

INFORMATICA

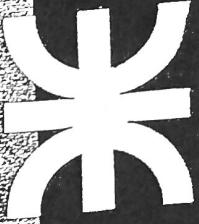
Ing. Jorge Argibay

Elementos de

Electrónica

R2CT18

UNIVERSIDAD DE EDUCACION NACIONAL
FACULTAD REGIONAL BUENOS AIRES



CEIT
F.R.B.A.

ELEMENTOS DE C++

PROGRAMACION ORIENTADA A OBJETOS (POO)

ANALISIS DEL PROBLEMA

A fin de visualizar la evolución de los lenguajes, recorreremos el camino del crecimiento de los programas, observando las dificultades que se presentan.

En el caso de programas pequeños, su simplicidad permite que el programador se tome algunas licencias, y realice el programa siguiendo sus propias reglas (o ninguna). De esta manera que los programas tienen una fuerte presencia de la inventiva del programador. Denominamos a estos, "programas artesanales".

A medida que los programas crecen, en tamaño y complejidad, la subjetividad manifiesta en el diseño del programa hace que el mismo sea difícil de analizar y comprender por otros programadores, y aun, por su mismo creador.

Considerando que un programa es un ente en evolución (ya sea por corrección de errores o por modificaciones), se hace necesario que sea comprensible, y que una modificación en una parte del mismo, no acarree cambios indeseados en otras partes.

El nombre de programas "tallarin" surge al observar los diagramas de flujo tradicionales, llenos de "GO TO" hacia todos lados. Tienen la apariencia de fideos en un plato de tallarines.

A fin de solucionar estos problemas, se determinaron una serie de reglas, seguramente conocidas por el lector, bajo el nombre de "diagramación estructurada", que conducen a la "programación estructurada".

A fin de adecuarse a dichas reglas, surgieron los lenguajes estructurados, como el PASCAL y el C.

Una de las características más notables de la programación estructurada es la MODULARIDAD. Esta consiste en fraccionar el programa en "módulos estancos" que permitan analizarlos, crearlos y probarlos en forma independiente unos de otros.

Estos módulos adoptan el nombre de subrutinas, funciones o procedimientos, según el lenguaje utilizado.

Hasta este punto del análisis, los lenguajes (tanto estructurados, como no estructurados) son "procedurales", es decir están orientados al procedimiento mismo.

Creo necesario aclarar que la programación estructurada es suficiente para una gran gama de problemas

A medida que los programas crecieron (superando las 25.000 líneas) se advirtió que el modelo procedural era imperfecto.

El punto débil de este modelo lo constituyen los DATOS.

Es decir, el modelo procedural, está enfocado en el "como" se hacen las cosas, y deja en segundo plano al motivo del programa mismo.

El programa tiene razón de ser, solamente por el RESULTADO. Este resultado es consecuencia del proceso de los DATOS.

Con este criterio, lo importante en todo el proceso, no es el programa mismo, sino los datos y los resultados.

Concretamente, en los lenguajes procedurales, todas las funciones tienen acceso a los datos. En muchas ocasiones, acceden a los datos que deben manejar, y a los que no deben, también.

Por ejemplo, en una base de datos de stock, precio y características de artículos, existirán una serie de procesos como actualización, búsqueda, altas y bajas, etc. tanto de precios como de cantidades, que serán llevadas a cabo por una serie de funciones.

Una función de cambio de precios puede tener acceso (incorrectamente) a la cantidad de elementos en stock, y, en caso de ser creada o modificada por un programador que desconoce el contexto, los datos pueden resultar erróneamente modificados.

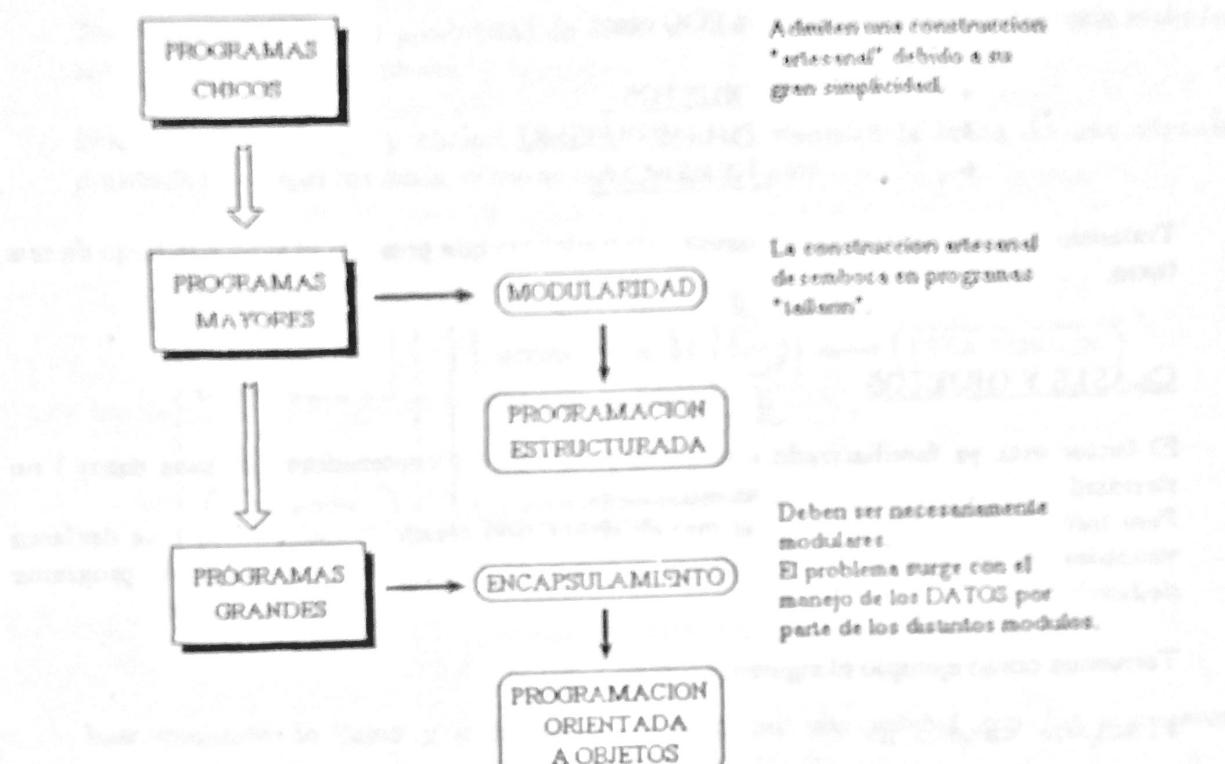
Nótese que la situación planteada exige que diferentes funciones accedan a los datos, y por lo tanto la protección de los mismos no puede resolverse ocultándolos como variables locales de las mismas.

La situación anteriormente descripta desemboca directamente en la necesidad de proteger a los datos, restringir el acceso a los mismo y ocultarlos a todo el programa, con excepción de algunas funciones "privilegiadas" destinadas a manejarlos directamente. Cuando cualquier función necesite el acceso a dichos datos, solo podrá realizarlo a través de las funciones "privilegiadas". Estas funciones reciben, en general, el nombre de "métodos", y en particular (en C++), el nombre de "funciones miembro".

Está claro que los datos y las funciones miembro están estrechamente vinculados. Tanto es así, que pasan a formar un solo ente denominado "objeto".
Un objeto es una cápsula dentro de la cual "viven" los datos y las funciones miembro que los manejan. Cualquier acceso a los datos se debe realizar a través de las funciones miembro.

Esta característica se denomina ENCAPSULAMIENTO y es el pilar fundamental de la POO.

ESQUEMA DEL PROBLEMA



CARACTERISTICAS DEL C++.

Hacia 1980, basándose en un lenguaje existente denominado SIMULA 67, Bjarne Stroustrup, quien trabajaba para los Laboratorios Bell en Murray Hill, NJ, añadió extensiones al lenguaje C, destinadas a solucionar los problemas mencionados anteriormente.

Llamó a este lenguaje "C con clases". Hacia 1983 se adoptaría el nombre de "C++". Tras una serie de revisiones del C++, la empresa Borland lanzó al mercado en 1990, el compilador Turbo C++.

Dado que C++ es un derivado de C, mantiene las características y ventajas de este. Podría considerarse que C++ es un "superconjunto" que contiene a C como elemento.

Sin embargo, es necesario ser muy cuidadoso al compilar bajo C++ programas construidos en C. Especialmente el programador acostumbrado a tomarse algunas "licencias" con el compilador Turbo C.

El compilador Turbo C++ es más estricto que el Turbo C en cuanto a conversiones de tipo, declaración de prototipos e inclusión de cabeceras.

Los elementos sobresalientes de la POO son :

- ◆ OBJETOS
- ◆ POLIMORFISMO
- ◆ HERENCIA

Trataremos estos temas junto a nuevas características que presenta el C++ a lo largo de este texto.

CLASES Y OBJETOS

El lector está ya familiarizado con datos y variables (contenedores de esos datos) no standard en C. Tal es el caso de las estructuras. Para trabajar con ellos, se crea un *tipo de dato* (tipo creado por el usuario), se declaran variables de ese tipo y se las manejan a través de funciones o fracciones de programa dedicadas a tal fin.

Tomemos como ejemplo el siguiente código :

```
#include <stdio.h>

typedef struct {
    int A ;
    float B ; } TIPO ;

TIPO ESTRUCTURA ; /* Variable global */

void FUNCION ( void ) ; /* Prototipos de funciones */
void OTRA_FUNCION ( void )
```

La situación queda reflejada en la siguiente figura :



TIPO es el nuevo tipo de dato creado mediante *typedef*. Solamente es el descriptor del dato y no ocupa memoria.

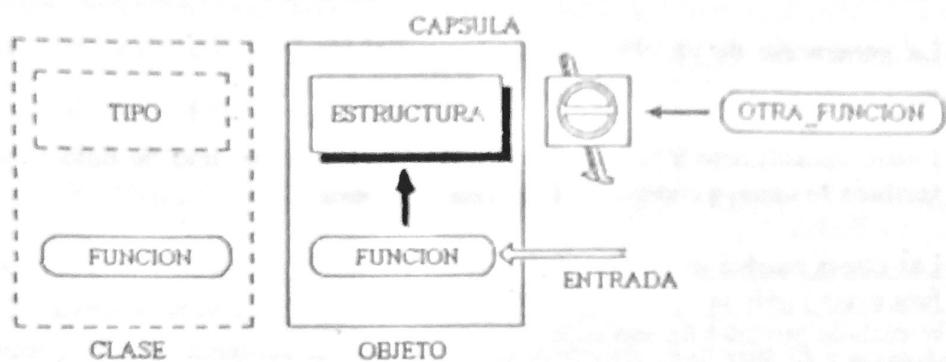
ESTRUCTURA es la variable declarada de tipo **TIPO**, ocupa memoria y contiene eventualmente al dato.

Ambas funciones pueden acceder a la variable **ESTRUCTURA**.

Consideremos que la función **OTRA_FUNCION()** no debería acceder al dato, pero lo hace erróneamente como se sugiere en la figura.

Podríamos considerar la posibilidad de aislar el dato y las funciones que si deben actuar sobre él, del resto del contexto.

Este conjunto de dato y código, aislados del resto tomarían la forma de una cápsula destinada a proteger los datos, como se muestra en la figura:



Este conjunto de datos y código pasa a constituir una entidad con las siguientes características:

- ◆ Se constituye un nuevo tipo de dato creado por el usuario. El tipo general de dato constituye una CLASE.
- ◆ La CLASE es solamente el descripto y no ocupa memoria.
- ◆ La CLASE involucra a tipos de datos y código destinado a manejarlos.
- ◆ La realización de la CLASE corresponde a un OBJETO. Es decir, la CLASE es al tipo de datos como el OBJETO es a la variable.
- ◆ las variables que integran el OBJETO en realidad son campos del mismo denominadas VARIABLES-MIEMBRO y, en principio, solo pueden ser accedidas por las funciones que integran el OBJETO.
- ◆ Las funciones que pertenecen al objeto se denominan FUNCIONES-MIEMBRO y pueden ser accedidas desde el exterior de la cápsula, o no.
- ◆ Las VARIABLES-MIEMBRO no pueden ser accedidas (a menos que se especifique lo contrario) por funciones ajenas al OBJETO. De esta forma, quedan ocultas al exterior.

