3.2) **Una clase para manejar tiempo.**

A continuación se presenta otra clase, la cual se puede implementar definiendo objetos miembro de este tipo dentro de una clase. La clase **tiempo** nos permite iniciar mediante la función **inicio()** un proceso, marcar con la función **marca()** el fin de un proceso, leer con la función **t_duracion()** en segundos el tiempo transcurrido en un proceso. Cuenta también con una función llamada **pausa()**, para detener un proceso un determinado número de segundos.

El archivo de cabecera contiene todos los metodos como funciones **inline**. Veamos.

```
// Programa PLCP79.H
// Una clase para menejo de retardo y tiempo
#ifndef TIEMPO_H_
#define TIEMPO_H_
#include <time.h>

class tiempo {
    clock_t ini_t, marca_t;
public:
    void inicio() {     // Inicia el conteo
        ini_t = clock();
    }
    void marca() {      // Marca el tiempo
        marca_t = clock();
    }
    clock_t duracion() { // Número de segundos transcurridos
        return (marca_t - ini_t)/CLOCKS_PER_SEC;

    }
    void pausa(int seg); // espera
};

#endif      // TIEMPO_H_
```

En la definición de la clase se usan como datos miembro del tipo **clock_t** él cual no es más que un sinónimo del tipo **long int**. El dato miembro **ini_t** almacena el valor inicial del tiempo, el cual es regresado por la función **clock()**. El dato miembro **marca_t** almacena el valor final del tiempo, también regresado por la función **clock()**. Esto nos pwermite medir la duración de un proceso.

Función: clock()

Sintaxis: #include <time.h>
clock_t clock(void);

Descripción: Esta función puede ser utilizada para medir el tiempo entre 2 eventos.
Para determinar el valor del tiempo en segundos, es necesario dividir el valor regresado por la función entre la macro CLOCK_PER_SEC o bien por la macro CLK_TCK, ambas tienen el mismo valor.
CLK_TCK = 18.2
NOTA: La computadora genera una interrupción cada 1/18.2 seg., este valor es tomado por la función.

V. regresado: Si se ejecuta, regresa el tiempo transcurrido desde la invocación del programa, en 1/18.2 seg.
Si hay un error, (El valor de tiempo no esta disponible en el procesador), regresa -1.

Los valores regresados por la función **clock()** son tomados por las funciones miembro **inicio()** y **marca()**.

La función **pausa()**, nos permite generar retardos del valor recibido como argumento. Su implementación se muestra en la siguiente aplicación, la cual nos permite probar la clase tiempo.

La función **duracion()** nos convierte un intervalo de tiempo generado por la función **clock()**, en un valor en segundos.

```
// Programa PLCP79.CPP
//Prueba de la clase tiempo
// Toma como argumento la duración de la pausa
// desde la línea de comandos.
#include <iostream.h>
#include <stdlib.h>
#include "plcp78.h"
#include "plcp79.h"

void tiempo::pausa(int seg) {  // espera
    inicio();
    do
        marca();
    while( duracion() < seg);
}

static ioserror error("Prueba de tiempo");


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

```
    if(argc > 2)
        error << "Use: Tiempo en segundos"<< termina;

    tiempo T1, T2;
    T1.inicio();
    T2.pausa(atoi(argv[1]));
    T1.marca();
    cout << "Duración: " << (int)T1.duracion()
         << " segundos" << endl;
}
```

OBSERVACIONES.

El programa anterior, trabaja de una manera muy simple, es necesario darle un valor de tiempo desde la línea de comandos, dicho valor es convertido a valor numérico y aplicado a la función **pausa()**, entonces, durante la ejecución del programa se va a estar verificando que se llegue al intervalo de tiempo dado, mientras, no va a hacer nada la computadora, una vez que se cumplió el intervalo de tiempo, llama a la función miembro **marca()**, para actualizar el valor del tiempo, llama a la función miembro **duracion()** para calcular el valor del intervalo en base a la llamada anterior de la función **marca()**, nos imprime en pantalla dicho valor y termina el programa.

Note que una instancia de la clase **error_handler** es usada como un objeto ordinario, no miembro, para la impresión de los mensajes de error, si el usuario no proporciona el número correcto de argumentos.

3.2.a) Un ejemplo más interesante del uso de la clase tiempo.

En la siguiente aplicación, se proporciona en la línea de comandos, además del nombre del programa, el nombre de un comando del DOS (puede ser otro programa, el cual se quiere ejecutar), el programa nos regresa el tiempo en segundos que tardó en ejecutarse el programa dado como argumento.

