// Declaración de una instancia u objeto de la clase persona
persona p1;

char nom[15],apellido[15],cedula[15];
// Invocación de los métodos sobre la estructura de dato de la clase persona

//printf("\n Los valores que contiene la clase actualmente son: %s \n");
//p1.presenta_valores();

printf("Por favor Introduzca el Nombre de la persona:  ");
gets(nom);
```

```
p1.asigna_nombre(nom);
printf("Por favor Intruduzca el Apellido de la persona:  ");
gets(apellido);
p1.asigna_apellido(apellido);
printf("Por favor Introduzca C,dula de la persona:  ");
gets(cedula);
p1.asigna_cedula(cedula);
// printf("\n Los valores que contienen la clase es: %s \n");
p1.presenta_valores();

}

/*********************************************************************/
/* Programa que permite mostrar el concepto de Herencia sencilla     */
/*********************************************************************/
# include<string.h>
# include<stdio.h>
# include "persona.h"
void main()
{
// programa que demuestra el uso de la Herencia Simple
// Declaraci¢n de una instancia u objeto de la clase persona
empleado e1;
// declaraci¢n de las variables locales
float sal;
char dept1[15],esc1[15];

printf("Introduzca el Salario: ");
scanf("%f",&sal);
e1.asigna_salario(sal);
printf("\n Introduzca el Departamento: ");
scanf("%15s",dept1);
e1.asigna_dept(dept1);
printf("\n Introduzca el Escalafon: ");
scanf("%15s",esc1);
e1.asigna_esc(esc1);
e1.asigna_nombre("Pedro");
e1.asigna_apellido("Camejo");
e1.asigna_cedula("V 8.345.560");

e1.presenta_valores();
e1.cumpleano();
}

/*********************************************************************/
/* Programa que permite mostrar el concepto de Herencia múltiple     */
/*********************************************************************/
# include<string.h>
# include<stdio.h>
# include "persona.h"
void main()
{
// programa que demuestra el uso de la Herencia Múltiple
// Declaración de una instancia u objeto de la clase persona
```

```
beca_trab b1;

// Declaración de las variables locales
float horas,sal;
printf("Introduzca la cantidad de Horas que trabaja al día : ");
scanf("%f",&horas);
b1.asigna_horas(horas);

// Métodos que pertenecen a la clase empleados
printf("Introduzca el Salario: ");
scanf("%f",&sal);
b1.asigna_salario(sal);
b1.asigna_dept("Computación");
b1.asigna_esc("profesor");
}

/*****************************************************************/
/* Código que muestra como implantar una lista en c++   (León94)          */
/***************************************************************** /
// LISTAS.H
#include <stdlib.h>
#include <stdio.h>
#define FINLST NULL
#define SIG(lista) *(void **)((char *)lista + largo)
class LISTAS {
private:
 void *actual, *cabeza;
 int largo, n, pos;

 void error (char *mens) {
    printf ("%s \n",mens);
    abort();
 }
 void nodo_falso (int tam);
public:
  LISTAS(int tam) {

    n = pos = 0;
    nodo_falso (tam);
  }
  void *pri_lst();
  void *suc_lst();
  void *sig_lst();
  void *pre_lst();
  void *ant_lst();
  void *cab_lst() { return SIG(cabeza); }
  void *con_lst() { return SIG(actual); }
  void *ins_lst(void *elem);
  void *eli_lst();
  void anu_lst();
  void des_lst();
  int num_lst() { return n; }
  int pos_lst() { return pos; }
};
```

```
/*********************************************************************/
/* Código que implanta los métodos de una lista en c++  (León94)     */
/*********************************************************************/
// LISTAS.CPP
#include "listas.h"
#include <alloc.h>
#include <mem.h>

#define SIG(lista) *(void **)((char *)lista + largo)

void LISTAS::nodo_falso (int tam) {
  void *nodo = (void *) malloc (tam + sizeof(char *));

  if (nodo != NULL) {
    largo = tam;
    actual = cabeza = nodo;
    SIG(nodo) = NULL;
    return;
  }
  error ("LISTAS(): Memoria no disponible !!!");
}

void *LISTAS::pri_lst () {

  pos = 0;
  actual = cabeza;
  return SIG(cabeza);
}

void *LISTAS::suc_lst() {

  if (SIG(actual)==FINLST)
     error("suc_lst(): Actual en fin lista !!!");
  return SIG(SIG(actual));
}

void *LISTAS::sig_lst() {

  if (SIG(actual)==FINLST)
     error ("sig_lst(): Actual en fin lista !!!");
  pos++;
  actual = SIG(actual);
  return SIG(actual);
}

void *LISTAS::ant_lst() {
  void *nodo = cabeza;

  if (pos == 0)
    error ("ant_lst(): Actual en pos = 0");
  while (SIG(nodo) != actual)
    nodo = SIG(nodo);
  pos--;
  actual = nodo;
```

