Curso de Introducción a C++

Para Programadores en C

Luis de Salvador

INDICE

1. Introducción a las Clases

Una clase es un tipo definido por el usuario, como las estructuras y uniones. La definición de clase se emplea para especificar un conjunto de datos que forman parte de la clase así como el conjunto de operaciones que se pueden emplear sobre los objetos de dicha clase.

Una clase se define básicamente de la siguiente forma:

class *nombre_de_la_clase* {
<parte privada>
public:
<parte pública>
};

En la parte privada se definen los datos (atributos) y las funciones (métodos) que son accesibles unicamente por funciones que sean miembros o **friend** (amigas) de las clase. Las funciones miembro son aquellas funciones que se encuentran definidas en el ambito de la clase y no fuera de ese ambito (ambito de la clase, que es diferente del ambito de declaración de la clase).

Por defecto, todas las definiciones realizadas antes de la palabra **public** son privadas. Se puede incluir la palabra **private** para declarar una parte privada. Se pueden incluir tantas partes privadas como públicas que se deseen.

En la parte publica se definen los datos y las funciones que son accesibles fuera del ambito de la declaración de la clase.

Ejemplo de definición de una clase

```
#include "stdio.h"

class string {
    /* objeto privado de la clase */
    char *str;
public:
    /* Constructor de la clase */
    string() {str = new char[10];
              *str = 'a';
              *(str+1) = NULL; }
  /* Función miembro de clase */
    void print()  {    printf("%s\n", str); }
    /* Función amiga de la clase */
    friend int null_test ( string s){
              return (*s.str == NULL);
    }
};

void main () {
string s;              /* inicialización del objeto*/

  if (!null_test(s))   /* manipulación de la clase */
              s.print();

return;
}
```

Notas sobre el ejemplo:

?? La parte privada de la clase es un puntero a una cadena de caracteres Por lo tanto, esta cadena de caracteres solo podrá ser accedida desde funciones que sean miembro o amigas de la clase.

?? Las funciones miembro de una clase no necesitan referenciar a los objetos privados de la clase via **clase.miembro**.

?? Las funciones miembro de una clase se invocan a partir de un objeto de la clase (**s.print()**). Este "**.**" se denomina operador de acceso a un miembro de la clase.

?? Una de las funciones definidas en la parte pública ha de ser una función ***constructora***. Esta función es la empleada para realizar las declaracion de una variable como de tipo la clase considerada y, a su vez, inicializarla. En esta función se prevee la devolución de ningún tipo.

?? La función para reservar espacio en memoria es **new**, la función para liberarlo es **delete**.

Es posible que un miembro de la clase sea del mismo tipo que la clase. De igual forma que ocurría en la declaración de estructuras, este miembro ha de ser puntero al tipo de la clase.

Una clase puede ser declarada y no definida, para ser definida posteriormente. La declaración de una clase sin cuerpo tiene la forma:

class nombre_de_clase;

Ejercicio 1: Crear un objeto llamado **nodo** que se el núcleo de un arbol binario. Por lo tanto, que sea capaz de almacenar dos punteros a dos nodos. Cada nodo puede almacenar los siguientas datos de forma excluyente: o bien un valor **double** o bien otro puntero a otro nodo del mismo arbol.

Escribir una función constructura y las siguientes funciones amigas:

void crear (nodo *, int)

Crea un nuevo nodo a la izquierda o a la derecha del nodo actual en función de un parámetro en int.

void cambiar_v (nodo *el_nodo, double *valor)

Añadir al nodo actual el valor especificado.

void cambiar_n (nodo *el_nodo, nodo *nuevo)

Añadir al nodo actual un nuevo puntero.

nodo* viajar (nodo *el_nodo, int valor)

Devuelve el nodo izquierdo o derecho en función del valor entero.

Escribir una función **main** que compruebe el correcto funcionamiento de las funciones.

2. Punteros a funciones

Tanto en C como en C++ se pueden utilizar punteros a funciones. El puntero a una función es el mismo nombre de la función. Dada la función:

float procesa **(int, int*);**

El puntero a dicha función es *procesa*. Para declarar que una variable es de tipo puntero a una función:

float (*p**) (int, int*);**

Este es un puntero *p* a una función que devuelve un flotante y tiene dos parámetros: un entero y un puntero a entero. El tipo genérico es el siguiente:

float (*) (int, int*)

Para declarar un array de punteros a funciones se realiza de la siguiente forma:

float (*p**[20*?*)(int, int*);**

O bien un puntero a un array es el siguiente:

float (**p**)**(int,int*)**

La ejecución de una función cuyo puntero está en un array es:

c = p[x?(a, b);

Donde **x** es un índice del array.

Ejercicio 2: Realizar una clase que almacene un vector de funciones que devuelven un entero y que tengan como argumento dos enteros. Manejar el vector como una pila. Crear varias funciones de operación entre enteros (suma, multi, div, ...). Crear dos funciones miembro de la clase: una que tome como argumento un puntero a función (función del tipo que devuelve un entero y tiene como argumentos dos enteros); otra que tome como argumentos dos enteros y devuelva la ejecución de los mismos bajo la función que está en la parte superior de la pila. Crear un programa principal que demuestre el buen funcionamiento de estas dos funciones miembro.

3. Declaración de funciones en una clase

En una clase se pueden definir dos tipos de funciones: las funciones miembro y las funciones amigas.

Las funciones miembro son aquellas que se declaran en el ámbito de una clase. Para invocarlas, es necesario utilizar el operador de acceso a una clase. Las funciones miembro se pueden definir dentro o fuera de la clase. Para ello es necesario utilizar el operador de resolución de ámbito "**::**" como en el siguiente ejemplo:

```
class prueba {
    char * str;
    int funcion (prueba);
};
int prueba::función( prueba p) { .....};
```

En cualquier caso, la definición se considera dentro del ámbito de definición de la clase por lo que se pueden utilizar los nombres de los miembros de la clase directamente, sin necesidad de hacer uso del operador de acceso a un miembro de una clase "**.**".

Cuando la definición de la función se realiza fuera del ámbito de la clase, esta se puede declarar como **inline**. En ese caso se considera reescrita dentro del ámbito de la función con las restricciones de ámbito asociadas. El siguiente ejemplo muestra la diferencia:

```
int b = 2;
class x {
    char* b;
    char* f();
};
char* x::f() { return b; };     // devuelve b, que es igual a 2
inline char* x::f() { return b; };  // devuelve x::b
```

Las amigas de una clase son funciones que no son miembros de la clase pero que pueden acceder a los nombres definidos en el ámbito de una clase. Las funciones amigas no se ejecutan utilizando el operador de acceso a miembros de una clase (a menos que sean miembros de otra clase).

Una función amiga puede ser definida fuera de la declaración de clase. En caso contrario, la función se considera como **inline** y se aplican las reglas de reescritura que corresponden a ella. La funciones amigas definidas en una clase se consideran semánticamente en el ámbito de la clase en cuestion.

Una función amiga puede serlo de dos clases o más simultáneamente. Una función amiga de una clase, puede ser miembro de otra. Todas las funciones de una clase pueden ser amigas de otra clase, por ejemplo:

```
class X { ... };
class Y {
    friend class X; };
```

Ejercicio 3: Declarar y definir dos clases. La primera es la clase *vector*. Esta clase contiene un vector de flotantes y un indice de ocupación para indicar cual es la primera posición libre en el vector. Define un constructor que inicialize el tamaño de vector a un valor MAX (indicado en un #define). Define una función miembro que añada un valor al vector e incremente el índice:

void annade(float);

Define una clase *string*. Esta clase contiene un vector de caracteres. Define un constructor que inicialize el tamaño a un valor MAX+1. Define una función miembro que imprima el string:

void print();

Define una función amiga en ambas clase que convierta directamente los valores contenidos un objeto de clase *vector* en un objeto de clase *string* mediante un **casting** de cada uno de los elementos del vector al string.

Crea una función **main** que compruebe el buen funcionamiento de las funciones definidas.

4. Otras características de las funciones en C++

Características de las funciones en C++ sobre las definidas en C son las siguientes:

?? Sobrecarga (overload).

?? Operadores

?? Funciones en línea (inline).

?? Argumentos con inicializadores por defecto.

4.1 Sobrecarga

Una función se dice que está sobrecargada si un mismo nombre de función se emplea para implementar distintas funciones que trabajan con distintos tipos de datos. Por ejemplo:

```
void print (int v) { printf("%d", v); return;};
void print(float) { printf("%f", v); return;);
```

En este caso, las funciones se distinguen por el tipo. Es necesario que el tipo sea lo suficientemente distinto para que se realice la distinción. Por ejemplo, esta distinción es insuficiente:

```
void print (int v) { ... };
void print (int& v) { ... };
```

Lo mismo ocurre con parámetros como **const** y **volatile** o aquellos distinguidos en el **typedef**. Tampoco es suficiente que solo difieran en el tipo de retorno.