```
// Programa PLCP79A.CPP
// Demuestra el uso de la clase iostream
// Para correr el programa dé en la línea de comandos
// algunos comando para ser ejecutados por el sistema
// operativo.
// Al terminar el programa nos mostrará el tiempo
// que tardo el sistema en ejecutar los comandos
// dados
#include <iostream.h>
#include <stdlib.h>
#include <strstrea.h>
```

```
#include "plcp78.h"
#include "plcp79.h"

void tiempo::pausa(int seg) {  // espera
    inicio();
    do
        marca();
    while( duracion() < seg);
}

static ioserror error("Tiempo comando");

void main(int argc, char *argv[])
{
    if(argc < 2)
        error << nl << "\tUse: tiempo com en línea de comandos"
              << termina;
    ostrstream com;
    // Copia el resto de la línea en un solo buffer
    for(int i = 1; i < argc; i++)
        com << argv[i] << " ";
    com << ends;

    tiempo T1;
    T1.inicio();
    cout << com.str() << endl;
    system(com.str());
    T1.marca();
    cout << "Duración: " << (int)T1.duracion()
         << " segundos" << endl;
    delete com.str();   // limpia el ostrstream
}
```

OBSERVACIONES.

El programa puede recibir más de un argumento en la línea de comandos, éstos son almacenados en el objeto **com**, él cual es del tipo **ostrstream**. ostrstream nos proporciona un stream de salida, desde un arreglo usando streambuf. El constructor empleado crea un arreglo dinámico para almacenar las cadenas dadas desde la línea de comandos.

ostrstream es una clase derivada de la clase **strstreambase** y **ostream**.

La función miembro **str()**, regresa la dirección del buffer usado por el objeto **com**. Esto nos permite llamar a un comando desde la línea de comandos, e inclusive darle a éste su propio argumento. Una vez usada la función miembro **str()** es responsabilidad del programador liberar la memoria usada, si fué realizado acceso dinámico, mediante el operador **delete**.

En el **for** el programa llena el buffer con los argumentos dados desde la línea de comandos, al final le inserta un NULO mediante el manipulador **ends**.

A continuación, definimos un objeto T1 y llamamos con él a la función miembro inicia(), para definir el primer valor del intervalo de tiempo.

Luego imprimimos el contenido del buffer, cuya dirección es regresada por la función miembro de ostrstream, **str()**. Este contenido es enviado a la función global **system()**, la cual hace uso del command.com del sistema operativo para ejecutar el comando que le fué pasado como argumento.

Una vez terminado de ejecutarse el comando, se llama a la función miembro **marca()**, para actualizar el segundo valor del tiempo y mediante la función **duracion()**, calcular el tiempo que duró la ejecución del comando, realizada mediante la función global **system()**.

Por último se libera la memoria del heap, solicitada por el constructor de **ostrstream**, cuando fué declarado el objeto **com**.

4.- **Reutilizando código mediante el uso de herencia.**

La herencia se puede utilizar de 2 maneras; La primera es cuando se tiene ya una clase y se quiere hacer algunas pequeñas modificaciones, sin embargo, no es conveniente cambiar dicha clase. La segunda forma de utilizar la herencia es expresar un problema como una jerarquía de clases. La interface de la clase base es común a todas las clases derivadas. Las subclases pueden manipularse utilizando esta interface común.

En esta sección se examina la forma de usar una clase existente, de tal manera de construir una nueva clase sobre ella usando el concepto de herencia. Se puede pensar que tal vez resulte más sensato, modificar la clase base que usar herencia, sin embargo a continuación se enumeran algunas buenas razones para usar herencia.

1.- No se tiene disponible el código de la clase base.

2.- No se puede modificar la clase base.

3.- Cada clase dentro de la herencia puede representar un concepto diferente, él cual es útil por sí mismo y no como parte de una clase grande.

4.- Cuando se esta escribiendo código, la herencia nos facilita el trabajo de rastreo (Debug) de una porción del programa, ya que al usar una clase derivada, su código es más reducido y no se pierde con el código de la clase base, pudiendo más facilmente detectar errores.

4.1) Clase semilla.

Si analizamos la segunda forma de utilizar la herencia, en donde en primera instancia se concive una idea general del problema, de la cual resulta la definición de algunas clases, y posteriormente nos damos cuenta que existe repetición de código en estas clases, entonces, debemos de sacar el código repetido relegandolo a un pariente, la clase base, y definiendo clases derivadas en las cuales no exista código repetido. Cuando se hace ésto, en donde se define una clase para compartir código, a esta clase se le llama **clase semilla**. Este tipo de clases no necesariamente deben de ser abstractas.

Veamos la primera aproximación, en donde se definen 2 clases las cuales cuentan con atributos comunes.

