Manual básico de Programación en C++

Apoyo a la Investigación C.P.D.
Servicios Informáticos U.C.M.

INDICE

1. INTRODUCCION

C++ es una mejoría sobre muchas de las características de C, y proporciona capacidades de P.O.O. que promete mucho para incrementar la productividad, calidad y reutilización del software.

En **C**, la unidad de programación es la **función**, con lo cual, se trata de una programación orientada a la acción.

En **C++**, la unidad de programación es la **clase** a partir de la cual, los objetos son producidos. Se trata, pues, de una programación orientada al objeto.

Las bibliotecas estándar de C++ proporcionan un conjunto extenso de capacidades de entrada/salida. C++ usa entradas/salidas de tipo seguro; no podrán introducirse datos equivocados dentro del sistema.

Se pueden especificar

equivo9is05.1ndar de C+. Est 173.25 0 /F1 12 5f 0.07 Tc 10 Tw (C++) To de cibiyzaci a mejor ca

A continuación se muestra dos tablas de palabras reservadas en C++ :

C y C++				
auto	break	case	char	const
continue	default	do	double	else
enum	extern	float	for	goto
if	int	long	register	return
short	signed	sizeof	static	struct
switch	typedef	union	unsigned	void
volatile	while			

C++ únicamente	
asm	Medio definido por la puesta en práctica de lenguaje de ensamble a lo largo de C++. (Vea los manuales correspondientes a su sistema).
catch	Maneja una excepción generada por un **throw.**
class	Define una nueva clase. Pueden crearse objetos de esta clase.
delete	Destruye un objeto de memoria creado con **new.**
friend	Declara una función o una clase que sea un "amigo" de otra clase. Los amigos pueden tener acceso a todos los miembros de datos y a todas las funciones miembro de una clase.
inline	Avisa al compilador que una función particular deberá ser generada en línea, en vez de requerir de una llamada de función.
new	Asigna dinámicamente un objeto de memoria en la memoria adicional disponible para el programa en tiempo de ejecución. Determina automáticamente el tamaño del objeto.
operator	Declara un operador "homónimo".
private	Un miembro de clase accesible a funciones miembro y a funciones **friend** de la clase de miembros **private.**
protected	Una forma extendida de acceso **private**; también se puede tener acceso a los miembros **protected** por funciones miembro de clases derivadas y amigos de clases derivadas.
public	Un miembro de clase accesible a cualquier función.
template	Declara cómo construir una clase o una función, usando una variedad de tipos.
this	Un apuntador declarado en forma implícita en toda función de miembro no **static** de una clase. Señala al objeto al cual esta función miembro ha sido invocada.
throw	Transfiere control a un manejador de excepción o termina la ejecución del programa si no puede ser localizado un manejador apropiado.
try	Crea un bloque que contiene un conjunto de números que pudieran generar excepciones, y habilita el manejo de excepciones para cualquier excepción generada.
virtual	Declara una función virtual.

2. C++ COMO UN ''C MEJORADO''

2.1. COMENTARIOS Y DECLARACIONES EN C++

C++ permite empezar un **comentario** con // y usar el resto de la línea para texto del comentario; el fin de la línea da de manera automática por terminado el comentario. También es aceptable la forma de C : **/* */**.

En C++, a diferencia de C, las **declaraciones** pueden ser colocadas en cualquier parte de un enunciado ejecutable, siempre y cuando las declaraciones antecedan el uso de lo que se está declarando. También, las variables pueden ser declaradas en la sección de inicialización de una estructura **for**; dichas variables se mantienen en alcance hasta el final del bloque en el cual la estructura **for** está definida.

El alcance de una variable local C++ empieza en su declaración y se extiende hasta la llave derecha de cierre. Las declaraciones de variables no pueden ser colocadas en la condición de una estructura **while**, **do/while**, **for** o **if**.

2.2. FLUJO DE ENTRADA/SALIDA DE C++

C++ ofrece una alternativa a las llamadas de función *printf* y *scanf* para manejar la entrada/salida de los tipos y cadenas de datos estándar. Así, en lugar de *printf* usamos el *flujo estándar de salida* **cout** y el operador << ("colocar en"); y en lugar de *scanf* usamos el *flujo de entrada estándar* **cin** y el operador >> ("obtener de"). Estos operadores de inserción y extracción de flujo, a diferencia de *printf* y *scanf*, no requieren de cadenas de formato y de especificadores de conversión para indicar los tipos de datos que son extraídos o introducidos.

Para utilizar entradas/salidas de flujo, se debe incluir el archivo de cabecera **iostream.h**.

2.3. COMO CREAR NUEVOS TIPOS DE DATOS EN C++

C++ proporciona la capacidad de crear tipos definidos por el usuario mediante el uso de las palabras reservadas **enum**, **struct**, **union** y la nueva palabra reservada **class**. A diferencia de C, una enumeración (*enum*) en C++, cuando se declara, se convierte en un tipo nuevo. Para declarar la variable del nuevo tipo, la palabra reservada *enum* no es requerida. Lo mismo para s*truct*, *union* y *class*. Los nombres de etiqueta pueden ser utilizados para declarar variables. Las enumeraciones por omisión son evaluadas iniciándose en cero.

2.4. PROTOTIPOS DE FUNCION Y VERIFICACION DE TIPO

C++ requiere que se declaren todos los parámetros de función en los paréntesis de la definición de función y del prototipo. Una lista vacía de parámetros se especifica escribiendo **void** o absolutamente nada en los paréntesis.

2.5. FUNCIONES EN LINEA

Existen *funciones en línea* que ayudan a reducir la sobrecarga por llamadas de función especial para pequeñas funciones. El compilador puede ignorar el calificador **inline** y típicamente así lo hará para todo, a excepción de las funciones más pequeñas. El calificador *inline* deberá ser utilizado sólo tratándose de funciones pequeñas, de uso frecuente. Usa funciones *inline* puede reducir el tiempo de ejecución, pero puede aumentar el tamaño del programa.

Ejemplo:

```
// Usamos la función inline para calcular el volumen de un cubo.

#include <iostream.h>

inline   float   cubo( const float s )  { return s * s * s; }

main( )
{
     cout  <<  "Introduce la longitud del lado de tu cubo:  ";
     float   lado;
     cin  >>  lado;
     cout  <<  "El volumen del cubo de lado "
     <<  lado  <<  "es"  <<  cubo( lado )  <<  '\n';

     return 0;
}
```

2.6. PARAMETROS POR REFERENCIA

C++ ofrece *parámetros por referencia*. Un parámetro de referencia es un seudónimo ("alias") de su argumento correspondiente. Para indicar que un parámetro de función es pasado por referencia, sólo hay que colocar un ampersand (&) después del tipo del parámetro en el prototipo de función.

Debemos usar apuntadores para pasar argumentos que pudieran ser modificados por la función llamada, y usar referencias a constantes para pasar argumentos extensos, que no serán modificados.

Las variables de referencia deben ser inicializadas en sus declaraciones, y no pueden ser reasignadas como seudónimos a otras variables.

Cuando se regresa un apuntador o una referencia a una variable declarada en la función llamada, la variable deberá ser declarada **static** dentro de dicha función.

Las referencias pueden ser usadas como argumentos de funciones y regresar valores.

Ejemplo:

```
int   ix;              /* ix es la variable "real" */
int   &rx = ix;        /* rx es el "alias" de ix */
ix = 1;                /* también rx == 1 */
rx = 2;                /* también ix == 2 */
```

2.7. EL CALIFICADOR CONST

El calificador *const* se usa en la lista de parámetros de una función para especificar que un argumento pasado a la función no es modificable en dicha función. También puede ser utilizado para declarar las llamadas *"variables constantes"*. Estas pueden ser colocadas en cualquier parte en que se espere una expresión constante. También en archivos de cabecera.

La variable constante debe ser inicializada al declararse.

Otro uso común para el calificador *const* es para poder declarar un apuntador constante.

Tabla con expresiones de declaración de constantes:

DECLARACION	NOMBRE ES...
const *tipo nombre* = *valor*;	Tipo constante.
tipo *const *nombre* = *valor*;	Apuntador constante a tipo.
const *tipo* **nombre* = *valor*;	(Variable) apuntador a la constante tipo.
const *tipo* *const *nombre* = *valor*;	Apuntador constante a tipo constante.

