MATERIA PROGRAMACIÓN III
ESCUELA: ANÁLISIS DE SISTEMAS
ÚLTIMA ACTUALIZACIÓN: FEBRERO 2019

ELABORADO POR: ING. MARIA NEUS
OBJETIVOS GENERALES
- Identificar los elementos que componen el Lenguaje de programación C++.
- Codificar instrucciones utilizando las normas específicas del Lenguaje C++.
- Aplicar los conocimientos teóricos prácticos en el desarrollo de programa
CONTENIDO
UNIDAD 1. INTRODUCCIÓN AL LENGAJE C++
1.1 Origen del lenguaje
1.2 Estructura básica de un programa
UNIDAD 2. VARIABLES, CONSTANTES, OPERADORES
2.1 Tipos de datos
2.2 Declaración y tipos de variables
2.3 Funciones de Entrada/Salida
2.4 Modificadores de formato
2.5 Operadores
UNIDAD 3. ESTRUCTURAS
3.1 Vectores Paralelos
3.2 Matrices
UNIDAD 4. PUNTEROS
4.1 Declaración de un puntero
4.2 Punteros a tipos de dato numérico
4.3 Parámetros de métodos tipo puntero
UNIDAD 5. CLASES
5.1 Definición y declaración
5.2 Clases amigas
5.3 Herencia
UNIDAD 6. ENTRADA, SALIDA DE ARCHIVOS EN DISCO
6.1 Conceptos generales
6.2 Memoria dinámica
6.3 Trabajar con ficheros

UNIDAD 1. INTRODUCCION AL LENGUAJE "C++"

1.1 Origen del Lenguaje.

La historia del lenguaje de programación C++ comienza a principios de los años 70, con un programador de nombre Dennis Ritchie que trabajaba en los laboratorios de AT&T Bell. Trabajando con un lenguaje llamado BCPL inventado por Martin Richards (que luego influyó para crear el B de Ken Thompson), Dennis deseaba un lenguaje que le permitiese manejar el

hardware de la misma manera que el ensamblador pero con algo de programación estructurada como los lenguajes de alto nivel. Fue entonces que creó el C que primeramente corría en computadoras PDP-7 y PDP-11 con el sistema operativo UNIX. Pero los verdaderos alcances de lo que sería éste, se verían poco tiempo después cuando Dennis volvió a escribir el compilador C de UNIX en el mismo C, y luego Ken Thompson (diseñador del sistema) escribió UNIX completamente en C y ya no en ensamblador. Al momento de que AT&T cedió (a un precio bastante bajo) el sistema operativo a varias universidades, el auge de C comenzaba. Cuando fueron comerciales las computadoras personales, empezaron a diseñarse varias versiones de compiladores C, éste se convirtió en el lenguaje favorito para crear aplicaciones. En 1983, el Instituto Americano de Normalización (ANSI) se dio a la tarea de estandarizar el lenguaje C, aunque esta tarea tardó 6 años en completarse, y además con la ayuda de la Organización Internacional de Normalización (ISO), en el año de 1989 definió el C Estándar.

A partir de éste, se dio pie para evolucionar el lenguaje de programación C. Fue en los mismos laboratorios de AT&T Bell, que Bjarnes Stroutstrup diseñó y desarrolló C++ buscando un lenguaje con las opciones de programación orientada a objetos. Ahora el desarrollo del estándar de C++ acaparaba la atención de los diseñadores. En el año 1995, se incluyeron algunas bibliotecas de funciones al lenguaje C. Y con base en ellas, se pudo en 1998 definir el estándar de C++. Algunas personas podrían pensar que entonces C++ desplazó a C, y en algunos aspectos podría ser cierto, pero también es cierto que algunas soluciones a problemas requieren de la estructura simple de C más que la de C++, C generalmente es usado por comodidad para escribir controladores de dispositivos y para programas de computadoras con recursos limitados. La base del lenguaje fue creada por programadores y para programadores, a diferencia de otros lenguajes como Basic o Cobol que fueron creados para que los usuarios resolvieran pequeños problemas de sus ordenadores y el segundo para que los no programadores pudiesen entender partes del programa. C++ es un lenguaje de nivel medio pero no porque sea menos potente que otro, sino porque combina la programación estructurada de los lenguajes de alto nivel con la flexibilidad del ensamblador. La siguiente tabla muestra el lugar del lenguaje respecto a otros.

En la actualidad los lenguajes que originalmente eran no estructurados han sido modificados para que cumplan con esta característica, tal es el caso de Basic, que actualmente soporta la programación orientada a objetos. Podemos notar cuando un lenguaje de programación es viejo si vemos que no cumple con la programación estructurada. C++ es, también, un lenguaje orientado a objetos, y es el mismo caso de Java. Al referirnos a lenguaje estructurado debemos pensar en funciones, y también a sentencias de control (if, while, etc.)

C++ es un superconjunto de C, cualquier compilador de C++ debe ser capaz de compilar un programa en C. De hecho la mayoría admite tanto código en C como en C++ en un archivo. Por esto, la mayoría de desarrolladores compilan con C++ su código escrito en C, incluso hay quienes, siendo código en C ponen la extensión CPP (extensión de los archivos de código

C++) y lo compilan con C++ (hasta hace unos días yo hacía esto), lo cual no es recomendable por norma al programar. Cuando se compila código C en un compilador C++ este debe cumplir con los estándares definidos en 1989, cualquier palabra definida en el estándar de 1999 no funcionará. La evolución de C++ continúa, y la diversidad de compiladores contribuye a ello.

1.2 Estructura básica de un programa

```
#include <iostream.h>                    }declaración de librerias

int main(void){                          }función main
    cout<<"hola mundo"<<endl;            }secuencia de instrucciones
    return 0;                            }valor de retorno de la
                                         función
}                                        }llaves de cierre de la
```

Analizaremos la estructura del programa:

#include <iostream.h>

La parte del #include se refiere a la biblioteca de funciones que vamos a utilizar. Es decir para llamar a una biblioteca en particular debemos hacer lo siguiente:

#include <librería_solicitada>

El estándar de C++ incluye varias bibliotecas de funciones, y dependiendo del compilador que se esté usando, puede aumentar el número.

int main(void){

Todo programa en C++ comienza con una función main(), y sólo puede haber una.

En C++ el main() siempre regresa un entero, es por eso se antepone “int” a la palabra “main”. Los paréntesis que le siguen contienen lo que se le va a mandar a la función. En este caso se puso la palabra “void” que significa vacío, es decir que a la función main no se le está mandando nada, podría omitirse el void dentro de los paréntesis, el compilador asume que no se enviará nada. La llave que se abre significa que se iniciará un bloque de instrucciones.

cout<<”hola mundo”<<endl;

Esta es una instrucción. La instrucción cout está definida dentro de la biblioteca iostream.h, que previamente declaramos que íbamos a utilizar. Una función, en este caso main() siempre comienza su ejecución con una instrucción (la que se encuentra en la parte superior), y continúa así hasta que se llegue a la última instrucción (de la parte inferior). Para terminar una instrucción siempre se coloca “;”.

