UNIDAD 3

Uso de Clases Predefinidas.

1.- **INTRODUCCION.**

Los tipos de datos definidos por el usuario o clases es lo que distingue al C++ de los lenguajes procedurales tradicionales. Una clase es un nuevo tipo de dato que se crea para resolver un problema en particular. Una vez que una clase es creada, puede usarse sin la necesidad de conocer como se implementó o como trabaja.

En la Unidad manejamos las clases como si se tratara de otro tipo de dato disponible para ser usado en el desarrollo de programas.

Una vez creadas las clases, éstas pueden ser empaquetadas en archivos, que algunos les llaman librerías.

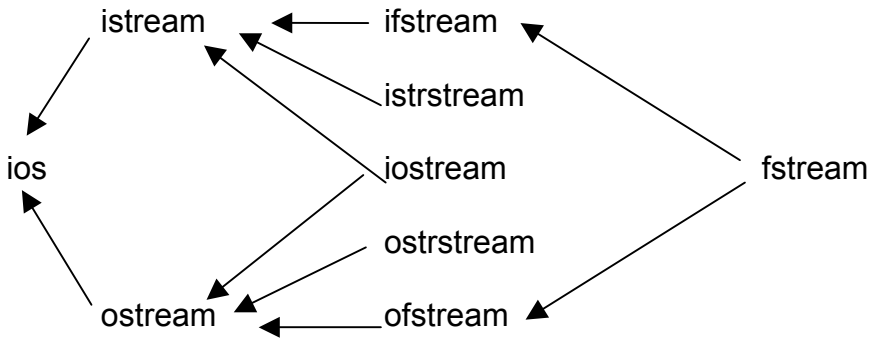
En esta Unidad vamos a usar la librería de clases **iostream**, la cual acompaña a la implementación del paquete. La librería **iostream** es una manera muy práctica de leer información de archivos o del teclado y escribir en archivos o en la pantalla. Vamos a ver lo fácil que resulta utilizar una librería predefinida de clases.

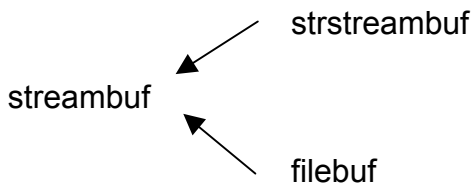
2.- **QUE ES UN stream?.**

Un **stream** es una abstracción referida como cualquier flujo de datos de una FUENTE a un DESTINO, en ocasiones también se maneja el sinónimo de extracción u obtención cuando hablamos de caracteres de entrada a una FUENTE e inserción o almacenamiento, cuando hablamos de salida de caracteres hacia un DESTINO.

3.- **La librería iostream**.

3.a) Derivación de la clase ios.





3.b) Descripción de las clases de una libreria iostream típica.

CLASE	DESCRIPCION
filebuf	Nos permite el manejo de caracteres mediante un archivo tanto para inserción como para extracción.
fstream	Se usa para construir archivos mediante la definición de objetos stream, tanto para leer como para escribir información en disco.
ifstream	Nos brinda rentrada formateada para manejar archivos en disco usando un objeto filebuf.
ios	Es la clase base, en algunas implementaciones se encuentran declaradas variables de estado y funciones miembro para manipular dichas variables. No necesariamente es una clase abstracta, sin embargo no se debe intentar instanciar objetos de dicha clase.
iostream	No pertenece al estandar ANSI C++, sin embargo, se encuentra disponible en muchos compiladores para construir objetos que puedan ser manejados tanto en inserción como en extracción.
istream	Nos brinda una interface para transferir caracteres formateados como datos desde un objeto del tipo streambuf. El objeto fuente **cin** es del tipo istream (o de alguna clase derivada).
istrstream	No brinda operaciones de entrada para objetos del tipo strstreambuf, usados en memoria.
ofstream	Nos brinda una salida formateada a disco, usando objetos del tipo filebuf.
ostream	Nos brinda una interface para transferir datos como caracteres formateados a un objeto del tipo streambuf. Los objetos cout, cerr y clog son de este tipo (o de una clase derivada).
ostrstream	Nos brinda operaciones de salida para objetos del tipo streambuf, usados en memoria.
streambuf	Es la clase base de filebuf y strstreambuf, nos brinda funciones que manejan buffer de E/S para lectura y escritura. Puede o no ser implementada en una clase abstracta, pero

	de ninguna manera deberán instanciarse objetos de esta clase.directamente en un programa.
strstreambuf	Es un manejador de cadenas, se implementa como un puntero a un arreglo del tipo char Esta clase es usada por las clases istrstream y ostrstream para menejar cadenas en memoria.

3.c) Objetos para menejo del teclado y pantalla.

cin La entrada estandar, equivalente a stdin.

cout La salida estandar, equivalente a stdout.

cerr La salida estandar de error, equivalente a stderr. La diferencia con **cout** es que no puede ser redireccionado

clog Es una versión de **cerr**, pero con buffer. Este hecho puede incrementar la velocidad de muestreo en la pantalla.