2.8. ASIGNACION DINAMICA DE MEMORIA MEDIANTE NEW Y DELETE

En C++, el enunciado

ptr = new typeName;

asigna memoria para un objeto del tipo *typeName.* El operador **new** crea automáticamente un objeto del tamaño apropiado, y regresa un apuntador (ptr)del tipo apropiado. Si mediante **new** no se puede asignar memoria, se regresa un apuntador nulo.

Para liberar el espacio para este objeto se usa **delete ptr**;

El operador **delete** sólo puede ser utilizado para desasignar memoria ya asignada mediante el operando **new**.

2.9. OPERADOR DE RESOLUCION DE ALCANCE UNARIO

Es posible declarar variables locales y globales con un mismo nombre. C++ dispone del *operador de resolución de alcance unario* (**::**) para tener acceso a una variable global cuando está en alcance una variable local con el mismo nombre. No puede ser utilizado para tener acceso a una variable del mismo nombre en un bloque externo.

Ejemplo:

```
#include <iostream.h>
float v;
int main( )
{
   int v = 7;
   ::v = 10.5;    // Utilizar la variable global v
   cout << "variable local    v = " << v << "\n";
   cout << "variable global  v = " << ::v << "\n";
   return 0;
}
```

2.10. HOMONIMIA DE FUNCIONES

C++ permite que sean definidas varias funciones del mismo nombre (sobrecarga de funciones), siempre que estos nombres indiquen distintos conjuntos de parámetros. Esta capacidad se llama *homonimia de funciones*. La homonimia de la función se utiliza por lo común para crear varias funciones del mismo nombre, que ejecuten tareas similares, sobre tipos de datos distintos.

Las funciones homónimas deben tener distintas listas de parámetros.

La homonimia ayuda a eliminar el uso de las macros #define de C.

Ejemplo:

```
// Definimos dos funciones distintas max( ), una que regrese el mayor de
// dos enteros y otra que regrese el mayor de dos strings.

#include <stdio.h>
int  max( int a,  int b )
{
   if ( a > b ) return a;
   return b;
}
char  *max( char *a, char *b )
{
   if ( strcmp (a,b) > 0 )  return a;
   return b;
}
int  main( )
{
   printf ( "max( 19, 69 ) = %d\n", max( 19, 69 ) );
   printf ( "max( abc,def ) = %s\n", max( "abc","def" ) );
   return 0;
}
```

El programa del ejemplo define dos funciones que difieren en su lista de parámetros, de ahí que defina dos funciones distintas.

Las referencias se pueden usar para proveer una función con un alias de un argumento real de llamada de función. Esto permite cambiar el valor del argumento de llamada de función tal como se conoce de otros lenguajes de programación de llamada-por-referencia:

```
void  mira( int  porValor,  int  &porReferencia )
{
    porValor = 42;
    porReferencia = 42;
}
void  mar( )
{
   int  ix, jx;
   ix = jx = 1;
   mira( ix, jx );
   // ix == 1,  jx == 42
}
```

2.11. PLANTILLAS DE FUNCION

Las plantillas de función permiten la creación de funciones que ejecuten las mismas operaciones sobre distintos tipos de datos, pero la plantilla de función se define sólo una vez.

Las plantillas de función proporcionan, como las macros, una solución compacta, pero permiten verificación completa de tipo.

Todas las definiciones de plantillas de función empiezan con la palabra reservada **template** (ver apdo.4.8, pág. 31), seguida por una lista de parámetros formales a la plantilla de función encerrados en corchetes angulares (< >). Cada parámetro formal es precedido por la palabra reservada **class**. A continuación se coloca la definición de función y se define como cualquier otra función.

3. PRIMERAS EXTENSIONES ORIENTADAS A OBJETOS

3.1. COMO PONER EN PRACTICA UN TDA CON UNA CLASE

Las clases permiten que el programador modele objetos que tienen *atributos (miembros de datos)* y *comportamientos* u *operaciones* (*funciones miembro*). Los tipos contienen miembros de datos y funciones miembro, y son definidos mediante la palabra reservada **class.**

El nombre de la clase (una vez definida) puede ser utilizado para declarar objetos de dicha clase.

Ejemplo:

```
class   Punto        // Declaración de la clase Punto.
{
   int   _x,  _y;     // Coordenadas del punto.
public:               // Principio de la declaración de interface.
   void    setX  (const  int  val);
   void    setY  (const  int  val);
   int     getX( )    {return   _x;}
   int     getY( )    {return   _y;}
};
Punto    apunto;    // Definición del objeto apunto.
```

La definición de una clase comienza con la palabra reservada **class** . El *cuerpo* de la definición de clase se delimita mediante llaves. La definición de clase termina con un punto y coma. En el cuerpo de la definición existen partes nuevas: la etiqueta **public:** y **private:** se conocen como especificadores de acceso de miembro. Cualquier miembro de datos o función miembro declarado después del especificador de acceso de miembro **public:** (y antes del siguiente especificador de acceso de miembro) es accesible, siempre que el programa tenga acceso a un objeto de la clase. Cualquier miembro de datos o función miembro declarada después del especificador de acceso de miembro **private:** (y hasta el siguiente especificador de acceso de miembro) sólo es accesible a las funciones miembro de la clase.

Estos especificadores pueden aparecer varias veces en una definición de clase. Por defecto, los elementos de las clases son privados (**private:**).

Por lo regular, los miembros de datos aparecen listados en la porción **private:** de una clase, y normalmente, las funciones de miembro aparecen listadas en la porción **public:**, aunque puede darse el caso contrario.

Los miembros de datos de una clase no pueden ser inicializados donde son declarados en el cuerpo de la clase. Deberán ser inicializados por el constructor de la clase, o las funciones "set" les podrán asignar valores.

La función con el mismo nombre que la clase, pero precedido por un carácter tilde (~) se llama el *destructor* de dicha clase.

Cuando una función miembro se define por fuera de la definición de clase, el nombre de función es antecedido por el nombre de la clase y por el operador de resolución de alcance binario (**::**). Dicha función miembro queda dentro del *alcance de la clase*.

Ejemplo:

```
void   Punto :: setX  (const   int   val)   // Definimos el método setX
                                            // ámbito de la clase Punto.
{
    _x  =  val;
 }
void   Punto :: setY  (const   int   val)
{
    _y  =  val;
}
```

El objeto *apunto* puede usar estos métodos para establecer y para obtener información sobre sí mismo:

```
            Punto   apunto;
            apunto.setX(1);         // Inicialización.
            apunto.setY(1);
//
 // x es necesaria a partir de aquí, de modo que la definimos aquí y la
// inicializamos con el valor de la coordenada _x de apunto.
//
            int   x  =  apunto.getX( );
```

3.2. ALCANCE DE CLASE Y ACCESO A MIEMBROS DE CLASE

Los nombres de variables y los de función declarados en una definición de clase, y los nombres de datos y funciones miembro de una clase, pertenecen al *alcance de dicha clase*.

Dentro del alcance de clase, los miembros de clase son accesibles por todas las funciones miembro de dicha clase y pueden ser referenciados sólo por su nombre. Fuera del alcance de una clase, los miembros de clase se referencian, ya sea a través del nombre del objeto, una referencia a un objeto, o un apuntador a un objeto.

3.3. COMO CONTROLAR EL ACCESO A MIEMBROS

Se puede tener acceso a los miembros de clase privado sólo por miembros (y amigos) de dicha clase. Se puede tener acceso a los miembros públicos de una clase mediante cualquier función del programa.

El cliente de una clase puede ser una función miembro de otra clase, o puede ser una función global.

Los miembros de clase públicos presentan una vista de los servicios que proporciona la clase.

Para miembros de una clase, el acceso por omisión es privado.

El acceso a los datos privados de una clase puede ser controlado con cuidado mediante el uso de las funciones miembro conocidas como *funciones de acceso*. Si una clase desea permitir que los clientes lean datos privados, la clase puede proporcionar una función "*get*". Para permitir que los clientes modifiquen datos privados, la clase puede proveer una función "*set"*.