Pero además de instrucciones se pueden invocar funciones definidas por el usuario (por supuesto diferentes de main) como se verá más adelante.

return 0;

Esta es otra instrucción, en este caso la instrucción return determina que es lo que se devolverá de la función main(). Habíamos declarado que main devolvería un entero, así que la instrucción return devuelve 0. Lo cual a su vez significa que no han ocurrido errores

durante su ejecución. } La llave de cierre de la función main() indica el termino del bloque de instrucciones. En algunos programas de ejemplo, notará el uso de dobles diagonales (“//”). Estas diagonales se usan para escribir comentarios de una línea dentro del código del programa. Además podrá encontrar el uso de “/*” “*/” estos caracteres encierran un comentario de varias líneas y cualquier cosa que se escriba dentro de ella no influenciará en el desempeño del programa. También verá que muchas veces utilizaremos una diagonal invertida (“\”). Este signo se utiliza cuando una instrucción ocupará varias líneas y por razones de espacio en la hoja es mejor dividirla en partes.

EVALUACIÓN

Identifique la función de cada línea de código del siguiente ejercicio:

```
#include<iostream.h>
#include<conio.h>
int n1, n2, n3;
float p;
char nom[20];
void main(){
clrscr();
cout<<"\n\r Ingrese el Nombre del Alumno ";
cin>>nom;
gotoxy(10,5);
cout<<"\n\r Ingrese las notas \n\r";
cin>>n1>>n2>>n3;
p=(n1+n2+n3)/3;
if(p>10.4)
cout<<"\n\r Su promedio es "<<p<<" y su condicione es Aprobado";
else
cout<<"\n\r Su promedio es "<<p<<" y su condicione es Desaprobado";
getch();
}
```

UNIDAD 2.VARIABLES, CONSTANTES, OPERADORES, EXPRESIONES:

2.1- Tipos de datos.

En la sección anterior vimos la forma general de un programa, un programa sumamente sencillo. Ahora veamos un programa muy parecido al anterior:

```
#include <iostream.h>
int main( ){
int variable;
variable=5;
cout<<variable;
```

```
return 0;
}
```

Notemos en esta ocasión sólo la parte: int variable;

A esta sección se le denomina declaración. Se trata de la declaración de una variable de nombre

"variable".

Una variable es una posición de memoria con nombre que se usa para mantener un valor que puede ser modificado por el programa. Las variables son declaradas, usadas y liberadas. Una declaración se encuentra ligada a un tipo, a un nombre y a un valor.

C++ tiene los siguientes tipos fundamentales:

Caracteres: char (también es un entero), wchar_t

Enteros: short, int, long, long long

Números en coma flotante: float, double, long double

Booleanos: bool

Vacío: void

El modificador unsigned se puede aplicar a enteros para obtener números sin signo (por omisión los enteros contienen signo), con lo que se consigue un rango mayor de números naturales.

2.2 Declaración y tipos de variables.

Básicamente, la declaración de una variable presenta el siguiente aspecto:

tipo nombre [=valor];

Las variables se pueden declarar en tres sitios básicos: dentro de las funciones (ya sea la función main u otras creadas por el programador), estas variables son llamadas locales; en la definición de parámetros de una función, como se verá más adelante; y fuera de todas las funciones, variables globales.

Ejemplo:

```
#include <iostream.h>
int variable_global=10;
int main(){
int variable_local=20;
   cout<<"\nprograma que muestra los usos de variables "\
"globales y locales\n"<<endl;
   cout<<"la variable global tiene asignado un: "\
  <<variable_global<<endl;
   cout<<"\nla variable local tiene asignado un: "\
<<variable_local<<endl;
   return 0;
}
```

Una variable global puede ser modificada en cualquier parte del programa, mientras que una variable local sólo puede ser modificada y utilizada dentro de la función en la que se ha

declarado. Por supuesto, antes de utilizar una variable y hacer operaciones con ella, hay que declararla.

Por lo general, siempre se trata de utilizar lo menos posible la declaración de variables globales. El siguiente ejemplo muestra que se pueden declarar variables en cualquier parte del programa, siempre y cuando se declaren antes de usarlas.

```
#include <iostream.h>
int main( ){
int variable1=10;
cout<<"la variable 1 local tiene almacenado un: "\
<<variable1<<endl;
variable1=50;
int variable2=variable1+30;
cout<<"\nla variable 2 almacena un: "\
<<variable2<<endl;
return 0;
}
```

En un programa puede que necesitemos declarar un dato y asignarle un nombre, pero que éste no pueda ser modificado. En este caso debemos declarar una constante.

2.3 Funciones de Entrada/Salida

En los programas hechos hasta el momento, hemos utilizado la instrucción cout<< para mandar mensajes a la pantalla.

La mayoría de los programas en C++ incluyen el archivo de encabezado <iostream.h>, el cual contiene la información básica requerida para todas las operaciones de entrada y salida (E/S) de flujo.

Cuando usamos la instrucción:

```
cout<<"Mensaje a la pantalla"<<endl;
```

Estamos enviando una cadena de caracteres ("Mensaje a la pantalla") al dispositivo de salida estándar (la pantalla). Luego, el manipulador de flujo endl da el efecto de la secuencia de escape '\n'.

Pruebe el siguiente programa:

```
#include <iostream.h>
int main(){
cout<<"cadena de caracteres"<<endl;
cout<<2+2<<endl; //imprime un entero
cout<<9/2<<endl; //imprime un flotante
cout<<(int)(3.141592+2)<<endl; //imprime un entero
return 0; }
```

La instrucción cout<< puede imprimir tanto números enteros como flotantes sin necesidad de decirle específicamente el tipo de datos del que se trata, pero, por supuesto notemos

que al enviarle una cadena de caracteres esta debe de estar entre comillas. Ya en ejemplos anteriores vimos cómo se mandaba a la pantalla el valor de una variable, así que no hace falta más ilustración al respecto.

La interacción con el usuario es algo muy importante en la programación, imaginemos que en este preciso momento y con los conocimientos que tenemos hasta ahora, necesitamos hacer un programa que calcule la distancia a la que caerá un proyectil lanzado a determinada velocidad y ángulo, o simplemente un programa que calcule las raíces de una ecuación cuadrática. Sería muy molesto estar cambiando los valores de las variables directamente en el código para cada caso que queramos calcular. Por eso debemos ver cuanto antes la forma de leer datos desde el teclado.