Estos stream son abiertos de manera automática cuando se inicia el programa, nos brindan la interface entre el programa y el usuario.
El **cin** es asociado con el teclado.
El **cout**, **cerr** y **clog** se asocian con la pantalla.

4.- **La salida.**

El stream de salida se ejecuta con la inserción del operador **<<**. El operador estandar de corrimiento a la izquierda es sobrecargado para funcionar como operador de salida, su operador de la izquierda es un objeto del tipo clase ostream, el operador de la derecha es cualquier tipo para el cual el stream de salida fué definido. El operador **<<** es sobrecargado por el tipo **type**, llamado: **type insert**, por ejemplo:

cout << "Hola muchachas\n";

Esta sentencia escribe la constante de cadena a **cout** (stream de salida, normalmente la pantalla). El operador **<<** es del tipo: **char *insert**.

Veamos un ejemplo:

```
// Programa # 1, uso simple de cout
#include <iostream.h>

void main()
{
```

```
    /* printf("Hola muchachas\n"); */
    cout << "Hola muchachas" << "\n";
}
```

OBSERVACIONES.

En primer lugar, note que los comentarios se pueden representar por medio de [*//*], el comentario se termina con el retorno de carro.

Este programa fué compilado usando el TurboC++, por lo que las clases de manejo de stream se encuentran definidas en el archivo de cabecera **iostream.h**, por lo cual, para usar **cout** éste se debe incluir como cabecera del programa.

Podemos notar en el programa, que el C++ acepta ambos tipos de comentarios [*//*] y [*/* */*], ésto se usa para escribir la sentencia equivalente en "C", para la misma salida que **cout**.

En el programa el salto de línea, en la sentencia **cout** se escribe de manera encadenada usando el operador **<<**, sin embargo, también se puede escribir como parte de la cadena.

```
cout << "Hola muchachas\n";
```

4.a) Encadenamiento de inserciones.

El operador **<<** es asociado a la izquierda y regresa una referencia al objeto **ostream** para el cual es invocado. Esto nos permite varias inserciones en cadena, de la siguiente manera:

```
void muestra(int a, double b)
{
        cout << "a = " << a << ", b = ", << b << "\n";
}
```

Suponiendo, fueron recibidos los valores, la salida a pantalla nos resulta:

```
a = 30, b = 45.658
```

Note que la sobrecarga del operador << no altera su precedencia, de tal manera que podemos escribir:

```
cout << "suma = " << c+d << "\n";
```

en este caso no son necesarios los paréntesis, sin embargo, en la siguiente sentencia, si son necesarios, de lo contrario se produce un error.

```
cout << (c&d) << "\n";
```

Esto se debe a que de acuerdo a la sintaxis de la expresión, se debe primero tomar el operador << y luego el operando (c&d), que en este caso se comporta como un argumento para el operador <<.

```
// Programa # 2, presedencia de operadores

#include <iostream.h>


void main()
{
     int a = 10, b = 5;

     cout << "\n";
     cout << "La suma:           " << a+b << "\n";
     cout << "La intersección:   " << (a&b) << "\n";
}
```

OBSERVACIONES.

Sin los paréntesis, nos aparece el siguiente error:

// ERROR en a&b, si se usa sin paréntesis.
// Uso ilegal de apuntador en función main.

Considera que estamos manejando al operador [&] como un apuntador, sin embargo no encuentra relación con ninguna variable puntero.

4.b) Tipos definidos para el operador **<<**.

Los tipos soportados por el operador **<<**, son:

- signed char
- unsigned char
- signed short
- unsigned short
- signed int
- unsigned int
- signed long
- unsigned long
- char *(cadena)
- float
- double
- long double
- void *

Los enteros (int) son manejados de acuerdo a las reglas definidas para printf, y pueden modificarse usando las banderas (flags) definidas en **ios**. veamos el siguiente programa:

```
// Programa # 3, reglas de default

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

void main()
{
    int i = 20000;
    long l = 200000;
    float f = 45.5678;
    char cad[] = "Clases Predefinidas";

    printf( "\nUsando printf con enteros : %d, %ld", i, l);
    cout << "\nUsando cout con enteros   : " << i << ", " << l;
    printf( "\nUsando printf con float   : %f", f);
    cout << "\nUsando cout con float     : " << f;
    printf( "\nUsando printf con [char *]: %s", cad);
    cout << "\nUsando cout con [char *]  : " << cad;
    printf( "\nUsando printf con [void *]: %p", &i);
    cout << "\nUsando cout con [void *]  : " << &i << "\n";
}
```

OBSERVACIONES.