```
  return SIG(actual);
}

void *LISTAS::pre_lst() {

 if (pos == 0)
   error ("pre_lst(): Actual en pos = 0 !!!");
 return actual;
}

void *LISTAS::ins_lst (void *elem) {
  void *nodo = (void *) malloc (largo + sizeof(char *));

  if (nodo != NULL) {
    SIG(nodo) = SIG(actual);
    SIG(actual) = nodo;
    n++;
    return memmove (nodo,elem,largo);
  }
  return NULL;
}

void *LISTAS::eli_lst() {
  void *nodo = SIG(actual);

  if (n == 0)
    error ("eli_lst(): Lista vacia !!!");
  n--;
  SIG(actual) = SIG(nodo);
  free (nodo);
  return NULL;
}

void LISTAS::anu_lst() {
  void *nodo;

  pri_lst();
  while (con_lst() != FINLST) {
    eli_lst();
  }
}

void LISTAS::des_lst() {

  anu_lst();
  free(cabeza);
}
```

```
/*****************************************************************/
/*Código que muestra el uso de la clase lista (León94)           */
/*****************************************************************/

// PRULIST.CPP

#include "listas.h"
#include <conio.h>
#include <stdio.h>
#include <iostream.h>
#include <alloc.h>

#define escribe(x,y,num)  gotoxy(x,y);cprintf("%i",num)
#define color(x,y)  textbackground(x);textcolor(y)

void esc_lista (LISTAS lista) {
  int *nodo, pos_act, i=7;

  pos_act = lista.pos_lst();
  nodo = (int *)lista.pri_lst();
  gotoxy (1,17); printf ("lista : ");
  while (nodo != FINLST) {
    i+=3;
    if (lista.pos_lst() != pos_act) {
            escribe (i,17,*nodo);
    } else {
            color(1,5);
            escribe (i,17,*nodo);
            color(0,15);
    }
    nodo = (int *)lista.sig_lst();
  }
  gotoxy (i+3,17);
  if (pos_act == lista.num_lst()) {
    color(1,5);
    cprintf ("<f>");
    color(0,15);
  } else
    printf ("<f>");
}

void status_lista (LISTAS lista) {
  int p,n,pred,suce;

  n = lista.num_lst();
  gotoxy (1,19); printf ("Nro. elementos  : %i ",n);
  p = lista.pos_lst();
  gotoxy (1,20); printf ("Posicion Actual : %i",p);
  pred = (p > 0) ? *(int *)lista.pre_lst() : -1;
  gotoxy (1,21); printf ("Predecesor Act. : %i ",pred);
  suce = (p < n-1) ? *(int *)lista.suc_lst() : (p==n) ? -1 : 0;
  gotoxy (1,22);
  if (suce)
     printf ("Sucesor Actual  : %i ",suce);
```

```
   else
      printf ("Sucesor Actual  : <f> ");
}

void main (void) {
  unsigned long m1 = coreleft(), m2;
  LISTAS lista(sizeof(int));
  int num, opcion;

  do  {
     clrscr();
     gotoxy (1,1);
     printf ("1.- Insertar \n");
     printf ("2.- Eliminar \n");
     printf ("3.- Primero \n");
     printf ("4.- Siguiente \n");
     printf ("5.- Anterior \n");
     printf ("6.- Anular \n");
     printf ("7.- Finalizar \n");
     printf ("8.- Abandonar \n");
     printf ("opcion : ");
     status_lista (lista);
     esc_lista (lista);
     gotoxy(10,9); cin >> opcion;
     switch (opcion) {
       case 1 : printf ("introducir numero  : ");
                        cin >> num;
                        lista.ins_lst(&num);
                        break;
       case 2 : lista.eli_lst();
                        break;
       case 4 : lista.sig_lst();
                        break;
       case 6 : lista.anu_lst();
                        break;
       case 5 : lista.ant_lst();
                        break;
       case 3 : lista.pri_lst();
                        break;
       case 7 : lista.des_lst();
       case 8 : break;

       default : printf ("opcion no valida !!!");
     }
  } while (opcion != 8 && opcion != 7);
  m2 = coreleft();
  printf ("Memorias: Entrar => (%lu,%lu) <= Salir",m1,m2);
  getch();
}
```