CLASES

Se enumeraran a continuación una serie de características de las clases :

- ♦ Como se vio anteriormente, la clase es el descriptor del objeto. Por lo tanto, para generar un objeto es necesario crear primero su clase.
- ♦ La generación de un objeto a partir de una clase se denomina instancia. Entonces, instanciar un objeto es declararlo a partir de una clase.
- ♦ Puede considerarse a la clase, como una extensión del tipo de dato struct, en el que también se incluye código, en forma de funciones.
- ♦ Las clases pueden tener una parte pública y/o una parte privada. Los miembros (datos y funciones) públicos pueden ser accesibles desde el exterior del objeto, mientras que los miembros privados no son accesibles desde el exterior.
- ♦ Para definir la situación anterior se utilizan las palabras reservadas public y private.
- ♦ Si nada se especifica, la categoría de un miembro es privada. Es decir, el default de acceso para los miembros de clases es privado.
- ♦ Al crear una clase, no es necesario que el cuerpo de las funciones miembro integren la declaración. Basta con incluir su prototipo.
- ♦ Para definir una clase se usa el nombre reservado class.

Sintaxis de declaración :

```
class [ nombre_de_clase ] {
    [ private : ]
        | datos y funciones privadas ;
    public :
        | datos y funciones publicas ;
    } [ declaración de objetos ] ;
```

Los elementos entre corchetes, son opcionales, pero, al igual que en el caso de las estructuras, debe figurar o el nombre del tipo o el de las variables, o ambos. La indicación de categoría de acceso privada podría omitirse debido a que es la elección por omisión.

A continuación se muestra la declaración de una clase llamada CLASE y la instanciacción de un objeto llamado OBJETO.

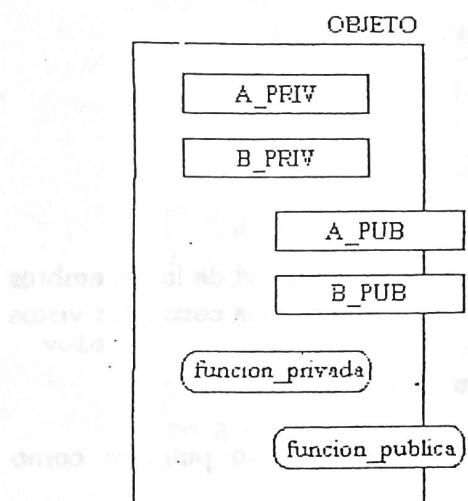
```
class CLASE {  
    private :  
        int A_PRIV ;  
        float B_PRIV ;  
        void funcion_privada () ;  
    public :  
        int A_PUB ;  
        float B_PUB ;  
        void funcion_publica () ;  
} OBJETO ;
```

MIEMBROS PUBLICOS Y MIEMBROS PRIVADOS

Lo que diferencia a los miembros públicos de los privados es su acceso o visibilidad. Los miembros públicos (tanto los campos como las funciones) son visibles y accesibles desde el exterior del objeto.

Los miembros privados (tanto los campos como las funciones) solo son accesibles por miembros del mismo objeto.

NOTA : Algunas de estas consideraciones pueden ser ligeramente modificadas en las próximas páginas, pero por ahora, por razones de simplicidad, se aceptaran de esta manera.



Notación de EGE :

En esta representación de objetos se propone mostrar a los miembros públicos "saliendo" del rectángulo o cápsula del objeto, indicando que son "visibles" desde el exterior del mismo.

Los miembros privados quedan totalmente "protegidos" por la cápsula y no son visibles desde el exterior.

El objeto OBJETO del ejemplo anterior se representa como muestra la figura.

CUERPO DE LAS FUNCIONES

El cuerpo de las funciones miembro se puede colocar dentro de la declaración de clase, o bien, indicar solo el prototipo dentro de la clase, y explicitar el cuerpo de la función fuera de ella.

Si se realiza de esta manera es necesario indicar que la función pertenece a una determinada clase, dado que funciones con el mismo nombre, pertenecientes a clases diferentes, son distintas.

Para realizar lo mencionado se utiliza el operador membresía “::”.

Sintaxis :

```
tipo nombre_de_clase :: función ( parámetros )
{
    < CUERPO DE LA FUNCION >
}
```

A fin de invocar la función miembro desde el exterior (como así también a las variables miembro) se utiliza el operador miembro “.”, en forma similar su uso en estructuras.

Sintaxis :

nombre_de_objeto . función (argumentos)

PROG01.C PP

VERIFICACION DE ACCESIBILIDAD

Este ejemplo está destinado a comprobar las condiciones de accesibilidad de los miembros públicos y privados de una clase, y a familiarizarnos con la sintaxis y los conceptos vistos hasta el momento.

También se introducen dos nuevos elementos que se comentaran oportunamente.

Se declara una clase llamada CLASE que contiene miembros, tanto públicos como privados.

El campo DATO es una estructura privada de dos campos.

La función DOBLAR () tiene como objetivo duplicar el valor de los datos de la estructura. Nótese que ni DOBLAR () ni DATO son accesibles desde el exterior. Cualquier intento de acceso dará un error le compilación.

En la parte pública de la clase se encuentra el campo N, accesible desde el exterior, como así también las funciones MIRAR(), CARGAR() y DUPLICAR() que constituyen la puerta de entrada al objeto y pueden acceder a la parte privada del mismo.

Las funciones MIRAR() y CARGAR() se encargan de enviar los datos de la estructura a pantalla y cargar los campos de la estructura desde el exterior del objeto, mientras que la función DUPLICAR() se encarga de duplicar el valor de dichos datos, pero lo hace a través de la invocación de la función DOBLAR().

Como se mencionó anteriormente, se introdujeron un par de nuevos elementos al programa.

Uno de ellos es la doble barra " // " para indicar comentarios. Este símbolo tiene validez para una sola línea.

La forma tradicional de C para comentarios " /* */ " sigue siendo válida.

El otro elemento es el lugar de la declaración de la variable de control de lazo I. Esta no fue declarada al inicio del programa, sino justo antes de ser utilizada.

Esto no es permitido por el compilador Turbo C. En él las declaraciones deben realizarse antes de cualquier sentencia ejecutable o llamado a función.

Este punto no representa una gran ventaja, dado que al momento de compilar ya se sabe que se utilizará esta variable y se podría haber colocado su declaración en el sitio habitual.

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

class CLASE {
    struct ESTRUCTURA {
        int A ;
        float B ; } ;
    struct ESTRUCTURA DATO ; /* Dato privado */
    void DOBLAR (void) ; /* Función privada */
public:
    int N ; /* Dato público */
    void MIRAR ( void ) ; /* Funciones públicas */
    void CARGAR ( int , float ) ;
    void DUPLICAR ( void ) ;
};

void CLASE::CARGAR ( int X , float Y )
{
    DATO.A = X ;
    DATO.B = Y ;
}

void CLASE::MIRAR ( )
{
    printf("\n\nMostrando valores");
    printf("\n%6d%10f",DATO.A,DATO.B);
}
```

```

void CLASE::DOBLAR ( )
{
    DATO.A *= 2 ;
    DATO.B *= 2 ;
}

void CLASE::DUPLICAR ( )
{
    DOBLAR () ;
}

void main ( void )
{
    CLASE OBJETO ; /* Declaracion del objeto */
    int ENTERO ;
    float REAL ;
    clrscr();
    // OBJETO.DATO.A = 2 ; /* 1 */
    // Da que CLASE::DATO no es accesible en funcion main

    printf("\n\nIngrese el numero de repeticiones\n\n");
    sc.inf("%d",&(OBJETO.N)) ; /* Se accede porque es publico */

    p.intf("\n\nIngrese un valor entero y uno real\n\n");
    sc.anf("%d %f",&ENTERO,&REAL);

    OBJETO.CARGAR(ENTERO,REAL);

    // OBJETO.DOBLEAR () ; /* 2 */
    // Da que CLASE::DOBLAR() no es accesible desde funcion main.

    OBJETO.DUPLICAR () ;

    for (int I = 0 ; I < OBJETO.N ; I++ )
        OBJETO.MIRAR();

    getch();
}

```

Se sugiere que se ejecute este programa con (y sin) la inclusión de las líneas (1) y (2) a fin de comprobar las restricciones de acceso mencionadas.

FLUJOS

Como ya se vio en C, los flujos se utilizan para dar un formato uniforme a las distintas entradas y salidas.

En C++ se dispone de un formato muy simple :

Flujo de entrada >> Variable

Flujo de salida << Constantes o variables

Nótese el uso de los operadores " << " y " >> ". Estos siguen teniendo validez como operadores de desplazamiento a nivel de bit, pero además, permiten la transferencia de datos hacia o desde los flujos.

Esta situación representa un ejemplo de "sobrecarga" de operadores, que se comentará mas adelante.

El funcionamiento de estos operadores relacionados a los flujos esta sustentado en una jerarquía de clases predefinidas cuyas declaraciones se encuentran en las cabeceras **iostream.h** y **stream.h** lo que hace necesaria su inclusión.

En C++ el flujo de entrada standard (teclado) recibe el nombre de **cin** , mientras que el flujo de salida standard (pantalla) recibe el nombre de **cout** .

De esta forma, una secuencia de entrada y salida simple podria escribirse como :

```
int A ;  
cin >> A ;  
cout << A ;
```

PROG02.CPP

FLUJOS STANDARD

Este es un programa sencillo que pretende mostrar el comportamiento de los flujos de entrada y salida, como asi tambien su manejo.

Es de notar la ausencia de caracteres de formato (%). El formato de los datos que ingresan o egresan esta dado por el tipo mismo del dato.

```
#include <iostream.h>  
#include <conio.h>  
void main ( )  
{  
    clrscr ( ) ;  
  
    int I ;  
    cout << "Ingrese un entero \n" ;  
    cin >> I ;
```

```

float F ;
cout << "Ingrese un flotante \n" ;
cin >> F ;

char C ;
cout << "Ingrese un caracter \n" ;
cin >> C ;

long L ;
cout << "Ingrese un long \n" ;
cin >> L ;

char palabra[20] ;
cout << "Ingrese una palabra \n" ;
cin >> palabra ;

cout << "\n Entero = " << I << "\n Flotante = " << F \
<< "\n Caracter = " << C << "\n Long = " << L ;
cout << "\n Palabra = " << palabra ;

getch() ;
}

```

MANIPULADORES

Si bien los flujos de salida admiten los caracteres de formato de C (\n , \t , \r , etc.) , el C++ ofrece el uso de **manipuladores** para formatear la salida.

Los manipuladores son funciones que se insertan en las corrientes con el objeto de darles formato. Su uso requiere la inclusión de la cabecera **iomanip.h**.

Entre estos manipuladores encontramos :

dec , hex y oct	Establecen la base numérica
endl	End Line (fin de línea). Equivale a '\n'
ends	End of String. Equivale a '\0' .
flush	Fuerza el flujo de salida. Tiene el efecto de fflush().

PROG03.CPP

USO DE MANIPULADORES

Este programa ilustra el uso de manipuladores permitiendo observar sus efectos. Se sugiere que el lector compruebe las siguientes situaciones :

- ♦ Cuando se manda un manipulador de base numérica al flujo, este modo queda seteado hasta que se lo cambie por otro.
- ♦ El formato de base numérica solo afecta a los tipos *int* y *long*, y no a los *float* y *double*.

```

#include <iostream.h>
#include <conio.h>
#include <stdio.h>
#include <iomanip.h>

void main ( )
{
    clrscr ( );

    int OCTAL ;
    cout << "Ingrese un entero octal\n" ;
    cin >> oct >> OCTAL ;

    int I ;
    cout << "Ingrese un entero decimal\n" ;
    cin >> dec >> I ;

    int H ;
    cout << "Ingrese un entero hexadecimal\n" ;
    cin >> hex >> H ;

    float F ;
    cout << "Ingrese un flotante \n" ;
    cin >> F ;

    char C ;
    cout << "Ingrese un caracter \n" ;
    cin >> C ;

    long L ;
    cout << "Ingrese un long \n" ;
    cin >> L ;

    cout << "\n\nOCTAL\n" ;
    cout << oct << "\n Hexadecimal = " << H \
        << "\n Octal = " << OCTAL << "\n Decimal = " \
        << I << "\n Flotante = " << F \
        << "\n Caracter = " << C << "\n Long = " << L ;
    getch();

    cout << "\n\nDECIMAL\n" ;
    cout << dec << "\n Hexadecimal = " << H \
        << "\n Octal = " << OCTAL << "\n Decimal = " \
        << I << "\n Flotante = " << F \
        << "\n Caracter = " << C << "\n Long = " << L ;
    getch();

    cout << "\n\nHEXADECIMAL\n" ;
    cout << hex << "\n Hexadecimal = " << H \
        << "\n Octal = " << OCTAL << "\n Decimal = " \
        << I << "\n Flotante = " << F \
        << "\n Caracter = " << C << "\n Long = " << L ;
    getch();
}

```

C++ dispone de una serie de funciones manipuladores con parámetros que expondremos a continuación.

La sintaxis usual es :

```
cout << manipulador(argumento) << elemento de salida
```

Base de numeración : **setbase(int n)**

Donde **n** puede tomar los valores 0 (default), 8 (octal), 10 (decimal) o 16 (hexadecimal). Su funcionamiento es similar a los manipuladores **dec**, **oct** y **hex**. Si se asigna un valor de **n** no previsto, se ignora la función.

Ancho de los campos : **setw(int ancho)**

La función "set width" establece el ancho el ancho del campo a usar en un flujo de salida. Si el largo del dato de salida es menor al ancho especificado, se rellena con caracteres en blanco (a menos que se especifique lo contrario) y se coloca el dato justificado a la derecha.

El manipulador **setw()** solo afecta la siguiente corriente de datos, como se puede apreciar a continuación :

```
cout << 12345678901234567890 << endl ;
cout << 123 << setw(8) << 456 << 789 << endl ;
cout << 123 << setw(8) << 456 << setw(5) << 789 ;
```

da como salida :

```
12345678901234567890
123      456789
123      456 789
```

También puede establecerse el ancho de campo (válido hasta el próximo cambio) mediante la función miembro **width()**.

Esta función se invoca a través del objeto relacionado.