La salida nos resulta:

Usando cout con enteros   : 20000, 200000
Usando cout con float   : 45.567799
Usando cout con [char *] : Clases Predefinidas
Usando cout con [void *] : 0x3401fff4

Usando printf con enteros : 20000, 200000
Usando printf con float   : 45.567799
Usando printf con [char *]: Clases Predefinidas
Usando printf con [void *]: FFF4

Note que la salida es la misma excepto para &i, donde en el caso de printf, solo nos muestra el offset, en cambio para cout muestra tanto la dirección de segmento como la de offset y nos indica que su valor esta dado en hexadecimal [0x]. El orden de la salida primero para cout y luego para printf, obedece al hecho de haber mandado la salida a un archivo. (PLCP03.RPT). Si no se redirecciona la salida es de acuerdo al código del programa.

NOTA: La explicación anterior, se hace solo como observación al redireccionar la salida.

Antes de seguir analizando funciones y manipuladores para la salida, veamos los conceptos básicos para entrada de datos.

5.- **La entrada**.

El stream de entrada es similar al de salida solo que usa el operador de corrimiento a la derecha sobrecargado **>>**, al cual se le denomina **operador de extracción** (obtiene de) o **extractor**. Este operador nos proporciona una alternativa más clara compacta y legible que su homologo, scanf. El operador **>>** es un objeto de la clase tipo **istream**. Igual que con la salida, el operador **>>** puede manejar cualquier tipo para el cual el stream de entrada se definió.

Los tipos listados anteriormente para el operador de salida son válidos también para el operador de entrada, recuerde que se tiene la libertad de sobrecargar este operador para el manejo de un tipo definido por el usuario. El operador **>>** sobrecargado para un **tipo type**, se le llama el **type extractor**. ejemplo:

```
cin >> a;
```

La conversión realizada de las cadenas de entrada va a depender; del tipo de **a**, como su extractor es definido, y del estado de las banderas. Ejemplo de uso:

```
// Programa # 4, uso simple de cin
#include <iostream.h>

void main()
{
     char cad[80];
     int num;
     float fnum;

     cout << "\nDame tu nombre: ";
     cin >> cad;
     cout << "Dame un número entero: ";
     cin >> num;
     cout << "Dame un número flotante: ";
     cin >> fnum;
     cout << "\n" << "Te llamas: " << cad << "\n";
     cout << "Su producto es: " << num*fnum << "\n";
     cout << "La dirección de la cadena es: " << &cad << "\n";
}
```

OBSERVACIONES.

Al correr el programa nos resulta:
NOTA: Las palabras subrayadas significan datos dados del teclado.

Dame tu nombre: Velvet
Dame un número entero: 45
Dame un número flotante: 45.7896
Te llamas: Velvet
Su producto es: 2060.532017
La dirección de la cadena es: 0x34a3ff9c

En este caso estamos usando la sobrecarga del operador **>>** para una cadena, un entero y un float.

5.a) Encadenamiento de extractores.

De la misma manera que sucedió con el operador de inserción **<<**, podemos hacerlo con el operador de extracción **>>**, el cual es asociado a la izquierda y nos regresa su operando a la izquierda. El operando izquierdo es una referencia al objeto **istream**, por él cual es invocado, ésto nos permite que varias operaciones de entrada sean invocadas en una sentencia. Consideremos el siguiente ejemplo:

```
// Programa 4A, encadenamiento de extractores.
#include <iostream.h>

void main()
{
    int i;
    float f;
    char cad[20];

    cout << "\nDame un entero:\nUn real:\nUna cadena:";
    cout << "\nSepara los datos por un espacio\n";
    cin >> i >> f >> cad;
    cout << "\nEl entero: " << i;
    cout << "\nEl real  : " << f;
    cout << "\nLa cadena: " << cad;
}
```

OBSERVACIONES.

Al correr el programa se obtiene:
NOTA: Las palabras subrayadas significan datos dados del teclado.

Dame un entero:
Un real:
Una cadena:
Separa los datos por un espacio
67 56.786 Velvet

El entero: 67
El real : 56.785999
La cadena: Velvet

Por medio del encadenamiento de extracción pudimos introducir por teclado; un entero, un real y una cadena, todas sobre la misma sentencia.

La línea:

```
cin >> i >> f >> cad;
```

ocasiona que los espacios en blanco sean brincados; Los digitos leidos del teclado son convertidos internamente a una forma binaria y guardados en variables.

5.b) Extracción de enteros.

Para los tipos **short**, **int** y **long** (**signed** y **unsigned**) la acción por default de **>>** es saltar los espacios en blanco, conviertiendo a un valor entero los caracteres leidos de la entrada, hasta que encuentre uno que no es parte legal de la representación del tipo. Es posible expresar también caracteres en octal y hexadecimal anteponiendo al número un [0] para octal y [0x] para el hexadecimal.

5.c) Extracción de reales.

Para los tipos **float** y **double** el efecto del operador **>>** es el salto de espacios en blanco y convertir los caracteres leidos del teclado a un valor de punto flotante, hasta encontrar uno que no sea parte válida de la representación del punto flotante.