Por definición, dos declaraciones de funciones con el mismo nombre se refieren a la misma función si están en el mismo ámbito y tienen el mismo argumento.

Atención: Una función miembro de una clase derivada no se encuentra en el mismo ámbito que una función miembro de la clase base que tenga el mismo nombre. Asimismo, una función declarada localmente no esta en el mismo ámbito que una función declarada en el fichero. Ejemplo:

```
int f(char*)
void g()
{ extern f(int);
  f("asdf");      // ERROR }
```

Un puntero a una función sobrecargada se identifica por lista de parámetros que le acompaña:

```
int f(char*);
int (*puntero_a_funcion) (char*);
```

4.2 Operadores

En C++ es posible dar una mayor definición a los operadores aritméticos (y otros). Por ejemplo:

```
friend string operator + ( string, string);
string a, b, c;
c = a+b;
```

Como se denota en el ejemplo, las funciones **operator** se han de definir en el ámbito de una clase como funciones amigas. Son funciones sobrecargadas. Se pueden sobrecargar:

```
new delete + - * / % ^ & | ! =  < > += -= *= /= %= ^= &= != << >> >>= <<= == != <= >=
                        && || ++ -- , ->* -> () []
```

4.3 Funciones Inline

Una función se declara como función en línea cuando se añade la palabra clave **inline** antes de la definición de la función. El efecto que produce es que el código de la función se sustituye en los puntos de llamada a la función, consiguiendo de esta forma el ahorro en la llamada a la función. Por ejemplo:

```
inline int suma(int,int);
inline int suma(int a, int b) {
    return a+b; }
void main() {
int a = 5, b = 3,c;
    c = suma(a,b);
return; }
```

4.4 Constructores

Los constructores son funciones miembro de una clase que tiene el mismo nombre de la clase.

Un constructor por defecto siempre existe en una clase aunque no se haya especificado ninguna función que los implemente. Se consideran constructores por defecto aquellos que se declaran sin argumentos.

El constructor como conversión o inicializador con parámetros se declara como una función miembro con parámetros con inicialización por defecto. Por ejemplo:

```
class complex { float r, i;
public: complex ( float real = 0, float imag = 0) { r = real; i = imag; }; ....};
```

La declaración de esta función se puede emplear como sigue:

```
complex a (5, 3); complex b; complex c = complex (3,4);
```

Algunas de las características de los constructores, que se determinarán más adelante, son:

?? Un constructor puede ser invocado por un objeto **const** o **volatile**.

?? Un constructor no puede ser **virtual** ni **static**.

?? Los constructores no se heredan.

4.5 Funciones con un número variable de argumentos

Una función en C++ se puede llamar con un número variable de argumentos. Para ello, en la declaración de la función es necesario suministar inicializadores por defecto de los argumentos.

```
#include <stdio.h>
int funcion (int, int, int , int b = 3);
int funcion (int x, int y, int a = 1, int b ) {
return x+y+a+b; }
void main() {
int x = 5, y = 9, r;
    r = funcion(x,y);
  printf("%d", r);
return; }
```

La declaración de inicializadores por defecto se puede realizar tanto en la declaración del prototipo o en la definición del cuerpo de la función.

4.6 Funciones con un número indefinido de argumentos

Este tipo de funciones se puede realizar tanto en C como en C++. Son más flexibles que las funciones con número variable de argumentos. Para poder realizarlo hay que llamar a la librería **<stdarg.h>**.

En el prototipo de la función, se incluyen los parámetros fijos que se desean más un indicador del número de variables que se introducen en el caso concreto, más un indicativo de un número indefinido de variables:

```
int suma (int a, int b, int *lista, ....) { }
```

En esa lista se encuentran de alguna forma definidos los tipos y el número de los argumentos que se pasan en una llamada concreta. En el cuerpo del programa se declara una variable que contendrá el puntero a los argumentos indefinidos introducidos.

```
va_list ap;
```

Se asigna la dirección de los argumentos indicándole cual es el último argumento real introducido;

```
va_start(ap, lista);
```

Se extraen los argumentos indicando de que tipo son. El puntero se incrementa automáticamente:

```
va_arg(ap, tipo);
```

Finalmente, se anulan el espacio asignado a los argumentos:

```
va_end(ap);
```

Ejemplo:

```
#include <stdarg.h>
#include <stdio.h>
#define ENTERO 1
#define FLOTANTE 2
int suma ( int a, int *puntero_tipo_argumento,...) {
va_list ap;        // Declaro la variable que apuntara a los argumentos reales
  va_start(ap, puntero_tipo_argumento);  // Inicializa AP para almacenar el valor de los argumentos
  while (*puntero_tipo_argumento != NULL ) {  // Recorre la lista de argumentos
      switch (*puntero_tipo_argumento++) {
      case ENTERO:              // Si es entero
                        a = a+ va_arg(ap, int);      // Extrae
                        break;
      case FLOTANTE:            // Si es flotante
                        a = a+ (int) va_arg(ap,double);
        break; } }
  va_end(ap);   // Libera la lista de punteros
  return a; }
int main () {
  int result;
  int argumentos[] = { ENTERO, ENTERO, FLOTANTE, FLOTANTE, ENTERO, NULL };
  result = suma ( 2, argumentos, 2, 3, 1.5, 2.5, 3 );
  printf(" %d \n ", result);
  return 0;  }
```

Ejercicio 4: Declarar dos clases: números complejos en forma binomial(a+ib) y números complejos en forma polar ($a.e^b$). La primera se llamará *complex* y la segunda *polar*. Implementar los operadores { +, *, -, /, conj } para los nuevos tipos de datos. Asimismo, implementar las necesarias funciones de conversión. Implementar los constructores flexibles. Implementar una función llamada *acumula* que sume un número indefinido de números complejos y polares.

5. Introducción a la herencia

Herencia es un mecanismo que proporciona C++ para construir tipos de clase a partir de otras clases. Existen unos tipos de clases que se denominan *bases* mientras que las que se construyen a partir de estas se denominan clases *derivadas*.

La forma de definir una clase derivada de otra es:

class mi_derivada: **public** la_primitiva { .....};

La clase derivada hereda todos los miembros de la clase primitiva, incluyendo datos y funciones. La palabra **public** indica que los que es público en la clase primitiva también lo será en la derivada. Si se omite o se incluye **private**, los que es público en la primitiva será privado desde la derivada. Por ejemplo:

```
class lista {
    float elemento;
    lista *enlace;
    public:
    lista (float, lista*);
    lista * siguiente() { return enlace; }
    float valor();
};
class lista2 : public lista {
    lista2 *enlace;
public:
    lista2 (float, lista2*, lista2*);
    lista2 * anterior() { return enlace; }
}
```

De esta forma, se declara una lista doblemente enlazada de tipo **lista2** a partir de un lista simple. También se crea una jerarquía de ámbito de funciones. Fijarse cómo las dos variables tipo enlace no se confunden. Cuando se busca la variable **enlace** primero se busca en el ámbito de **lista2** y si no se encuentra se busca en **lista**. Para evitar confusiones y tener acceso, se puede usar el operador de ámbito, p.e.: **lista::enlace**.

La siguiente función accede a miembros heredados y normales:

```
lista2* antesde (lista2 *lst, float v ) {
lista2 *p;
    for ( p = lst; p; p = (lista2 *) p->siguiente())
            if (p->valor () == v)
                    break;
    if (p)
            return p->anterior();
    else
            return 0;
}
```

Observa como hay que hacer un **casting** del valor devuelto por la función **siguiente**. Es importante recordar que desde el ámbito de la clase derivada no se puede acceder a los miembros de la clase base. Lo siguiente es erróneo:

```
lista* lista2::siguiente() { return lista::enlace; }  // Incorrecto
```

Se podría incluir **return lista::siguiente().**

Para definir un constructor de la clase derivada se hace de la siguiente forma:

```
lista2: lista2( valor v, lista2 *p1, lista2 *p2) : lista ( v, *p1) { link = p2; };
```

En la definición, se añade a la declaración del constructor una llamada al constructor de la clase base, que inicializa sus variables miembro, y luego se inicializa las variables miembro que pertenecen exclusivamente a la clase derivada.

6. Entrada/Salida en C++

Las características adicionales de entrada/salida en C++ se consiguen incluyendo la librería

```
#include <iostream.h>
```

Esto permite que los operadores << y >> se encuentren sobrecargados con nuevas funcionalidades de entrada/salida. Estas se pueden usar con unos operandos que se denominan **streams** y que son:

cin	stream estándar de entrada
cout	stream estándar de salida
cerr	stream estándar de salida de error
clog	igual que el anterior, pero con un buffer que permite poder enviar una gran cantidad de mensajes simultáneamente.