```
/*****************************************************************/
/* Programa que permite mostrar el uso de clases paramétricas    */
/*****************************************************************/
// Ejemplo que demuestra las clases parametricas
#include <stdlib.h>
#include <iostream.h>
template<class T>
struct celda{
        T item;
        celda *prox;
};
template<class T>
class cola {
        celda<T>* frente;          // puntero al primero
        celda<T>* fondo;       // puntero al segundo
        int long;
public:
        cola();
        int vacio() {return frente == 0;} // chequea si la cola esta vacia
        void encole(T x);  // inserte al frente
        T decolar();                    // remueve un elemento de la cola
        int numelemt() {return long;}
        ~cola();
};
#define elemento(q)         q->item
#define  proximo(q)         q->prox
void Error(char *mensg){
   cout << endl << mensg << endl;
   exit(1);
}
template <class T> cola<T>::cola(){
        frente=fondo=long=0;
}
template <class T> void cola<T>::encole(T x){
        celda<T> * nuevo_elto =new celda<T>;
        elemento(nuevo_elto)=x;
        if (vacio())
          frente=nuevo_elto;
        else
          proximo(fondo)= nuevo_elto
        fondo = nuevo_elto;
        long++;
}
template <class T>cola<T>::decole(){
  if (vacia())
    Error("cola vacia");
  celda<T> *f = frente;
  T item = elemento(frente);
  frente = proximo(frente);
  delete frente;
  if (f== fondo)
    frente=fondo=0;
  long--;
  return item;
```

```
}
template <class T> cola<T>::~cola(){
        while(!vacio())
          decole();
}

void main(){
        cola<char> c;
        cola<double> d;
}

/*****************************************************************/
/* Código que muestra el uso de archivos en c++                  */
/*****************************************************************/
// ***************************************************************
// programa que permite insertar estudiante y consultar estudiantes
// Realizado por:Domingo Hernandez Hernandez
// fecha: 12-03-97
// ***************************************************************

#include <iostream.h>
#include <fstream.h>
#include<stdlib.h>


// ****  Definición de las constantes

typedef int bool;
const int true = 1;
const int false = 0;

// **** Definición del registro estudiante

typedef  struct Estudiante
{
   char     nombre[30];
   char        apellido[30];
   char        cedula[10];
   float       trab1,trab2,trab3,seminario,prom,porcen,parc1,parc2,promp,porcenp,def;
};

// ********** Declaración de prototipos
void insertar_datos_personales(void);
int leeopc(void);
int llamaopcion(int opc);
void consultar(void);
void writeRecord(ostream &os, const Estudiante &r);
void readRecord(istream &is,Estudiante &r);
void insertar_notas(void);
void calcular_nota_def(void);
void writePantalla(ostream &os, const Estudiante &r);

// **** declaración de procedimientos
```

```
//*************** Función que lee una opción del teclado ********

//******** Fuction Error

void error(char *mss1,char *mss2) {
   cerr << mss1 << ' ' << mss2 << '\n';
   exit(1);
  } // End of Function error


int leeopc(void) {
  bool time = true;
  int opc;
  do {
          cout << "Opciones: \n";
          cout << "(1) Insertar datos personales \n" << "(2) Insertar notas \n" << "(3) consultar \n"  << "(4)
calcular nota definitiva \n" << "(5) salir \n";
          cout << "Por favor introduzca el número de la opción: ";
          cin >> opc;
          if (opc < 1 || opc > 5)
            cout << '\n' << "Error -- el número de la opción no se encuentra en el rago \n";
          else
           time = false;
    } while (time);
  return  (opc);
  } // fin del procedimiento lee opción.