```
cout.width( 8 );
```

Esta función también es válida para limitar el ancho de los campos de entrada.

Relleno de los campos : setfill (int caracter)

Establece cual será el carácter de relleno en caso de haberse establecido un determinado ancho de campo. El relleno no tiene sentido si no se estableció antes un ancho de campo.

Una vez que se establece el relleno, sigue siendo válido hasta que se lo cambie.

Al igual que en el caso anterior, se puede utilizar la función miembro `fill()`. Esta función establece el nuevo relleno, pero además retorna el carácter de relleno actual. Por lo tanto, se puede utilizar para restituir posteriormente el anterior relleno.

PROG04.CPP

ANCHOS Y RELLENOS

Este programa ingresa e imprime una matriz de strings. Muestra el uso de los manipuladores anteriormente descriptos para establecer un ancho de campo fijo y relleno con caracteres '.'.

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

void main ( )
{
    clrscr ( );
    char mat[3][4][10];
    for ( int I=0 ; I<3 ; I++ )
        for ( int J=0 ; J<4 ; J++ )
            cout << "\nIngrese el nombre " << I << " " << J ;
            cin >> mat[I][J] ;
    cout << setfill ( '.' ) ;
    for ( I=0 ; I<3 ; I++ )
        cout << endl << endl << "\t" ;
        for ( J=0 ; J<4 ; J++ )
            cout << " " << setw(12) << mat[I][J] ;
    cout << "\n\n\n\n" ;
    getch();
}
```

Cantidad de decimales en números reales : setprecision (int num_dec)

Mediante este manipulador se establece la cantidad máxima de decimales para los números flotantes. Esta cantidad permanece hasta que se cambia nuevamente.

Flags

La clase `ios` posee una serie de propiedades (en forma de flags). Los flujos de entrada y salida son objetos de esa clase, y es posible setear o resetear esos flags a través de las funciones manipuladoras :

- ◆ `setiosflags (long flag)`
- ◆ `resetiosflags (long flag)`

A continuación se listan estos flags :

• <code>skipws</code>	Saltea espacios en blanco en operaciones de entrada.
• <code>left</code>	Justifica la salida a la izquierda en un campo.
• <code>right</code>	Justifica la salida a la derecha en un campo.
• <code>internal</code>	Rellena el campo después del signo o el indicador de base.
• <code>dec</code>	Activa conversión decimal.
• <code>oct</code>	Activa conversión octal.
• <code>hex</code>	Activa conversión hexadecimal.
• <code>showbase</code>	Visualiza el indicador de base numérica.
• <code>showpoint</code>	Visualiza el punto decimal en flotantes.
• <code>uppercase</code>	Visualiza valores hexadecimales en mayúscula.
• <code>showpos</code>	Precede a los enteros positivos del signo '+'.
• <code>scientific</code>	Establece notación científica en los flotantes.
• <code>fixed</code>	Establece notación de punto fijo para los flotantes.
• <code>unitbuf</code>	Vacia los buffers después de cada escritura.
• <code>stdio</code>	idem. sobre <code>stdout</code> y <code>stderr</code> .

El formato de uso incluye el operador de membresía como se muestra :

```
flujo << setiosflags(ios::flag)
```

Por ejemplo :

```
cout << setiosflags(ios::scientific) << .346.232323 << endl ;
```

Dado que la operación de establecer un flag, en definitiva equivale a colocar un '1' en un bit de una variable long, se pueden combinar estas operaciones mediante operadores OR, como se muestra :

```
flujo << setiosflags(ios::flag1|ios::flag2|ios::flag3) ;
```

PROG05.CPP

USO DE MANIPULADORES

En este programa se ingresa una matriz de flotantes y se la imprime fijando 2 decimales, utilizando conjuntamente `setprecision(2)` y `showpoint`.
 Asimismo se llenan los espacios en blanco con el carácter ‘~’ utilizando `setfill()`.

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

void main ( )
{
    clrscr ( );
    float mat[3][4];
    for ( int I=0 ; I<3 ; I++ )
        for ( int J=0 ; J<4 ; J++ ) {
            cout << "\nIngrese el numero " << I << " " << J ;
            cin >> mat[I][J] ;
        }
    cout << setfill ( '~' ) ;
    cout << setprecision ( 2 ) ;
    cout << setiosflags ( ios::showpoint ) ;
    for ( I=0 ; I<3 ; I++ ) {
        cout << endl << endl << "\t" ;
        for ( J=0 ; J<4 ; J++ )
            cout << " " << setw(12) << mat[I][J] ;
    }
    cout << "\n\n\n\n" ;
    getch();
}
```

PROG06.CPP

USO DEL FLAG `showbase`

Al setear el flag `showbase` el formato de los datos enteros es el que equivale a los números decimales, octales y hexadecimales como se indica:

123	Decimal
0123	Octal
0X123	Hexadecimal

```

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

void main ( )
{
    clrscr ( ) ;

    int NUM ;
    NUM = 25 ;

    cout << setiosflags ( ios::showbase ) ;

    cout << "\n\nHexadecimal " << hex << NUM ;
    cout << "\n\nDecimal " << dec << NUM ;
    cout << "\n\nOctal " << oct << NUM ;

    cout << "\n\n\n\n" ;
    getch() :
}

```

EJERCICIO PROPUESTO

Se propone al lector la resolución del siguiente problema, en donde aplique los conocimientos de clases y objetos recientemente adquiridos.

Se desea crear y manejar una pila compacta de 10 números enteros. El vector que de soporte a la pila deberá ser un miembro privado de una clase, al que solo se podrá acceder mediante las funciones miembro **PUSH()** y **PULL()**.
El puntero (stack pointer) que permite recorrer el vector, también será un miembro privado de la clase.

PROG07.CPP

MANEJO DE UNA PILA COMPACTA

Dado que el acceso a la pila está restringido a las funciones miembro **PUSH()** y **PULL()**, esta asegurado de esta forma, que no se producirá un desbalance de la misma, ni se excederán sus límites. Asimismo, el manejo del puntero **P** que recorre la pila, está vedado al exterior y solo puede manejarse a través de las funciones miembro.

Sin embargo, es necesario inicializar la pila haciendo que **P** apunte al inicio de la misma. Esta situación se resuelve mediante la función miembro **Inicializar()** que se invoca al crear el objeto “pila”.

No es posible realizar esta inicialización en la clase, pues en ella no existen variables. Es necesario implementarla luego de crear al objeto.

La función **Inicializar()** es un punto débil del encapsulamiento de la pila del que nos ocuparemos en la siguiente sección.

```

#include <conio.h>
#include <iostream.h>

#define MAX 10
#define MENU getch(); clrscr(); cout << "\n\n\n\t\t\t\t0\tsALIR"; \
    cout << "\n\n\n\t\t\t\t1\tpONER\n\n\n\t\t\t\t2\tsACAR\n\n\n";
}

class PILA {
    int VEC [ MAX ] ;           /* Datos pivados */
    int * P ;
public:
    void Inicializar ( void ) ; /* Funciones publicas */
    void PUSH ( int ) ;
    int PULL ( void ) ;
} ;

PILA Pila ;

void poner ( ) ;
void sacar ( ) ;

void main ( void )
{
    int SELEC = 1 ;

    Pila.Inicializar ( ) ;

    clrscr();

    while( SELEC ) {
        MENU ;
        cin >> SELEC ;
        switch ( SELEC ) {
            case 1 : poner ( ) ; break ;
            case 2 : sacar ( ) ; break ;
        }
    }
}

void poner ( )
{
    int DATO ;
    cout << "\nIngrese un entero      " ;
    cin >> DATO ;
    Pila PUSH ( DATO ) ;
}

void sacar ( )
{
    cout << "\nDato = " << Pila.PULL ( ) ;
}

```

```

void PILA::Inicializar ( )
{
    P = VEC ;
}

void PILA::PUSH ( int DATO )
{
    if ( P >= VEC + MAX )
        cout << "\nDesborde de pila.\nDato perdido.\n" ;
    else
    {
        *P = DATO ;
        P++ ;
    }
}

int PILA::PULL ( )
{
    if ( P <= VEC ) {
        cout << "\nPila vacia.\nDato nulo.\n" ;
        return 0 ;
    }
    P-- ;
    return *P ;
}

```

CONSTRUCTORES Y DESTRUCTORES

Se observó en la sección anterior que fue necesario introducir la función miembro **Iniciar()** a fin de apuntar el puntero de pila **P** al inicio de la misma. Esta función es pública a fin de invocarla desde el exterior. Una llamada irresponsable de esta función durante el programa, daría como resultado la reinicialización de la pila con la consecuente perdida de los datos que contiene.

Para solucionar este problema, sería necesario contar con una función que cumpla con las siguientes características :

- No sea accesible desde el exterior.
- Se invoque automáticamente al crear el objeto.
- Actúe solo una vez (justamente en la oportunidad anterior).

Esta función recibe el nombre de **CONSTRUCTOR**.

CONSTRUCTOR

Es una función especial, miembro de una clase y con el mismo nombre que ella, que solo actúa al crearse el objeto.

Un constructor tiene las siguientes características:

- Tiene el mismo nombre de la clase a la que pertenece.
 - No tiene tipo retornado (no es void, simplemente , no tiene).
 - Debe estar ubicada en la parte publica de la clase.
 - No puede ser invocado desde el exterior.

Ejemplo :

```
class CLASE {
    int A ;
    float B ;
    public :
    CLASE ( ) ; /* Constructor */
    void funcion ( ) ;
}

CLASE::CLASE ( )
{
    A = 0 ;
    B = 0 ;
}
```

Nótese que el constructor está ubicado en la parte pública de la clase, se llama igual que ella y no tiene tipo retornado. Tampoco lo tiene al definirla.

A continuación se repite el ejemplo anterior , pero inicializando la pila a través de un constructor.

PROG03.CPP

PROG08.CPP

MANEJO DE UNA PILA COMPACTA USANDO UN CONSTRUCTOR.

```

PILA Pila ;

void poner ( ) ;
void sacar ( ) ;

void main ( void )
{
    int SELEC = 1 ;
    clrscr();
    while( SELEC ) {
        MENU ;
        cin >> SELEC ;
        switch ( SELEC ) {
            case 1 : poner ( ) ; break ;
            case 2 : sacar ( ) ; break ;
        }
    }
}

void poner ( )
{
    int DATO ;
    cout << "\nIngrese un entero " ;
    cin >> DATO ;
    Pila.PUSH ( DATO ) ;
}

void sacar ( )
{
    cout << "\nDato = " << Pila.PULL ( );
}

PILA::PILA ( )
{
    cout << "\nConstructor en accion.\n" ;
    P = VEC ;
}

void PILA::PUSH ( int DATO )
{
    if ( P >= VEC + MAX )
        cout << "\nDesborde de pila.\nDato perdido.\n" ;
    else {
        *P = DATO ;
        P++ ;
    }
}

```

```

int PILA::PULL ( )
{
    if ( P <= VEC ) {
        cout << "\nPila vacia.\nDato nulo.\n" ;
        return 0 ;
    }
    P-- ;
    return *P ;
}

```

DESTRUCTOR

Es similar al anterior pero se invoca al destruir el objeto.

DESTRUCTOR

Es una función especial, miembro de una clase y con el mismo nombre que ella precedido de ‘~’, que solo actúa al destruirse el objeto.

Recordemos que un objeto local a un bloque o a una función, se destruye en el momento en que se abandona el bloque o función donde fue creado.

Del mismo modo, un objeto dinámico, se destruye al liberar la memoria que ocupa.

PROG09.CPP

CONSTRUCTORES Y DESTRUCTORES.

Este programa permite comprobar la oportunidad de acción de un constructor y un destructor.

En él se declara un objeto local a una función a fin de obtener los eventos de creación y destrucción (al salir de la función) del objeto, y por lo tanto se da oportunidad al accionar del constructor y del destructor.

En el programa y las funciones, simplemente se envían leyendas a la pantalla que permitan seguir el accionar del mismo.

Se tomaron los nombres de los dioses del hinduismo Brahma (el constructor) y Shiva (el destructor) para ilustrar los mensajes, dejando a Vishnu (el protector) para otra oportunidad.