La principal función para leer desde el teclado es cin>>, pero es mejor ver un ejemplo para tener la idea clara.

```
#include <iostream.h>
int main(){
int numero;
char car;
float otroNum;
cout<<"escribe un numero:"<<endl;
cin>>numero;
cout<<"\nel numero que tecleaste es: "<<numero<<endl;
cout<<"dame una letra"<<endl;
cin>>car;
cout<<"\ntecleaste: "<<car<<endl;
cout<<"escribe un numero flotante"<<endl;
cin>>otroNum;
cout<<"\neste es: "<<otroNum;
```

return 0; } En resumen, cin es el flujo de entrada asociado al teclado, cout es el flujo de salida estándar asociado a la pantalla, y existe otro, que aunque a lo largo de este trabajo casi no lo utilicemos, es necesario nombrarlo, cerr, que es el flujo de error estándar asociado a la pantalla. Los operadores << y >> son operadores de inserción y extracción de flujo respectivamente, y no deben confundirse con los de desplazamiento de bits. Estos operadores son muy eficaces porque no es necesario especificar formatos para presentación o lectura, ellos los presentan en función al tipo de datos de la variable. Aunque en ocasiones podría necesitar de nuestra ayuda para obtener los resultados específicos que queremos, y para eso están los modificadores de formato.

2.4 Modificadores de formato

```
cout<<(int)(3.141592+2)<<endl;
```

Si probó el código y lo ejecutó seguramente esperaba que apareciese en la pantalla

5.141592, esto habría pasado si no hubiésemos hecho una conversión de tipo. La inclusión de “(int)” dentro de la instrucción provocó que el resultado de la operación, que debía ser 5.141592, se transformara a un entero y por resultado perdiera sus valores decimales.

A esto se le llama hacer un cast, y es muy común hacerlo cuando no queremos que se pierdan determinadas propiedades en los resultados de las operaciones. Por ejemplo, las siguientes líneas de código, que son equivalentes tienen por efecto convertir el tipo de la variable de origen al de la variable destino:

```
int B; int B;
double A= (double) B; double A = double(B);
```

Otra de las herramientas para la especificación de formatos son los manipuladores de flujos, así como el manipulador de flujo “endl” da una secuencia de escape, los manipuladores dec, oct y hex, hacen que la variable a la que le anteceden sea presentada en formato decimal, octal o hexadecimal respectivamente.

```
#include <iostream.h>
int main(){
int numero=25;
int leido;
cout<<"numero es en octal: "<<oct<<numero<<endl;
cout<<"en hexadecimal: "<<hex<<numero<<endl;
cout<<"en decimal: "<<dec<<numero<<endl;
cout<<"ahora teclea un numero"<<endl;
cin>>hex>>leido;
cout<<"el leido vale: "<<hex<<leido<<endl;
cout<<"y en decimal: "<<dec<<leido<<endl;
cout<<"y en octal: "<<oct<<leido<<endl;
return 0;
}
```

Estos manipuladores de flujo no son exclusivos de la salida, también funcionan con la entrada de datos, por defecto, la entrada desde el teclado se lee en decimal y también la presentación de datos se hace en decimal, así que si se sabe que el usuario dará a la máquina un dato en hexadecimal, sería buena idea anticiparlo y poner el formato en que será leído.

2.5 Operadores.

Tabla de operadores

Operador	Descripción	Asociatividad
::	Resolución de ámbito (solo C++)	
++ -- () [] . ->typeid() const_cast dynamic_cast reinterpret_cast static_cast	Post- incremento y decremento Llamada a función Elemento de vector Selección de elemento por referencia Selección de elemento con puntero Información de tipo en tiempo de ejecución (solo C++) Conversión de tipo (solo C++) Conversión de tipo (solo C++) Conversión de tipo (solo C++) Conversión de tipo (solo C++)	Izquierda a derecha
++ -- + - ! ~ (type) * & sizeof new new[] delete delete[]	Pre- incremento y decremento Suma y resta unitaria NOT lógico y NOT binario Conversión de tipo Indirección Dirección de Tamaño de Asignación dinámica de memoria (solo C++) Desasignación dinámica de memoria (solo C++)	Derecha a izquierda
.* ->*	Puntero a miembro (solo C++)	Izquierda a derecha
* / %	Multiplicación, división y módulo	
<< >>	Operaciones binarias de desplazamiento	
< <= > >=	Operadores relaciones "menor que", "menor o igual que", "mayor que" y "mayor o igual que"	
== != &	Operadores relaciones "igual a" y "distinto de" AND binario	
^	XOR binario	
|	OR binario	
&&	AND lógico	
||	OR lógico	
c?t:f	Operador ternario	
= += -= *= /= %= <<= >>= &= ^= |= throw	Asignaciones Operador Throw (lanzamiento de excepciones, solo C++)	Derecha a izquierda
,	Coma	Izquierda a derecha

Más información sobre el uso de los operadores en el siguiente link http://profesores.fi-b.unam.mx/carlos/lcpi/p09/OPERADORES%20EN%20%20C++.pdf

EVALUACIÓN

Realice un programa que calcule el área de un círculo, haga uso de los operadores, variables y modifique los formatos correspondientes

Realice un programa que calcule el área de un triángulo haga uso de los operadores, variables y modifique los formatos correspondientes

UNIDAD 3 ESTRUCTURAS

3.1 Vectores Paralelos

Este concepto se da cuando hay una relación entre las componentes de igual subíndice (misma posición) de un vector y otro.

nombres	Juan	Ana	Marcos	Pablo	Laura
edades	12	21	27	14	21

Si tenemos dos vectores de 5 elementos cada uno. En uno se almacenan los nombres de personas en el otro las edades de dichas personas.
Decimos que el vector nombres es paralelo al vector edades si en la componente 0 de cada vector se almacena información relacionada a una persona (Juan - 12 años)
Es decir hay una relación entre cada componente de los dos vectores.
Esta relación la conoce únicamente el programador y se hace para facilitar el desarrollo de algoritmos que procesen los datos almacenados en las estructuras de datos.

Problema 1:

Desarrollar un programa que permita cargar 5 nombres de personas y sus edades respectivas. Luego de realizar la carga por teclado de todos los datos imprimir los nombres de las personas mayores de edad (mayores o iguales a 18 años)

Programa:

```
#include<iostream>

using namespace std;

class PersonasEdades {
private:
   char nombres[5][40];
   int edades[5];
public:
   void cargar();
   void mayoresEdad();
};

void PersonasEdades::cargar()
{
   for(int f=0;f < 5;f++)
   {
      cout <<"Ingrese nombre:";
      cin.getline(nombres[f],40);
      cout <<"Ingrese edad:";
```

```
      cin >>edades[f];
      cin.get();
   }
}

void PersonasEdades::mayoresEdad()
{
   cout <<"Personas mayores de edad.";
   cout <<"\n";
   for(int f=0;f < 5;f++)
   {
      if (edades[f] >= 18)
      {
         cout <<nombres[f];
         cout <<"\n";
      }
   }
   cin.get();
}

void main()
{
   PersonasEdades pe;
   pe.cargar();
   pe.mayoresEdad();
}
```

Definimos los dos vectores:

```
private:
   char nombres[5][40];
   int edades[5];
```

Como debemos cargar 5 nombres, luego debemos indicar en el primer subíndice un 5 y un 40 para indicar la cantidad de caracteres por nombre (tener en cuenta que las cadenas de caracteres se almacenan en forma distinta a los int y float).