 //******** Procedimiento que llama la opción seleccionada  **********************

int llamaopcion(int opc) {
 switch (opc) {
          case 1 :
                  insertar_datos_personales();
          break;

          case 2:
                  insertar_notas();
          break;

          case 3:
                  consultar();
                  break;

          case 4:
                  calcular_nota_def();
          break;

      case 5 : return opc;

    }
  return opc;

  }  // Fin del procedimiento que invoca las opciones
```

```
// captura los datos de un estudiante

void insertar_datos_personales(void){
            ofstream archivo_est("c:\domingo\programa\estudian.dat");
            Estudiante est;
            int numest,i;
            cout << "Introduzca el número de estudiantes a insertar:";
            cin >> numest;
            cout << "\n";
            for (i=1;i<=numest;i++) {
                cout << "Introduzca los siguientes datos para el estudiante número:" << i <<" \n";
                cout << "Nombre: ";
                cin >> est.nombre;
                cout << "Apellido: ";
                cin >> est.apellido;
                cout << "Cédula: ";
                cin >> est.cedula;
                est.trab1=0.0;est.trab2=0.0;est.trab3=0.0;est.seminario=0.0;est.prom=0.0;
                est.porcen=0.0;est.parc1=0.0;est.parc2=0.0;est.promp=0.0;est.porcenp=0.0;
                est.def=0.0;
                writeRecord(archivo_est,est);
            }
}

// consulta todos los estudiantes

void consultar(void){
   ifstream archivo_est("c:\domingo\programa\estudian.dat");
   if (!archivo_est) {
              error("Error -- cannot open input file","c:\domingo\programa\estudian.dat");
   }
   else {
      Estudiante *est = new (Estudiante);
      while (!archivo_est.eof()) {         // Read from of file
         readRecord(archivo_est,*est);
               writePantalla(cout,*est);   // Output to screen
      }
           delete est;
           archivo_est.close();
   }
}

// escribe el registro de un estudiante en el archivo estudiante


void writeRecord(ostream &os, const Estudiante &r) {
 os << r.nombre << ' '<< r.apellido << ' ' << r.cedula << ' ' << r.trab1 << ' ';
 os << r.trab2 << ' ' << r.trab3 << ' ' << r.seminario << ' ' << r.prom << ' ' ;
 os << r.porcen << ' ' << r.parc1 << ' ' << r.parc2 << ' ' << r.promp << ' ' ;
 os << r.porcenp << ' ' << r.def <<'\n';
 } // End funtion writeRecord

 // Escribe el registro de un estudiante por pantalla
```

```
void writePantalla(ostream &os, const Estudiante &r) {
 os << "          Datos del estudiante " << '\n';
 os << "Nombre: " << r.nombre << "  Apellido: " << r.apellido;
 os << "  Cédula: " << r.cedula << '\n';
 os << '\n';
 os << "          Notas de  los trabajos Prácticos " << '\n';
 os << "Trab1:" << r.trab1 << " Trab2:" << r.trab2;
 os << " Trab3:" << r.trab3 <<" Seminario:" << r.seminario;
 os << " Prom_Trab:" << r.prom << " Porcen_Trab:" << r.porcen <<'\n';
 os << '\n';
 os << "          Notas de los parciales " << '\n';
 os << "Parcial1:" << r.parc1 << " Parcial2:" << r.parc2;
 os << " Prom_parciales: " << r.promp <<" Porcen_parciales: " <<r.porcenp;
 os << " Def: " << r.def <<'\n';
 } // End funtion writeRecord


 //******* Función lee registro

void readRecord(istream &is,Estudiante &r) {
 is >> r.nombre >> r.apellido >> r.cedula >> r.trab1 >> r.trab2 >> r.trab3 >> r.seminario;
 is >> r.prom >> r.porcen >> r.parc1 >> r.parc2 >> r.promp >> r.porcenp >> r.def >>'\n';
} // End of function readRecod

// Procedimiento que permite insertar la nota de un estudiante..

void insertar_notas(void){
  cout << "Procedimiento no implantado aún....." << "\n";
}

// procedimiento que permite cálcular la nota definitiva .....

void calcular_nota_def(void){
  cout << "Procedimiento no implantado aún....." << "\n" ;
}

int main(){
   int opcion=1;
   while (opcion !=5)
     opcion=llamaopcion(leeopc());
   return 0;
}
```

Bibliografía

Rumbugh James, Willian Lorensen Object Oriented Modeling and Desing, Prentice Hall 1991

Borland C++ Getting Started 1991.

Sharan Hekmatpour, C++ Guía para Programadores en C, Prentice Hall 1991

Bjarne Stroustrup El C++ Lenguaje de Programación, Addison Wesley 1993

Stephen C, Stark Kathy Programing in C++, Prentice Hall 1989.

Kim Wom Objet Oriented Concepts, ACM press1989.