Ejemplo:

```
#include <iostream>
// Definición de la clase Cfecha
class Cfecha
{
 public:
    // Constructor ( véase apdo. siguiente)
    Cfecha( int dd = 1, int mm = 1, int aa = 1980 );
    // Funciones de acceso
    void get_fecha( int &, int &, int & );
    void set_fecha( int, int, int );
 private:
    int dia, mes, anyo;  // Datos miembro
};

// Constructor
Cfecha::Cfecha( int dd, int mm, int aa )
{
   set_fecha( dd, mm, aa );
 }
 // Obtener una fecha
 void Cfecha::get_fecha( int &dd, int &mm, int &aa )
 {
    dd = dia, mm = mes, aa = anyo;
  }
```

3.4. COMO INICIALIZAR OBJETOS DE CLASE : CONSTRUCTORES

Los constructores son métodos que se usan para inicializar un objeto al momento de su definición o para asignar almacenamiento en memoria. Cuando se declara un objeto de una clase, se pueden dar *inicializadores* en paréntesis a la derecha del nombre del objeto y antes del punto y coma. Estos inicializadores son pasados como argumentos al constructor de la clase.

Los constructores pueden tener argumentos por omisión.

Si para una clase no se define ningún constructor, el compilador creará un constructor por omisión. Dicho constructor no ejecutará ninguna inicialización.

Debemos incluir siempre un constructor que ejecute la inicialización adecuada para su clase.

Ejemplo:

```
class   Punto
{
        int   _x, _y;
public :
        Punto( )
        {
        _x  =  _y  =  0;
         }
         void    SetX( const   int   val );
         void    SetY( const   int   val );
         int    getX   { return   _x; }
         int    getY   { return   _y; }
};
```

Los constructores tienen el mismo nombre de la clase. No regresan ningún valor. Pueden llevar argumentos. Así:

```
class    Punto
{
      int    _x, _y;
public:
      Punto()
      {
      _x  =  _y  =  0;
      }
```

```
        // Continuamos con el ejemplo:
        Punto (const int x, const int y)
        {
            _x = x;          // Para poder inicializar un punto en
            _y = y;          // otras coordenadas que no sean (0,0).
        }
        void setX (const int val);
        void setY (const int val);
        int getX() { return _x;}
        int getY() { return _y;}
};
```

Los constructores son llamados implícitamente cuando definimos objetos de sus clases:

```
Punto apunto;              // Punto :: Punto()
Punto bpunto(12,34);       // Punto :: Punto(const int, const int)
```

3.5. DESTRUCTORES

Un destructor de clase es llamado automáticamente cuando un objeto de una clase se sale de alcance: debemos asegurarnos de que la memoria asignada sea liberada.

Este método especial llamado *destructor* es llamado una vez por cada objeto en el momento de su destrucción.

Un destructor no recibe parámetros ni regresa ningún valor. Una clase sólo puede tener un destructor: la homonimia de destructores no está permitida.

La destrucción de los objetos tiene lugar cuando el objeto abandona su ámbito de definición o es explícitamente destruido. Esto último sucede cuando nosotros dinámicamente asignamos un objeto y lo liberamos cuando ya no nos es necesario.

Los destructores se declaran con el nombre de la clase prefijado con una tilde (~):

```
class Punto
{
    int _x, _y;
public:
    Punto()
    {
        _x = _y = 0;
    }
```

```
        // Continuamos con el ejemplo:
        Punto (const   int   x, const   int   y)
        {
            _x  =  xval;
            _y  =  yval;
        }
        Punto (const   Punto   &desde)
        {
            _x  =  desde._x;
            _y  =  desde._y;
        }
        ~ Punto()    { /* ¡Nada que hacer! */}
        void   setX(const   int   val);
        void   setY(const   int   val);
        int   getX()    {return   _x;}
        int   getY()    {return   _y;}
};
```

3.6. CUANDO SON LLAMADOS LOS DESTRUCTORES Y CONSTRUCTORES

Por lo regular, son llamados de forma automática. En general, las llamadas de destructor se efectúan en orden inverso a las llamadas de constructor.

3.7. ASIGNACIÓN POR OMISION EN COPIA A NIVEL DE MIEMBRO

El operador de asignación (=) es utilizado para asignar un objeto a otro objeto del mismo tipo.

4. CLASES

4.1. OBJETOS CONSTANTES Y FUNCIONES MIEMBRO CONST

La palabra reservada **const** indica que un objeto no es modificable. Para objetos *const* no se permiten llamadas de función miembro. Se pueden declarar funciones miembro *const* : sólo éstas pueden operar sobre objetos *const*; estas funciones no pueden modificar el objeto.

Una función es definida como *const* tanto en su declaración como en su definición, insertando la palabra reservada *const* después de la lista de parámetros de la función, y, en el caso de la definición de función, antes de la llave izquierda que inicia el cuerpo de la función.

Ejemplo:

```
const Cfecha cumpleanyos;
cumpleanyos.ok_fecha( );

class Cfecha
{
public:
   // Funciones miembro de la clase
   //...
   int ok_fecha( ) const;
   //...
};
int Cfecha::ok_fecha( ) const
{
   // Cuerpo de la función
}
```

Si una función miembro *const* se define fuera de la definición de la clase, tanto la declaración como la definición de la función miembro deberán incluir *const*.

Una función miembro *const* puede ser homónima en una versión no *const*.

Para constructores y destructores de objetos *const* no se requiere la declaración *const*. A un constructor debe permitírsele poder modificar un objeto de forma tal que el objeto pueda ser inicializado de forma correcta. Un destructor debe ser capaz de ejecutar sus trabajos de terminación, antes de que un objeto sea destruido.

Un objeto *const* debe ser inicializado. Las asignaciones no son permitidas.

4.2. COMPOSICION: CLASES COMO MIEMBROS DE OTRAS CLASES

Una clase puede tener otras clases como miembros.

Cuando un objeto entra en alcance, su constructor es llamado automáticamente, por lo que es preciso especificar cómo se pasan argumentos a constructores de objetos miembro. Se construyen los objetos miembro antes de que los objetos de clase que los incluyen sean construidos.

4.3. FUNCIONES AMIGO Y CLASES AMIGO

Podemos definir que funciones o clases sean *amigos* de una clase para permitirles acceso directo a sus miembros de datos privados. Se puede declarar una función o toda una clase como un *friend* de otra clase.

Para declarar una función como un *friend* de una clase, en la definición de clase hay que preceder el prototipo de función con la palabra reservada **friend**.

Las declaraciones de amistad pueden ser colocadas en cualquier parte de la definición de clase.

Es posible especificar funciones homónimas como amigos de una clase. Cada función homónima que se desea como un amigo, debe ser declarada en forma explícita en la definición de clase como amigo de la clase.

Ejemplo:

```
class   Complejo
{
  ...
public:
  ...
  friend   Complejo   operator +
     (
        const   Complejo  &,
        const   Complejo  &
     );
 }
 Complejo   operator +(const   Complejo   &op1, const   Complejo   &op2)
 {
       double   real  =  op1._real  +  op2._real,
                imag  =  op1._imag  +  op2._imag;

       return   ( Complejo (real, imag));
  }
```

No se deben usar amigos muy seguido debido a que rompen con el principio de aislamiento de datos en sus fundamentos. Si los usamos de forma muy seguida, es señal de que tenemos que modificar nuestra gráfica de herencia.

4.4. COMO UTILIZAR UN APUNTADOR THIS

Cada objeto mantiene un apuntador a sí mismo llamado *apuntador* **this** que es un argumento implícito en todas las referencias a miembros incluidos dentro de dicho objeto. El apuntador *this* puede también ser utilizado de forma explícita. Mediante el uso de la palabra reservada *this*, cada objeto puede determinar su propia dirección.

El apuntador *this* es utilizado de manera implícita para referenciar tanto los miembros de datos como las funciones miembro de un objeto. El tipo de este apuntador *this* depende del tipo del objeto y de si es declarada *const* la función miembro en la cual *this* es utilizado.

Un uso del apuntador *this* es impedir que un objeto sea asignado a sí mismo.

Ejemplo:

```
// Declaramos un objeto fecha1 y a continuación le enviamos el
// mensaje set_fecha.

fecha1.set_fecha( );

// C++ define el puntero this para apuntar al objeto fecha1 de la
// forma:

Cfecha *const this = &fecha1;
```

```
// La función set_fecha puede ser definida también de la siguiente
// forma:

void Cfecha::set_fecha( )
{
   cout << "día,   ##    : "; cin >> this->dia;
   cout << "mes, ##     : "; cin >> this->mes;
   cout << "año, ####  : "; cin >> this->anyo;
 }
```

4.5. ASIGNACION DINAMICA DE MEMORIA MEDIANTE NEW Y DELETE

Los operadores **new** y **delete** ofrecen una mejor forma de efectuar la asignación dinámica de memoria, que mediante las llamadas de función **malloc** y **free** de C.