Mediante un for procedemos a la carga de los elementos de los vectores:

```
  for(int f=0;f < 5;f++)
   {
      cout <<"Ingrese nombre:";
      cin.getline(nombres[f],40);
      cout <<"Ingrese edad:";
      cin >>edades[f];
```

```
    cin.get();
  }
```

Luego de la carga de la edad de la persona se procede a cargar otro nombre por lo que debemos llamar al método get del objeto cin (esto debido a que en el buffer de teclado queda un enter)
Para imprimir los nombres de las personas mayores de edad verificamos cada componente del vector de edades, en caso que sea igual o mayor o 18 procedemos a mostrar el elemento de la misma posición del otro vector:

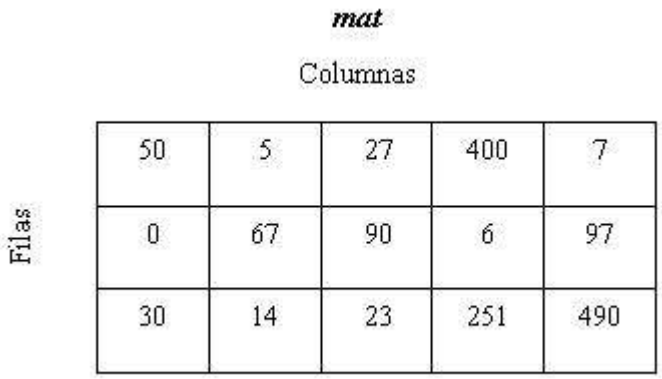
```
void PersonasEdades::mayoresEdad()
{
  cout <<"Personas mayores de edad.";
  cout <<"\n";
  for(int f=0;f < 5;f++)
  {
    if (edades[f] >= 18)
    {
      cout <<nombres[f];
      cout <<"\n";
    }
  }
  cin.get();
}
```

En la main creamos un objeto de la clase PersonasEdades y llamamos a sus dos métodos:

```
void main()
{
  PersonasEdades pe;
  pe.cargar();
  pe.mayoresEdad();
}
```

2.2 Matrices

Una matriz es una estructura de datos que permite almacenar un CONJUNTO de datos del MISMO tipo.
Con un único nombre se define la matriz y por medio de DOS subíndices hacemos referencia a cada elemento de la misma (componente)



Hemos graficado una matriz de 3 filas y 5 columnas. Para hacer referencia a cada elemento debemos indicar primero la fila y luego la columna, por ejemplo en la componente 1,4 se almacena el valor 97.

En este ejemplo almacenamos valores enteros. Todos los elementos de la matriz deben ser del mismo tipo (int, float etc.)

Las filas y columnas comienzan a numerarse a partir de cero, similar a los vectores.

Hemos utilizado anteriormente una matriz para almacenar un conjunto de cadenas de caracteres

Problema 1:

Crear una matriz de 3 filas por 5 columnas con elementos de tipo int, cargar sus componentes y luego imprimirlas.

Programa:

```
#include<iostream>

using namespace std;

class Matriz1 {
private:
   int mat[3][5];
public:
   void cargar();
   void imprimir();
};

void Matriz1::cargar()
{
   for(int f = 0;f < 3;f++)
   {
      for(int c = 0;c < 5;c++)
      {
         cout <<"Ingrese componente:";
```

```
            cin >>mat[f][c];
        }
    }
}

void Matriz1::imprimir()
{
    for(int f = 0;f < 3;f++)
    {
        for(int c = 0;c < 5;c++)
        {
            cout <<mat[f][c] <<" ";
        }
        cout <<"\n";;
    }
    cin.get();
    cin.get();
}

void main()
{
    Matriz1 m;
    m.cargar();
    m.imprimir();
}
```

Para definir una matriz debemos disponer como primer subíndice la cantidad de filas y como segundo subíndice la cantidad de columnas:

```
private:
    int mat[3][5];
```

Para cargar sus 15 componentes (cada fila almacena 5 componentes y tenemos 3 filas)
Lo más cómodo es utilizar un for anidado, el primer for que incrementa el contador "f" lo utilizamos para recorrer las filas y el contador interno llamado "c" lo utilizamos para recorrer las columnas.
Cada vez que se repite en forma completa el for interno se carga una fila completa, primero se carga la fila cero en forma completa, luego la fila uno y finalmente la fila 2.
Siempre que accedemos a una posición de la matriz debemos disponer dos subíndices que hagan referencia a la fila y columna mat[f][c]):

```
void Matriz1::cargar()
{
```

```
   for(int f = 0;f < 3;f++)
   {
      for(int c = 0;c < 5;c++)
      {
         cout <<"Ingrese componente:";
         cin >>mat[f][c];
      }
   }
}
```

Para imprimir la matriz de forma similar utilizamos dos for para acceder a cada elemento de la matriz:

```
void Matriz1::imprimir()
{
   for(int f = 0;f < 3;f++)
   {
      for(int c = 0;c < 5;c++)
      {
         cout <<mat[f][c] <<" ";
      }
      cout <<"\n";;
   }
   cin.get();
   cin.get();
}
```

Cada vez que se ejecuta todas las vueltas del for interno tenemos en pantalla una fila completa de la matriz, por eso pasamos a ejecutar un salto de línea (con esto logramos que en pantalla los datos aparezcan en forma matricial):

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

EVALUACIÓN

En un curso de 4 alumnos se registraron las notas de sus exámenes y se deben procesar de acuerdo a lo siguiente:
a) Ingresar Nombre y Nota de cada alumno (almacenar los datos en dos vectores paralelos)
b) Realizar un listado que muestre los nombres, notas y condición del alumno. En la condición, colocar "Muy Bueno" si la nota es mayor o igual a 8, "Bueno" si la nota está entre

4 y 7, y colocar "Insuficiente" si la nota es inferior a 4.
c) Imprimir cuantos alumnos tienen la leyenda ?Muy Bueno?.
Crear una clase que permita ingresar el nombre de 5 productos y sus respectivos precios. Definir dos vectores paralelos. Mostrar cuantos productos tienen un precio mayor al primer producto ingresado.
Crear y cargar una matriz de 4 filas por 4 columnas. Imprimir la diagonal principal.
Crear y cargar una matriz de 3 filas por 4 columnas. Imprimir la primera fila. Imprimir la última fila e imprimir la primera columna.