```

// Clases y objetos
// Constructores y destructores.

#include <conio.h>
#include <iostream.h>

```

```

class CLASE {
public:
    CLASE ( void ) ;
    ~CLASE ( void ) ; /* Constructor */
                           /* Destructor */

void funcion ( void ) ;

void main ( void )
{
    clrscr();
    cout << "\nPrograma principal.\n";
    getch();

    funcion ( );
    cout << "\nPrograma principal despues de la funcion.\n";
}

void funcion ( )
{
    CLASE OBJETO;
    cout << "\nDentro de la Funcion\n";
    getch();
}

/* Constructor */
CLASE::CLASE ( )
{
    cout << "\n\t\tBRAHMA es el CONSTRUCTOR\n";
    getch();
}

/* Destructor */
CLASE::~CLASE ( )
{
    cout << "\n\t\tSHIVA es el DESTRUCTOR\n";
    getch();
}

```

El resultado de la ejecución es como se muestra :

```

Programa principal.
        BRAHMA es el CONSTRUCTOR
Dentro de la Funcion.
        SHIVA es el DESTRUCTOR
Programa principal despues de la funcion.

```

Esto indica que las acciones del constructor y del destructor enmarcan la acción de la función, siendo el momento de su accionar el descripto.

CONSTRUCTORES PARAMETRIZADOS

Como se mostró anteriormente, los constructores inicializan algunas variables miembro del objeto.

En algunas ocasiones, se desconoce a priori, el valor de inicialización, o bien, este puede variar de ocasión en ocasión, siendo el usuario quien desee establecerlo en cada caso.

Contemplando este problema, existe la posibilidad de transferir al constructor argumentos de inicialización, de manera que estos valores estén determinados por el usuario. Dado que se transfieren argumentos, la función constructora deberá disponer de parámetros para recibirlas. Se les denomina entonces, *constructores parametrizados*.

Sintaxis de declaración de clase :

```
class nom_clase {  
    . . .  
    public :  
        nom_clase ( tipo parámetros ) ;  
    . . .  
}
```

Sintaxis de declaración del constructor :

```
nom_clase::nom_clase (tipo parámetros)  
{  
    < cuerpo >  
}
```

Sintaxis de declaración del objeto :

```
nom_clase nom_objeto (argumentos) ;
```

PROG10.CPP CONSTRUCTORES PARAMETRIZADOS.

Este ejemplo permite comprobar el funcionamiento de los constructores parametrizados. En el se declara una clase que tiene un string privado. Se permite el ingreso de un string de teclado y se lo asigna al objeto en el momento de su instancia.

La función pública del objeto mostrar() permite verificar que se realizó la inicialización prevista.

```

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

class CLASE {
    char nombre [ 20 ];
public:
    CLASE ( char * s ); /* Constructor */
    void mostrar ( void );
};

void main ( void )
{
    clrscr();
    char VEC [ 20 ];
    cout << "\nIngrese un nombre para inicializar el objeto.\n";
    cin >> VEC;
    CLASE OBJETO ( VEC );
    OBJETO.mostrar (); /* Declaracion del objeto */
}

void CLASE::mostrar ( )
{
    cout << "\n\nNombre de inicializacion : ";
    cout << nombre << "\n\n";
    getch();
}

/* Constructor */
CLASE::CLASE ( char * s )
{
    strcpy ( nombre , s );
}

```

CONSTRUCTORES PARAMETRIZADOS CON DEFAULT

Avanzando un poco mas sobre la conveniencia de asignación del valor inicial, podría ocurrir que se diera una situación en la cual el usuario pueda desechar elegir la inicialización de las propiedades del objeto (variables del mismo), o no desecharlo. Y en este ultimo caso, que dichas variables adopten valores predeterminados (incluso desconocidos por el usuario).

Los valores por omisión se indican en la declaración de la clase de la siguiente forma :

```

class CLASE {
    .....
    CLASE ( tipo parámetro = valor por omisión );
    .....
}

```

PROG11.CPP

CONSTRUCTORES PARAMETRIZADOS CON DEFAULT.

En este ejemplo se instancian dos objetos. A uno de ellos se le entregan argumentos y al otro, no.

Luego se comprueba el contenido de cada uno, verificando el valor por default.

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

class CLASE {
    char nombre [ 20 ] ;
public:
    CLASE ( char * S = "Filiberto" ) ; /* Constructor */
    void mostrar ( void ) ;
};

void main ( void )
{
    clrscr();
    char VEC [ 20 ] ;
    cout << "\nIngrese un nombre para inicializar el objeto.\n" ;
    cin >> VEC ;

    CLASE OBJETO ( VEC ) ;           /* Declaracion del objeto */
    OBJETO.mostrar ( ) ;

    CLASE OTRO_OBJETO ;            /* Declaracion del objeto */
    cout << "\n\nInicializacion por default.\n\n" ;
    OTRO_OBJETO.mostrar ( ) ;

    getch ( ) ;
}

void CLASE::mostrar ( )
{
    cout << "\n\nNombre de inicializacion : " ;
    cout << nombre << "\n\n" ;
}

/* Constructor */
CLASE::CLASE ( char * S )
{
    strcpy ( nombre , S ) ;
}
```

Es importante destacar que el valor por default debe indicarse en la declaración de la clase, y no en la función constructora.

Es necesario indicarlo en la declaración de la clase pues caso contrario, cuando se instancia un objeto sin argumento, el tipo del constructor no coincide con el que se específico en la clase, y se produce un error de "desapareamiento de tipos".

Si se indica el valor por omisión, además, en el cuerpo del constructor, se genera un error de compilación debido a que el valor por default está indicado dos veces.

ARGUMENTOS POR DEFECTO DE FUNCIONES

Hemos visto como es posible asignar argumentos por defecto a las funciones constructores. De manera similar, es posible hacerlo con cualquier función.

En este caso, la indicación del valor default se realiza en la declaración del cuerpo de la función.

Sintaxis :

```
tipo nom_fun(tipo parámetro = valor por omisión)
{
    < cuerpo >
}
```

Al invocar la función tenemos entonces dos alternativas :

- nom_fun (argumento);
- nom_fun ();

En el segundo caso, se utiliza el valor por omisión.

Se propone al lector que desarrolle un programa que permita verificar lo expuesto.

MIEMBROS ESTÁTICOS DE UNA CLASE

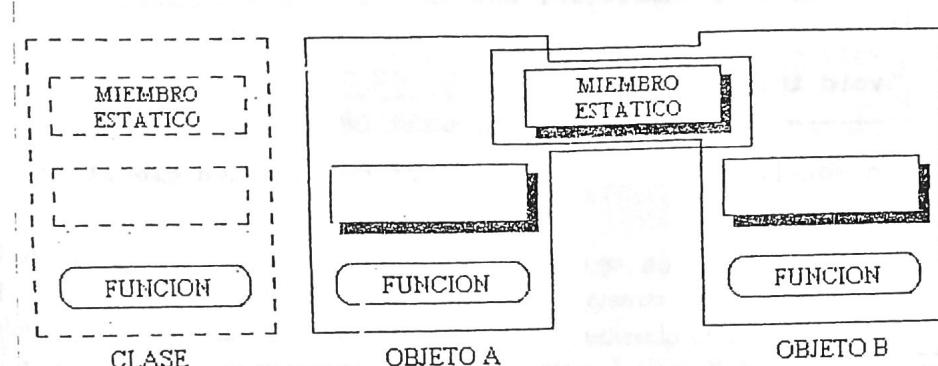
Cuando una variable es declarada como estática (utilizando el modificador *static*), se aloja en el área de memoria estática. Esta es el área donde se encuentran las variables y constantes globales.

Es posible utilizar el modificador static con los miembros de una clase en el momento de su declaración.

Dado que los campos estáticos del mismo nombre comparten memoria en el área estática, se trata de hecho, de la misma variable.

De esta forma, los mismos campos de distintos objetos de la misma clase, son en realidad la misma variable.

De esta manera se crea un vínculo entre objetos de la misma clase, aun si el miembro compartido es privado.



En algunos compiladores (como el Turbo C++) puede ser necesario además declarar globalmente (fuera de la clase) las variables estáticas de la misma, indicando a qué clase pertenecen mediante el operador de membresía.

Ejemplo :

```
int CLASE :: VARIABLE ; /* Declaración global */
```

PROG12.CPP MIEMBROS ESTATICOS

En este ejemplo se declaran dos objetos de una clase que tiene un miembro privado estático.

Se observa que la escritura en ese miembro en uno de los objetos, tiene el mismo efecto sobre el mismo campo estático del otro objeto.

Para darle generalidad a la comprobación, uno de los objetos se declaró localmente a una función.

Notese que la variable estática fue declarada globalmente fuera de la clase y después de haber sido declarada la misma. Esto último es necesario dado que se debe invocar el nombre de la clase en la declaración global, y por lo tanto, esta (la clase) ya debe estar definida.

```

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

class CLASE {
    static char nombre[10];
public:
    void escribir ( char * ) ;
    void leer ( void ) ;
};

char CLASE::nombre[10]; /* Declaracion global */

void funcion ( void ) ;

void main ( void )
{
    clrscr();
    CLASE OBJ;
    cout << "\nPrograma principal.\nIngrese un string.\n";
    char pal[10];
    cin >> pal;

    OBJ.escribir(pal);

    funcion();

    cout << "\nPrograma principal despues de la funcion.\n";
    OBJ.leer();
    getch();
}

void funcion ( )
{
    char STR[10];
    CLASE OBJETO;
    cout << "\nFuncion\n";
    (OBJETO.leer());

    cout << "\n \nDentro de la funcion.\nIngrese un nombre ";
    cin >> STR;
    (OBJETO.escribir(STR));
}

void CLASE::escribir ( char * s )
{
    strcpy ( nombre , s );
}

void CLASE::leer ( )
{
    cout << "\n" << nombre;
}

```

FUNCIONES AMIGAS

Son funciones que, sin formar parte de una clase, pueden acceder a todos los miembros de la misma, aun a los miembros privados.

Para tener derecho a este acceso, la función debe ser declarada como “amiga” dentro de la clase. Para ello se utiliza la palabra reservada *friend*.

Sintaxis:

```
class nom_clase {  
    .....  
    friend tipo nom_funcion (tipo parametros);  
    .....  
}
```

Hay que tener en claro que la “*amistad*” se entrega, no es algo que se toma. No es valido decir “*soy amigo del anfitrión, déjenme pasar a la fiesta*”, porque cualquier desconocido podría argumentar lo mismo, produciendo una violación de seguridad. Lo que corresponde es que el anfitrión (la clase) diga : “*este (la función) es mi amigo y por lo tanto puede pasar a la fiesta* (acceder a los miembros de la clase)”.

Dado que la función amiga es externa a la clase, nada impide que sea amiga de dos o más clases, transformándose en un nexo entre ellas.

PROG13.CPP

FUNCION AMIGA

En este ejemplo se presenta una modificación del programa de manejo de una pila compacta.

Se agrega una función que permite observar el contenido de la pila. Esta función es “ilegal” dado que no respecta el principio de lectura destructiva de la misma.

Por esta razón no se la incluye como miembro de la clase, pero se permite su uso. Para ello se la declara con o función amiga de la clase.

Por otro lado, se incluyó la variable privada RESTO en la clase, destinada a contener la cantidad de datos presentes en la pila.