El operador *new* crea en forma automática un objeto del tamaño apropiado, llama el constructor para el objeto (si hay uno disponible) y regresa un apuntador del tipo correcto. Si *new* no puede encontrar espacio, regresa un apuntador **0**.

```
char *ptr;
ptr = new char[longitud];
```

Mediante el operador **delete** liberamos espacio para ese objeto:

```
delete ptr;
```

C++ permite incluir un *inicializador* para un objeto recién creado.

New invoca al constructor de manera automática, y *delete* automáticamente invoca al destructor de la clase.

4.6. MIEMBROS DE CLASE ESTATICOS

Un miembro de datos estático representa información "aplicable a toda la clase". La declaración de un miembro estático empieza con la palabra reservada **static**.

Estos miembros de datos estáticos tienen alcance de clase. Pueden ser públicos, privados o protegidos.

Una función miembro puede ser declarada *static* si no tiene acceso a miembros de clase no estáticos. A diferencia de las funciones miembro no estáticas, una función miembro estática no tiene apuntador *this*.

Los miembros estáticos de una clase son accesibles a través de un objeto de dicha clase, o pueden ser accesibles a través del nombre de la clase mediante el uso del operador de resolución de alcance.

4.7. CLASES CONTENEDOR E ITERADORES

Las *clases contenedor* (o *clases colección*) están diseñadas para contener colecciones de objetos, como puedan ser arreglos y listas enlazadas.

Asociamos *iteradores* con la clase de colección. Un iterador es un objeto que regresa el siguiente elemento de una colección. Una clase contenedor puede tener varios iteradores operando sobre ella simultáneamente.

El recorrido de una estructura de datos es implementado usando *iteradores*. Estos garantizan la visita a cada ítem de su estructura de datos asociada en un orden bien definido. Deben proveer al menos las siguientes propiedades:

1. *Elemento actual*. El iterador visita los elementos de datos uno a la vez. El elemento que se visita actualmente es llamado el *"elemento actual"*.
2. *Función sucesor*. La ejecución del paso al siguiente elemento de datos depende de la estrategia de recorrido implementada por el iterador. La función sucesor se usa para regresar el elemento que será visitado enseguida: regresa el sucesor del elemento actual.
3. *Condición de terminación*. El iterador debe proveer un mecanismo que chequee si se han visitado todos los elementos, o si falta alguno por visitar.

4.8. CLASES PLANTILLA

En C++, los tipos de datos genéricos son llamados *plantillas de clase* o simplemente *plantillas (templates)*. Una plantilla de clase se parece a la definición de una clase normal, en la que algunos aspectos son representados por *sustitutos (placeholders)*.

Las definiciones de clases plantilla empiezan con la línea:

template <class T>

en la línea que antecede a la definición de clase. Puede existir más de un tipo parametrizado. Si es así, estarán separados por comas y cada tipo estará precedido por la palabra reservada **class**. (Ver apdo. 2.11., pág. 20).

Luego, con la palabra clave *template* se inician todas las declaraciones de plantillas. Los argumentos de una plantilla se encierran en corchetes angulares.

Cada argumento especifica un sustituto en la siguiente definición de clase.

Ejemplo:

```
template <class T>     // T es el sustituto
Class    Lista: ...
{
public:
      ...
      void   apéndice( const   T   dato);
      ...
 };
```

Una clase plantilla se produce especificando el tipo de la clase que sigue al nombre de la clase:

NombreClase<tipo> NombreObjeto;

Ejemplo :

```
Lista<int> listaDeEnteros
```

4.9. CLASES ABSTRACTAS

Existen muchas situaciones en las cuales resulta útil definir clases para las cuales el programador no tiene intención de producir ningún objeto. Estas clases se conocen como *clases abstractas*. A partir de una clase base abstracta no se pueden producir objetos.

El único fin de una clase abstracta es proporcionar una clase base apropiada, a partir de la cual las clases pueden heredar interfaz y/o puesta en práctica. Las clases a partir de las cuales los objetos se pueden producir, se conocen como *clases concretas*.

Se definen igual que las clases ordinarias. Sin embargo, algunos de sus métodos están designados para ser definidos necesariamente por sus subclases. Sólo mencionamos su *signature* (nombre del método más sus argumentos) incluyendo el tipo que regresa, pero no una definición. Se podría decir que omitimos el cuerpo del método. Esto se expresa añadiendo "=0" después de las "signatures" de los métodos:

```
class   ObjetoDesplegable
{
    ...
public:
    ...
    virtual void print( ) = 0;
};
```

Estas declaraciones de métodos son también llamadas *métodos puros*. También deben ser declarados virtuales, debido a que sólo queremos usar objetos de clases derivadas. Las clases que definen métodos puros son llamadas *clases abstractas*.

5. HOMONIMIA DE OPERADORES

5.1. FUNDAMENTOS DE LA HOMONIMIA DE OPERADORES

C++ permite *sobrecargar* casi todos sus operadores por tipos recién creados.

Se hace la homonimia de operadores escribiendo una definición de función (con encabezado y cuerpo) como se haría normalmente, excepto que ahora el nombre de la función se convierte en la palabra clave **operator**, seguida por el símbolo correspondiente al operador homónimo:

```
<tipo devuelto> operator <operador>(<arg1>,<arg2>)
{
        <cuerpo de la función>
 }
```

Para utilizar un operador sobre objetos de clase, dicho operador *deberá* ser un homónimo con dos excepciones. El operador de asignación (=) puede ser utilizado sin homónimo con cualquier clase. El operador de dirección (**&**) también puede ser utilizado sin homonimia con objetos de cualquier clase.

La homonimia de operadores no es automática; para llevar a cabo las operaciones deseadas, el programador deberá escribir funciones de homonimia de operadores.

5.2. RESTRICCIONES SOBRE LA HOMONIMIA DE OPERADORES

Se puede hacer la homonimia de la mayor parte de los operadores de C++:

Operadores que pueden tener homónimos							
+	-	*	/	%	^	&	|
~	!	=	<	>	+=	-=	*=
/=	%=	^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&	||	++
--	->*	'	->	[ ]	( )	new	delete

Operadores que no pueden tener homónimos				
.	**.***	**::**	**?:**	sizeof

La precedencia y la asociatividad de un operador no puede ser modificada por la homonimia. Para hacer la homonimia de un operador no pueden utilizarse argumentos por omisión. Además, la sobrecarga sólo es válida dentro del contexto de la clase en la que ésta ocurre.

No es posible modificar el número de operandos que toma un operador. Tampoco se pueden crear operadores nuevos; sólo se puede hacer la homonimia de operadores existentes.

La homonimia de operador funciona sólo con objetos de tipos definidos por el usuario o por una combinación de un objeto de tipo definido por el usuario y un objeto de tipo incorporado.

5.3. HOMONIMIA DE OPERADORES DE INSERCION Y DE EXTRACCION DE FLUJO

Se hace la homonimia de estos operadores (<<, >>) para procesar cada tipo de datos estándar, incluyendo cadenas y direcciones de memoria; también para ejecutar entradas y salidas para tipos definidos por usuario.

Los operadores de entrada y de salida homónimos deben ser declarados como amigo, si han de tener acceso directo a los miembros de clase no públicos.

5.4. HOMONIMIA DE OPERADORES UNARIOS

Se puede hacer la homonimia de un operador unario para una clase como función miembro no estática sin argumentos, o como función no miembro con un argumento; dicho argumento debe ser un objeto de la clase o una referencia a un objeto de la clase.

5.5. HOMONIMIA DE OPERADORES BINARIOS

Se puede hacer la homonimia de un operador binario como función miembro no estática con un argumento o como función no miembro con dos argumentos (uno de dichos argumentos debiendo ser un objeto de clase o una referencia a un objeto de clase).

Así, por ejemplo, podríamos definir un operador "+" de la siguiente forma:

```
Class   Complejo
{
    ...
public :
    ...
    Complejo  operator  + ( const  Complejo  &op)
    {
        double  real  =  _real  +  op._real,
                imag  =  _imag  +  op._imag;
        return   (Complejo ( real, imag));
    }
    ...
};
```

En este caso, se ha hecho del + un miembro de la clase *Complejo.* Así, una expresión de la forma:

```
c = a + b;
```

es traducida a una llamada a método :

```
c = a.operator +(b);
```

Así, el operador binario + sólo necesita un argumento. El primer argumento es provisto implícitamente por el objeto invocante (en este caso *a*).