5.d) Extracción de caracteres.

Para el tipo **char** (signed y unsigned) el efecto del operador **>>** es el salto de los espacios en blanco almacenando el primer caracter que no sea espacio en blanco, terminando la lectura cuando encuentre un espacio en blanco.

Para leer cadenas que involucren espacios en blanco es necesario usar una función miembro de **cin**.

6.- **Funciones miembro, y manipuladores de cout y cin.**

Funciones miembro y manipuladores de cout y cin nos dan una mayor versatilidad para el manejo de salidas y entradas.

6.a) La función miembro **get**.

La función get tiene 2 formas; una nos permite ir extrayendo caracter por caracter, la otra nos extrae una cadena completa. (esta versión no esta disponible en TurboC++ Ver 1.01, aunque lo debería estar, ya que sí se encuentra su prototipo).

Función get. Miembro de istream.

Descripcion Extrae un solo caracter, colocandolo en la referencia a caracter definida.

Sintaxis istream get(unsigned char&)
istream get(signed char&)

Ejemplo:

```
// Programa # 5, uso de la función get

#include <iostream.h>
# define LONG 80

void main()
{
     char cad[LONG], a = '\0';

     cout << "\nDame tu nombre completo: ";
     for(int i = 0; a != '\n'; i++) {
          cin.get(a);
          cad[i] = a;
     }
     cad[i] = '\0';
     cout << "Gracias " << cad;
}
```

OBSERVACIONES.

Una función miembro de una clase se llama indicando la instancia y a continuación un punto y el nombre de la función, es similar a hacer referencia a un elemento de una estructura.

La función miembro **get()**, va almacenando cada caracter dado del teclado en la variable **a**, y ésta a su vez la almacena en el arreglo **cad**, pudo haberse almacenado directamente, no se hizo por cuestión de lógica del programa.

La definición en la sintaxis de la función get(), tal vez no esté muy clara, ya que maneja una referencia a char, indicandola como [char&]. Si pasamos el argumento por valor no habría forma de que éste regresará el valor capturado dentro de la función, por lo tanto es necesario pasar a la función get() su dirección, lo cual se hace utilizando el paso por referencia del C++.

Esta manera de manejar punteros como argumento a una función es una nueva forma definida en el C++. Veamos como funciona.

PASE POR REFERENCIA.

Si un argumento a una función es definido como pase por referencia, el compilador toma la dirección de la variable cuando la función es llamada.

Para una mejor comprensión de este proceso podemos analizar el siguiente programa:

```
// Programa 5D, Pase por referencia.
#include <stdio.h>
void cuadrado(int &);
int cubo(int);

void main()
{
    int y, x = 15;

    y = cubo(x);
    cuadrado(x);
    printf("\nEl cubo de  x:  %d", y);
    printf("\nCuadrado de x:  %d", x);
}

void cuadrado(int &y)
{
    y = y*y;
}

int cubo(int z)
{
    return z*z*z;
}
```

OBSERVACIONES.

El programa a pesar de aparentemente no tener nada de C++, fué compilado como un programa en C++ (Extensión .CPP), no hubiese resultado si se hubiese compilado como .C, en cuyo caso el compilador lo interpresta como un programa en "C" estandar, y marcaría un error en la definición de la función **cuadrado(int &y)**
Note que en esta función se pasa la dirección de **x** a **y**, en cambio en la función **cubo(int z)** se pasa **x** por valor, y para poder obtener el resultado es necesario que la función nos regrese su valor mediante un return, no así en la función **cuadrado()**, en donde el valor es regresado en el mismo argumento **x**. La corrida nos resulta:

El cubo de x: 3375
Cuadrado de x: 225

Note que en C++ se debe incluir el archivo de cabecera **stdio.h**, lo exije el compilador, no así en el "C", en el cual es optativo, aunque recomendable.

Función get. Miembro de istream

Descripción Extrae caracteres hacia char *, hasta que el delimitador (tercer parámetro) o EOF es encontrado o hasta que son leidos (**len** - 1) bytes.
Al final de la cadena, siempre se agrega el caracter nulo, el delimitador no forma parte de la cadena.
Por default el delimitador es el salto de línea, pero puede ser cambiado.
La última definición extrae caracteres de una cadena hasta que se encuentra el delimitador.

Sintaxis istream& get(signed char*, int **len**, char = '\n')
istream& get(unsigned char*, int **len**, char = '\n')
istream& get(streambuf&, char = '\n')

Veamos un ejemplo, el cual fué compilado en Zortech Ver 2.0.

```
// Programa # 5A, uso de la función get
#include <iostream.h>
# define LONG 80

int main(void)
{
     char cad[LONG];
```

```
      cout << "\nDame tu nombre completo: ";
      cin.get(cad, LONG);
      cout << "Gracias " << cad;
      return 0;
}
```

OBSERVACIONES.