El tipo de **cin** es una clase denominada **istream**. La clase de **cout, cerr** y **clog** es **ostream**. Ambas clases son derivadas de **ios**.

6.1 Salida

El operador de salida también se llama de inserción. Para sacar un conjunto de caracteres por pantalla se ejecuta el comando:

```
cout << mi_string << '=' << i + j;        // Tantos strings como se desee
```

Los tipos que soporta este operador son:

char, short, int, long, const char*, float, double, long double, void (direccion hexadecimal)

Para sacar un único carácter:

```
cout.put(ch);
```

Para sacar **n** bytes consecutivos empezando en la dirección **a**:

```
cout.write(a, n);
```

Existen manipuladores para adaptar la salida proporcionada por **cout** con algún formato. Se encuentran en el fichero de cabecera **<iomanip.h>**. Estos manipuladores son:

setw(n)	Saca la salida en un campo de **n** espacios
setfill(ch)	Utiliza como carácter de relleno **ch**
dec	Salida en decimal (por defecto)
hex	Salida en hexadecimal
oct	Salida en octal
endl	Inserta un *new-line*
ends	Inserta un caracter **null**
flush	Saca por la salida todos los datos que se encuentren en el buffer
setbase(n)	Establece la base de conversión a **n** (8,10,16)
setprecision(n)	Establece la precisión para flotantes en **n** dígitos
setiosflags(f)	Establece los bits de formato definidos en **f** que es tipo **long**
resetiosflags(f)	Resetea los bits de formato definidos en **f** que es tipo **long**

Ejemplos de salida son:

```
cout << setw(4) << x;// salida en un campo de 4 digitos
cout << setw(6) << x << endl;// salida en campo de 6 dígitos con cambio de línea
cout << setw(10) << setprecision(10) << x << endl;
```

Incluida en la clase **ios** se encuentran cada uno de los **flags** que se pueden actualizar vía las sentencias **re/setiosflags**.

```
enum    {
        skipws    = 0x0001,    // salta los espacios en blanco en la entrada
        left      = 0x0002,    // salida ajustada a la izquierda
        right     = 0x0004,    // salida ajustada a la derecha
        internal  = 0x0008,    // desplaza despues del signo o indicador de base
        dec   = 0x0010,        // conversion decimal
        oct   = 0x0020,        // conversión octal
        hex   = 0x0040,        // conversión hexadecimal
        showbase  = 0x0080,    // indicar base
        showpoint = 0x0100,    // forzar el punto decimal en la salida flotante
        uppercase = 0x0200,    // salida hexadecimal en mayúsculas
        showpos   = 0x0400,    // añade '+' a los enteros positivos
        scientific= 0x0800,    // notación científica
        fixed     = 0x1000,    // punto fijo
        unitbuf   = 0x2000,    // limpia el buffer despues de una entrada
        stdio     = 0x4000     // limpia lel buffer de salida y de error despues entrada
        };
```

Un ejemplo de empleo es:

```
cout << setiosflags(ios::fixed); // Flotantes en punto fijo
cout << resetiosflags(ios::uppercase); // salida hexadecimal en minúsculas
```

Se puede emplear en vez de **setiosflags** el comando **cout.setf().**

Se puede crear un insertador para un tipo definido por el usuario de la siguiente forma. Sea **vector** un tipo formado por dos valores flotantes:

```
ostream &operator<< (ostream &s, vector &v)
{ return s << "x = " << v.x " y = " << v.y << endl;
}
```

6.2 Entrada

Para la entrada se utiliza el operador >>. Si su primer operando es tipo **istream** tiene significado de entrada o extracción. Dada una variable **x** tipo **float** la sentencia:

```
cin >> x;
```

Lee un valor numérico de entrada. Se puede utilizar también como:

```
cin >> x >> y >> z;
```

Siendo **x, y,z** de los tipos declarados anteriormente. Tal y como está definida, la entrada se salta todos los caracteres en blanco. Para conseguir un único carácter, independientemente de que sea blanco o no se emplea:

```
cin.get(ch);          // ch de tipo char;
```

Para leer un carácter en el **stream** de entrada, pero sin extraerlo y que se pueda leer más tarde se emplea:

```
cin.putback(ch);
```

Para leer un string de tamaño fijo:

```
cin.get(str, 40);     // 39 caracteres + NULL
```

La lectura se cierra cuando se lee un carácter **\n**. Pero este último no se lee. Por lo tanto, para leer líneas es más interesante:

```
cin.getline(str,40); // Esta funcion detecta y lee el caracter de '\n'
```

Para leer un número de caracteres independientemente de cuales sean sus características se emplea:

cin.read(str,40);

Los manipuladores que soporta son:

setw(n), dec, hex, oct, setiosflags(f), resetiosflags(f)

además **ws** que salta cualquier tipo de carácter no imprimible. Por ejemplo:

cin >> **setw**(40) >> buf;

Lee 39 caracteres más un carácter **null** o bien lee hasta que encuentra un carácter no imprimible. Hay que tener en cuenta que todos los blancos iniciales se los ha saltado.

Se puede crear un extractor para un tipo definido por el usuario de la siguiente forma. Sea **vector** un tipo formado por dos valores flotantes:

```
istream &operator >> (istream& s, vector& u)
{   float x, y;
    s >> x >> y;
    u = vector(x, y);
    return s;
}
vector v;
cin >> v;
```

Ejercicio 5: Construir una clase, denominada *persona* que tenga como datos miembro *nombre*, *apellido*, *edad* (las variables tipo *string* definirlas como puntero a carácter). Definir funciones constructoras, así como funciones que saquen por pantalla los valores de una variable de tipo *persona* utilizando las facilidades de C++ y funciones de entrada por pantalla de los datos correspondientes a dicho tipo. Construir tipos derivados de la clase persona denominados *consumidor* y *deportista*. El primero con las mismas características y funciones que persona pero añadiendo variables que determinen su consumo de *gasolina*, *electricidad* y *gas* (flotantes) y funciones que permitan su actualización. El segundo añadiendo las variables *tipo_deporte* y *marca* (este último flotante). Prepara las salidas de estos últimos para que los números flotantes siempre queden alineados en su punto decimal.

7. Paso por referencia

Las referencias permiten establecer alias para los objetos que se han definido en el programa. Se emplean para pasar a las funciones los argumentos por referencia en vez de por valor. La referencia se indica con **&**.

Importante: una referencia siempre ha de tener una inicialización. Una vez se ha inicializado, esta referencia no puede cambiarse. Ejemplo:

```
int i;
int &r = i;  // r es una alias de i
```

El inicializador ha de ser el nombre de un objeto del tipo que se está referenciando.

```
r = 3;        // i toma el valor 3
int j, *ip;
j = i*r;      // j toma el valor 9
ip = &r;      // ip consigue la dirección de i
```

Si el inicializador para la referencia no es del tipo correcto, se crea un **objeto anonimo** del cual la referencia es un alias. De igual forma ocurre cuando se inicializa a referencia con algo que no es un objeto. Ejemplo:

```
double d;
int &r = d; // Se crea un objeto anonimo que no tiene que ver con d
int &s = 3;
```

Un uso de las referencias es para pasar argumentos por referencia en vez de usar punteros:

```
void input (int&, int&, int&);
int a, b, c;
input(a, b, c);        // Los valores de a, b y c son alterados en el interior de la
función
```

Como en cualquier paso por referencias tiene dos ventajas: permite la actualización de los parámetros de entrada y evita la sobrecarga de operaciones por inicialización de variables. Esto permite tambien eliminar la sobrecarga de operaciones en la devolución de parámetros con funciones:

```
lista& input ( parametros&);
```

La creación de objetos anonimos para inicializar referencias permite realizar conversiones para poder emparejar argumentos. Por ejemplo:

```
class complex {
    .....
    friend complex operator * (complex&, complex&);
};
complex V, Z;
......
V = Z * 12;
```

En este caso se ha creado un objeto anónimo tipo **complex** que se inicializa con **12**.