Obsérvese que RESTO es inicializada en cero por el constructor

```

class PILA {
    int VEC [ MAX ] ;           /* Datos pivados */
    int * p ;
    int RESTO ;
    friend void MIRAR ( PILA ) ;
public:
    PILA ( void ) ;           /* Constructor */
    void PUSH ( int ) ;
    int PULL ( void ) ;
} ;

PILA Pila ;

void poner ( ) ;
void sacar ( ) ;
void MIRAR ( PILA ) ;

void main ( void )
{
    int SELEC = 1 ;

    clrscr();
    while( SELEC ) {
        MENU ;
        cin >> SELEC ;
        switch ( SELEC ) {
            case 1 : poner ( ) ; break ;
            case 2 : sacar ( ) ; break ;
            case 3 : MIRAR ( Pila ) ; break ;
        }
    }
}

void poner ( )
{
    int DATO ;
    cout << "\nIngrese un entero " ;
    cin >> DATO ;
    Pila.PUSH ( DATO ) ;
}

void sacar ( )
{
    cout << "\ndato = " << Pila.PULL ( ) ;
    getch() ;
}

void MIRAR ( PILA PP )
{
    int * p , CANT ;
    p = PP.VEC ;
    clrscr();
    for ( CANT = 0 ; CANT < PP.RESTO ; CANT++ , p++ )
        cout << "\n" << *p ;
    getch();
}

```

```

PILA::PILA ( )
{
    cout << "\nConstructor en accion.\n" ;
    P = VEC ;
    RESTO = 0 ;
}

void PILA::PUSH ( int DATO )
{
    if ( P >= VEC + MAX )
        cout << "\nDesborde de pila.\nDato perdido.\n" ;
    else
    {
        *P = DATO ;
        P++ ;
        RESTO++ ;
    }
}

int PILA::PULL ( )
{
    if ( P <= VEC ) {
        cout << "\nPila vacia.\nDato nulo.\n" ;
        return 0 ;
    }
    P-- ;
    RESTO-- ;
    return *P ;
}

```

PROG|4.CPP

FUNCION AMIGA DE CLASES DIFERENTES

En este caso, la función **mismo()** es compartida, como amiga, por las clases UNO y DOS. Nótese que la función **mismo()** recibe como argumentos, objetos de ambas clases.

Cuando se declara la clase DOS, es necesario hacer referencia a la función amiga **mismo()**, que recibe como argumento un objeto de clase UNO. Pero esta clase aun no fue declarada, y por lo tanto es desconocida.

Esto hace necesario avisar previamente que la clase UNO será usada, mediante la linea "class UNO ;".

Convieneclarar que la situación anterior no se soluciona invirtiendo el orden de declaración de las clases.

```

#include <conio.h>
#include <iostream.h>

class UNO ;

class DOS {
    int VAR_DOS ;
    friend void mismo ( UNO , DOS ) ;
public:
    DOS ( void ) ; /* Constructor */
    void SET ( int ) ;
} ;

class UNO {
    int VAR_UNO ;
    friend void mismo ( UNO , DOS ) ;
public:
    UNO ( void ) ; /* Constructor */
    void SET ( int ) ;
} ;

void mismo ( UNO , DOS ) ;

void main ( void )
{
    int A=1 , B=2 ;

    UNO uno ;
    DOS dos ;

    clrscr();

    while ( (A!='Q') && (B!='Q') ) {
        A = getch() ;
        cout << "\n\n\n" << A << "\n" ;
        uno.SET ( A ) ;
        B = getch() ;
        cout << "\n" << B << "\n" ;
        dos.SET ( B ) ;
        mismo ( uno , dos ) ;
    }
}

UNO::UNO ( )
{
    VAR_UNO = 1 ;
}

DOS::DOS ( )
{
    VAR_DOS = 2 ;
}

void UNO::SET ( int VALOR )
{
    VAR_UNO = VALOR ;
}

```

```

void DOS::SET ( int VALOR )
{
    VAR_DOS = VALOR ;
}

/* Funcion amiga */
void mismo ( UNO one , DOS two )
{
    if ( one.VAR_UNO == two.VAR_DOS )
        cout << "\n\nSON IGUALES \n\n" ;
    else
        cout << "\n\nSON DISTINTOS \n\n" ;
}

```

FUNCIONES MIEMBRO INLINE

El modificador **inline** solicita al compilador que se inserte el cuerpo de la función en el programa que la invoca. De esta forma, la función es tratada como una macro, con las ventajas y desventajas que esto implica.

PROG15.CPP

FUNCION INLINE

En este ejemplo se declaran dos funciones, **cuad1()** y **cuad2()**. Ambas son idénticas en cuanto al desarrollo. Retornan el cuadrado del argumento que reciben.

La diferencia entre ambas estriba en que **cuad2()** se declara como función **inline**.

```

#include <conio.h>
#include <iostream.h>

class CLASE {
public:
    void cuad1 ( int ) ;
    inline void cuad2 ( int ) ;
};

void main ( void )
{
    int A=3 , B=2 ;
    CLASE OBJETO ;
    clrscr () ;
    OBJETO.cuad1 ( 2+A + B ) ;
    OBJETO cuad2 ( 2*A + B ) ;
}

```

```

void CLASE::cuad1 ( int x )
{
    cout << "\n\n" << x*x ;
}

void CLASE::cuad2 ( int x )
{
    cout << "\n\n" << x*x ;
}

```

La ejecución del programa arrojó el mismo resultado en ambos casos.
 Se propone al lector que verifique este comportamiento y determine si la función declarada **inline** tiene el comportamiento de una **macro**.
 Para esto se sugiere construir una verdadera macro y chequear su comportamiento.

Problema propuesto : Realice un programa que permita verificar el comportamiento **inline** de una función, comparando el tamaño del programa resultante.

TRANSFERENCIA POR REFERENCIA

Cuando programamos en C, consideramos dos maneras de transferir argumentos a funciones, a las que llamaremos *transferencia por valor* y *transferencia por referencia*. En esta última, lo que en realidad se realiza, es transferir la dirección del argumento, y recibirla en un parámetro de tipo puntero.

De esta forma, la transferencia realizada es en realidad, una *transferencia por valor* de la dirección del argumento, o una "*transferencia por referencia simulada*".

La programación en C++ posibilita la creación y uso de parámetros "*de referencia*". Este tipo de variables constituyen un alias o referencia a una variable "*original*".

Para referencia, una variable se utiliza el operador "&". Podemos asumir que el significado de este operador (en esta aplicación) es "*referencia a ...*".

Este operador no pierde su significado en cuanto a su utilización en el ámbito de los punteros. En ese caso representa "*la dirección de ...*".

La situación descripta muestra un ejemplo de **sobrecarga** de operadores.

Sintaxis de declaración :

```
tipo & referencia ;
```

Por ejemplo :

```
int & REF ;
```

Es lo que significa que REF será un alias de alguna variable entera (que aún no ha sido asignada).

En el siguiente ejemplo se puede ver como la referencia hace mención a la variable original, la que es en definitiva modificada.

```
int VALOR = 1 ;
int & REF = VALOR ;
REF++ ;
cout << VALOR ;
```

El valor resultante de este código es 2.

Pasaje por referencia de argumentos a función :

Observemos esta sección de programa en donde se transfiere el argumento A, por referencia, a la función llamada *funcion()*.

```
int A = 4 ;
funcion ( A ) ;
. . . . .
int funcion ( int & REF ) ;
{
    REF += 2 ;
}
```

E. prototipo de la función toma la forma :

```
int funcion ( int & ) ;
```

Nótese que la transferencia desde el programa invocante, y el manejo dentro de la función son idénticos a los de la transferencia por valor.

Debido a esta razón, podría confundirse su uso. Esto es, suponer que se está realizando una transferencia por valor, donde no se trabaja sobre la variable original, y modificar accidentalmente el contenido de la misma.

Por lo tanto, se realiza la siguiente sugerencia :

Si el contenido de la variable original transferida a una función por referencia, no debe ser modificado dentro de la misma, se recomienda recibirla en un parámetro afectado por el modificador *const* de manera de hacerlo invariante dentro de la mencionada función.

PROG16.CPP

TRANSFERENCIA POR REFERENCIA

En este sencillo ejemplo de comprobación se muestra la transferencia por valor, por referencia simulada (por dirección) y por referencia, observándose los valores afectados por las diferentes funciones.

Prediga los valores a obtener y compruébelos ejecutando el programa.

```
#include <stdio.h>
/* Prueba de pasajes por referencia */
#include <conio.h>
#include <iostream.h>

void funcion_1 ( int ) ;
void funcion_2 ( int * ) ;
void funcion_3 ( int & ) ;

void main( )
{
    int VALOR_1 , VALOR_2 , VALOR_3 ;
    clrscr ( );
    VALOR_1 = 2 ;
    VALOR_2 = 3 ;
    VALOR_3 = 4 ;

    funcion_1 ( VALOR_1 ) ;
    funcion_2 ( &VALOR_2 ) ;
    funcion_3 ( VALOR_3 ) ;

    cout << "\n\n" << VALOR_1 << "\t" << VALOR_2 << "\t" << VALOR_3 ;
    getch();
}

void funcion_1 ( int VAR )
{
    VAR *= 2 ;
}

void funcion_2 ( int *P )
{
    *P *= 2 ;
}

void funcion_3 ( int &REF )
{
    REF *= 2 ;
}
```

Ventajas de la transferencia por referencia :

La variable "alias" es en realidad la misma que la original, por tal motivo, no hay verdadera transferencia de los contenidos. Esto implica que no se pierde tiempo de copia de valores, ni hay duplicación del espacio de memoria como en la transferencia por valor.

Si bien el manejo es parecido a la transferencia por valor, la transferencia por referencia esta intimamente ligada a la transferencia por dirección.

TRANSFERENCIA DE OBJETOS A FUNCIONES

Los objetos pueden ser considerados como estructuras con código asociado a ellas. Por lo tanto, pueden ser tratados de manera similar.

Dado que son variables como las estructuras, pueden ser copiados y apuntados. De esta forma pueden ser transferidos a funciones de diversas maneras.

PROG17.CPP

TRANSFERENCIA DE OBJETOS A FUNCIONES

Este ejemplo es una extensión del precedente, en el que se muestran las tres formas de transferencia mencionadas en la sección anterior.

Sin embargo, es conveniente prestar atención a varios puntos :

La clase creada tiene dos variables privadas A y B, un constructor destinado a inicializar los contenidos de dichas variables con los valores 2 y 3.5, una función de acceso público llamada DOBLE(), cuyo objetivo es duplicar los contenidos de las variables de su objeto, y finalmente se declara la función MIRAR() como "amiga". Esta última es la responsable de mostrarnos los contenidos de las variables de los distintos objetos.

Obsérvese las diferentes formas en que se invoca a la función DOBLE(), especialmente a través del puntero P.

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

class CLASE {
    int A;
    float B;
public :
    CLASE ( );
    void DOBLE ( );
    friend void MIRAR ( CLASE );
};
```

```

// Prototipos
CLASE funcion_1 ( CLASE ) ;
void funcion_2 ( CLASE * ) ;
void funcion_3 ( CLASE & ) ;

void main( )
{
    CLASE OBJETO1 , OBJETO2 , OBJETO3 ;
    clrscr ( ) ;

    cout << "\n\n\nValores antes de los llamados a funcion" ;
    MIRAR ( OBJETO1 ) ;
    MIRAR ( OBJETO2 ) ;
    MIRAR ( OBJETO3 ) ;

    OBJETO1 = funcion_1 ( OBJETO1 ) ;
    funcion_2 ( & OBJETO2 ) ;
    funcion_3 ( OBJETO3 ) ;

    cout << "\n\n\nPasaje por valor" ;
    MIRAR ( OBJETO1 ) ;

    cout << "\n\n\nPasaje por puntero" ;
    MIRAR ( OBJETO2 ) ;

    cout << "\n\n\nPasaje por referencia" ;
    MIRAR ( OBJETO3 ) ;
    getch() ;
}

CLASE funcion_1 ( CLASE OBJ )
{
    OBJ.DOBLE ( ) ;
    return OBJ ;
}

void funcion_2 ( CLASE *F )
{
    F->DOBIE ( ) ;
}

void funcion_3 ( CLASE & REF )
{
    REF.DOBLE ( ) ;
}

void MIRAR ( CLASE OBJ )
{
    cout << "\n\n" << OBJ.A << "\t\t" << OBJ.B ;
}

CLASE::CLASE ( )
{
    A = 2 ;
    B = 3.5 ;
}

```

```
void CLASE::DOBLE ( )
{
    A *= 2 ;
    B *= 2 ;
}
```

CLASES, ESTRUCTURAS Y UNIONES

Como vimos anteriormente, las clases son extensiones de las estructuras. Pero las estructuras mismas tienen en C++, un concepto más extenso que en C. Las estructuras pueden tener código asociado, al igual que las clases. De hecho la diferencia con estas es mínima.

En C++ se pueden definir para las estructuras, campos públicos y privados al igual que en las clases. La diferencia con estas últimas, es que el default de definición, es el acceso público.

Las uniones son estructuras cuyos campos comienzan en la misma posición de memoria y por lo tanto se superponen. Por lo demás, tienen, en C++ un comportamiento análogo a las estructuras.

PROG18.CPP

UNIONES EN C++

En este ejemplo se utiliza una unión para observar los cuatro bytes componentes de un flotante.

La unión tiene un campo flotante y un vector de 4 caracteres (bytes) para producir la superposición.

En el programa se carga al flotante con un valor y se observa el contenido de los 4 bytes del vector de caracteres. Estas dos tareas se realizan mediante las funciones miembro de la unión, `lee_float()` y `muestra_byte()`.

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

union UNION {
    void lee_float ( ) ;
    void muestra_byte ( ) ;
    float F ;
    unsigned char V[4] ;
};
```

```

void main ( void )
{
    UNION U ;

    clrscr();
    U.lee_float ( ) ;
    U.muestra_byte ( ) ;
    getch();
}

void UNION::lee_float ( )
{
    cout << "\nIngrese un flotante " ;
    cin >> F ;
}

void UNION::muestra_byte ( )
{
    cout << "\n\n\n" ;
    int I ;
    for ( I = 3 ; I >= 0 ; I-- )
        printf("%02X ", V[I]) ;
}

```

POLIMORFISMO Y SOBRECARGA

El polimorfismo es una característica de C++ mediante la cual un operador o una función se comportan de manera diferente según cual sea el tipo de sus operandos, o bien, el tipo o cantidad de sus argumentos.