```
// Programa PLCP80.CPP
// Clases con atributos comunes.
#include <iostream.h>

class librero {
    int color;
    int ancho, alto;
    int divisiones;
public:
    librero(int, int, int, int);
    int getcolor() { return color; }
    int getancho() { return ancho; }
    int getalto()  { return alto;  }
    int getdiv()  { return divisiones; }
};

class escritorio {
    int color;
    int ancho, alto;
    int dibujo;
    int material;
public:
    escritorio(int, int, int, int, int);
    int getcolor() { return color; }
    int getancho() { return ancho; }
    int getalto()  { return alto;  }
    int getdib()   { return dibujo; }
    int getmat()   { return material; }
};

librero::librero(int a, int b, int c, int d)
```

```
{
    color = a;
    ancho = b;
    alto = c;
    divisiones = d;
}

escritorio::escritorio(int a, int b, int c, int d, int e)
{
    color = a;
    ancho = b;
    alto = c;
    dibujo = d;
    material = e;
}


void main()
{
    escritorio E(1, 2, 3, 4, 5);
    librero L(10, 20, 30, 40);

    cout << "color      = " << L.getcolor() << endl;
    cout << "ancho      = " << L.getancho() << endl;
    cout << "alto       = " << L.getalto() << endl;
    cout << "divisiones = " << L.getdiv() << endl << endl;

    cout << "color      = " << E.getcolor() << endl;
    cout << "ancho      = " << E.getancho() << endl;
    cout << "alto       = " << E.getalto() << endl;
    cout << "dibujo     = " << E.getdib() << endl;
    cout << "material   = " << E.getmat() << endl;
}
```

OBSERVACIONES.

Estas clases tienen atributos comunes y otros no lo son, de la misma forma se estan repitiendo funciones miembro.

La salida nos resulta:

```
color      = 10
ancho      = 20
alto       = 30
divisiones = 40

color      = 1
```

```
ancho      = 2
alto       = 3
dibujo     = 4
material   = 5
```

Lo anterior se puede modificar si definimos una clase que contenga los atributos y las funciones miembro comunes de las clases librero y escritorio, veamos la modificación, a la clase común, a la cual le llamaremos, **articulo**.

```
// Programa PLCP80A.CPP
// Clases con atributos comunes.
// Declarando una clase base.
#include <iostream.h>

class articulo {
    int color;
    int ancho, alto;
public:
    articulo(int, int, int);
    int getcolor() { return color; }
    int getancho() { return ancho; }
    int getalto()  { return alto;  }
};

articulo::articulo(int a, int b, int c)
{
    color = a;
    ancho = b;
    alto = c;
}

// Se simplifican las clases derivadas.

class librero : public articulo {
    int divisiones;
public:
    librero(int, int, int, int);
    int getdiv()  { return divisiones; }
};

class escritorio : public articulo {
    int dibujo;
    int material;
public:
    escritorio(int, int, int, int, int);
    int getdib()   { return dibujo; }
    int getmat()   { return material; }
```