8. Tipos Calificados como Constantes

El calificador **const** indica que un objeto de cualquier tipo no puede ser cambiado, ni directamente ni a través de un puntero. Los parametros por referencia que son usados para prevenir la copia de todo el argumento pero que no desean ser alterados pueden ser declarados como **const**.

```
complex operator * (const complex&, const complex&);
```

Cuando se utiliza el calificador **const** en la declaración de punteros, la posición en la declaración indica si se desea que sea constante la dirección del puntero o bien el valor hacia el que apunta el puntero.

```
const char *p = buffer;
p = dir;      // correcto
*p = 'x';     // error
char * const q = buffer;
q = dir;      // error
*q = 'x';     // correcto
```

La dirección de un objeto constante no puede ser asignada a un puntero no constante, porque tal asignación puede derivar en un cambio del valor constante a través del puntero.

```
const char espacio = ' ';
const char *p = &espacio;   // correcto
char *p = &espacio;                    // error
```

9. Funciones virtuales

La utilización de clases derivadas tiene una gran ventaja. Es posible realizar una conversión automática de un puntero de la clase derivada a un puntero de la clase base:

```
base * p1;
derivada * p2;
p1->miembros_clase_base;
p1 = p2;     // Esta conversión es automática
```

De esta forma, puedo construir una lista de elementos de clase base que contenga realmente elementos de la clase derivada:

```
class animal {
public:
    print() { cout << "animal"; }};
};
class pez: animal {
public:
    print() { cout << "pez"; }};
class perro: animal {
public:
    print() { cout << "perro"; }};
void main () {
    animal *p[5];
    p[0] = new pez;
    p[1] = new perro;
....
};
```

Pero el vector **p** es de tipo animal. Si a continuación se ejecutan las siguientes sentencias:

```
p[0]->print();          // Resultado "animal"
p[1]->print();          // Resultado "animal"
(pez*) p[0])->print();          //Resultado "pez"
```

Esto es debido al ambito en el que se encuentran definidas las funciones miembro y a que dichas funciones son asignadas en tiempo de compilación. Para evitar esto se utilizan las funciones virtuales. Si se reescribe la clase animal como:

```
class animal {
public:
```

```
    virtual print() { cout << “animal”; }};
};
```

En este caso, la función **print** de la clase base **animal** es una función virtual al haber incluido la palabra clave **virtual**. La salida es:

```
p[0]->print();      // Resultado “pez”
p[1]->print();      // Resultado “perro”
```

La utilización de funciones virtuales permite la asignación dinámica de la función a emplear en tiempo de ejecución. Atención: si no existiese la definición de función **print** en la clase base, esta función no se considera miembro de los elementos de la lista, pues estos son de la clase base.

Ejercicio 6: Realizar un programa que implemente una lista cuyos elementos son de clase **componente**. Los miembros de esta clase son datos como: num_piezas, codigo, tipo, proveedor, ...; y funciones constructoras, de salida por pantalla y de consulta de la información. Realiza un conjunto de clases derivadas de la componente como son: **pantalla, teclado** y **ucp** (de unidad central). Cada una de estas clases derivadas tiene datos miembro adicionales: pulgadas, teclas y dimensiones. Además, cuando se manda a imprimir se sacan estos valores acompañados de la indicación que se trata de un rodamiento, etc. Utiliza funciones virtuales y paso por referencia en la implementación.

10. Templates

Las templates son un recurso de C++ que permite la generación de funciones y clases que sean adecuadas para un número indeterminado de tipos diferentes.

Sea el siguiente programa:

```
#include <iostream.h>
template <class T>
void swap(T &x, T &y) { T w = x; x = y; y = w;}
void main () {
int i=1, j=2;
float a=1.0,b=2.0;
    swap (i,j);
    swap (a,b);
    cout << i << j << endl;
    cout << a << b << endl;
return;
}
```

La función **swap()** se ha llamado con diferentes argumentos. Pero realmente se ha llamado a dos funciones diferentes que tienen el mismo nombre sobrecargado que son:

```
void swap(int &x, int &y) { int w = x; x = y; y = w;}
void swap(float &x, float &y) { float w = x; x = y; y = w;}
```

El parámetro **T** ha sido sustituido por el compilador por **float** e **int** ya que en el programa se realiza una llamada con cada uno de esos parámetros. **T** se considera como un parámetro que puede ser adaptado debido a la declaración:

```
template <class T>
```

Aunque en este caso, **T** no era una clase.

Para usar funciones definidas a través de una **template** y que puedan ser empleadas en varios ficheros hay que incluir un fichero de cabecera que contenga la declaración de tal función. El fichero de cabecera podría contener unas instrucciones como:

```
template <class T>
void swap(T &x, T &y) { T w = x; x = y; y = w;}
```

En esta caso, la función **swap** no está definida, aunque parezca lo contrario. Sólo está declarada. Recordar que no se deben incluir definiciones en un fichero .h.

Las **templates** pueden tener varios argumentos, por ejemplo:

```
template <class ta, class tb>
tb potencia (ta a, int n, tb &x) {
x = 1;
for (int i=1; i<=n; i++) x*=a;
return x;
}
```

Este ejemplo muestra como las variables afectadas por el **template** también pueden ser variables de salida de la función. Algo importante que se ve en el ejemplo es que es necesario que todas las variables que son afectadas por el **template** sean utilizadas en los parámetros de llamada a la función. Por esto en el ejemplo hay un paso de parámetros redundante.

Las **templates** se pueden utilizar en las clases, cuando son más o menos idénticas y sólo difieren en el tipo o clase sobre el que están construidas. Un ejemplo típico es el de una clase de vectores de enteros y flotantes. Por ejemplo:

```
template <class T>
class vector {
```

```
T xx;
T yy;
public:
    vector(T x=0, T y=0) {xx = x; yy = y}
    void printvec()const;
    friend vector<T> operator+(vector<T> &a,vector<T>&b); };

template <class T>
void vector<T>::printvec()const {
    cout << xx << "     " << yy << endl; };

template <class T>
vector<T> operator+(vector<T> &a, vector<T> &b) {
    return vector <T>(a.xx+b.xx,a.yy+b.yy); };

void main () {
vector<int> iu(1,2), iv(3,4), isom;
vector <float> fu(1.1,2.2), fv(3.3,4.4), fsom;
isom = iu+iv;
fsom = fu+fv;
isom.printvec();
return 0;
}
```

La notación **vector<T>** indica que vector por sí solo no define la clase, sino que es necesario que se complemente con la definición de **T**. El tipo vector se encuentra parametrizado con **T**.

Es posible utilizar los **templates** con argumentos que no sean de tipo. Por ejemplo:

```
template <int n>
class vectorfloat { float vf[n]; };
```

De esta manera se puede definir:

```
vectorfloat<100> a;
vector float<200> b;
```

Los argumentos normales y de tipo se pueden alternar en el **template** de forma transparente.

11. El Puntero This

En una función miembro de una clase, que no sea estática, existe una palabra clave **this** que es el puntero al objeto por el que la función ha sido llamado. Por ejemplo:

```
class objeto {
    int x;
public:
    objeto(int b) { this->x = b } };
```

El tipo de **this** es, en este ejemplo, **objeto***. Si una función miembro se declara como **const**, por ejemplo:

```
void print() const;
```

Significa que el puntero **this** es un de tipo **objeto* const**. Por lo tanto, los objetos a los que apunta no se pueden cambiar, es decir, los datos miembros de la clase. En el anterior ejemplo:

```
fun_objeto(int b)const { x = b }  // error
```

Las funciones miembro constantes pueden ser llamadas por objetos constantes y no constantes. Las funciones miembro no-constantes sólo pueden ser llamadas por objetos no constantes. Ni los constructores ni los destructores pueden ser declarados constantes.

Ejercicio 7: Crear una clase genérica que sea una cola de cualquier tipo de objeto. Crear la función constructora y las funciones miembro de poner y quitar elementos, e imprimir el contenido de la cola. Esta última, que se una función constantes. Construir un programa que compruebe el buen funcionamiento de la cola con una variable tipo float y con una clase tipo persona como la que se ha definido en otros ejercicios.

12. Manejo de Excepciones

El manejo de excepciones es una cualidad del C++ de tratar errores y otras situaciones excepcionales que se producen en la ejecución de un programa debido a:

?? Fin de memoria
?? Error al abrir un fichero
?? División por cero
?? Y otras circunstancias.

Para el manejo de excepciones C++ emplea tres palabras reservadas: **try**, **catch** y **throw**. Ejemplo de utilización:

```
void detalle (int k) {
    cout << "Comienzo detalle" << endl;
    if (k==0) throw 123;
    cout << "Fin detalle" << endl;
return; };
void proceso (int i) {
    cout << "Comienzo proceso" << endl;
    detalle(i);
    cout << "Fin proceso" << endl;
return; };
void main() {
    int x = 0;
    try {
            proceso(x); }
    catch (int i) {
            cout << i << endl; }
    return; };
```

La ejecución de este proceso tiene la siguiente salida:

```
Comienzo proceso
Comienzo detalle
123
```

La instrucción:

```
    catch (int i) { cout << i << endl; }
```

Se conoce como el manejador de la excepción, y una sucesión de al menos un manejador se conoce como la lista de manejadores. Si no es necesario utilizar ninguna variable dentro del manejador, se puede utilizar como argumento **(int)**.