5.6. ESTUDIO DE CASO : UNA CLASE ARRAY

La clase permite que, mediante el operador de asignación, un objeto de arreglo sea asignado a otro. Al pasar el arreglo a una función, su tamaño no necesita ser pasado como argumento. Mediante los operadores de extracción y de inserción de flujo, respectivamente, es posible introducir o extraer arreglos completos. Se pueden comparar arreglos utilizando los operadores de igualdad == y !=.

En una declaración se llama al constructor de copia cuando un objeto de la clase *Array* es producido e inicializado con otro objeto de la clase *Array.*

Para evitar que un objeto de clase sea asignado a otro, debemos definir el operador de asignación como miembro privado de la clase.

5.7. CONVERSION ENTRE TIPOS

El compilador no sabe de forma automática cómo convertir entre tipos definidos por usuario y tipos incorporados. El programador deberá especificar en forma explícita cómo deberán ocurrir dichas conversiones. Estas conversiones pueden ser ejecutadas mediante *constructores de conversión*: constructores de un solo argumento, que sólo conviertan objetos de otros tipos en objetos de una clase particular.

Un *operador de conversión* (*operador de conversión explícita cast*) puede ser utilizado para convertir un objeto de una clase en un objeto de otra clase o en un objeto de un tipo incorporado. Este operador no puede ser una función amigo; debe ser una función miembro no estática.

En C++, una construcción **cast** puede expresarse de la forma siguiente:

nombre-de-tipo(expresión)

Ejemplos:

```
sqrt (double(n+2))
```

```
int *p = ( int * ) 0x1F5;
```

Notación funcional:

```
typedef int * pint;
int *p = pint( 0x1F5 );
```

6. HERENCIA

La herencia es una forma de reutilización del software, en la cual se crean clases nuevas a partir de clases existentes, mediante la absorción de sus atributos y comportamientos, y embelleciendo éstos con las capacidades que las clases nuevas requieren.

Al crear una clase nueva, en vez de escribir en su totalidad miembros de datos y funciones miembro nuevos, el programador puede determinar que la clase nueva debe *heredar* los miembros de datos y las funciones miembro provenientes de una *clase base* ya definida. La clase nueva se conoce como *clase derivada*.

La herencia forma estructuras jerárquicas de apariencia arborescente. Una clase base existe en una relación jerárquica con sus clases derivadas.

Mediante la *herencia única,* una clase es derivada de una única clase base. Con la *herencia múltiple* una clase derivada hereda de múltiples clases base.

En C++, "hereda de" se reemplaza por dos puntos (**:**). Luego, la sintaxis para escribir una clase derivada es:

class tipo_clase_derivada: (public/private/protected) tipo_clase_base { };

Ejemplo:

```
class    Puntos3D : public   Punto
{
      int   _z;
public:
      Puntos3D ( )
      {
            setX(0);
            setY(0);
            _z  =  0;
      }
     Puntos3D (const   int   x, const   int   y, const   int   z)
     {
           setX(x);
           setY(y);
           _z  =  z;
     }
     ~ Puntos3D( )     { /* Nada que hacer */}
     int   getZ( )   { return   _z;}
     void   setZ ( const   int   val)  { _z  =  val;}
};
```

6.1. TIPOS DE HERENCIA

C++ distingue dos tipos de herencia : **pública** y **privada**. Por defecto, las clases se derivan unas de otras en forma privada. Si queremos herencia pública, debemos decírselo explícitamente al compilador.

El tipo de herencia influye sobre los privilegios de acceso a elementos de las diversas superclases.

Los miembros públicos de una clase base son accesibles a todas las funciones en el programa. Los miembros privados de clase base son accesibles sólo para las funciones miembro y los amigos de la clase base.

El tipo **protected** se usa para elementos que deberían ser usados directamente en las subclases, pero que no deberían estar accesibles desde fuera. Luego, el acceso protegido sirve como nivel intermedio de protección entre el acceso público y el privado. Los miembros protegidos de clase base son accesibles sólo por miembros y amigos de la clase base, y por miembros y amigos de las clases derivadas.

Los miembros de clases derivadas pueden hacer referencia a miembros públicos y protegidos de la clase base sólo utilizando los nombres de los miembros.

6.2. CLASES BASE PUBLICAS, PROTEGIDAS Y PRIVADAS

Al derivar una clase a partir de una clase base, la clase base puede ser heredada como *public*, *protected* o *private.*

Al derivar una clase a partir de una clase base pública, los miembros públicos de la base se convierten en miembros públicos de la clase derivada, y los miembros protegidos de la clase base se convierten en miembros protegidos de la clase derivada. Los miembros privados de una clase base nunca son accesibles en forma directa desde una clase derivada.

Al derivar una clase a partir de una clase base protegida, los miembros públicos y protegidos de la clase base se convierten en miembros protegidos de la clase derivada.

Cuando se deriva una clase a partir de una clase base privada, los miembros públicos y protegidos de la clase base se convierten en miembros privados de la clase derivada.

ESPECIFICADOR DE ACCESO	ACCESIBLE DESDE SU PROPIA CLASE	ACCESIBLE DESDE CLASE DERIVADA	ACCESIBLE DESDE OBJETOS FUERA DE LA CLASE
public	si	si	si
protected	si	si	no
private	si	no	no

6.3. CONSTRUCCION Y DESTRUCCION

En general, anterior a la ejecución del cuerpo particular del constructor, los **constructores** de cada superclase son llamados para inicializar su parte del objeto creado. Así, al constructor de clase derivada siempre llamará primero al constructor de su clase base, a fin de crear y de inicializar los miembros de la clase base de la clase derivada.

Podemos especificar los constructores deseados después de un signo de dos puntos (**:**).

Ejemplo:

```
class    Puntos3D : public   Punto
{
   ...
public:
   Puntos3D ( )   {...}
   Puntos3D (
      const   int   x,
      const   int   y,
      const   int   z) : Punto (x,y)
      {
          _z  =  z;
      }
    ...
};
```

Si hay más superclases, proveemos sus llamadas a constructor como una lista separada por comas. Este mecanismo se usa también para crear objetos contenidos.

La inicialización dinámica puede ser usada con tipos de datos integrados. Por ejemplo:

```
Punto ( ):   _x(0),   _y(0)   { }
Punto ( const   int   x, const   int   y) : _x(x), _y(y)   { }
```

Los **destructores** serán llamados en orden inverso a las llamadas de constructor, por lo que un destructor de clase derivada será llamado antes del destructor de su clase base (o superclase). (Ver apdo. 3.5., pág. 25).

6.4. HERENCIA MULTIPLE :

La herencia múltiple significa que una clase derivada (subclase) hereda los miembros de varias superclases.

Sintaxis:

class tipo_base_derivada: (public/private/protected)tipo_clase_base1,
(public/private/protected)tipo_clase_base2 { };

7. POLIMORFISMO

Es la capacidad de objetos de clases distintas, relacionados mediante la herencia, a responder de forma diferente a una misma llamada de función miembro. En C++, el polimorfismo se realiza por uno de los métodos siguientes: sobrecarga de operadores o funciones virtuales.

Podemos declarar que los métodos de las clases sean **virtual** con el fin de forzar que su evaluación se base en el contenido de los objetos más que en su tipo. Con el uso de esta palabra clave, una función puede estar definida en una clase base y en su clase derivada bajo el mismo nombre. La función no deberá ser declarada virtual más que en la clase base:

```
class  ObjetoDesplegable
{
public:
      virtual  void  print( );
};
```

La clase ObjetoDesplegable define un método print( ), el cual es virtual. De esta clase podemos derivar otras clases:

```
class  Punto: public  ObjetoDesplegable
{
   ...
public:
   ...
   void  print( )    { ... }
};
```

Nuevamente, print( ) es un método virtual debido a que hereda esta propiedad de ObjetoDesplegable. La función *display( ),* que es capaz de dibujar cualquier tipo de objeto desplegable, puede por tanto ser definida como:

```
void  display  ( const  ObjetoDesplegable  &obj)
{
   obj.print ( );
};
```

Cuando se usan métodos virtuales, el destructor de la clase correspondiente debe ser declarado también virtual cuando se usan apuntadores a subclases (virtuales) cuando llega el momento de destruirlas. Debido a que el apuntador está declarado como superclase, su destructor normalmente sería llamado.