Para ilustrar la situación anterior imaginemos una simple suma, llevada a cabo mediante el operador "+" o mediante la función SUMAR(), de las siguientes formas:

$$\begin{aligned} R &= A + B ; \\ R &= \text{SUMAR}(A, B) ; \end{aligned}$$

Podríamos esperar que si A vale 2, y B vale 3, obtengamos el valor 5 en R. Sin embargo, en ningún momento se dijo que A, B y R debían ser variables de tipo float o int. Bien podrían ser objetos de clases que representen vectores tridimensionales, o cualquier otro tipo.

Gracias al polimorfismo, podemos lograr que tanto el operador "+" como la función SUMAR() adecuen su comportamiento al tipo de operandos, o argumentos que se les presenten.

Esta característica también es denominada sobrecarga de operadores y sobrecarga de funciones.

SOBRECARGA DE FUNCIONES

El hecho de tener funciones que se comporten de manera diferente según la cantidad y/o tipo de sus argumentos, como lo indica el polimorfismo, puede ser interpretado de manera diferente.

Es posible tener, bajo ciertas condiciones, **diferentes funciones con el mismo nombre**. Debido a que son diferentes funciones, su comportamiento también será diferente.

Las "ciertas condiciones" mencionadas anteriormente establecen que las funciones con igual nombre o **funciones sobrecargadas** deben tener distinto tipo o diferente cantidad de argumentos.

No alcanza como condición que tengan distinto tipo retornado. C++ determinará cual función se activará según los argumentos que reciba.

PROG19.CPP

SOBRECARGA DE FUNCIONES

En este programa se construyen 3 funciones llamadas **cuadrado()**. Cualquiera de ellas retorna el cuadrado de lo que recibe.

La diferencia estriba en que una opera con enteros, otra con flotantes y la tercera con números complejos, implementados mediante un vector de 2 enteros (uno para la parte real, y otro para la imaginaria).

```
#include <conio.h>
#include <iostream.h>
typedef int* complex ;
complex cuadrado ( complex ) ;
int cuadrado ( int ) ;
float cuadrado ( float ) ;

void main ( )
{
    int N ;
    float F ;
    int COMPLEJO[2] ;

    clrscr() ;

    cout << "\n\nIngrese un entero    " ;
    cin >> N ;
    cout << "\n\nIngrese un flotante   " ;
    cin >> F ;
    cout << "\n\nIngrese un complejo ( parte real ( entera ) )    " ;
    cin >> COMPLEJO[0] ;
    cout << "\n\nIngrese un complejo (parte imaginaria(entera))    " ;
    cin >> COMPLEJO[1] ;

    cuadrado ( N ) ;
    cuadrado ( F ) ;
    cuadrado ( COMPLEJO ) ;
}
```

```

complex cuadrado ( complex v )
{
    int R[2];
    R[0] = v[0]*v[0] - v[1]*v[1];
    R[1] = 2*v[0]*v[1];
    char C = '+';
    int IMAG = R[1];
    if ( IMAG < 0 ) {
        C = '-';
        IMAG = -IMAG;
    }
    cout << "\n\nEl complejo cuadrado es      ";
    cout << R[0] << C << IMAG << "i \n\n";
    return R;
}

int cuadrado ( int A )
{
    A = A * A;
    cout << "\n\nEl entero cuadrado es " << A;
    return A;
}

float cuadrado ( float F )
{
    F = F * F;
    cout << "\n\nEl flotante cuadrado es " << F;
    return F;
}

```

SOBRECARGA DE OPERADORES

La sobrecarga de operadores es similar a la de funciones. En este caso, se crean funciones que se invocan al utilizar un operador simbólico.

Ya hemos tenido ejemplos de sobrecarga de operadores. Uno de ellos lo constituye “*”, que significa multiplicación cuando actúa como operador binario, tanto para operadores *int* como *char*, *float*, etc., mientras que como operador monario se aplica a punteros permitiendo acceder al valor apuntado por el puntero.

Función operador

La forma de sobrecargar un operador es asignarle una función especial que se invoca cuando se utiliza el operador asociado, bajo ciertas condiciones. Denominamos a estas funciones *función operador*.

Para definir una función operador se utiliza la palabra reservada "operator", y su sintaxis es como sigue :

```
tipo operator <operador> ( lista de parametros )
{
    < cuerpo de la función >
};
```

Donde <operador> es algún operador monario o binario permitido.
No se pueden sobrecargar los siguientes operadores :

.	miembro
*	indirección de acceso a miembro
::	membresía
?:	if aritmético

Otra restricción es que una función operador debe ser una función miembro, o bien tener al menos un parámetro que sea de tipo clase.

El objeto de esta restricción es evitar cambiar la operatoria de los operadores sobre los tipos intrínsecos de datos.

Un intento no válido de redefinición se muestra a continuación :

```
int operator + ( int A , int B )
{
    return A - B ;
}
```

En él se pretendía redefinir el operador suma de enteros, para que los reste.

PROG20.CPP

SOBRECARGA DE OPERADORES

En el siguiente programa se muestra la operación de suma de dos vectores tridimensionales, utilizando el operador "+".

Recordemos que el vector resultante es aquel formado por la suma de los componentes en cada uno de los ejes de los vectores sumandos.

En el programa se declara los vectores tridimensionales V1, V2 y VR (vector resultante) como objetos de la clase "vector".

Nótese la simplicidad de la operación : $VR = V1 + V2 ;$

El resultado que se observa es :

5 6 11

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include "vector.h"

main()
{
    vector V1 , V2 , VR ;
    V1.X = 2 ;
    V1.Y = 4 ;
    V1.Z = 6 ;

    V2.X = 3 ;
    V2.Y = 2 ;
    V2.Z = 5 ;

    VR = V1 + V2 ;

    cout << "\n\n" << VR.X << '\t' << VR.Y << '\t' << VR.Z ;
    getch() ;
}

```

Claro está que la operación anterior se puede llevar a cabo gracias al trabajo realizado en la cabecera propia `vector.h`.

En esta cabecera se declara la clase “`vector`” con tres miembros enteros y públicos, y también se declara la función operador `+`, que retorna un objeto `vector`, y tiene como parámetros dos objetos de clase `vector`.

// Archivo VECTOR.H

```

class vector {
public :
    int X ;
    int Y ;
    int Z ; } ;

vector operator + ( const vector V , const vector W )
{
    vector R ;
    R.X = V.X + W.X ;
    R.Y = V.Y + W.Y ;
    R.Z = V.Z + W.Z ;
    return R ;
}

```

OPERADORES NEW Y DELETE

Los operadores `new` y `delete` están relacionados con la asignación de memoria dinámica. Se supone que el lector está familiarizado con el tema por lo que no entrará en detalles sobre el mismo.

Operador new

El operador **new** gestiona un bloque de memoria dinámica del tamaño del tipo de dato que se le indica, y retorna la dirección de inicio de ese bloque con formato de puntero al tipo especificado.

En caso de fracasar la asignación, el valor retornado es NULL.

La sintaxis es como sigue :

```
tipo * puntero = new tipo ;  
tipo * puntero = new tipo [ tamaño ] ;
```

En el segundo caso se está gestionando memoria para un vector de elementos de tipo *tipo*.

El funcionamiento de “**new tipo**” es similar a la utilización de : `malloc(sizeof(tipo))` ; Pero en el caso de **new** , no se requiere la inclusión de ninguna cabecera especial dado que está disponible de forma inmediata.

En el siguiente ejemplo se gestiona dinámicamente un objeto de clase ALFA :

```
class ALFA {  
public:  
    int A;  
    float B;  
};  
ALFA * P ;  
P = new ALFA ;
```

Operador delete

El operador **delete** libera la memoria gestionada a través de **new**. Su funcionamiento es similar al de la función **free()** , pero no se necesita incluir ninguna cabecera especial, dado que **delete** está disponible en forma inmediata.

La sintaxis es :

```
delete < dirección del bloque > ;
```

En el ejemplo anterior :

```
delete P ;
```

Debe tenerse en consideración que al gestionarse memoria para un objeto o al liberarla, se entiende que se está creando o destruyendo al mismo.
Por lo tanto acuñará los constructores y destructores asociados a él.

PROG21.CPP

LISTA ENLAZADA

En el siguiente programa se propone el manejo de una lista enlazada de nodos constituidos por estructuras NODO.

Se utilizan los operadores **new** y **delete** para la gestión de memoria dinámica. En el programa se muestra su uso.

En el programa principal se permite agregar nodos, eliminar un nodo, observar el contenido de la lista y salir del menú.

Es importante comprender la estructura de la clase LISTA. En ella se define el tipo de dato struct NODO. Nótese que no hay un campo estructura dentro de la clase, sino solamente la definición del nuevo tipo. Se declara un puntero a estructura NODO como miembro privado de la clase, y también la función miembro privada destruir(), destinada a eliminar la totalidad de la lista enlazada. Estos miembros son privados, por lo que no pueden ser accedidos directamente desde el exterior.

La clase contiene un constructor destinado a inicializar el puntero al INICIO, y un destructor cuyo objetivo es eliminar la totalidad de la lista invocando a la función `destruir()`.

El destructor actuará en el momento en que el objeto deje de existir. Esto ocurre, en este programa, cuando el mismo finaliza. Pero si el proceso continuara, el objetivo de estas funciones destructivas es liberar la memoria dinámica que se gestionó.

En todo el programa, solo se instancia un objeto, que constituirá la puerta de entrada a toda la lista. Este objeto contiene el puntero al inicio y las funciones que pueden accederlo, por lo que éste y toda la lista quedan encapsulados.