Un bloque **try** se construye como:

try {sentencia-compuesta} lista-manejadores

Cuando **throw** se ejecuta, el control se transfiere al manejador cuyo bloque **try** fue más recientemente ejecutado y no salido de él. Si hay varios manejadores para un mismo bloque **try**, la expresión de la sentencia **throw** determina cual de ellos se ejecuta ya que ha de coincidir con el tipo de expresión especificado en el manejador. El tipo puede ser especificado en una clase definida por el usuario.

La expresión **throw** únicamente puede aparecer en una parte de código llamada directa o indirectamente por un manejador. Una expresión **throw** con una expresión a la que no le corresponde ningún manejador lanza la ejecución de una función **teminate**.

Importante: los bloques **try** se pueden anidar.

12.1 Función terminate

Si se dispara un **throw** con un objeto al que no corresponde ningún manejador se dispara la función **terminate**, que generalmente aborta el programa. Para insertar la propia función de terminación se utiliza

set_terminate(nombre_función);

Esta función devuelve un puntero a la ultima función establecida como **terminate**, por lo que se puede llamar varias veces para cambiar a lo largo de la ejecución que función va a ser tratada como **terminate**. La función ha de finalizar con la instrucción **exit(1)**. Dependiendo del compilador, la no inclusión de esta instrucción puede generar un error grave en la ejecución del programa.

12.2 Especificación de Excepciones

Se puede limitar las excepciones que se pueden llamar desde una función con la declaración:

void f() throw (x,y);

Donde **x** e **y** indican tipos de excepciones. La lista de excepciones que se pueden lanzar puede ser vacía **throw()** y así no permitir lanzar ningún tipo de excepción. Si no se incluye un especificador de excepciones, todas se consideran admitidas.

Si se dispara una excepción que no se encontraba permitida, se ejecuta la función **terminate** que termina abortando el programa. Para evitarlo se puede utilizar la función **set_unexpected** que se encuentra en el fichero **except.h**, aunque en algunos compiladores no es necesario realizar ninguna inclusión de nuevos ficheros para poder emplearla. Esta función tiene un comportamiento similar a la función **set_terminate**.

12.3 Manejador de errores con New

Con el constructor se puede realizar un manejador de errores de la siguiente forma. En primer lugar se incluye una librería del manejador de interrupciones:

include <new.h>

La función que detecta y trata los errores al ejecutar un constructor se declara y se incluye la siguiente llamada:

set_new_handler(nombre_de_función)

De esta manera se puede ejecutar el manejador que se desee. En la función manejadora, es necesario incluir la sentencia **exit(1)**, ya que si no se regresa a la ejecución de la sentencia **new** y se ejecutaría el ciclo indefinidamente.

Otra forma es coger la excepción con una manejador de excepción **catch** pero que sea de tipo **xalloc** (Borland C++). La clase **xalloc** se encuentra declarada en el fichero **except.h**. El bloque tratado también hay que declararlo con **try** pero no es necesario incluir ningún **throw**, pues este es la propia sentencia **new**. La función manejadora llamada desde el manejador también es necesario terminarla con **exit(1)**, pero en este caso si no se incluye se regresa al programa principal y continua la ejecución normal del programa.

Ejercicio 8: Crear una función **tomadatos** que tome los datos *nombre, apellidos* y *edad* desde pantalla. Estos datos son de tipo: **char**, **char** y **entero**. Lanzar un error de tipo carácter si el *nombre* o los *apellidos* tienen una longitud menor a 2 caracteres. En la función de captura del error es necesario que se indique la cadena leída. Lanzar un error de tipo entero si la *edad* es menor que cero, y en la captura del error que se indique la edad errónea introducida.

Lanza un error de distinto tipo (puntero carácter) en la función **tomadatos** si la *edad* es menor de 10. No construir el manejador y programar una función **terminate** para capturar el error.

Amplia el ejercicio incluyendo una función **calculo** en la función **tomadatos** que evalúa un *descuento* en función de la *edad*. Poner la función **calculo** dentro de un bloque **try** en la misma función **tomadatos** de forma que esté anidado con el anterior. Lanza un error de tipo entero si la edad es mayor de 70 y programa el manejador del error.

Especifica la función **calculo** como limitada a errores de tipo entero. Lanza un error de tipo carácter dentro de la función si la *edad* es mayor que 60 y crea una función manejadora de errores inesperados que capture dicho error.

Reserva un vector tipo doble de 999.999.999 elementos en la función **calculo** y construye un manejador de errores de reserva de memoria.

13. Funciones Destructoras

Es la función contraria a la función constructora. Cuando se invoca, el objeto deja de existir. Para objetos que son variables locales no estáticas, el destructor se invoca cuando la función en la que se encuentra definido el objeto ejecuta la sentencia **return**. Para objetos que se definen como estáticos, el destructor es llamado cuando termina el programa.

Cuando se termina el programa con una instrucción **exit**(), todos los destructores son llamados y eliminan los objetos que se encuentren definidos.

Como ocurre con los constructores, los destructores también existen por defecto. Si se quieren declarar han de seguir las siguientes reglas:

Tienen el mismo nombre que el constructor, pero precedido de una tilde.

No tiene argumentos ni valores de retorno.

Por ejemplo:

```
class mi_objeto {
    char *nombre;
    float *numero;
public:
    mi_objeto() {                       // constructor
            nombre = new char[20];
            numero = new float;
    }
    ?mi_objeto() {              // destructor
            delete nombre;
            delete numero;
    }
};
```

14. Herencia Múltiple

Anteriormente se ha visto un ejemplo de cómo se puede derivar una clase a partir de otra clase base. Es posible realizar la derivación de una clase a partir de más de una clase base. Por ejemplo:

```
class motor {
    float potencia;
public:
    motor (float pot) {potencia = pot};
};

class rueda {
    float radio;
public:
    rueda (float rad) {radio = rad};
};

class coche: public motor, public rueda {
    int cod_modelo;
public:
```

```
    coche(float pot, float rad, int cod): motor(pot), ruedad(rad) {
            cod_modelo = cod; }
}
```

Importante destacar que en la declaración de herencia múltiple es necesario indicar que la herencia es **public** en cada uno de los casos. Si no, toma por defecto **private**.

El orden en el que se invocan los constructores viene determinado por la derivación. En este caso, en primer lugar motor, a continuación rueda y finalmente la construcción propia de la nueva clase.

Es posible asignar a una variable de tipo base una variable de tipo derivado como en el caso de la herencia sencilla:

```
rueda r;
coche c;
r = c;
```

En este caso se pierden las características adicionales del objeto tipo coche respecto del objeto tipo rueda.

De igual forma que en la herencia sencilla, es posible asignar punteros de la clase derivadas a punteros de cualquier variable base:

```
rueda *r;
motor *m;
coche c;
r = &c;
m = &c;
```

Ejercicio 9: Crear tres clases bases: *unidadcentral*, *pantalla*, *teclado*. Define como privadas las variables que identifican cada uno de estos elementos por separado. Por ejemplo, para la *unidadcentral*: char *codigo*, char *cpu*, float *velocidad*, int *memoria*, int *discoduro*. Define funciones constructoras, destructoras, de impresión y de acceso.

Crea una clase derivada de las tres anteriores llamada *ordenador*. Esta tendrá como dato miembro un *string* que define su nombre comercial y un *string* que también es su código. Crear funciones constructoras, destructoras, de impresión y de acceso a los datos miembro.

Realiza un programa principal en el que se declare un vector de objetos *unidadcentral*. Realiza un bucle en que por pantalla se piden unidades centrales u ordenadores, con sus características. Almacena ambos en el mismo vector.

15. Composición de Clases

La composición de clases aparece cuando se tiene miembros de una clase que son a su vez objetos de una clase. Tiene cierta similitud cuando en C una estructura estaba formada por distintas estructuras.

Su empleo es trivial. Por ejemplo:

```
class cumpleanios {
    int dia,mes,anio;
public:
    cumpleanios() { dia = 0; mes = 0; anio = 0; };
};

class empleado {
    int sueldo;
    cumpleanios diacum;
public:
    empleado(int,cumpleanios);
};

empleado::empleado(int elsueldo, cumpleanios cumple) {
    sueldo = 0;
  diacum = cumple;
};
```

16. Funciones como parte de una Estructura

Cuando se han definido las clases se ha visto como es posible definir como miembros de una clase tanto datos como funciones.

En C++ también se pueden incluir funciones en la definición de una estructura. Por ejemplo:

```
struct complex {
    float a;
    float b;
    void salida(void);
};

void complex::salida(void) {
    cout << "real " << a << " imag " << b;
return;
}

void main() {
struct complex x;
    x.a = 1.0;
    x.b = 2.0;
    x.salida();
return;
}
```

Esto conduce a la definición de una forma menor de clases, o una versión de estructuras un poco más compleja. El tamaño (**sizeof**()) de la estructura depende de sus datos miembros, no del conjunto de funciones que contiene.