Note que en la función miembro get() no se incluyó el tercer parámetro, por lo que se esta usando el default ("\n").

El C++ cuenta con otra nueva característica, podemos inicializar parámetros por default, sí éstos no son definidos como argumentos, el compilador toma la definición de default, lo que nos permite invocar la función con una número menor de parámetros que los definidos en ella.

VALORES POR DEFAULT PARA PARAMETROS DE FUNCION.

Se puede tanto, omitir los parámétros definidos por default, o bien, se les puede modificar su valor.

Los valores de default se asignan en la declaración de la función y no en su definición. Para utilizar valores de default en parámetros se deben tomar en cuenta los siguientes puntos:

1) Los argumentos de default se deben pasar por valor, no pueden ser pasados por referencia.

2) Los valores de default pueden ser valores literales o valores constantes (const). No pueden ser valores especificados por una variable.

3) Todos los argumentos con valores de default deben estar al final de la lista de parámetros, en la sección de declaraciones.

Veamos un ejemplo:

```
// Programa 5E, Parámetros por default
#include <stdio.h>

int cubo(int s, int t, int u = 5);

void main()
{
    int a = 2, b = 3, c = 4, d;

    d = cubo(a, b, c);
    printf("\nEl valor del cubo es = %d", d);
```

```
    d = cubo(a, b);
    printf("\nEl valor del cubo es = %d", d);
}

int cubo(int x, int y, int z) { return x*y*z; }
```

OBSERVACIONES.

En este programa calculamos el cubo, en el primer caso, damos los tres argumentos, y el valor en la función cubo(), es calculado como: 2*3*4 = 24.

En el segundo caso solo damos 2 parámetros, a = 2 y b = 3, el tercer parámetro es tomado de la declaración y el valor de la función cubo(), es calculado como 2*3*5 = 30.

6.b) La función miembro put().

Función put(). Miembro de ostream.

Descripción Inserta un caracter. Envia un caracter a la pantalla.

Sintaxis ostream& put(char)

Ejemplo:

```
// Programa # 5B, uso de las funciones get y put

#include <iostream.h>

# define LONG 80

void main()
{
     char a = 0, cad[LONG];

     cout << "\nDame tu nombre completo: ";
     for(int i = 0; a != '\n'; i++) {
          cin.get(a);
          cad[i] = a;
     }
     cad[i] = '\0';
     cout << "Gracias ";
     for(i = 0; cad[i] != '\0'; i++)
          cout.put(cad[i]);
}
```

OBSERVACIONES.

Si analizamos su comportamiento vemos que; es la contraparte de la función get() en su modalidad de extraer un caracter, la función put() inserta un caracter en la pantalla, en este caso enviamos toda la cadena capturada con la función get() a la pantalla hasta encontrar el NULO.

6.c) La función miembro write().

Función write(). Función miembro de ostream.

Descripción Inserta **n** caracteres incluyendo el NULO, Envia a la pantalla **n** caracteres incluyendo el NULO.

Sintaxis ostream& write(const signed char *, int n)
ostream& write(const unsigned char *, int n)

Ejemplo:

```
// Programa # 5C, uso de las funciones get y write
#include <string.h>
#include <iostream.h>
# define LONG 40

void main()
{
     char a = 0, cad[LONG];
     cout << "\nDame tu nombre completo: ";
     for(int i = 0; a != '\n'; i++) {
          cin.get(a);
          cad[i] = a;
     }
     cad[i] = '\0';
     cout << "Gracias ";
     cout.write(cad, strlen(cad));
}
```

OBSERVACIONES.

En este programa estamos enviando la cadena solicitada mediante la función get(), sin incluir el NULO, ya que la función strlen(), al calcular la longitud de una cadena no incluye el NULO,

Podemos usar la función miembro write(), para escribir cualquier número de caracteres y de tipos, convertido éste a caracter. Veamos un ejemplo:

```
// Programa # 5C1, uso de la función write
#include <iostream.h>

void main()
{
     unsigned long x = 1634496328;

     cout << "Tecleaste: ";
     cout.write((char *)&x, sizeof(x));
}
```

En el programa anterior damos un long int x = 1634,496,328, el número es almacenado en memoria en la variable **x**, guardandolo en binario, siendo su equivalente hexadecimal (486F6C61). Este contenido de memoria (4 bytes) son enviados a la pantalla pasando a **x** por **referencia** y con un cast a (char *), imprimiendo:

Tecleaste: Hola

Es el equivalente del número leido de la parte menos significativa a la más signiFicativa, ésto es: 61 6C 6F 48.

7.- **banderas de formato de estado.**

Existe una sútil diferencia entre el operador formateado **<<** y las funciones no formateadas **put** y **write**. Con el operador formateado se puede vaciar el tied stream, (tied stream, son aquellos stream que cuando uno es usado el otro también es afectado, ejemplo; cuando cin es usado cout es vaciado) además puede contar con un ancho de campo asociado con él, y las funciones NO. Así que las expresiones:

```
cout << 'a';
cout.put('a');
```

pueden producir difrentes resultados. Todas las banderas de formato se aplican al operador **<<** pero no a las funciones **put** y **write**.