```

#include <conio.h>
#include <iostream.h>
#include <iomanip.h>
#include <string.h>

#define MENU clrscr(); cout << "\n\n\n\t\t\t0\tsALIR";\
    cout << "\n\n\n\t\t\t1\tpONER\n\n\n\t\t\t2\tsACAR";\
    cout << "\n\n\n\t\t\t3\tmIRAR\n";

class LISTA {
    struct NODO {
        char NOM [20] ;
        struct NODO * sig ; } ;
    struct NODO * INICIO ;
    void destruir ( struct NODO * ) ; /* Elimina la lista */
public:
    LISTA ( void ) ; /* Constructor */
    ~LISTA ( void ) ; /* Destructor */
    int PONE ( char * ) ;
    void LEER ( void ) ;
    int SACA ( char * ) ;
} ;

```

```

LISTA Lista ;

void poner ( void ) ;
void sacar ( void ) ;

void main ( void )
{
    int SELEC = 1 ;

    clrscr();

    while( SELEC ) {
        MENU ;
        cin >> SELEC ;
        switch ( SELEC ) {
            case 1 : poner ( ) ; break ;
            case 2 : sacar ( ) ; break ;
            case 3 : Lista.LEER ( ) ; break ;
        }
    }

    void poner ( void )
    {
        char VEC[20] ;
        cout << "\nIngrese un nombre " ;
        cin >> VEC ;
        if ( Lista.PONE ( VEC ) ) {
            cout << "\n Error de acceso " ;
            getch( ) ;
        }
    }

    void sacar ( void )
    {
        char VEC[20] ;
        cout << "\nIngrese el nombre a eliminar " ;
        cin >> VEC ;
        if ( Lista.SACA ( VEC ) ) {
            cout << "\n No se encontro en la lista " ;
            getch( ) ;
        }
    }
}

LISTA::LISTA ( )
{
    INICIO = NULL ;
}

LISTA::~LISTA ( )
{
    destruir ( INICIO ) ;
}

```

```

void LISTA::destruir ( struct NODO * p )
{
    if ( !p ) return ;
    if ( p -> sig ) destruir ( p -> sig ) ; // Invocacion recursiva
}

int LISTA::PONE ( char * s )
{
    struct NODO * nuevo ;
    if ( ! ( nuevo = new NODO ) )
        return 1 ;
    strcpy ( nuevo -> NOM , s ) ;
    nuevo -> sig = NULL ;
    if ( ! INICIO ) {
        INICIO = nuevo ;
        return 0 ;
    }

    struct NODO * p = INICIO ;
    while ( p -> sig ) p = p -> sig ;
    p -> sig = nuevo ;
    return 0 ;
}

int LISTA::SACA ( char * s )
{
    struct NODO * p = INICIO , * q = NULL ;
    /* Busca el nodo a eliminar */
    while ( p ) {
        if ( !strcmp ( p -> NOM , s ) ) {
            q = p ;
            p = NULL ;
        }
        else
            p = p -> sig ;
    }
    /* No se encontró */
    if ( !q ) return 1 ;

    if ( p == INICIO ) { /* Elimina primer nodo */
        INICIO = p -> sig ;
        delete p ;
        return 0 ;
    }

    while ( p -> sig != q ) p = p -> sig ; /* Ubica el anterior */
    p -> sig = q -> sig ;
    delete q ;
    return 0 ;
}

```

```

void LISTA::LEER ( void )
{
    struct NODO * p = INICIO ;
    clrscr( ) ;
    while ( p ) {
        cout << p -> NOM << endl ;
        p = p -> sig ;
    }
    getch() ;
}

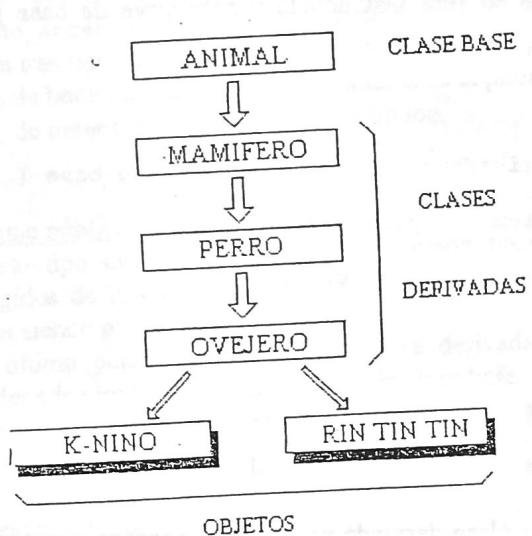
```

HERENCIA

La herencia es una propiedad de la programación orientada a objetos mediante la cual una clase (llamada *clase derivada*) adopta los atributos de una o mas clases (llamadas *clases base*).

Esta propiedad permite que algunos objetos sean creados a partir de otros objetos. Es decir, que los nuevos objetos incorporen estructuras de datos y funciones previstos para sus antecesores, y además agreguen estructuras y funciones propias. De esta manera se logra la reutilización de código ya desarrollado para clases anteriores (base).

Se establece de esta manera una jerarquía de clases (de tipo árbol), que contempla una clase base, y clases derivadas de ella en una o mas “*generaciones*”.



La figura ilustra la jerarquía mencionada.

La clase ANIMAL contiene las características generales del reino. MAMIFERO contiene también las características de la clase que hereda, y le agrega las propias.

El mismo diccionario utiliza para la definición una jerarquía de clases a fin de evitar definiciones demasiado largas. (mamifero = dicese del animal cuyas hembras alimentan a sus crías con leche de sus mamas).

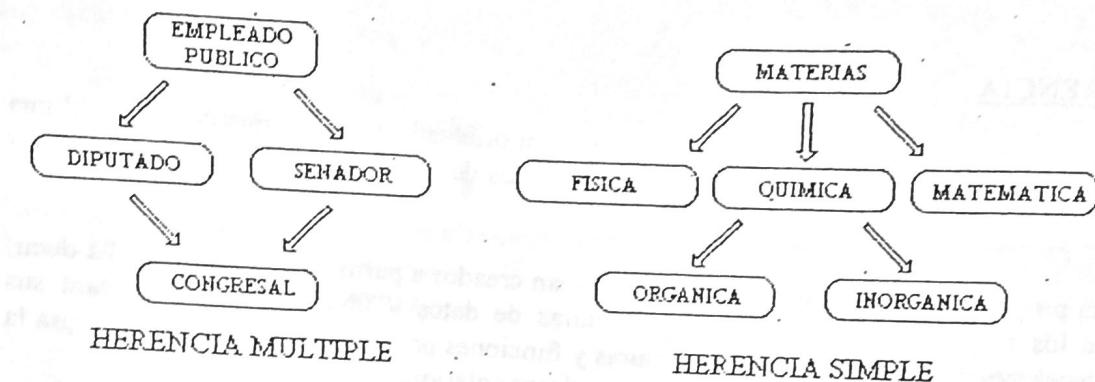
La clase PERRO a su vez hereda a MAMIFERO e indirectamente a ANIMAL.

Lo propio ocurre con la clase OVEJERO, la cual hereda a las anteriores.

Finalmente los objetos K-NINO y RIN TIN TIN son los elementos “*reales*” que tienen las características de las clases anteriores.

Existen dos tipos de herencia utilizados :

- Herencia simple : cada clase solamente puede tener un antepasado inmediato.
- Herencia múltiple : cada clase puede tener mas de un antepasado inmediato.



Es posible crear una clase con el único efecto de posibilitar que otras clases hereden sus características, se la denomina entonces, clase abstracta.

Clase abstracta : es aquella clase que no será instanciada y solo sirve de base para ser heredada.

Sintaxis de herencia :

```
class clase_derivada : tipo_de_acceso clase_base {  
    public :  
        miembros_publicos ;  
    private :  
        miembros_privados ;  
}
```

MIEMBROS PROTEGIDOS

Los miembros privados de la clase base son inaccesibles desde la clase derivada, mientras que los públicos son accesibles.

Esto significa que en un objeto de una clase derivada no se podrá acceder directamente a los datos y funciones privadas de la clase base, sino usando funciones públicas de la clase base que accedan a aquellos.

Una opción para tener un acceso directo a estos miembros en la clase derivada, seria hacerlos públicos en la clase base, pero de esta manera quedarían desprotegidos de los accesos directos desde el exterior ; hacia un objeto de la clase base.

Como lograr ambas cosas ? Protección en clase base y acceso directo en la derivada ?

Es tiempo de presentar una nueva categoría de miembros : los miembros protegidos.
Estos se declaran utilizando la palabra reservada `protected`.

Cuando un miembro se declara como protegido, solo se puede acceder a él mediante funciones miembro o amigas (al igual que los miembros privados), pero el tipo de acceso se hereda, es decir, en clases derivadas se podrá acceder a los mencionados miembros.

Si se va a trabajar sin emplear herencia (como hemos hecho hasta ahora), no tiene objeto el uso de miembros protegidos.

La existencia de miembros protegidos en una clase, indica que ésta está preparada para ser heredada.

ACCESIBILIDAD DE LOS MIEMBROS		
MIEMBRO	ACCESO DESDE EL EXTERIOR	ACCESO DESDE CLASE DERIVADA
PUBLICO	SI	SI
PRIVADO	NO	NO
PROTEGIDO	NO	SI

ACCESIBILIDAD EN LA HERENCIA

Cuando se describió la sintaxis de herencia se mostró un término denominado `tipo_de_acceso` en la herencia, o directamente, el *tipo de herencia*.

Existen tres tipos de herencia : **privado**, **público** y **protegido**.

El tipo de herencia utilizado habitualmente es el público.

El tipo de herencia por omisión es el público.

Herencia pública

En este tipo de herencia, la clase derivada tiene acceso a los miembros públicos y protegidos de la clase base. El status de herencia determina que los miembros públicos siguen siendo públicos en la clase derivada, y los protegidos siguen siendo protegidos. Este último punto indica que una clase derivada en segunda instancia también podrá acceder a los miembros protegidos de la clase base.

Los miembros privados de la clase base no son accesibles por las clases derivadas.

Herencia privada

En este caso, tanto los miembros públicos como protegidos de la clase base, adquieren el status de privados en la clase derivada, por lo que no son accesibles desde el exterior de la misma (solo lo son por miembros de ésta), ni por sus clases derivadas. Los miembros privados de la clase base no son accesibles por las clases derivadas.

Herencia protegida

La herencia protegida provee un nivel intermedio de accesibilidad, dado que los miembros públicos y protegidos de la clase base, pasan a ser protegidos en la clase derivada. Esto significa que si bien no son accesibles desde el exterior de ésta, si lo son por clases derivadas de ellas.

Los miembros privados de la clase base no son accesibles por las clases derivadas.

El siguiente cuadro resume el status que adquieren los diferentes miembros de la clase base al ser heredados mediante las tres variantes de herencia :

ACCESO A MIEMBROS HEREDADOS EN CLASE DERIVADA			
MIEMBRO EN CLASE BASE	HERENCIA PÚBLICA	HERENCIA PROTEGIDA	HERENCIA PRIVADA
PUBLICO	PUBLICO	PROTEGIDO	PRIVADO
PROTEGIDO	PROTEGIDO	PROTEGIDO	PRIVADO
PRIVADO	NO ACCESIBLE	NO ACCESIBLE	NO ACCESIBLE

PROG22.CPP

VECTOR DE OBJETOS DE CLASE DERIVADA

En este programa se ingresarán los datos de los 5 miembros del staff de una empresa y luego se mostrarán en pantalla dichos datos.

Para ello se declara una clase llamada EMPLEADO y otra llamada STAFF que hereda públicamente las características de la primera y agrega las suyas propias, considerando que un miembro del staff de la empresa es también un empleado.

El programa en sí es muy simple, y se sugiere que se vuelque la atención al proceso de herencia y los accesos a sus distintos datos a través de las funciones miembro.

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

#define N 5

class EMPLEADO {
protected :
    int LEGAJO ;
    char NOM[20] ;
    float SUELDO ;
public :
    void LEER ( void ) ;
    void MOSTRAR ( void ) ;
} ;
```

```

class STAFF : public EMPLEADO {
protected :
    char CARGO[20] ;
    int PERS_A_CARGO ;
public :
    void INGRESAR ( void ) ;
    void IMPRIMIR ( void ) ;
} ;

void main ( )
{
    STAFF VEC[N] ;
    int I ;

    for ( I=0 ; I < N ; I++ ) {
        clrscr ( ) ;
        cout << I ;
        VEC[I].INGRESAR ( ) ;
    }

    clrscr ( ) ;
    for ( I=0 ; I < N ; I++ )
        VEC[I].IMPRIMIR ( ) ;
    getch() ;
}

void EMPLEADO :: LEER ( )
{
    cout << "\nIngrese el legajo : " ;
    cin >> LEGAJO ;
    cout << "\nIngrese el nombre : " ;
    cin >> NOM ;
    cout << "\nIngrese el sueldo : " ;
    cin >> SUELDO ;
}

void EMPLEADO :: MOSTRAR ( )
{
    cout << "\n" << LEGAJO ;
    cout << "\t\t" << NOM ;
    cout << "\t\t" << SUELDO ;
}

void STAFF :: INGRESAR ( )
{
    LEER ( ) ;
    cout << "\nIngrese el cargo : " ;
    cin >> CARGO ;
    cout << "\nIngrese la cantidad de personas a cargo : " ;
    cin >> PERS_A_CARGO ;
}

```

```
void STAFF :: IMPRIMIR ( )
{
    MOSTRAR ( ) ;
    cout << "\t\t" << CARGO ;
    cout << "\t\t" << PERS_A_CARGO ;
}
```

TEMPLATES (PLANTILLAS)

Como hemos visto, en muchos casos se utilizan algoritmos que no cambian al variar el tipo de datos involucrado.

Sin embargo, al implementarlos mediante algún lenguaje, surge código que sí cambia según el tipo de datos.