17. Diferencias entre Public, Private y Protected

Los miembros de una clase pueden ser **Public**, **Private** y **Protected**.

?? **Private**: el nombre de un miembro privado puede se utilizado únicamente por las funciones miembro, los constructores y las amigas de la clase en la que está declarado.

?? **Protected**: el nombre de un miembro privado puede se utilizado únicamente por las funciones miembro, los constructores y las amigas de la clase en la que está declarado, además de las funciones miembro y amigas de la clase derivada.

?? **Public**: el nombre de un miembro publico puede ser utilizado por cualquier función o iniciador.

Veamos un ejemplo de utilización:

```
class uno {
private:
    float a;
protected:
    float b;
public:
    float c;
    uno() { a = 1.0; b = 2.0; c = 3.0; }
};

class dos: uno {
public:
    void acceso() {
            // a = 0.0; no es accesible
            b = 4.0;
            c = 9.0;
  return; };
};

void main() {
uno x;
dos y;
    // x.a = 3; no es accesible
    // x.b = 6; no es accesible
    x.c = 9;
    y.acceso();
return;
};
```

18. Campos de Bit

La especificación de campos de bit se realiza en C++ de la siguiente forma:

```
int a:4;
```

Esto indica que la varible **a** es un campo de 4 bits.

Los campos se han de empaquetar en alguna unidad de asignación direccionable, por ejemplo una estructura:

```
struct campo {
    int a:4;
    int b:2; };
```

Cómo se alinean los campos de bits en la palabra del ordenador depende de la implementación. Además se pueden asignar de izquierda a derecha en algunas máquinas o de derecha a izquierda en otras.

Un campo de bits sin nombre resulta útil como relleno para adaptares a esquemas impuestos externamente. Como caso especial, un campo de bits sin nombre con ancho 0 especifica la alineación del próximo campo de bits en la frontera de la unidad de almacenamiento de memoria.

Un campo de bits sin nombre no es un miembro y no se puede iniciar.

Los campos de bits han de ser de tipo entero. Depende de la implementación, si un campo de bits tipo **int** puede considerarse de tipo con signo o sin signo. Puede explicitarse **unsigned int**.

El operador **&** no se puede emplear en un campo de bits. No hay punteros ni referencias a un campo de bits.

19. Diferencias entre compilar un programa en C como C++

19.1 Función sizeof

En C las siguientes funciones tienen el mismo resultado:

```
sizeof(int) = sizeof ('c')
```

Esto es debido a que los tipos **char**, cuando se pasan como argumentos a las funciones, son pasados como enteros. Por lo tanto, el compilador de C trata a una constante 'c' como una constante entera.

En C++ no ocurre lo mismo. De esta forma:

```
cualquier_funcion(10) <----------> cualquier_funcion('c')
```

Son la misma función en C pero no en C++.

19.2 Prototipos

El C++ es más estricto en la declaración de prototipos que el C. La siguiente declaración:

```
void funcion();
```

En C significa que la función no devuelve parámetros pero que sus argumentos pueden ser cualquiera. No los especifica.

En C++ significa que la función no toma ningún argumento. En ese caso es preferible usar la fórmula:

```
void funcion(void);
```

Ejercicio: Construir una clase llamada *numero* que almacene números enteros de cualquier tamaño (en el ejemplo limitarlo a 10 dígitos, pero que se sea capaz de incrementar a infinitos dígitos). Aprovechar el almacenamiento en memoria introduciendo dos dígitos en cada byte, es decir, usando notación BCD. Crear las funciones constructores y de salida por pantalla. Estas últimas que implementen la salida decimal y la salida del contenido individual de cada byte (el resultado la composición binaria de dos dígitos BCD). Crear las funciones que soporten los operadores sobrecargados de suma y resta.

20. Soluciones a los ejercicios

20.1 Ejercicio 1:

```
#include <stdio.h>
#include <stdlib.h>

class nodo {
    nodo *inodo;
     nodo *dnodo;
     int  flag;
     union {
       double valor;
       nodo *pvalor;
     };
  public:
     nodo () {
              inodo = NULL;
              dnodo = NULL;
              flag  = 0;
              valor = 5.0;
     }
     friend void crear (nodo *el_nodo, int selec) {
              if (selec == 0) el_nodo->inodo = new nodo;
              else  el_nodo->dnodo = new nodo;
     }
     friend void cambiar_v (nodo *el_nodo, double *valor){
              el_nodo->flag = 0;
              el_nodo->valor = *valor;
     }
     friend void cambiar_n (nodo *el_nodo, nodo *nuevo) {
              el_nodo->flag = 1;
              el_nodo->pvalor = nuevo;
     }
     friend nodo* viajar (nodo *el_nodo, int valor) {
     return ((valor == 0) ? el_nodo->inodo: el_nodo->dnodo);
     }
};


void main () {
nodo   raiz, *nuevo;
double valor = 20.0;

  crear ( &raiz, 0);
  cambiar_v ( &raiz, &valor);
  nuevo = viajar ( &raiz, 0);
  cambiar_n (nuevo, &raiz);
}
```

20.2 Ejercicio 2:

```
#include <stdio.h>
#define TAMANO 40

int suma (int,int);
int resta (int,int);
int multi (int, int);

int suma(int a, int b) {
return a+b; };

int resta(int a, int b) {
return a-b; };

int multi (int a, int b) {
return a*b; };

int div (int a, int b) {
return a/b; };

class opera {
    int indx;
    int (**f) (int, int);
public:
    opera() { f = new (int (*[TAMANO])(int,int));
                                    indx = 0; };
    int pon( int(*x)(int,int)) {
            int error = 0;
            if (indx < TAMANO-1)
                    f[indx++] = x;
            else
                    error = 1;
            return error;
    }
    int saca (int a, int b) {
            return f[--indx](a,b);
  }
};

void main () {
opera memo;
int error, a = 25, b = 8, result;
  error = memo.pon(suma);
  error = memo.pon(multi);
  error = memo.pon(div);
  result = memo.saca(memo.saca(a,b),a);
  printf("%i", result);
return;
}
```

20.3 Ejercicio 3:

```
#include <stdio.h>
#define MAX 20
class string;

class vector {
    float * vec;
  int longitud;
public:
    vector();
    void annade(float);
    friend void convierte (vector, string); };
vector::vector() {
    vec = new float[MAX];
    longitud = 0;
return; };
inline void vector::annade(float valor) {
    if (longitud < MAX)
            vec[longitud++] = valor;
return; };

class string {
    char * str;
public:
    string();
    void print();
    friend void convierte (vector, string); };
string::string() {
    str = new char[MAX+1];
    str[MAX] = NULL; }

inline void string::print() {
    printf("%s\n", str);
return; };

void convierte(vector v, string s) {
int indx;
    for (indx = 0; indx < v.longitud; indx++)
            s.str[indx] = (char) v.vec[indx];
  s.str[indx] = NULL;
return; };

void main () {
vector v;
string s;
    v.annade(49.0);
    v.annade(51.0);
    convierte(v,s);
    s.print();
return; }
```