8. PLANTILLAS PARA LOS NODOS

El **nodo** es el bloque básico de construcción de una lista. Un nodo no tiene nada más que un apuntador a otro nodo.

Declaración de la clase Nodo:

```
class    Nodo
{
      Nodo   *_derecha;  // Asumimos que este nodo vecino siempre
                         // está del lado derecho.
public :
      Nodo ( Nodo   *derecha  =  NULL) :  _derecha ( derecha)     { }
      Nodo ( const   Nodo   &val) :  _derecha ( val._derecha)     { }

// Const justo antes del cuerpo del método: declara constante al método
// en lo que respecta a los elementos del objeto invocante. Sólo se puede
// usar este mecanismo en declaraciones de métodos o definiciones.

      const   Nodo   *derecha ( )   const    { return   _derecha;}
      Nodo   *&derecha( )    {return   _derecha;}

      Nodo   &operator  =  ( const   Nodo   &val)
      {
            _derecha  =  val._derecha;
            return   *this;
       }
       const   int   operator  ==  ( const   Nodo   &val)   const     {
            return   _derecha  ==  val._derecha;
       }
       const   int   operator  !=  ( const   Nodo   &val)   const     {
            return   ! ( *this  ==  val);
       }
 };
```

Cuando usamos un *const* justo antes del cuerpo del método (como ocurre en *derecha*()), *const* declara al método constante en lo que respecta a los elementos del objeto invocante. Sólo está permitido usar este mecanismo en declaraciones de métodos o definiciones, respectivamente.

Este tipo de modificador *const* también se usa para checar sobrecarga. Así:

```
class    Arboles
{
     ...
     int   arboles( )   const;
     int   arboles( );
};
```

En el ejemplo anterior, el modificador *const* declara dos métodos distintos. El primero se usa en contextos constantes y el segundo en contextos variables.

This apunta al objeto invocador.

Las aplicaciones reales requieren que los nodos lleven datos. Esto significa especializar los nodos. Los datos pueden ser de cualquier tipo; usamos la construcción de plantilla:

```
template   <class  T>
class    DatoNodo  :  public   Nodo

// La plantilla DatoNodo especializa la clase Nodo para que transporte da-
// tos de cualquier tipo. Añade funcionalidad para accesar su elemento de
// datos y también ofrece el mismo conjunto de funcionalidad estándar:
// Copy Constructor, operator = ( ) y operator = = ( ).

{
     T _data;
public :
     DatoNodo ( const   T   dato,   DatoNodo   *derecha   =   NULL) :
              Nodo ( derecha ),   _dato ( dato)     { }
     DatoNodo ( const   DatoNodo   &val) :
              Nodo ( val),   _dato ( val._dato)     { }

     const   DatoNodo   *derecha ( )   const     {
              return   ((DatoNodo   *)   Nodo   ::   derecha( ) );
     }
     DatoNodo   *&derecha( )   {return   ((DatoNodo *&) Nodo :: derecha( ) );}

     const   T   &dato( )   const     { return   _dato ;}
     T   &dato ( )     { return   _dato ;}

     DatoNodo   &operator   =   ( const   DatoNodo   &val)     {
              Nodo   ::   operator   =   (val);
              _dato   =   val._dato;
              return   *this;
      }
```

```
// Continuamos :

    const int operator == ( const DatoNodo &val) const {
        return (
            Nodo :: operator == (val) &&
            _data == val._dato ) ;
    }
    const int operator != ( const DatoNodo &val) const {
        return !( *this == val);
    }
};
```

9. COMPILAMOS UN PROGRAMA

Un proyecto grande debe ser repartido en secciones manejables, llamadas frecuentemente *módulos*. Dichos módulos se implementan en archivos separados.

A grandes rasgos, los módulos consisten en dos tipos de archivos :

- Descripciones de interface, y
- archivos de implementación.

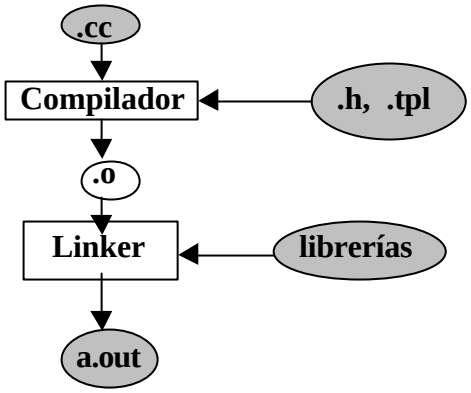
Para distinguir estos tipos, se usa un conjunto de sufijos cuando se compilan programas de C y C++ :

Extensión	Tipo de archivo
.h, .hxx, .hpp	Descripciones de interface. ("cabecera" o "archivos incluidos")
.c	Implementación de archivos de C.
.cc, .C, .cxx, .cpp, .c++	Implementación de archivos de C++.
.tpl	Descripción de interface. (templates) (definición de plantillas).

9.1. PASOS DE LA COMPILACION

El proceso de compilación toma los archivos **.cc**, los preprocesa y los traduce en *archivos objeto.*

A continuación, el conjunto de archivos objeto es procesado por un *linker*. Este programa combina los archivos, añade las bibliotecas necesarias y crea un ejecutable.



En los ordenadores Alpha OSF/1 de la U.C.M., para compilar y linkar (enlazar) un programa basta con hacer:

cxx nombre_del_programa.cxx

A continuación se explican las opciones de compilación más comunes.

Para obtener más información, teclear el comando:

man cxx

10. FLUJO DE ENTRADA/SALIDA DE C++

Los usuarios pueden especificar entradas/salidas de tipos definidos por usuario, así como de tipos estándar.

10.1. ARCHIVOS DE CABECERA DE BIBLIOTECA IOSTREAM

La mayor parte de los programas de C++ deben incluir el archivo de cabecera **iostream.h**, que contiene información básica requerida para todas las operaciones de flujo de entrada/salida. Este archivo contiene los objetos **cin, cout, cerr** y **clog**, los cuales corresponden al flujo de entrada estándar, flujo de salida estándar, flujo de error estándar y versión con memoria intermedia de *cerr*, respectivamente.

Además del operador <<, la clase **ostream** proporciona las funciones **put** para visualizar caracteres, y **write** para visualizar cadenas:

```
cout.put('w');          // Visualiza el carácter w, equivale a cout<<'w';
cout.write(a,n);  // Escribe n bytes consecutivos comenzando en la dirección a.
```

Otros encabezados básicos se muestran en la siguiente tabla:

ENCABEZADO	DESCRIPCION
iomanip.h	Contiene información útil para llevar a cabo entradas/salidas con formato, mediante *manipuladores de flujo parametrizados.*
fstream.h	Contiene información importante para llevar a cabo las operaciones de proceso de archivo controlados por el usuario.
strstream.h	Contiene información de importancia para llevar a cabo *formato en memoria*.
stdiostream.h	Contiene información de importancia para aquellos programas que combinen los estilos de C y C++ para entradas/salidas.

10.2. FUNCIONES MIEMBRO GET Y GETLINE

La función miembro **get** sin ningún argumento, introduce un carácter del flujo designado y devuelve dicho carácter como el valor de la llamada de función.

La función miembro **get** con un argumento de carácter, introduce el carácter siguiente del flujo de entrada.

La función miembro **getline** lee caracteres a partir del flujo de entrada.

10.3. ENTRADAS/SALIDAS SIN FORMATO UTILIZANDO READ, GCOUNT Y WRITE

Se lleva a cabo *entradas/salidas sin formato* mediante las funciones miembro **read** y **write**. Cada una de estas funciones introduce o extrae, respectivamente, un número designado de caracteres de o desde un arreglo de caracteres existente en memoria.

La función **gcount** determina el número de caracteres introducido.

10.4. MANIPULADORES DE FLUJO

C++ proporciona varios *manipuladores de flujo* que cambian el formato por defecto y el estado de la anchura Para utilizar manipuladores con parámetros, se debe incluir el archivo "**iomanip.h**":

MANIPULADOR	DESCRIPCION
hex	Cambia la base con la cual se interpretan los enteros dentro de un flujo a hexadecimal.
oct	Define a base octal.
dec	Devuelve la base del flujo a decimal.
setbase	Toma un argumento entero **10, 8** o **16** para definir la base. Necesita el archivo de cabecera **iomanip.h**.
setprecision	(Sin argumento). Devuelve el ajuste actual de precisión de los números en punto flotante. (Lo mismo la función miembro **precision**).
width	Define el ancho de campo y devuelve el ancho anterior. Sin argumento, devuelve el ajuste presente.
setw	Define el ancho de campo.