```
};


librero::librero(int a, int b, int c, int d) : articulo(a, b, c)
{
    divisiones = d;
}

escritorio::escritorio(int a, int b, int c, int d, int e)
: articulo(a, b, c)
{
    dibujo = d;
    material = e;
}

void main()
{
    escritorio E(1, 2, 3, 4, 5);
    librero L(10, 20, 30, 40);

    cout << "color      = " << L.getcolor() << endl;
    cout << "ancho      = " << L.getancho() << endl;
    cout << "alto       = " << L.getalto() << endl;
    cout << "divisiones = " << L.getdiv() << endl << endl;

    cout << "color      = " << E.getcolor() << endl;
    cout << "ancho      = " << E.getancho() << endl;
    cout << "alto       = " << E.getalto() << endl;
    cout << "dibujo     = " << E.getdib() << endl;
    cout << "material   = " << E.getmat() << endl;
}
```

OBSERVACIONES.

Ahora definimos una clase común llamada **articulo** (clase semilla), la cual contiene los atributos y funciones comunes de las clases librero y escritorio. También podemos observar que la aplicación no sufrió ninguna modificación a pesar de haber modificado la estructura de las clases.

En los constructores de cada una de las clases derivadas fué necesario llamar al constructor de la clase base.

La salida por lo tanto, será igual a la anterior.

El uso de una clase semilla, es frecuente, incluso si todavía no se conocen cuales elementos van a ser comunes a las clases derivadas, se puede declarar esta clase semilla y dejarla vacía. Veamos de nuevo nuestro ejemplo, usando esta forma de diseño.

```
// Programa PLCP80X.CPP
// Clases con atributos comunes.
// Usando una clase semilla vacía.

#include <iostream.h>


class articulo { };

class librero : public articulo {
    int color;
    int ancho, alto;
    int divisiones;
public:
    librero(int, int, int, int);
    int getcolor() { return color; }
    int getancho() { return ancho; }
    int getalto()  { return alto;  }
    int getdiv()   { return divisiones; }
};


class escritorio : public articulo {
    int color;
    int ancho, alto;
    int dibujo;
    int material;
public:
    escritorio(int, int, int, int, int);
    int getcolor() { return color; }
    int getancho() { return ancho; }
    int getalto()  { return alto;  }
    int getdib()   { return dibujo; }
    int getmat()   { return material; }
};
```

Note que solo se declaró la clase **articulo**, y también fué adicionada como clase base a las clases **librero** y **escritorio**. El código restante del progrma PLCP80.CPP no sufrió modificación.

4.2) Conversión de tipo, usando clases derivadas.

Cuando se usa herencia en un programa en C++, es posible emplear la jerarquía de clases para realizar ciertas conversiones entre clases a diferentes niveles del árbol de clases. Consideremos las siguientes 2 clases.

```
class A {};
class B : public A {};
```

Ya que la clase B es derivada de la clase A, los miembros de A estarán disponibles para B. Con lo anterior podemos decir que; la clase B es en cierta forma un superconjunto de A, por lo tanto es posible convertir un objeto del tipo B en un objeto del tipo A pero NO a la inversa.

```
// Programa PLCP80C.CPP
// Conversión de tipos entre clases.

class A {};
class B : public A {};

void func(A *d) {}

void  main()
{
    A a;
    B b;

    A *ap = new A;
    B *bp = new B;

    a = b;
    ap = bp;

     func(ap);
     func(bp);

     delete ap;
     delete bp;
}
```

OBSERVACIONES.

La conversión también ocurre de manera impliita en la llamada a una función.

La capacidad de C++ de soportar conversión de clases se da de manera natural, el polimorfismo es una extensión de este concepto.

NOTA: El programa anterior, a pesar que no hace nada fué compilado, se hace notar que el compilador solo generó 2 advertencias por el no uso de los objetos **a** y **ap**.

Veamos un ejemplo usando la estructura de clases definida en los programas anteriores.

```
// Programa PLCP80B.CPP
// Clases con atributos comunes.
// Declarando una clase base.