UNIDAD 4. PUNTEROS

4.1 Declaración de un puntero

Los punteros son variables que almacenan direcciones de memoria de otra variable.
El concepto de punteros es bastante complejo en un principio y puede llevar a pensar que no tienen una gran utilidad, muy lejos está la realidad.
El manejo de punteros es fundamental para conceptos futuros como la creación y liberación de objetos en tiempo de ejecución de un programa.
Hemos visto las estructuras de datos tipo vector y matriz, pero hay otro tipo de estructuras llamadas estructuras dinámicas que requieren obligatoriamente el empleo de punteros y resuelven otro conjunto de problemas que las estructuras estáticas no pueden.
Un puntero se define de la siguiente manera:

```
<tipo de dato al que apunta> * <nombre del puntero>;
```

4.2. Punteros a tipo de dato numérico

Definición de un puntero que apunta a un entero:

```
int *pe;
```

Asignación de contenido a un puntero:

```
int x=9;
 pe=&x;
```

Un puntero contiene una dirección, aquí le asignamos la dirección de la variable entera x, por eso debemos anteceder el símbolo &.
Podemos asignar un valor a lo apuntado por el puntero:

```
int x=9;
 pe=&x;
 *pe=5;  // la variable x almacena 5
 cout <<x; //5
```

Aquí le asignamos el valor 5 a la dirección a la cual apunta el puntero pe, es decir, a la variable entera x. Para indicar que el valor 5 es asignado a donde apunta el puntero pe, antecedemos al nombre del puntero el símbolo *.

Impresión:

No se puede imprimir el contenido de un puntero, que es una dirección de memoria, lo que imprimimos es el contenido de la variable a la cual apunta el puntero:

```
int x=9;
 pe=&x;
 cout <<*pe; // imprime 9
```

Problema 1:

Confeccionar un programa que defina dos punteros a tipos de datos int y float. Acceder mediante estos punteros a otras variables de tipo int y float.

Programa:

```
#include <iostream>

using namespace std;

void main()
{
   int x = 10;
   int *pe;
   pe = &x;
   cout << x;       // 10
   cout << "\n";
   cout << *pe;      // imprime lo apuntado por pe: 10
   cout << "\n";
   *pe = 5;         //asignamos 5 a lo apuntado por pe
   cout << x;       // 5
   cout << "\n";
   float valor = 10.9;
   float *pf;
   pf = &valor;
   cout << *pf;     //imprime lo apuntado por pf: 10.9
   cin.get();
}
```

4.3 Parámetros de métodos tipos puntero

Hemos visto que cuando un método debe retornar un dato disponemos al principio del método el tipo de dato que retornará (int, float etc.) y dentro del método especificamos con la palabra return el valor que devuelve.

¿Qué sucede si queremos retornar dos valores o más en un solo método? Una forma de solucionar este problema es pasar la dirección de dos variables y que las reciban dos parámetros de tipo puntero, luego por medio de estos punteros modificamos las variables que les pasamos por ejemplo desde la main.

Problema:

Confeccionar una clase que permita administrar un vector de 5 enteros. Definir dos métodos: uno que permita cargar el vector y otro que retorne el mayor y el menor valor del vector. Emplear dos punteros para poder retornar dos datos en un único método.

Programa:

```
#include<iostream>

using namespace std;

class Vector {
   int vec[5];
public:
   void cargar();
   void retornarMayorMenor(int *pmay, int *pmen);
};

void Vector::cargar()
{
   for (int f = 0; f < 5; f++)
   {
      cout << "Ingrese componente:";
      cin >> vec[f];
   }
}

void Vector::retornarMayorMenor(int *pmay, int *pmen)
{
   *pmay = vec[0];
   *pmen = vec[0];
   for (int f = 1; f < 5; f++)
   {
      if (vec[f]>*pmay)
      {
         *pmay = vec[f];
      }
      else
      {
```

```
        if (vec[f]<*pmen)
        {
           *pmen = vec[f];
        }
      }
   }
}


void main()
{
   Vector vector1;
   vector1.cargar();
   int ma, me;
   vector1.retornarMayorMenor(&ma, &me);
   cout << "El elemento mayor del vector es:" << ma << "\n";
   cout << "El elemento menor del vector es:" << me << "\n";
   cin.get();
   cin.get();
}
```

Es muy importante entender cómo funciona el método retornarMayorMenor. En la main definimos dos variables de tipo entera:

```
int ma, me;
```

Y llamamos al método rotornarMayorMenor enviándole la dirección de estas dos variables (recordemos que mediante el operador & se accede a la dirección de una variable):

```
vector1.retornarMayorMenor(&ma, &me);
```

Como estamos enviando direcciones de variables el método debe tener parámetros de tipo puntero como lo son *pmay y *pmen:

```
void Vector::retornarMayorMenor(int *pmay, int *pmen)
```

Mediante estos dos punteros estamos accediendo a las variables definidas en la main llamadas: ma y me.

Cargamos en las dos variables definidas en la main (accediendo a las mismas por los punteros) la primer componente del vector:

```
*pmay = vec[0];
 *pmen = vec[0];
```

Luego mediante un for recorremos el resto del vector para ver si hay alguna componente más grande a la que hemos considerado hasta ahora la mayor (lo mismo hacemos para la menor componente):

```
for (int f = 1; f < 5; f++)
 {
    if (vec[f]>*pmay)
    {
       *pmay = vec[f];
    }
    else
    {
       if (vec[f]<*pmen)
       {
          *pmen = vec[f];
       }
    }
 }
```

Como podemos observar luego en la main cuando imprimimos los contenidos de las variables ma y me aparecen en pantalla el valor mayor y menor del vector:

```
cout << "El elemento mayor del vector es:" << ma << "\n";
cout << "El elemento menor del vector es:" << me << "\n";
```

Como vemos en ningún momento asignamos valores directamente a las variables ma y me, sino que lo hicimos con los parámetros que apuntan a dichas variables.

EVALUACIÓN

Definir tres variables enteras e inicializarlas con los valores 5,10 y 15. Luego definir una variable puntero a entero. Hacer que dicha variable apunte sucesivamente a las distintas variables definidas previamente e imprimir su contenido.
Definir dos variables float y un puntero a un tipo de dato float. Hacer que el puntero guarde sucesivamente las direcciones de la primera y segunda variable y cambiar el contenido de las mismas por asignación. Imprimir las dos variables de tipo float.

UNIDAD 5. CLASES

5.1 Definición y declaración

En C++ podemos definir un método que se ejecute inicialmente y en forma automática. Este método se lo llama constructor.
El constructor tiene las siguientes características:

Tiene el mismo nombre de la clase.
Es el primer método que se ejecuta.
Se ejecuta en forma automática.
No puede retornar datos.
Se ejecuta una única vez.
Un constructor tiene por objetivo inicializar atributos.