Los usuarios pueden crear sus propios manipuladores de flujo.

11. EJERCICIOS

1. Escriba un programa en C++ que convierta la longitud definida por el usuario en metros y centímetros.

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

void main ( void )
{
   float   fMedidaUsuario,
           fConvMetros,
           fConvCentim;

   cout  <<  " Longitud a convertir:  ";
   cin    >>  fMedidaUsuario;

   while ( fMedidaUsuario  >  0 )  {

      fConvMetros  =  fMedidaUsuario  *  12  *  2.54;
      fConvCentim  =  fConvMetros/100;
      cout  <<  "Medida inglesa:      "
            <<  setw             ( 9           )
            <<  setprecision   ( 2           )
            <<  setiosflag      ( ios::fixed )
            <<  fMedidaUsuario
            <<  "\n";
      cout  <<  "Medida en metros:      "
            <<  setw             ( 9           )
            <<  setprecision   ( 2           )
            <<  fConvMetros
            <<  "\n";
      cout  <<  "\nIntroduzca otro valor a \n"              ;
      cout  <<"convertir ( 0 termina el programa) :  "   ;
      cin    >>  fMedidaUsuario                              ;
   }
   cout  <<  " Terminación del programa "                 ;
}
```

2. Escriba un programa que obtenga los valores trigonométricos de un ángulo dado.

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

const   double   grad_a_rad  =  0.0174532925;  // Grado = 0.0174532925 radianes


class   grado   {
   double   valor_dato;

public:
   void   pone_valor        ( double ) ;
   double   calc_sen         ( void    ) ;
   double   calc_cos         ( void    ) ;
   double   calc_tang        ( void    ) ;
   double   calc_sec         ( void    ) ;
```

```
  double  calc_cosec      ( void    ) ;
  double  calc_cotang     ( void    ) ;
} grad

void  grado::pone_valor  ( double  ang )
  {
    pone_valor = ang  ;
  }

double  grado::calc_sen( void )
  {
    double  respuesta  ;

    respuesta = sin( grad_a_rad * valor_dato )  ;
    return( respuesta )  ;
  }

double  grado::calc_cos( void )
  {
    double  respuesta  ;

    respuesta = cos( grad_a_rad * valor_dato )  ;
    return( respuesta )  ;
  }

double  grado::calc_tang( void )
  {
    double  respuesta  ;

    respuesta = tan( grad_a_rad * valor_dato )  ;
    return( respuesta )  ;
  }

double  grado::calc_sec( void )
  {
    double  respuesta  ;
    respuesta = 1.0 / sin( grad_a_rad * valor_dato )  ;
    return( respuesta )  ;
  }


double  grado::calc_cosec( void )
  {
    double  respuesta  ;

    respuesta = 1.0 / cos( grad_a_rad * valor_dato )  ;
    return( respuesta )  ;
  }

double  grado::calc_cotang( void )
  {
    double  respuesta  ;

    respuesta = 1.0 / tan( grad_a_rad * valor_dato )  ;
    return( respuesta )  ;
  }

void  main( void )
  {
    // Pone ángulo a 25.0 grados
```

```
grad.pone_valor( 25.0 ) ;
cout << "El seno del ángulo es :" << grad.calc_sen( ) << endl;
cout << "El coseno del ángulo es :" << grad.calc_cos( ) << endl;
cout << "La tangente del ángulo es :" << grad.calc_tang( ) << endl;
cout << "La secante del ángulo es :" << grad.calc_sec( ) << endl;
cout << "La cosecante del ángulo es :" << grad.calc_cosec( ) << endl;
cout << "La cotangente del ángulo es :" << grad.calc_cotang( ) << endl;
}
```

3. **Escriba un programa que calcule el sueldo de un empleado.**

```
# include <iostream.h>

class empleado {
   struct nombre_emp {
      char nombre[20] ;
      char primer_apel[20] ;
      char segundo_apel[20] ;
   } nom ;

   struct horas_emp {
      double horas ;
      double salario_base ;
      double salario_extra ;
   } hor ;




public:
   void entrada_emp( void ) ;
   void salida_emp( void ) ;
} ;

void empleado::entrada_emp( void ) {
   char cr ;

   cout << "Nombre del empleado: " ;
   cin >> nom.nombre ;
   cin.get( cr ) ; //Pasar el cambio de línea

   cout << "Primer apellido del empleado: " ;
   cin >> nom.primer_apel ;
   cin.get( cr ) ;

   cout << "Segundo apellido del empleado: " ;
   cin >> nom.segundo_apel ;
   cin.get( cr ) ;

   cout << "Total de horas trabajadas: " ;
   cin >> hor.horas ;

   cout << "Sueldo base por hora: " ;
   cin >> hor.salario_base ;

   cout << "Pago por horas extra: " ;
   cin >> hor.salario_extra ;
```

```
  cout << "\n\n" ;
}

void empleado::salida_emp( void ) {

  cout << nom.nombre << " "
       << nom.primer_apel << " "
       << nom.segundo_apel << endl ;

  if ( hor.horas <= 40 )
    cout << "Pago básico: $"
         <<hor.horas * hor.salario_base
         << endl ;
    else {
      cout << "Pago básico: $"
           << 40 * hor.salario_base
           endl ;




      cout << "Pago extra: $"
           << (hor.horas - 40) * hor.salario_extra
           << endl ;
      }
  }

  void main( void ) {

    empleado emp_corte_inglés; // Crea un objeto emp_corte_ingles de la clase
                               // empleado.

    emp_corte_ingles.entrada_emp( ) ;
    emp_corte_ingles.salida_emp( ) ;
}
```

4. **Escriba un programa que convierta pesetas a monedas de duro, cinco_duros y veinte_duros.**

```
# include <iostream.h>

const int VEINTEDUROS = 100 ;
const int CINCODUROS = 25 ;
const int DURO = 5 ;

class monedas {
  int numero ;

public:
  monedas( ) {cout << "¡Comienza la conversión! \n"; } // constructor
  ~monedas( ) {cout << "\n¡Termina la conversión!";} // destructor

void entra_pesetas( int ) ;
int convertir_veinteduros( void ) ;
int convertir_cincoduros( int ) ;
int convertir_duros( int ) ;
} ;

void monedas::entra_pesetas( int pts )
  {
```

```
        numero = pts ;
        cout << numero
             << " pts, se convierten a :"
             << endl ;
    }



int monedas::convertir_veinteduros( void )
    {
      cout << numero / VEINTEDUROS << " de a veinte duros " ;
      return( numero % VEINTEDUROS ) ;
}

int monedas::convertir_cincoduros( int cincoduros )
    {
      cout << cincoduros / CINCODUROS << " de a cinco duros " ;
      return( cincoduros % CINCODUROS ) ;
}

int monedas::convertir_duros( int duro )
    {
      cout << duro / DURO << " de a duro y " ;
      return( duro % DURO ) ;
}

void main( void )
{
    int monedas, de_cinco, duros, pts ;
    cout << "Introduzca, en monedas de una peseta, la cantidad a convertir: " ;
    cin >> monedas ;

// Creamos un objeto de la clase monedas y asociamos la cantidad
// en efectivo.
monedas efectivo ;

    efectivo.entra_pesetas( monedas ) ;
    de_cinco = efectivo.convertir_veinteduros( ) ;
    duros = efectivo.convertir_cincoduros( de_cinco ) ;
    pts = efectivo.convertir_duros( duros ) ;
    cout << pts << " peseta(s). " ;
}
```

5. **Escriba un programa que devuelva el valor absoluto de un entero o real double. (Sobrecarga de funciones).**

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

class absoluto {
public:
    int ab( int val1 ) ;
    double ab( double val2 ) ;
} ;
```

```
int  absoluto::ab( int  val1 )
   {
     int  temp  ;
     temp  =  abs( val1 )  ;
     return( temp )  ;
}

double  absoluto::ab( double  val2 )
   {
     double  temp  ;
     temp  =  abs( val2 )  ;
     return( temp )  ;
}

main( )
   {
     absoluto  numero  ;
     cout  <<  " El valor absoluto de -123 es  "  <<  numero.ab(-123)  ;
     cout  <<  " el valor absoluto de -123.11221 es  "  <<  numero.ab(-123.11221)  ;
     return (0)  ;
}
```

6. Escriba un programa que sobrecargue el operador '+' para poder sumar directamente varios ángulos, en el formato de grados minutos y segundos.