7.a) Formateando la salida.

El formato tanto para entrada como para salida se determina por varias banderas de formato de estado definidas en la clase **ios** por un tipo enumerado como sigue:

```
// formatting flags
enum    {
   skipws          = 0x0001, // skip whitespace on input
   left            = 0x0002, // left-adjust output
   right           = 0x0004, // right-adjust output
   internal        = 0x0008, // padding after sign or base indicator
   dec             = 0x0010, // decimal conversion
   oct             = 0x0020, // octal conversion
   hex             = 0x0040, // hexadecimal conversion
   showbase        = 0x0080, // use base indicator on output
   showpoint        = 0x0100, // force decimal point (floating output)
   uppercase       = 0x0200, // upper-case hex output
   showpos         = 0x0400, // add '+' to positive integers
   scientific      = 0x0800, // use 1.2345E2 floating notation
   fixed           = 0x1000, // use 123.45 floating notation
   unitbuf         = 0x2000, // flush all streams after insertion
   stdio           = 0x4000  // flush stdout, stderr after insertion
   };
```

Su estado es determinado mediante bits en un tipo long int, de acuerdo al enumerado anterior.

Estas banderas son heredadas por la clase derivada **ostream** e **istream**. Cuando el usuario no define una acción específica, se usa el valor de default, él cual se mostró en los ejemplos anteriores. Existen funciones para; definir, limpiar, probar estas banderas ya sea en forma individual o en grupo. Algunas banderas se limpian automaticamente después de cada salida o entrada.

7.b) Conversiones de base.

Por default los enteros se insertan en notación decimal, ésto puede cambiarse redefiniendo las banderas correspondientes. Si son CERO, la inserción de default tiene lugar en decimal. Luego veremos ejemplos de modificación de estas banderas.

8.- **Anchura de campo.**

La anchura de campos para inserción es el menor número de caracteres necesarios para representar una salida. Para modificar este ancho se puede usar la función miembro de conversión de anchura **width**.

Función width(). Miembro de ios

Descripción Su valor de default es CERO.
Si se usa sin argumento, regresa el ancho actual.
Si se usa con argumento, regresa el ancho actual y lo modifica al ancho especificado.
Si el número de caracteres de salida sobrepasa la anchura especificada, no los trunca, ignora el ancho especificado. Se comporta como si fuera CERO.
El ancho de campo especificado con width, es limpiado (puesto a CERO) después de cada inserción formateada.

Sintaxis int width()
int width(int)

Ejemplo:

```
// Programa 6, uso de la función width
#include <iostream.h>

void main()
{
    int v, w, x = 200, y = 400;

    w = cout.width();
    cout << "\nAncho actual = " << w;
    cout << "\n# campos       1234567890";
    cout << "\nValor de x, y =";
    cout.width(10);
    v = cout.width();
    cout << x << " " << y;
    cout << "\nAncho antes de última salida = " << v;
     w = cout.width();
    cout << "\nAncho actual = " << w;
}
```

OBSERVACIONES.

La salida nos resulta:

```
Ancho actual = 0
# campos       1234567890
Valor de x, y =   200 400
Ancho antes de última salida = 10
Ancho actual = 0
```

Al inicio del programa vemos que el valor de default es CERO, posteriormente definimos el ancho a 10 campos y lo guardamos en la variable **v**, esta anchura se aplica a **x** pero no a **y**, como puede apreciarse con ayuda de la numeración arriba del 200 y

400, antes de salir del programa el ancho a regresado a CERO, lo podemos comprobar asignandole a **w** el ancho actual.

9.- **La función miembro fill(), Llenado.**

Función fill(), Miembro de ios.

Descripción Si no se dan argumentos solo rellena con el caracter de llenado de campos no usados, por default es, espacio en blanco.
Llena los campos no usados con el caracter definido y regresa el valor del caracter de llenado anterior.
fill no se resetea después de ejecutarse.

Sintaxis char fill()
char fill(char)

Ejemplo:

```
// Programa # 6A, Uso de la función fill(), Llenado
#include <iostream.h>

void main()
{
    int x = 300;

                          // Inicia llenado de espacios.
    cout << "Llenando espacios: ";
    cout.fill('*');
    cout.width(10);
    cout << x << "\n";

     cout << "Llenando espacios: ";
     cout.fill(32);      // Para modificar el llenado se debe
    cout.width(10);      // redefinir.
    cout << x << "\n";
}
```

OBSERVACIONES.

El programa redefine primero el caracter de llenado de espacios, siendo éste (*), luego define 10 campos e imprime un valor, note que los espacios no usados se rellenan con el (*).