Problema:
Se desea guardar los sueldos de 5 operarios en un vector. Realizar la carga del vector en el constructor.
Programa:

```
#include<iostream>

using namespace std;

class Operarios {
private:
   float sueldos[5];
public:
   Operarios();
   void imprimir();
};

Operarios::Operarios()
{
   cout <<"Carga de sueldos." <<"\n";
   for(int f=0;f < 5;f++)
   {
      cout <<"Ingrese el sueldo:";
      cin >>sueldos[f];
   }
}

void Operarios::imprimir()
{
   for(int f = 0; f < 5; f++)
   {
      cout <<sueldos[f] <<"\n";;
   }
   cin.get();
   cin.get();
}
```

```
void main()
{
   Operarios op;
   op.imprimir();
}
```

Problema:

Plantear una clase llamada Alumno y definir como atributos su nombre y su edad. En el constructor realizar la carga de datos. Definir otros dos métodos para imprimir los datos ingresados y un mensaje si es mayor o no de edad (edad >=18)

Programa:

```
#include<iostream>

using namespace std;

class Alumno {
private:
   char nombre[40];
   int edad;
public:
   Alumno();
   void imprimir();
   void esMayorEdad();
};

Alumno::Alumno()
{
   cout <<"Ingrese nombre:";
   cin.getline(nombre,40);
   cout <<"Ingrese la edad:";
   cin >>edad;
}

void Alumno::imprimir()
{
   cout <<"Nombre:" <<nombre <<"\n";
   cout <<"Edad:" <<edad <<"\n";
}

void Alumno::esMayorEdad()
{
```

```
   if (edad >= 18)
   {
      cout <<"es mayor de edad.";
   }
   else
   {
      cout <<"no es mayor de edad.";
   }
   cin.get();
   cin.get();
}

void main()
{
   Alumno alumno1;
   alumno1.imprimir();
   alumno1.esMayorEdad();
}
```

Declaramos la clase Alumno, sus dos atributos y definimos el constructor con el mismo nombre de la clase:

```
class Alumno {
private:
   char nombre[40];
   int edad;
public:
   Alumno();
   void imprimir();
   void esMayorEdad();
};
```

En el constructor realizamos la carga de los dos atributos por teclado:

```
Alumno::Alumno()
{
   cout <<"Ingrese nombre:";
   cin.getline(nombre,40);
   cout <<"Ingrese la edad:";
   cin >>edad;
}
```

En la main el constructor se llama en forma automática cuando definimos un objeto de la clase Alumno:

```
void main()
{
   Alumno alumno1;
```

Los otros dos métodos deben llamarse por su nombre y en el orden que necesitemos:

```
  alumno1.imprimir();
   alumno1.esMayorEdad();
```

5.2 Clases Amigas

Otra posibilidad en C++ es que una clase sea amiga. Esto hace que todos los métodos de la clase amiga tengan acceso a los atributos privados y protegidos.

Problema:

Declarar dos clases independientes: Nodo y ListaGenerica. En la clase nodo definir el atributo info, los punteros sig y ant e indicar que tiene una clase amiga llamada ListaGenerica por lo que tendrá acceso a sus atributos.

Programa:

```
#include<iostream>
using namespace std;
class Nodo {
   int info;
   Nodo *ant, *sig;
public:
   Nodo(int x){ info = x; };
   friend class ListaGenerica;
};


class ListaGenerica {
   Nodo *raiz;
public:
   ListaGenerica() { raiz = NULL; };
   ~ListaGenerica();
   void insertarPrimero(int x);
   void imprimir();
};

ListaGenerica::~ListaGenerica()
{
   Nodo *reco = raiz;
   Nodo *bor;
```

```
   while (reco != NULL)
   {
      bor = reco;
      reco = reco->sig;
      delete bor;
   }
}

void ListaGenerica::insertarPrimero(int x)
{
   Nodo *nuevo = new Nodo(x);
   nuevo->ant = NULL;
   if (raiz == NULL)
   {
      nuevo->sig = NULL;
      raiz = nuevo;
   }
   else
   {
      nuevo->sig = raiz;
      raiz->ant = raiz;
      raiz = nuevo;
   }
}

void ListaGenerica::imprimir()
{
   Nodo *reco = raiz;
   while (reco != NULL)
   {
      cout << reco->info << "-";
      reco = reco->sig;
   }
   cout << "\n";
}


void main()
{
   ListaGenerica *lista1 = new ListaGenerica();
   lista1->insertarPrimero(10);
```

```
   lista1->insertarPrimero(20);
   lista1->insertarPrimero(5);
   lista1->imprimir();
   delete lista1;
   cin.get();
}
```

Declaramos la clase Nodo definiendo los atributos info,ant y sig en la parte privada, esto significa que no podrán ser accedidos desde métodos fuera de la clase, pero hacemos la excepción con la clase ListaGenerica indicando que se trata de una clase amiga (friend class):

```
class Nodo {
   int info;
   Nodo *ant, *sig;
public:
   Nodo(int x){ info = x; };
   friend class ListaGenerica;
};
```

Esta característica hace que dentro de la clase ListaGenerica podamos acceder a los atributos privados y protegidos sin problemas:

```
  Nodo *nuevo = new Nodo(x);
   nuevo->ant = NULL;
```

Si bien se podrían haber definido métodos para acceder a dichos atributos lo más eficiente a nivel de ejecución es acceder a los mismos directamente.

En la declaración de la clase ListaGenerica no hay ninguna referencia a la otra clase, solo definimos el puntero raiz que tendrá la dirección al primer nodo de la lista:

```
class ListaGenerica {
   Nodo *raiz;
public:
   ListaGenerica() { raiz = NULL; };
   ~ListaGenerica();
   void insertarPrimero(int x);
   void imprimir();
};
```

Ahora en el método insertarPrimero podemos crear un objeto de la clase Nodo y acceder a sus tres atributos privados sin problema mediante el operador ->:

```
void ListaGenerica::insertarPrimero(int x)
{
   Nodo *nuevo = new Nodo(x);
```

```
    nuevo->ant = NULL;
    if (raiz == NULL)
    {
       nuevo->sig = NULL;
       raiz = nuevo;
    }
    else
    {
       nuevo->sig = raiz;
       raiz->ant = raiz;
       raiz = nuevo;
    }
}
```

5.3 Herencia

La herencia significa que se pueden crear nuevas clases partiendo de clases existentes, que tendrá todos los atributos y los métodos de su 'superclase' o 'clase padre' y además se le podrán añadir otros atributos y métodos propios.

Clase padre
Clase de la que desciende o deriva una clase. Las clases hijas (descendientes) heredan (incorporan) automáticamente los atributos y métodos de la clase padre.

Subclase
Clase que desciende de otra. Hereda automáticamente los atributos y métodos de su superclase. Es una especialización de otra clase. Admiten la definición de nuevos atributos y métodos para aumentar la especialización de la clase.