#include <iostream.h>


class articulo {
    int color;
    int ancho, alto;
public:
    articulo(int, int, int);
    int getcolor() { return color; }
    int getancho() { return ancho; }
    int getalto()  { return alto;  }
};

articulo::articulo(int a, int b, int c)
{
    color = a;
    ancho = b;
    alto = c;
}

// Se simplifican las clases derivadas.

class librero : public articulo {
    int divisiones;
public:
    librero(int, int, int, int);
    int getdiv()  { return divisiones; }
};


class escritorio : public articulo {
    int dibujo;
    int material;
public:
    escritorio(int, int, int, int, int);
    int getdib()   { return dibujo; }
    int getmat()   { return material; }
};


librero::librero(int a, int b, int c, int d) : articulo(a, b, c)
{
```

```
    divisiones = d;
}


escritorio::escritorio(int a, int b, int c, int d, int e)
: articulo(a, b, c)
{
    dibujo = d;
    material = e;
}

void imprime(articulo *x)
{
    cout << "Salida de la función imprime: " << x -> getcolor()
         << endl;
}

void main()
{
    articulo A(22, 33, 44);
    escritorio E(1, 2, 3, 4, 5);
    librero L(10, 20, 30, 40);

    cout << "Objeto A" << endl;
    cout << "color      = " << A.getcolor() << endl;
    cout << "ancho      = " << A.getancho() << endl;
    cout << "alto       = " << A.getalto() << endl << endl;

    A = E;

    cout << "Objeto A" << endl;
    cout << "color      = " << A.getcolor() << endl;
    cout << "ancho      = " << A.getancho() << endl;
    cout << "alto       = " << A.getalto() << endl << endl;

    A = L;

    cout << "Objeto A" << endl;
    cout << "color      = " << A.getcolor() << endl;
    cout << "ancho      = " << A.getancho() << endl;
    cout << "alto       = " << A.getalto() << endl << endl;

    cout << "Objeto L" << endl;
    cout << "color      = " << L.getcolor() << endl;
    cout << "ancho      = " << L.getancho() << endl;
    cout << "alto       = " << L.getalto() << endl;
    cout << "divisiones = " << L.getdiv() << endl << endl;

    cout << "Objeto E" << endl;
```

```
    cout << "color      = " << E.getcolor() << endl;
    cout << "ancho      = " << E.getancho() << endl;
    cout << "alto       = " << E.getalto() << endl;
    cout << "dibujo     = " << E.getdib() << endl;
    cout << "material   = " << E.getmat() << endl << endl;

    articulo *ap = new articulo(100, 200, 300);
    librero *lp  = new librero(11, 12, 13, 14);

    cout << "Objeto *ap" << endl;
    cout << "color      = " << ap -> getcolor() << endl;
    cout << "ancho      = " << ap -> getancho() << endl;
    cout << "alto       = " << ap -> getalto() << endl << endl;

    cout << "Objeto *lp" << endl;
    cout << "color      = " << lp -> getcolor() << endl;
    cout << "ancho      = " << lp -> getancho() << endl;
    cout << "alto       = " << lp -> getalto() << endl;
    cout << "divisiones = " << lp -> getdiv() << endl << endl;

    *ap = *lp;

    cout << "Objeto *ap" << endl;
    cout << "color      = " << ap -> getcolor() << endl;
    cout << "ancho      = " << ap -> getancho() << endl;
    cout << "alto       = " << ap -> getalto() << endl << endl;

    imprime(ap);
    imprime(lp);

    delete ap;
    delete lp;
}
```

OBSERVACIONES.

En el programa anterior se trata de demostar que es posible asignar un objeto de una clase derivada a una clase base, así mismo se presenta la conversión realizada en el pase de un argumento de una clase derivada a un parámetro de una clase base, en la función global **imprime()**.

Recuerde que en la llamada a **new**, este operador realiza una llamada al constructor de la clase involucrada.

Veamos la salida del programa.

```
Objeto A            // Valor original
color      = 22
```

```
ancho      = 33
alto       = 44

Objeto A           // Después de copiar A = E;
color      = 1
ancho      = 2
alto       = 3

Objeto A           // Después de copiar A = L:
color      = 10
ancho      = 20
alto       = 30

Objeto L           // Valor original
color      = 10
ancho      = 20
alto       = 30
divisiones = 40

Objeto E           // Valor original
color      = 1
ancho      = 2
alto       = 3
dibujo     = 4
material   = 5

Objeto *ap         // Valor original
color      = 100
ancho      = 200
alto       = 300

Objeto *lp         // Valor original
color      = 11
ancho      = 12
alto       = 13
divisiones = 14

Objeto *ap         // Después de copiar *ap = *lp;
color      = 11
ancho      = 12
alto       = 13

Salida de la función imprime: 11
Salida de la función imprime: 11
```

En la salida se trata de explicar lo ocurrido, usando como referencia el símbolo de comentario de C++ [//].

4.3) Resolución de ámbito.

Derivando una clase de otra es posible hacer referencia a un dato miembro o a una función miembro, las cuales tienen el mismo nombre en ambas clases. Para hacer ésto es necesario decirle al compilador exactamente sobre cual dato o función se esta haciendo referencia. Veamos un ejemplo simple:

```
// Programa PLCP81.CPP

// Dos clases que usan una función miembro
// con el mismo nombre.