```
# include  <strstream.h>
# include  <string.h>        // para el prototipo de la función strtok ( divide la cadena
                             // en tokens, cada uno delimitado por un carácter del 2º ar-
                             // gumento. Obtendrá un token en cada llamada ).
# include  <stdlib.h>        // para el prototipo de la función atoi ( convierte una cade-
                             // na de caracteres en su número entero ).

class  valor_angulo  {
   int  grados,  minutos,  segundos  ;

public:
   valor_angulo( )  {  grados  =  0,
                       minutos  =  0,
                       segundos  =  0;  }      // constructor
   valor_angulo( char* )  ;
   valor_angulo  operator+( valor_angulo );
   char  *salida_info( void );
}

valor_angulo::valor_angulo( char  *suma_angulo )
   {
     grados  =  atoi( strtok( suma_angulo,  "ó" )  );


     minutos  =  atoi( strtok( 0,  "´" )  );
     segundos  =  atoi( strtok( 0,  "\"" )  );
   }

valor_angulo  valor_angulo::operator+( valor_angulo  suma_angulo )
   {
     valor_angulo  ang;

     ang.segundos  =  ( segundos  +  suma_angulo.segundos )  %  60;
```

```
    ang.minutos = ( ( segundos + suma_angulo.segundos ) / 60 +
                      minutos + suma_angulo.minutos ) % 60;

    ang.grados = ( ( segundos + suma_angulo.segundos ) / 60 +
                     minutos + suma_angulo.minutos ) / 60;

    ang.grados += grados + suma_angulo.grados;
    return ang;
}

char *valor_angulo::salida_info( void )
  {
    char *ang[15];

    // se requiere strstrea.h para el formato

    ostrstream( *ang, sizeof(ang) ) << grados << "ó"
                                    << minutos << "´"
                                    << segundos <<"\" "
                                    << endl;
    return *ang;
  }

void main( void )
  {
    valor_angulo angulo1( "37ó 15´ 56\" " );
    valor_angulo angulo2( "10ó 44´ 44\" " );
    valor_angulo angulo3( "75ó 17´ 59\" " );
    valor_angulo angulo4( "130ó 32´ 54\" " );
    valor_angulo suma_de_angulos;
    suma_de_angulos = angulo1 + angulo2 + angulo3 + angulo4;
      cout << "La suma de los ángulos es "
           << suma_de_angulos.salida_info( ) << endl;
}
```

7. Escriba un programa con una clase padre que contenga el nombre, calle, ciudad, estado y código postal, y clases derivadas que añadan información sobre el kilometraje acumulado en una línea aérea y en la renta de autos.

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

char cr;
class cliente {
  char nombre   [60];
       calle    [60];
       ciudad   [20];
       provincia [15];
       c_postal  [10];
 public:
  void salida_datos( void );
  void entrada_datos( void );
};

void cliente::salida_datos( );
```

```
   {
      cout << "Nombre   : " << nombre   << endl;
      cout << "Calle    : " << calle    << endl;
      cout << "Ciudad   : " << ciudad   << endl;
      cout << "Provincia : " << provincia << endl;
      cout << "C Postal  : " << c_postal << endl;
   }

void cliente::entrada_datos( )
   {
      cout << "Introduzca el nombre completo del individuo: ";
      cin.get( nombre, 59, '\n' );
      cin.get( cr );   // recorre hasta el fin de la línea
      cout << "Introduzca el nombre de la calle: ";
      cin.get( calle, 59, '\n' );
      cin.get( cr );
      cout << "Introduzca la ciudad: ";
      cin.get( ciudad, 19, '\n' );
      cin.get( cr );
      cout << "Introduzca la provincia: ";
      cin.get( provincia, 14, '\n' );
      cin.get( cr );
      cout << "Introduzca el código postal: ";
      cin.get( c_postal, 9, '\n' );
      cin.get( cr );
}




class linea_aerea: public cliente {
   char tipo_linea_aerea[20];
   float kms_vuelo_acum;

public:
   void cliente_linea_aerea( );
   void escribe_kms_volados( );
};
void linea_aerea::cliente_linea_aerea( )
   {
      entrada_datos( );
      cout << "Introduzca el tipo de Línea Aérea: ";
      cin.get( tipo_linea_aerea, 19, '\n' );
      cin.get( cr );
      cout << "Introduzca kms. de vuelo acumulados: ";
      cin >> kms_vuelo_acum;
      cin.get( cr );  // recorre hasta el fin de línea
   }
void linea_aerea::escribe_kms_volados( )
   {
      salida_datos( );
      cout << "Tipo de Línea Aérea: " << tipo_linea_aerea << endl;
      cout << "Kilómetros volados: " << kms_vuelo_acum << endl;
   }

class renta_de_autos: public cliente {
   char tipo_renta_de_autos[20];
   float kms_auto_acum;
public:
   void cliente_renta_de_autos( );
```

```
   void  escribe_kms_rodados( );
};
void  renta_de_autos::cliente_renta_de_autos( )
   {
      entrada_datos( );
      cout << "Entre tipo de Renta de Autos: ";
      cin.get( tipo_renta_de_autos, 19, '\n' );
      cin.get( cr );
      cout << "Introduzca kms. rodados: ";
      cin >> kms_auto_acum;
      cin.get( cr );
   }
void  renta_de_autos::escribe_kms_rodados( )
   {
      salida_datos( );
      cout << "Tipo Renta de Autos: " << tipo_renta_de_autos << endl;
      cout << "Kilómetros rodados: " << kms_auto_acum << endl;
   }


void  main( void )
   {
     // Asociar los nombres de variables con las clases
      linea_aerea  Iberia;
      renta_de_autos  Avis;

   // Entrada de información de línea aérea.
      cout << "\n - Cliente Línea Aérea - \n";
      Iberia.cliente_linea_aerea( );

   // Entrada de información de renta de autos
      cout << "\n - Cliente renta de autos - \n";
      Avis.cliente_renta_de_autos( );

   // Escribe toda la información del cliente
      cout << "\n - Cliente Línea aérea - \n";
      Iberia.escribe_kms_volados( );
      cout << "\n - Cliente Renta de Autos - \n";
      Avis.escribe_kms_rodados( );
   }
```

8. Escriba un programa que demuestre cómo pasar un único elemento del array por valor, variable y por referencia.

```
# include <iostream.h>

struct  Estudiante_Est  {
   char  nombre[66],
         Direccion[66],
         Ciudad[26],
         EstadoCivil[3],
         Telef[13];
   int  cursos;
   float  GPA;
};

void  LlamadaPorValor( Estudiante_Est  unEstud );
void  LlamadaPorVariable( Estudiante_Est  *punEstud );
void  LlamadaPorReferencia( Estudiante_Est  &runEstud );
```

```
void   main( void )
   {
      Estudiante_Est   GrupoGrande[100];
      GrupoGrande[0].cursos  =  10;

      LlamadaPorValor(  GrupoGrande[0] );
      cout  <<  GrupoGrande[0].cursos  <<  "\n";  // Cursos continúa siendo 10


      LlamadaPorVariable(  &GrupoGrande[0] );
      cout  <<  GrupoGrande[0].cursos  <<  "\n";  // Cursos  =  20

      LlamadaPorReferencia(  GrupoGrande[0] );
      cout  <<  GrupoGrande[0].cursos  <<  "\n";  // Cursos  =  30
}

void   LlamadaPorValor(  Estudiante_Est   unEstudiante )
   {
      //  Sintaxis normal
      unEstudiante.cursos  +=  10;
   }

void   LlamadaPorVariable(  Estudiante_Est   *punEstudiante )
   {
      // Sintaxis de puntero
      punEstudiante->cursos  +=  10;
   }

void   LlamadaPorReferencia(  Estudiante_Est   &runEstudiante )
   {
      // Sintaxis más simple con el paso por referencia
      runEstudiante.cursos  +=  10;
   }
```

12. BIBLIOGRAFIA

Cómo programar en C/C++. H.M. Deitel / P.J. Deitel. Prentice Hall, 2ª ed. 1995.

Programación orientada a objetos con C++. Fco. Javier Ceballos. Ed. ra-ma 1993.

Programación en C++. Enrique Hernández / José Hernández. Ed. Paraninfo 1993.