20.4 Ejercicio 4:

```
#include <math.h>
#include <stdio.h>
#include <stdarg.h>
#define POLAR 1
#define COMPLEX 2

class polar;
class complex;
polar convierte(complex);


class complex {
    float r;
    float i;
public:
    complex (float real= 0, float imag= 0){
            r = real;
            i = imag; };
    void print() {
            printf( "%f +i%f\n", r, i);
    }
    friend complex operator + (complex a,complex b) {
    complex c;
            c.r = a.r+b.r;
            c.i = a.i+b.i;
    return c;
    }
    friend complex operator * (complex a,complex b) {
    complex c;
            c.r = a.r*b.r - a.i*b.i;
            c.i = a.r*b.i + a.i*b.r;
    return c;
    }
    friend polar convierte (complex);
  friend complex convierte (polar);
  friend complex acumula (int, ...);
};

class polar {
    float r;
    float t;
public:
    polar (float real= 0, float angle= 0){
            r = real;
            t = angle;};
    void print() {
            printf( "%f*e%f\n", r, t);
    }
```

```
    friend polar operator + (polar a,polar b) {
    polar c;
  float m1,m2,n1,n2,m,n;
              m1 = a.r*cos(a.t);
              m2 = a.r*sin(a.t);
              n1 = b.r*cos(b.t);
              n2 = b.r*sin(b.t);
              m = m1+n1;
              n = n1+n2;
              c.r = sqrt(m*m+n*n);
     c.t = atan(n/m);
    return c;
    }
    friend polar operator * (polar a,polar b) {
    polar c;
              c.r = a.r*b.r;
              c.t = a.t+b.t;
    return c;
    }
    friend polar convierte (complex);
  friend complex convierte (polar);
  friend complex acumula ( int,... );
};


polar convierte(complex a) {
polar b;
    b.r = sqrt(a.r*a.r+a.i*a.i);
    b.t = atan(a.i/a.r);
return b;
}

complex convierte(polar a) {
complex b;
    b.r = a.r*cos(a.t);
    b.i = a.r*sin(a.t);
    return b;
}

complex acumula ( int *puntero_tipo_argumento,...) {
va_list ap;        // Declaro la variable que apuntara a los argumentos reales
polar pvar;
complex cvar;
complex acum;

  va_start(ap, puntero_tipo_argumento);  // Inicializa AP para almacenar el valor de
los argumentos
  while (*puntero_tipo_argumento != NULL ) {  // Recorre la lista de argumentos
      switch (*puntero_tipo_argumento++) {
      case POLAR:              // Si es entero
                        pvar = va_arg(ap, polar);      // Extrae
        acum = acum + convierte(pvar);
                        break;
```

```
      case COMPLEX:            // Si es flotante
                        cvar = va_arg(ap, complex);
        acum = acum + cvar;
        break;
      }
  }
  va_end(ap);   // Libera la lista de punteros
  return acum;
}


void main () {
polar a (5,5), x, y;
complex b (2,4), c (1,1), d, acum;
int lista[] = {COMPLEX, COMPLEX, POLAR, NULL};
     d = b+c;
     x = convierte(d);
     y = a+x;
     d.print();
   x.print();
     y.print();
   y = polar(5,0);
     acum = acumula(lista, b, c, y );
     acum.print();
return;
}
```

Ejercicio 5:

```
#include <iostream.h>
#include <iomanip.h>

class persona {
    char* nombre;
    char* apellidos;
    int       edad;
public:
    persona( char* x = NULL, char* y = NULL, int z = 0) {
              nombre = x;
              apellidos = y;
              edad = z; }
    void print() {
              cout << "Nombre: " << nombre << " Apellidos: " << apellidos << "
edad: " << setw(4) << edad;
              return;
              }
    char *get_nombre () { return nombre;};
    char *get_apellidos () { return apellidos;};
    int get_edad () { return edad; };
    friend void lee_persona (persona &p) {
              p.nombre = new char[20];
              p.apellidos = new char[20];
              cout << "Nombre: ";
     cin >> p.nombre;
              cin >> p.apellidos >> p.edad;
    }
};

class consumidor: public persona {
    float gasolina;
    float electricidad;
    float gas;
public:
    consumidor ( char* x, char* y, int z, float a = 0, float b = 0, float c = 0 ):
persona(x,y,z) {
              gasolina = a;
              electricidad = b;
              gas = c; }
    void print() {
      persona::print();

              cout << setiosflags(ios::fixed) << setiosflags(ios::showpoint) << "
Gasolina "<< setw(12) << setprecision(8) << gasolina
              << " Electricidad: " << setw(12) << setprecision(8) <<electricidad << "
Gas: " <<setw(12) << setprecision(8) << gas << endl;
              return;
              }
    friend void actualiza ( consumidor &con, float a, float b, float c ) {
```

```
                con.gasolina = a;
                con.electricidad = b;
                con.gas = c;
                return;
     }
};

class deportista: public persona {
    char* tipo_deporte;
    float marca;
public:
    deportista (char* x, char* y, int z, char *a = 0, float b = 0): persona(x,y,z) {
                tipo_deporte = a;
                marca = b; }
    void print() {
                persona::print();
                cout << " Tipo Deporte: " << tipo_deporte << " Marca: " << marca <<
endl;
                return;
                }
};


void main () {
consumidor con1 ("pedro", "bernar", 25), con2 ("rafa", "olmedo", 30);
persona per1;

    actualiza(con1, 12.5, 1.3333, 444.0);
  actualiza(con2, 1.1, 3.3, 0.0005);
    con1.print();
    con2.print();
    cout << con1.get_edad() << endl;
  lee_persona(per1);
  per1.print();

return;
}
```

20.5 Ejercicio 6

```
#include <iostream.h>

class componente {
    int precio;
    int num_serie;
  char* proveedor;
public:
    componente () {};
    componente ( int a, int b, char* c ) {
            precio = a;
            num_serie = b;
            proveedor = c;
            return;
  }
    virtual void print() {      // Prueba ejecutar este programa sin la palabra virtual
            cout << proveedor << endl;
            return;
    }
};

class rodar: public componente {
    float diametro;
public:
    rodar(int a, int b, char* c, float d):componente(a,b,c) {
            diametro = d;
  }
    void print() {
            cout << "Diametro " << diametro << endl;
    }
};

class engrana: public componente {
    int estrias;
public:
    engrana(int a, int b, char* c, int d):componente(a,b,c) {
            estrias = d;
  }
    void print() {
            cout << "Estrias " << estrias;
    }
};

class lista {
    componente *elemento;
  lista* siguiente;
public:
    lista (){
    elemento = new componente;
            siguiente = NULL;};
    friend lista* anade (lista& a, componente* const b) {
```

```
            a.siguiente= new lista;
            a.elemento = b;
            a.siguiente->siguiente = NULL;
            return a.siguiente;
    };
    friend void print (lista* a) {
            if (a != NULL)
                    if (a->siguiente != NULL) {
                    a->elemento->print();
                            print(a->siguiente);
                    }
  }
};

void main() {
lista ini,*fin;
rodar c (1,1,"Pedro", 7.0);
engrana d (1,1, "Luis", 14);
    fin = anade(ini, &c);
    fin = anade(*fin, &d);
    print (&ini);
return;
}
```

20.6 Ejercicio 7

```
#include <iostream.h>
#include <iomanip.h>

template <class T, int n>
class cola {
    T vector[n];
    int ini;
  int fin;
public:
    cola();
    char poner (T&);
    char quitar (T&);
    void print()const;
};

template<class T,int n>
cola<T,n>:: cola() {
    ini = 0;
    fin = 0;
};

template<class T, int n>
char cola<T,n>:: poner (T& x) {
char error = 'n';
    if ((ini < n-1) && (ini+1 != fin)) ini++;
    else if ((ini == n-1) && (fin != 0)) ini = 0;
    else error = 'y';
    if (error == 'n') vector[ini] = x;
return error;
}

template<class T, int n>
char cola<T,n>:: quitar (T& x) {
char error = 'n';
    if (fin == ini) error = 'y';
  else if (fin < n-1) fin++;
    else if (fin == n-1) fin = 0;
    if (error == 'n') x = vector[fin];
return error;
}

template <class T, int n>
void cola<T,n>::print()const {
int contador;
    for(contador=0; contador < n; contador ++)
            cout << vector[contador] << endl;
return;
}
```

```
void main () {
cola<float,10> la_cola;
float f1;
char test;

     test = la_cola.poner(5.0);
     cout << test << endl;
   test = la_cola.poner(6.0);
     cout << test << endl;
   la_cola.print();
     test = la_cola.quitar(f1);
     cout << test << " " << f1 << endl;
     test = la_cola.quitar(f1);
     cout << test << " " << f1 << endl;
     test = la_cola.quitar(f1);
     cout << test << " " << f1 << endl;

return;
}
```

20.7 Ejercicio 8

```
#include <iostream.h>
#include <iomanip.h>
#include <new.h>

void tomadatos(float &descuento);
float calcula (char*,int) throw (int) ;
void mi_terminate(void);
void mi_terminate1(void);
void mi_inesperada(void);
void mi_new(void);

float calcula(char* a, int b) throw (int) {
double* memo;
  if (b > 70 ) throw "Excepcion inesperada"; // Produce un error al no cogerla con
unexpected
  if (b > 60 ) throw 0;
  memo = new double[999999999];
return b/3.0;
}

void mi_terminate(void) {
   cout << "Se ha lanzado una excepcion no controlada en try_0" << endl;
   exit(1);
}

void mi_terminate1(void) {
   cout << "Se ha lanzado una excepcion no controlada en try_1" << endl;
   exit(1);
}

void mi_inesperada(void) {
   cout << "Se ha lanzado una excepcion inesperada" << endl;
   exit(1);
}

void mi_new(void) {
   cout << "Se ha producido un error al solicitar memoria" << endl;
   exit(1);
}

void tomadatos(float &descuento) {
char nombre[10], apellido[20], ciudad[20];
int edad;
   cout << "Escribe el nombre: ";
   cin >> nombre;
   if (strlen(nombre) <2) throw nombre[0];  // comprueba que mandar el puntero da
error
   cout << endl << "Escribe el apellido: ";
```

```
   cin >> apellido;
   if (strlen(apellido) < 2) throw apellido[0];
   cout << endl << "Escribe la edad: ";
   cin >> edad;
   if (edad <0) throw 0;
   if (edad >100) throw 1;
   if (edad <10) throw "incontrolada";
   // TRY_1 ************************
   try {
      set_terminate(mi_terminate1);
      descuento = calcula (ciudad,edad);
      set_terminate(mi_terminate);
   }  catch (int) {
      cout << "Error de calculo de descuento" << endl;
      throw "incontrolada";
   };
return;
};

int main () {
float descuento;
   set_terminate(mi_terminate);
   set_unexpected(mi_inesperada);
   set_new_handler(mi_new);
   // TRY_0 ************************
   try {
      tomadatos(descuento);
   } catch(int i) {
     switch (i) {
        case 0: cout << "Edad demasiado baja" << endl;
                break;
        case 1: cout << "Edad demasiado alta" << endl;
                break;
     };
    }
    catch(char texto) {
        cout << "El texto: " << texto << " es demasiado corto" << endl;
    };

   cout << "El descuento: " << descuento << endl;
return 0;
};
```