Veamos algunos ejemplos teóricos de herencia:

1) Imaginemos la clase Vehículo. ¿Qué clases podrían derivar de ella?

Vehículo

Colectivo Moto Auto

FordK Renault 19

Siempre hacia abajo en la jerarquía hay una especialización (las subclases añaden nuevos atributos y métodos)

2) Imaginemos la clase Software. Qué clases podrían derivar de ella?

Software

De_Aplicacion De_Base

Procesador_Texto Planilla_De_Calculo Sistema_Operativo

Word WordPerfect Excel Lotus123 Linux Windows Mac OS X

El primer tipo de relación que habíamos visto entre dos clases, es la de colaboración. Recordemos que es cuando una clase contiene un objeto de otra clase como atributo. Cuando la relación entre dos clases es del tipo "...tiene un..." o "...es parte de...", no debemos implementar herencia. Estamos frente a una relación de colaboración de clases no de herencia.

Si tenemos una ClaseA y otra ClaseB y notamos que entre ellas existe una relación de tipo "... tiene un...", no debe implementarse herencia sino declarar en la clase ClaseA un atributo de la clase ClaseB.

Por ejemplo: tenemos una clase Auto, una clase Rueda y una clase Volante. Vemos que la relación entre ellas es: Auto "...tiene 4..." Rueda, Volante "...es parte de..." Auto; pero la clase Auto no debe derivar de Rueda ni Volante de Auto porque la relación no es de tipo-subtipo sino de colaboración. Debemos declarar en la clase Auto 4 atributos de tipo Rueda y 1 de tipo Volante.

Luego si vemos que dos clase responden a la pregunta ClaseA "..es un.." ClaseB es posible que haya una relación de herencia.

Por ejemplo:

Auto "es un" Vehiculo
Circulo "es una" Figura
Mouse "es un" Dispositivo de Entrada
Suma "es una" Operación

Problema 1:

Ahora plantearemos el primer problema utilizando herencia. Supongamos que necesitamos implementar dos clases que llamaremos Suma y Resta. Cada clase tiene como atributo valor1, valor2 y resultado. Los métodos a definir son cargar1 (que inicializa el atributo valor1), carga2 (que inicializa el atributo valor2), operar (que en el caso de la clase "Suma" suma los dos atributos y en el caso de la clase "Resta" hace la diferencia entre valor1 y

valor2, y otro método mostrarResultado)

Si analizamos ambas clases encontramos que muchos atributos y métodos son idénticos. En estos casos es bueno definir una clase padre que agrupe dichos atributos y responsabilidades comunes.

La relación de herencia que podemos disponer para este problema es:

Operacion

Suma Resta

Solamente el método operar es distinto para las clases Suma y Resta (esto hace que no lo podamos disponer en la clase Operacion), luego los métodos cargar1, cargar2 y mostrarResultado son idénticos a las dos clases, esto hace que podamos disponerlos en la clase Operacion. Lo mismo los atributos valor1, valor2 y resultado se definirán en la clase padre Operacion.

Programa:

```
#include<iostream>

using namespace std;

class Operacion {
protected:
   int valor1;
   int valor2;
   int resultado;
public:
   void cargar1();
   void cargar2();
   void mostrarResultado();
};


class Suma: public Operacion{
public:
   void operar();
};


class Resta : public Operacion {
public:
```

```
   void operar();
};


void Operacion::cargar1()
{
   cout << "Ingrese primer valor:";
   cin >> valor1;
}

void Operacion::cargar2()
{
   cout << "Ingrese segundo valor:";
   cin >> valor2;
}

void  Operacion::mostrarResultado()
{
   cout <<resultado<<"\n";
}

void Suma::operar()
{
   resultado = valor1 + valor2;
}

void Resta::operar()
{
   resultado = valor1 - valor2;
}


void main()
{
   Suma suma1;
   suma1.cargar1();
   suma1.cargar2();
   suma1.operar();
   cout << "La suma de los dos valores es:";
   suma1.mostrarResultado();
```

```
    Resta resta1;
    resta1.cargar1();
    resta1.cargar2();
    resta1.operar();
    cout << "La diferencia de los dos valores es:";
    resta1.mostrarResultado();

    cin.get();
    cin.get();
}
```

La clase Operación define los tres atributos:

```
class Operacion {
protected:
    int valor1;
    int valor2;
    int resultado;
public:
    void cargar1();
    void cargar2();
    void mostrarResultado();
};
```

Este proyecto lo puede descargar en un zip desde este enlace : Herencia1.zip

Ya veremos que definimos los atributos con este nuevo modificador de acceso (protected) para que la subclase tenga acceso a dichos atributos. Si los definimos private las subclases no pueden acceder a dichos atributos.

Los métodos de la clase Operacion son:

```
void Operacion::cargar1()
{
    cout << "Ingrese primer valor:";
    cin >> valor1;
}
void Operacion::cargar2()
{
    cout << "Ingrese segundo valor:";
    cin >> valor2;
}
```

```
void  Operacion::mostrarResultado()
{
   cout <<resultado<<"\n";
}
```

Ahora veamos cómo es la sintaxis para indicar que una clase hereda de otra:

```
class Suma: public Operacion{
public:
   void operar();
};
```

Utilizamos el caracter dos puntos y seguidamente el nombre de la clase padre precedido por el modificador public (con esto estamos indicando que todos los métodos y atributos de la clase Operación son también métodos de la clase Suma)

Luego la característica que añade la clase Suma es el siguiente método:

```
void Suma::operar()
{
   resultado = valor1 + valor2;
}
```

El método operar puede acceder a los atributos heredados (siempre y cuando los mismos se declaren protected, en caso que sean private si bien lo hereda de la clase padre solo los pueden modificar métodos de dicha clase padre)

Ahora podemos decir que la clase Suma tiene cuatro métodos (tres heredados y uno propio) y 3 atributos (todos heredados)

Luego en la main creamos un objeto de la clase Suma y otro de la clase Resta:

```
void main()
{
   Suma suma1;
   suma1.cargar1();
   suma1.cargar2();
   suma1.operar();
   cout << "La suma de los dos valores es:";
   suma1.mostrarResultado();

   Resta resta1;
   resta1.cargar1();
   resta1.cargar2();
```

```
    resta1.operar();
    cout << "La diferencia de los dos valores es:";
    resta1.mostrarResultado();

    cin.get();
    cin.get();
}
```

Podemos llamar tanto al método propio de la clase Suma "operar()" como a los métodos heredados. Quien utilice la clase Suma solo debe conocer que métodos públicos tiene (independientemente que pertenezcan a la clase Suma o a una clase superior), no podemos acceder a los métodos privados y protegidos.

La lógica es similar para declarar la clase Resta.