La segunda parte del programa, retorna como caracter de llenado el espacio en blanco e imprime de nuevo el mismo valor.

La salida será:

```
Llenando espacios: *******300
Llenando espacios:        300
```

10.- **La función miembro precision().**

Función precision(). Miembro de ios.

Descripción Si no se da argumento, regresa el número de decimales definidos. Si se dá argumento actualiza el número de decimales y regresa el valor anterior.
precision(), conserva su definición.

Sintaxis int precision()
int precision(int)

Ejemplo:

```
// Programa # 6B, uso de la función miembro precision()
// Determina el número de cifras decimales.
#include <iostream.h>

void main()
{
     int a, c, d;
     double b;

     cout << "\nDame un entero y un real: ";
     cin >> a >> b;
     cout.width(20);
     cout << "Entero:  " << a << "\n";
     cout.precision(2);
     cout.width(20);
     cout << "Real:    " << b << "\n";
     cout << "00000000011111111112\n";
     cout << "12345678901234567890\n";
}
```

OBSERBACIONES.

La corrida nos resulta:

```
Dame un entero y un real: 45 78.5389346
         Entero:  45
      Real:    78.54
00000000011111111112
12345678901234567890
```

En este caso la corrida lo dice todo, la precisión fué actualizada a 2 digitos, son los que se muestran, note que el número fué redondeado. El número de campos es definido para las cadenas, no para los numeros. Los numeros inferiores, solo nos sirven para contar el número de campos.

11.- **La función miembro flags().**

Función flags(). Miembro de ios.

Descripción Si no se dá argumento, regresa el estado actual de las **banderas de formato de estado** en un tipo long int.
Si se dá argumento, cambia el estado de las **banderas de formato de estado** y regresa su valor anterior.

Sintaxis long flags()
long flags(long)

Ejemplo:

```
// Programa 6C, uso de la función miembro flags.
// Actualiza y regresa el estado de las banderas.
#include <iostream.h>

void main()
{
    long a, b = 8257, w;
    int x = 300;
    float y = 45.6754;

    a = cout.flags();
    cout << "\nValor de las banderas = " << a << "\n";
    cout << "\nModificando las banderas a *8257, *0x2041";
    cout.flags(b);
    w = cout.flags();
    cout << "\nValor de flags: " << w << "\n";


    // Dá formato a salida y realiza conversiones.
```

```
    cout << "\nSe dió a x un valor de *x = 300\n";
    cout << "\nValor de x = " << x << "\n";

    // Inicia manejo de números reales.
    cout << "Números reales *y = 45.6754";
    cout << "\nValor de y = " << y << "\n";
}
```

OBSERVACIONES.

En el programa primero checamos el estado de las banderas, luego lo actualizamos y verificamos si éste fué cambiado.

En el programa vamos a definir la bandera de conversión hexadecimal, con lo que todas las salidas de enteros y sus similares deberán ahora ser enviadas a la pantalla en hexadecimal. Este hecho podemos verlo en las demás salidas, note que el tipo real no se afecta con el cambio de la bandera.

Veamos la corrida.

Valor de las banderas = 8193

```
Modificando las banderas a *8257, *0x2041
Valor de flags: 2041

Se dió a x un valor de *x = 300

Valor de x = 12c
Números reales *y = 45.6754
Valor de y = 45.6754
```

NOTA: Los números o variables que inician con un asterisco significa que se manejaron como cadenas, y no representan la salida del programa, solo se imprimieron como referencia.

Primero, la bandera tiene un valor de 8193 en decimal, en hexadecimal será 2001, lo que nos dice, que hay dos banderas activas:

skipws = 0x0001, // skip whitespace on input
unitbuf = 0x2000, // flush all streams after insertion

Nosotros vamos a activar la bandera para conversión hexadecimal:

hex = 0x0040, // hexidecimal conversion

Por lo que el número a enviar como argumento será en hexadecimal = 2041, convirtiendolo a decimal = 8257

Lo mandamos y verificamos que las banderas hayan sido actualiuzadas. Lo fueron.

Al imprimir el valor de x = 300 en decimal, ahora lo hace en hexadecimal, siendo x = 12c.

Por último comprobamos que los numeros reales no son afectados por esta bandera.

12.- **Manipuladores.**

Otra forma de cambiar las banderas, es usando **manipuladores**, éstos toman una referencia a stream como argumento y regresan una referencia al mismo stream, por lo tanto los **manipuladores** pueden ser colocados en una cadena de inserción (extracción), de tal manera de alterar el estado del stream, como un efecto colateral. Ejemplo:

```
cout << setw(4) << i << setw(6) << j;
```

Esta línea es equivalente a:

```
cout.width(4);
cout << i;
cout.width(6);
cout << j;
```

setw es un manipulador parametrizado declarado en **iomanip.h**, el cual nos define un número de campos **n**. Para usar este manipulador es necesario incluir el archivo de cabecera iomanip.h.