20.8 Ejercicio 9

```
#include <iostream.h>
#include <iomanip.h>
#include <new.h>

void tomadatos(float &descuento);
float calcula (char*,int) throw (int) ;
void mi_terminate(void);
void mi_terminate1(void);
void mi_inesperada(void);
void mi_new(void);

float calcula(char* a, int b) throw (int) {
double* memo;
  if (b > 70 ) throw "Excepcion inesperada"; // Produce un error al no cogerla con
unexpected
  if (b > 60 ) throw 0;
  memo = new double[999999999];
return b/3.0;
}

void mi_terminate(void) {
   cout << "Se ha lanzado una excepcion no controlada en try_0" << endl;
   exit(1);
}

void mi_terminate1(void) {
   cout << "Se ha lanzado una excepcion no controlada en try_1" << endl;
   exit(1);
}

void mi_inesperada(void) {
   cout << "Se ha lanzado una excepcion inesperada" << endl;
   exit(1);
}

void mi_new(void) {
   cout << "Se ha producido un error al solicitar memoria" << endl;
   exit(1);
}

void tomadatos(float &descuento) {
char nombre[10], apellido[20], ciudad[20];
int edad;
   cout << "Escribe el nombre: ";
   cin >> nombre;
   if (strlen(nombre) <2) throw nombre[0];  // comprueba que mandar el puntero da
error
   cout << endl << "Escribe el apellido: ";
   cin >> apellido;
```

```
   if (strlen(apellido) < 2) throw apellido[0];
   cout << endl << "Escribe la edad: ";
   cin >> edad;
   if (edad <0) throw 0;
   if (edad >100) throw 1;
   if (edad <10) throw "incontrolada";
   // TRY_1 ***************************
   try {
      set_terminate(mi_terminate1);
      descuento = calcula (ciudad,edad);
      set_terminate(mi_terminate);
   }  catch (int) {
      cout << "Error de calculo de descuento" << endl;
      throw "incontrolada";
   };
return;
};

int main () {
float descuento;
   set_terminate(mi_terminate);
   set_unexpected(mi_inesperada);
   set_new_handler(mi_new);
   // TRY_0 ***************************
   try {
      tomadatos(descuento);
   } catch(int i) {
     switch (i) {
        case 0: cout << "Edad demasiado baja" << endl;
                break;
        case 1: cout << "Edad demasiado alta" << endl;
                break;
     };
    }
    catch(char texto) {
        cout << "El texto: " << texto << " es demasiado corto" << endl;
    };

   cout << "El descuento: " << descuento << endl;
return 0;
};
```

20.9 Ejercicio 10

```
#include <iostream.h>
#include <string.h>


//----------------------------------------
// Clase Unidad Central
//----------------------------------------

class unidadcentral {
     float speed;
     float memo;
     char* cpu;
public:
     unidadcentral (float, float, char*);
     ~unidadcentral();
     void salida();
   void pide();
};

unidadcentral::unidadcentral(float mspeed = 0.0, float mmemo = 0.0, char* mcpu =
NULL) {
int lencpu;
     speed = mspeed;
     memo = mmemo;
     lencpu = strlen(mcpu); // Con copia me aseguro que el puntero char solo
pertenece a un objeto
   cpu = new char[lencpu];
     strcpy(cpu,mcpu);
};

unidadcentral::~unidadcentral() {
     delete cpu;
};

void unidadcentral::salida() {
     cout << cpu << " " << memo << " Mbytes " << speed << " MHz" << endl;
return;
};

void unidadcentral::pide() {
   cpu = new char [40];
     cout << "Tipo de cpu: ";
     cin  >> cpu;
     cout << "Memoria: ";
     cin  >> memo;
     cout << "Velocidad: ";
     cin  >> speed;
   cout << endl;
return;
```

```
};


//-----------------------------------------
// Clase Teclado
//-----------------------------------------
class teclado {
    char *tipo;
    int numteclas;
public:
    teclado ( char*, int);
    ~teclado ();
    void salida();
  void pide();
};

teclado::teclado (char* mtipo = NULL, int mnumteclas = 0) {
int lentipo;
    numteclas = mnumteclas;
    lentipo = strlen(mtipo);
    tipo = new char[lentipo];
    strcpy(tipo,mtipo);
};

teclado::~teclado() {
    delete tipo;
};

void teclado::salida() {
    cout << tipo << " " << numteclas << " teclas" << endl;
return;
};

void teclado::pide() {
    tipo = new char [40];
    cout << "Tipo: ";
    cin >> tipo;
    cout << "Numero de teclas: ";
    cin >> numteclas;
    cout << endl;
return;
};

//-----------------------------------------------
// Clase ordenador
//-----------------------------------------------
class ordenador:public unidadcentral, public teclado {
    char *nombre;
public:
    ordenador(float speed = 0.0, float memo = 0.0, char* cpu = NULL, char* tipo =
NULL, int num =0, char* mnombre = NULL);
    ~ordenador();
    void salida();
```

```
   void pide();
};

ordenador::ordenador(float speed, float memo, char* cpu, char* tipo, int num, char*
mnombre):
                        unidadcentral(speed,memo,cpu),teclado(tipo,num) {
int lennombre;
    lennombre = strlen(mnombre);
    nombre = new char[lennombre];
    strcpy(nombre,mnombre);
};

ordenador::~ordenador(){
    delete nombre;
};

void ordenador::salida() {
    cout << nombre << endl;
    unidadcentral::salida();
    teclado::salida();
return;
};

void ordenador::pide() {
    cout << "Nombre del ordenador: ";
    cin >> nombre;
    unidadcentral::pide();
    teclado::pide();
return;
};

//-------------------------------------------------------
// Funcion Principal
//-------------------------------------------------------

void main () {
unidadcentral* b;
ordenador a;
    a.pide();
    a.salida(); // Salida de la caracteristicas del ordenador
    b = &a;
    b->salida(); // Salida de las caracteristicas de la unidad central
return;
};
```

20.10 Ejercicio 11

```
#include <iostream.h>

union palabra {
    char a;
    struct {
            unsigned int low:4;
            unsigned int high:4;
    } dig;
};

class numero {
    palabra num [5];
public:
  numero(){};
    numero(int* a) {
    int indx;
            for (indx=0;indx<5; indx++) {
                    num[indx].dig.low = a[indx*2];
                    num[indx].dig.high = a[indx*2+1];
            }
    };
    void salida() {
    int indx;
            for (indx = 4; indx >= 0; indx--)
                    cout << (int) num[indx].dig.high << (int) num[indx].dig.low;
            cout << endl;
    return;
    }
    void hex() {
    int indx;
            for (indx = 4; indx >= 0; indx--)
                    cout << (unsigned int) num[indx].a;
            cout << endl;
    return;
   }

    friend numero operator +(numero,numero);
};

numero operator + (numero a, numero b) {
numero c;
int indx;
int x = 0;
    for (indx=0; indx < 5; indx++) {
            x = (int) a.num[indx].dig.low + (int) b.num[indx].dig.low + x;
            if (x > 9) {
       c.num[indx].dig.low = x-10;
                    x = 1; }
            else {
```

```
                        c.num[indx].dig.low = x;
        x = 0; };
                x = (int) a.num[indx].dig.high + (int) b.num[indx].dig.high + x;
                if (x > 9) {
        c.num[indx].dig.high = x-10;
                        x = 1;   }
                else {
                        c.num[indx].dig.high = x;
                        x = 0; };
    };
return c;
};


void main() {
int x[10] =  {1,2,3,4,5,6,7,8,9,0};
int y[10] =  {2,2,2,2,2,2,2,2,2,2};
numero a ( x );
numero b;
numero c;
  a.hex();
    a.salida();
  b.salida();
    c = a+b;
  c.salida();
return;
};
```