#include <iostream.h>

class A {
public:
    int func() { return 1; }
};

class B : public A {
public:
    int func() { return 2; }
};


void main()
{
    A a;
    B b;

    int i = a.func();
    int j = b.func();

    cout << "Valor de i = " << i << endl;
    cout << "Valor de j = " << j << endl;
}
```

OBSERVACIONES.

Cuando se empieza a trabajar con C++, puede producir confución sobre cual función es llamada y que valor será almacenado en **i** y en **j**. La clase **B** tiene 2 funciones **func()**, una es heredada de la clase **A** y la otra es nativa de la clase **B**. Esencialmente el compilador usa la versión de la función de la clase **B**, y luego haría

una busqueda hacia arriba del árbol de clases. Si pensamos en una herencia múltiple la trayectoria se vuelve más complicada.

En el ejemplo presentado arriba, sí observamos la salida, vemos que:

```
Valor de i = 1
Valor de j = 2
```

Al hacer referencia el compilador al objeto **b**, se da cuenta que existen 2 funciones con el mismo nombre. La regla de ámbito de C++ permite que esto ocurra;

Si un nombre en una clase base es redeclarado en una clase derivada, el nombre en la clase derivada oculta el nombre en la clase base.

Es el mismo caso que se presenta cuando se declaran variables en ámbitos diferentes.

En C++ se puede forzar al compilador para que vea fuera del ámbito actual y tenga acceso a nombres que de otra forma son ocultados. Este proceso se realiza usando el operador de resolución de ámbito, de acuerdo a la siguiente sintáxis:

<class nombre> <class identificador>

Donde:

`class nombre:` Es el nombre de cualquier clase base o clase derivada.

`class identificador` Es el nombre de cualquier variable o función declarada en la clase.

Para usar ésto, considere la siguiente modificación en el programa:

```
// Programa PLCP82.CPP
// Dos clases que usan una función miembro
// con el mismo nombre.
#include <iostream.h>

class A {

public:
    int func() { return 1; }
};

class B : public A {
public:
```

```
    int func() { return 2; }
};


void main()
{
    A a;
    B b;

    int i = a.func();
    int j = b.func();
    int k = b.A::func();

    cout << "Valor de i = " << i << endl;
    cout << "Valor de j = " << j << endl;
    cout << "Valor de k = " << k << endl;
}
```

OBSERVACIONES.

En el programa se está usando el operador de resolución de ámbito para indicarle al compilador que se está invocando la función miembro **func()** de la clase base.

La salida nos resulta:

```
Valor de i = 1
Valor de j = 2
Valor de k = 1
```

Note que aunque se esta haciendo referencia al objeto **b**, el compilador hace el llamado a la función miembro de la clase base.

También es posible crear una función auxiliar dentro de la clase derivada para tener acceso a la función con el mismo nombre en la clase base.

Veamos un ejemplo:

```
// Programa PLCP82A.CPP
// Dos clases que usan una función miembro
// con el mismo nombre.

#include <iostream.h>

class A {
public:
    int func() { return 1; }
```

```
};


class B : public A {
public:
    int func() { return 2; }
    int fua()  { return A::func(); }
};


void main()
{
    A a;
    B b;
    B *bp = new B;


    int i = a.func();
    int j = b.func();
    int k = b.fua();

    cout << "Valor de i = " << i << endl;
    cout << "Valor de j = " << j << endl;
    cout << "Valor de k = " << k << endl;
    cout << "Valor de l = " << bp -> func() << endl;
}
```

OBSERVACIONES.

En el ejemplo se puede ver como se definió una función auxiliar **fua()**, mediante la cual se hace acceso a la función **func()** de la clase base, en esta función se usó el operador de resolución de ámbirto.

Adicionalmente en el programa se presenta la manera de llamar a una función utilizando un puntero a objeto, él cual sigue las mismas reglas que cuando se hace referencia a un objeto.

La salida nos resulta:

```
Valor de i = 1
Valor de j = 2
Valor de k = 1
Valor de l = 2
```

En **k** se almacena el valor regresado por la función **func()** de la clase base.

En **l**, solo se usa como referencia de salida, se imprime el valor regresado por **func()** de la clase derivada.

Cuando se cuenta con varios niveles de herencia, el operador de resolución de ámbito nos permite accesar los miembros de cualquier clase base para la cual se cuenta con los privilegios de acceso necesarios.

Considere una clase D, la cual necesita hacer acceso a funciones con el mismo nombre, pero las cuales son declaradas en un nivel diferente de herencia.

En el siguiente ejemplo se muestran 4 niveles de jerarquía de clases, y desde la clase D, queremos tener acceso a la función **func()**, la cual se encuentra definida también en los demás niveles.

```
// Programa PLCP82B.CPP
// Usando unámbito de resolución de varios niveles.
#include <iostream.h>

class A {
public:
    int func() { return 1; }
};

class B : public A {
public:
    int func() { return 2; }
};

class C : public B {
public:
    int func() { return 3; }

};

class D : public C {
public:
    int func() { return 4; }
};

void main()
{
    D d;

    cout << "Función de A = " << d.A::func() << endl;
    cout << "Función de B = " << d.B::func() << endl;
    cout << "Función de C = " << d.C::func() << endl;
    cout << "Función de D = " << d.func() << endl;
}
```

OBSERVACIONES.

La salida nos resulta:

```
Función de A = 1
Función de B = 2
Función de C = 3
Función de D = 4
```

Otra manera de hacer ésto es definir funciones auxiliares en la clase D, que nos permitan accesar cada una de las funciones **func()** de las clases; A, B y C.

```
class D : public C {
public:
          int func() { return 4; }
          int f1() { return A::func(); }
          int f2() { return B::func(); }
          int f3() { return C::func(); }
          int f4() { return D::func(); } // Es redundante pero
};                                       // válido.
```

Las funciones miembro de la clase **D** contienen la definición de resolución de ámbito para todos los niveles de la jerarquía. Note como es necesario especificar el operador de resolución de ámbito solamente cuando una clase derivada usa un identificador él cual, también es usado en la clase base.

4.3.a) Resolución de ámbito para datos miembro.

En el caso de datos miembro, es el mismo proceso, veamos un ejemplo:

```
// Programa PLCP83.CPP
// Ambito de resolución para datos miembro.

#include <iostream.h>

class A {
    int valor;
public:
    A(int x) { valor = x; }
    void imprime() {
        cout << "Valor en A = " << valor << endl;
    }
```

```
};

class B : public A {
    int valor;
public:
    B(int x, int y) : A(x) { valor = y; }
    void imprime() {
        cout << "Valor en B = " << valor << endl;
    }

};

void main()
{
    B b(55, 77);

    b.A::imprime();
    b.imprime();
}
```

OBSERVACIONES.

La salida nos resulta:

```
Valor en A = 55
Valor en B = 77
```

En este caso los datos miembro son privados de cada clase, por lo tanto, solo pueden ser accesados por sus funciones miembro, que pasa cuando los declaramos públicos, veamos ésto, haciendo algunas pequeñas modificaciones al programa anterior.

```
// Programa PLCP83A.CPP
// Ambito de resolución para datos miembro.
#include <iostream.h>

class A {
public:
    int valor;
    A(int x) { valor = x; }
};

class B : public A {
public:
    int valor;
public:
    B(int x, int y) : A(x) { valor = y; }
```

```
};

void main()
{
    B b(55, 77);

    cout << "Valor en A = " << b.A::valor << endl;
    cout << "Valor en B = " << b.valor    << endl;
}
```

OBSERVACIONES.

La salida nos resulta:

```
Valor en A = 55
Valor en B = 77
```

Se puede apreciar que se usa la misma sintaxis que con las funciones miembro, mediante el uso del operador de resolución de ámbito. El resultado es el mismo que en el ejemplo anterior.

La misma sintaxis puede ser empleada con punteros a objeto y referencias.