Esta problemática se puede paliar en parte, **sobrecargando** la función que desarrolle el mencionado algoritmo, de manera de disponer de una función para cada tipo de dato. Esto implica la construcción de tantas funciones homónimas como tipos se deseen manejar, lo cual es, en definitiva, tarea del programador.

Sin embargo, existe un mecanismo mediante el cual se delega parte de esta tarea en el compilador.

Dado que el algoritmo es independiente del tipo de dato utilizado, se puede crear un "molde" o plantilla (template) de la función, en donde el tipo de dato a utilizar se establezca como "genérico" mediante algún tipo simbólico.

En tiempos de compilación, el compilador observa en el programa cual es el tipo real de dato que utiliza la función, y en ese momento crea el código correspondiente (basándose obviamente en la plantilla proporcionada).

La situación descripta hace referencia específicamente a plantillas de funciones, sin embargo, es posible extender el concepto a plantillas de clases, es decir "moldes" de clases cuyos datos miembros sean de tipos genéricos. De esta manera se logra *genericidad*.

GENERICIDAD²

La genericidad es una propiedad que permite definir una clase o una función, sin especificar el tipo de datos de uno o mas de sus miembros o parámetros, y de esta forma, poder cambiar la clase o la función para adaptarla a los diferentes tipos de datos utilizados, sin necesidad de reescribirla.

PLANTILLAS DE FUNCIONES

Como hemos visto, es posible crear una función genérica o *plantilla de función* independiente del tipo de datos, y dejar que el compilador instancie la función real o *función de plantilla*, adecuándola a los tipos reales utilizados.

A continuación veremos como hacerlo :

Creación de la plantilla

Para indicarle al compilador que se creará una plantilla de función se utiliza la palabra reservada `template`.

Sintaxis :

```
template < class TIPO >
    tipo función ( lista de parámetros )
    {
        cuerpo de la función
    } ;
```

Donde `TIPO` representa un tipo de dato simbólico (podría haberse utilizado cualquier nombre). No confundir con `tipo` que es el tipo de dato retornado por la función. Este tipo de dato simbólico se utilizará en todo lugar de la función donde el compilador deba interpretar que se utilizará el tipo de dato particular del caso.

Aclaremos esto con un ejemplo :

Se construirá la plantilla de la función `cuadrado()` que recibe un valor (de algún tipo) y retorna su cuadrado.

```
template < class TIPO >
    TIPO cuadrado ( TIPO valor )
    {
        return valor * valor ;
    } ;
```

Nótese que se está indicando que `TIPO` es un tipo de dato genérico, que la función recibe un argumento de ese tipo y retorna un valor del mismo.

Cuando el compilador detecte la invocación de la función `cuadrado()` tendrá en cuenta el tipo de sus argumentos, y de acuerdo con ello instanciará la función plantilla correspondiente.

PROG23.CPP USO DE PLANTILLA DE FUNCIONES

Se muestra a continuación el ejemplo completo de creación de la plantilla y su utilización en un programa.

En este caso, se invoca a la función `cuadrado()` con un valor entero, en un caso, y un valor flotante, a continuación.

El compilador generará entonces, dos funciones plantilla.

```

#include <iostream.h>
#include <conio.h>

template < class TIPO >
TIPO cuadrado ( TIPO valor )
{
    return valor * valor ;
}

void main ()
{
    int A ;
    float B ;
    cin >> A ;
    cin >> B ;
    cout << "\n\n\n" << cuadrado(A) ;
    cout << "\n\n"   << cuadrado(B) ;
    getch() ;
}

```

PROG24.CPP

OTRO EJEMPLO DE PLANTILLA DE FUNCIONES

En este caso se realizará la plantilla correspondiente a la función de intercambio de elementos de un vector a ordenar (`swapping()`).

En el programa se utilizan dos vectores de objetos. Uno pertenece a una clase cuyos miembros son un entero y un flotante, mientras que la otra clase está formada por un miembro string. La plantilla "no sabe" de que tipo es el objeto que recibirá. Obsérvese que la función `swapping()`, en un caso recibe objetos de la clase ESTRUCTURA y en otro caso recibe objetos de la clase STRING.

La función plantilla recibe sus argumentos por referencia (alias) dado que debe trabajar sobre el original de los mismos a fin de intercambiarlos.

En el programa se lee, ordena e imprime los vectores mencionados.

```

#include <iostream.h>
#include <conio.h>
#include <string.h>
#define N 5

template < class ESTRU >
void swapping ( ESTRU & A , ESTRU & B )
{
    ESTRU AUX ;
    AUX = A ;
    A = B ;
    B = AUX ;
}

```

```

void main ()
{
    class ESTRUCTURA {
        public :
            int A ;
            float B ;
    } ;

    class STRING {
        public :
            char NOM[10] ;
    } ;

    ESTRUCTURA VEC[N] ;
    STRING      STR[N] ;

    for ( int I = 0 ; I < N ; I++ ) {
        cin >> VEC[I].A ;
        cin >> VEC[I].B ;
        cin >> STR[I].NOM ;
    }

    for ( I=0 ; I<N-1 ; I++ )
        for( int J=0 ; J<N-I-1 ; J++ )
            if ( VEC[J].A < VEC[J+1].A )
                swapping ( VEC[J] , VEC[J+1] ) ;

    for ( I=0 ; I<N-1 ; I++ )
        for( int J=0 ; J<N-I-1 ; J++ )
            if ( strcmp(STR[J].NOM,STR[J+1].NOM) > 0 )
                swapping ( STR[J] , STR[J+1] ) ;

    for ( I = 0 ; I < N ; I++ ) {
        cout << VEC[I].A << "\t\t" << VEC[I].B ;
        cout << "\t\t" << STR[I].NOM << "\n" ;
    }

    getch() ;
}

```

PLANTILLAS DE CLASES

Así como las plantillas de funciones permiten definir funciones genéricas, las plantillas de clases permiten definir **clases genéricas**. Estos "moldes" de clases pueden contener objetos parametrizados, es decir, objetos cuyos tipos se entreguen como argumentos de la clase, en el momento de la instanciaación.

Dado que la clase así creada trabaja sobre datos de tipo genérico, las funciones miembro de la misma, también lo harán y por lo tanto, se deberá aplicar el concepto de plantilla de funciones, al definirlas.

A fin de definirlas se utiliza también la palabra reservada **template**.

Sintaxis :

```
template <class tipo_generico>
class nombre_de_clase
{
    .......
```

La sintaxis de instanciaión es :

```
nombre_de_clase < tipo_parámetro > nombre_de_objeto ;
```

Como ejemplo crearemos un vector de 10 elementos de algun tipo :

```
template < class T >
class VECTOR
{
    private :
        T VEC[10] ;
    public :
        .....
} ;

VECTOR < int > VEC_INT ;
VECTOR < float > VEC_FLOAT ;
```

VEC_INT y VEC_FLOAT son objetos compuestos por vectores de 10 enteros y 10 flotantes respectivamente.

Podemos incorporar un parámetro adicional a la plantilla, a fin de considerar el tamaño del vector.

Debemos recordar que la declaración de un array exige que la cantidad de elementos sea una constante. Por esta razón, en este caso, el vector se creará dinámicamente, dejándole esa tarea al constructor.

```
template < int NUM , class TIPO >
class VECTOR
{
    private :
        TIPO *VEC ;
    public :
        VECTOR { VEC = new TIPO[NUM]; } ;
        .....
} ;

int N = 25 ;
VECTOR < N , int > VEC_INT ;
VECTOR < N , float > VEC_FLOAT ;
```

En este caso, `VEC_INT` es un objeto compuesto por un vector de 8 enteros, mientras que `VEC_FLOAT` es un objeto compuesto por un vector de 25 flotantes.

Otra manera de transferir el tamaño del vector, es parametrizar el constructor de manera que lo reciba. Esta forma se utilizará en el siguiente programa ejemplo.

PROG25.CPP

MANEJO DE PILA CON PLANTILLA DE CLASES

Se implementará una plantilla de clases que genere y maneje una pila de elementos genéricos. La declaración de dicha plantilla de clases se incorporará a la cabecera propia `PILA.H`.

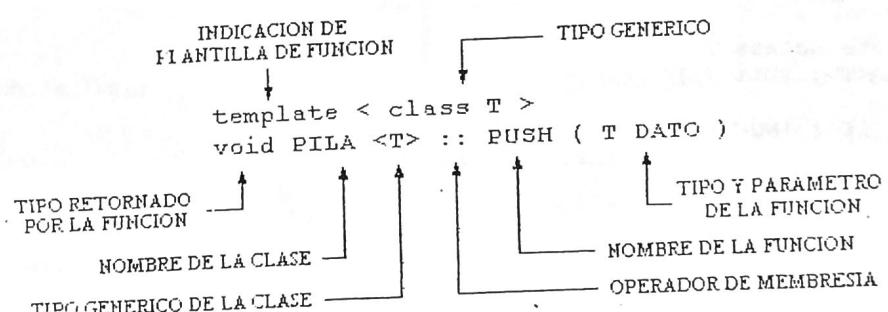
En esta plantilla se declaran los miembros privados `INICIO` (puntero al inicio del vector dinámico de objetos genéricos), `SP` (stack pointer = puntero utilizado para realizar el recorrido del vector), `CANT` (variable entera que guarda el tamaño del vector. Es necesaria dado que el parámetro del constructor, `N`, es local al mismo y no es visible por las otras funciones miembro.) y `NUM` (variable entera que guarda la cantidad actual de datos en la pila. Podría haberse prescindido de ella, pero se la incorporó por razones de claridad.)

El **constructor** recibe como argumento (con un default de 10 elementos) el tamaño del vector - pila, inicializa todas las variables y gestiona memoria dinámica para alojar al vector - pila.

Dado que el vector - pila se gestiona dinámicamente, la función del **destructor** es liberar la memoria pedida cuando se elimina el objeto.

Las funciones `PUSH()` y `PULL()` son las encargadas de incorporar o extraer un dato de tipo genérico `T` de la pila, comprobando si la misma está llena o vacía, e informando tal situación.

Obsérvese que la declaración de las funciones miembro se realiza a través de plantillas de funciones con el siguiente encabezamiento :



A continuación se muestra el contenido de la cabecera PILA.H :

```
// PILA.H

template < class T >
class PILA {
private :
    T *INICIO , *SP ;
    int CANT , NUM ;
public :
    PILA ( int N = 10 ) ;
    ~PILA ( ) { delete INICIO ; } ;
    void PUSH ( T ) ;
    T PULL ( void ) ;
};

template < class T >
PILA <T> :: PILA ( int N = 10 )
{
    CANT = N ;
    NUM = 0 ;
    INICIO = new T[N] ;
    SP = INICIO ;
} ;

template < class T >
void PILA <T> :: PUSH ( T DATO )
{
    if ( NUM == CANT ) {
        cout << "\n\nPila llena\n\nDato perdido\n\n" ;
        getch() ;
        return ;
    }
    *SP = DATO ;
    SP++ ;
    NUM++ ;
}

template <class T>
T PILA<T>::PULL ( )
{
    if ( !NUM )
        cout << "\n\nPila vacia\n\n" ;
    else {
        SP-- ;
        NUM-- ;
    }
    return *SP ;
}
```

El programa principal incluye la cabecera PILA.H .

En él se declara la clase ESTRUCTURA compuesta por un entero y un string, esta clase se utilizará para instanciar la plantilla de clases PILA.

También se solicita que se ingrese por teclado el tamaño deseado de la pila.

A continuación se muestra el listado del programa principal :

```
#include <iostream.h>
#include <conio.h>
#include "C:/BC/BIN/PILA.H"
#define MENU cout << ".\n\n1. PONER\n2. SACAR\n\n" ;

void main ()
{
    int TAM ;
    class ESTRUCTURA {
        public :
            int A ;
            char NOM[10] ;
    } ;

    ESTRUCTURA DATO ;

    clrscr () ;
    cout << "\n\nIngrese el tamaño de la pila : " ;
    cin  >> TAM ;

    // Instanciacion de la clase
    PILA <ESTRUCTURA> Pila (TAM) ;

    int SEL=1 ;
    do {
        clrscr() ;
        MENU ;
        cout << "\n\nSELECCION : " ;
        cin  >> SEL ;
        switch (SEL) {
            case 1 : cout << "\n\nIngrese numero y nombre \n\n" ;
                       cin >> DATO.A ;
                       cin >> DATO.NOM ;
                       Pila.PUSH(DATO) ;
                       break ;

            case 2 : DATO = Pila.PULL() ;
                       cout << "\n\n" << DATO.A << "\t\t" << DATO.NOM ;
                       getch() ;
                       break ;
        }
    } while (SEL) ;
}
```