La clase Operación agrupa en este caso un conjunto de atributos y métodos comunes a un conjunto de subclases (Suma, Resta). No tiene sentido definir objetos de la clase Operacion.

El planteo de jerarquías de clases es una tarea compleja que requiere un perfecto entendimiento de todas las clases que intervienen en un problema, cuales son sus atributos y responsabilidades.

EVALUACIÓN

Confeccionar una clase Persona que tenga como atributos el nombre y la edad. Definir como responsabilidades un método que cargue los datos personales y otro que los imprima.
Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir su sueldo.
Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a su método

UNIDAD 6.ENTRADA, SALIDA, ARCHIVOS EN DISCO

6.1 Conceptos generales

La entrada y salida por archivos es uno de los temas más importantes para la programación, es necesario para poder programar desde una pequeña agenda hasta una completa base de datos, pero antes de entrar de lleno a este tema, y aprovechando los conocimientos adquiridos hasta el momento, quiero mostrar el uso de la memoria dinámica.

6.2 Memoria dinámica

Hasta el momento no contamos con una forma de "administrar" la memoria utilizada en nuestros programas, cuando declaramos una variable se asigna memoria para almacenar datos dentro de ella, y ésta no se destruye hasta que termina el bloque en el que fue declarada, se crea y se destruye cada que se pasa por ese bloque, quizá puede parecer poco importante, pero cuando se cuentan con grandes cantidades de datos, es necesario tener un mejor control de lo que se pone en memoria. Una de las formas en que podemos asegurarnos que una variable declarada dentro de un bloque no sea borrada al término de éste, es mediante la utilización del calificador static. De ésta manera, la variable perdurará hasta el término de todo el programa.

static tipo_variable mi_variable;

Pero en algunos casos esto nos será insuficiente, vamos a necesitar "destruir" variables antes de terminar el programa, para hacer esto contamos con los operadores new y delete.

Se puede utilizar el operador new para crear cualquier tipo de variables, en todos los casos devuelve un puntero a la variable creada. En el momento en que se quiera borrar esta variable deberemos utilizar el operador delete, todas las variables creadas mediante el operador new deben de ser borradas con el otro operador.

En el siguiente ejemplo utilizaremos estos dos operadores para mostrar sus ventajas.

```
#include<iostream.h>
int main(){
	char *cadena2;
	int fincontador=10;
	cout<<"programa de prueba"<<endl;
	for(int i=0;i<fincontador;++i){
		cout<<"contador funcionando "<<i<<endl;
	}
	int respuesta;
	cout<<"crear variable?, 1 para si";
	cin>>respuesta;
	char cadena[3]="hi";
	if(respuesta==1){
		int LongCad;
		cout<<"variable creada"<<endl;
		cout<<"longitud de cadena: ";
		cin>>LongCad;
		cadena2=new char[LongCad];
		cout<<"escribe: "<<endl;
		cin.ignore();
		cin.getline(cadena2,LongCad);
```

```
        cout<<"cadena escrita: "<<cadena2<<endl;
    }else{
        char cadena[10]="rayos, no";
    }
    cout<<cadena<<endl;
 cout<<cadena2<<endl;
    delete cadena2;
    cin.ignore();
    cin.get();
    return 0;
}
```

Las partes que nos interesan de este programa son la declaración de la variable tipo apuntador que se encuentra al principio del main, la asignación de espacio dentro del bloque if, y por último la destrucción de nuestra variable lineas antes del término del programa. Si hubiésemos declarado el apuntador dentro del bloque if, cuando se saliera de éste entonces la variable ya no existiría y habría un error en tiempo de compilación. El uso de la memoria dinámica nos será muy útil en muchos casos, en otros quizá prefiramos hacerlo de la manera más común. Estos operadores funcionan tanto para variables simples como para compuestas (definidas por el programador). De nueva cuenta debo mencionar a las clases, porque en éstas se maneja mucho este concepto, y de nuevo invito al lector a que, terminando esta lectura, se adentre más en el mundo de la programación consultando otra bibliografía. Si se anima a trabajar con clases lo felicito, antes quisiera darle una idea manejando estructuras con funciones. El siguiente programa es una modificación del último que vimos en el capítulo anterior, la diferencia está en el manejo de los operadores descritos hace unos párrafos, y por ende en el manejo de apuntadores a estructuras.

6.3 Trabajar con ficheros

Para poder trabajar los ficheros como flujos es necesario incluir la librería fstream.h, y según la utilización que queramos dar a este fichero (lectura o escritura) deberemos declarar el tipo de flujo. Para crear un archivo de salida declaramos una variable de tipo ofstream, el cual ya está declarado dentro de nuestra librería. Es mejor ver un ejemplo de base.

```
#include<fstream.h>
int main(){
        ofstream archivo;
        archivo.open("miarchivo.txt");
        archivo<<"hola desde este archivo\n";
        archivo<<"ya he escrito algo\n";
        archivo.close();
}
```

Aquí declaramos a “archivo” como variable tipo ofstream, y posteriormente utilizamos su función miembro open para asociarla a un archivo, se puede asociar directamente en la declaración de la siguiente manera: ofstream archivo(“miarchivo.txt”);

6.4 Metodos de apertura de un archivo
Los metodos de apertura de archivos se detallan a continuacion:
ios::app Se escribe al final de archivo
ios::out El archivo se abre para escritura
ios::trunc Si el archivo existe se eliminará su contenido
ios::in El archivo se abre para lectura, el archivo original no será modificado
ios::binary El archivo se abre en modo binario
Además de éstas, existen otras funciones que nos serán muy útiles para que no sea tan secuencial la forma en la que leemos o escribimos el archivo.
tellg() Devuelve la posición de lectura s
eekg() Se coloca dentro del flujo en la posición pasada como argumento para poder leer.
tellp() Devuelve la posición de escritura
seekp() Se coloca dentro del flujo en la posición pasada como argumento para poder escribir.

EVALUACIÓN

Realizar una aplicación que permita crear y abrir un archivo con el siguiente contenido:
hola como estas?
bien que haces?
nada pues ponte a hacer algo, a que te dedicas?
estudio y que estudias? i
nformatica
ha que bueno!!, de donde eres? ixtapaluca que interesante, hay algo que quieras preguntarme?
como te llamas?
acaso eso importa?
como te llamas pregunta de nuevo

REFERENCIAS

Deitel, Harvey M.; Deitel, Paul J. C++, cómo programar Edit. Pearson Educación. 4ª ed., 2003. 1320 pp

Schildt, Herbert. C, Manual de referencia Edit. Osborne McGraw-Hill. 4ª ed., 2000. 709 pp.

http://profesores.fi-b.unam.mx/carlos/lcpi/p09/OPERADORES%20EN%20%20C++.pdf

https://www.tutorialesprogramacionya.com/cmasmasya/index.php?inicio=0