Manipuladores.

Sintaxis	Descripción
outs << dec ins >> dec	Pone bandera de conv. decimal.
outs << hex ins >> hex	Pone bandera de conv. hexadecimal
outs << oct ins >> oct	Pone bandera de conv. octal
ins >> ws	Extrae espacios en blanco

outs << endl	Inserta nueva línea y vacia stream
outs << ends	Inserta terminación NULO en cadena
outs << flush	Vacia un ostream
ins >> setfill(n) outs << setfill(n)	Coloca el caracter de llenado en **n** donde **n** es tipo int.
ins >> setprecision(n) outs << setprecision(n)	Define la precisión a **n** digitos, donde **n** es int.
ins >> setw(n) outs << setw(n)	Pone el número de campos en **n**, donde **n** es tipo int.
ins >> resetiosflags(l) outs >> resetiosflags(l)	Limpia el bit de formato especificado en **l**, donde **l** es de tipo long.
ins >> setiosflags(l) outs << setiosflags(l)	Pone el bit de formato especificado por **l**, donde **l** es de tipo long.
outs << setbase(n)	Pone el formato de conversión de base a **n**, donde **n** es del tipo int. **n** puede ser (0, 8, 10, 16), 0 es el default base de- cimal.

Los maipuladores no parametrizados **dec**, **hex** y **oct** son declarados en ios.h.

Ejemplo:

```
// Programa # 7, Formateando E/S
#include <iostream.h>
#include <iomanip.h>

void main()
{
     int x = 300;
     float y = 45.678454;

     // Dá formato a salida y realiza conversiones.
     cout << setw(10) << hex << x << setw(5) << oct << x << endl;
     cout << setw(10) <<  x;
     cout << setw(5) <<   dec << x << endl;
```

```
     // Inicia manejo de números reales.
     cout << "Números reales" << endl;
     cout << "Valor de y = " << y << endl;
     cout << setw(10) << setprecision(2) << y << endl;
}
```

OBSERVACIONES.

La corrida nos resulta:

```
00000000011111111112    Dados solo como referencia de campos.
12345678901234567890

       12c  454
       454  300
Números reales
Valor de y = 45.678455
     45.68
```

En el primer caso el número 300 (dec) es impreso en hexadecimal como 12c (hex), en seguida se dá un campo de 5 y se imprime el mismo número ahora en octal.

En el segundo renglón no se especificó formato de conversión, se usó el anterior. (octal), por último para regresar a la base decimal se uso el manipulador (dec). el número de campos es definido de la misma forma que en el primer renglón.

Para la impresión del número 45.68, se usó setprecision(2) y para los campos setw(10).

Otro Ejemplo:

```
// Programa 8, uso de manipuladores.
#include <iostream.h>


#include <iomanip.h>

void main()
{
 int x = 300, y = 400, z = 500;

 cout << setbase(16);
 cout << setw(10) << x << setw(10) << y << setw(10) << z << endl;
 cout << resetiosflags(64L);
 cout << setw(10) << x << setw(10) << y << setw(10) << z << endl;
}
```

OBSERVACIONES.

La corrida nos resulta:

```
12c       190       1f4
300       400       500
```

En el primer renglón fijamos la base a hexadecimal y todos los numeros son enviados a la pantalla en ese tipo. En el tercer renglón del programa fuente, limpiamos la bandera de conversión (base 16), quedando el default base 10, enviamos de nuevo las 3 variables a pantalla, siendo impresas ahora en decimal.

Se puede resetear más de una bandera a la vez.

Otro Ejemplo:

```
// Programa 9, uso de ins con manipuladores

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



void main()
{
    int x, y, z;

    cout << "\nDame un entero en HEX: ";
    cin >> hex >> x;
    cout << "\nDame un entero en OCT: ";
    cin >> oct >> y;
    cout << "\nDame un entero en DEC: ";
    cin >> resetiosflags(32L);
    cin >> z;
    cout << "\nValor de x = " << x << endl;
    cout << "\nValor de y = " << y << endl;
    cout << "\nValor de z = " << z << endl;
}
```

OBSERVACIONES.

En este programa estamos analizando la operación de algunos manipuladores usados con **cin**, los primeros me permiten introducir numeros hexadecimales y octales, para que luego me pueda aceptar un número decimal es necesario resetear la bandera de octal, esto se hace con el manipulador resetiosflags(), por último se muestran los tres numeros dados, en decimal. Veamos la corrida:

```
Dame un entero en HEX: 45
Dame un entero en OCT: 45
Dame un entero en DEC: 45

Valor de x = 69

Valor de y = 37

Valor de z = 45
```

NOTA: La bandera **internal** (0x0008), cuando se activa y sobran campos, pone el signo o indicador de base, en el primer campo, los campos sobrantes se dejan en blanco (o se llenan con el caracter de relleno), justificando a la derecha el número.