7.d) El uso de **union** en C++.

C++ permite a la **union** contar con; constructor, destructor y funciones miembro. No es permitido en una **union** el uso de las palabras; public, private o protected. Veamos un ejemplo:

```
// Programa 30, union con constructores y funciones miembro
#include <iostream.h>
union uno {
    int i;
    char c;
    uno(int a) { i = a; }
    uno(char cc) { c = cc; }
    ~uno() { c = 0; }
    int lee_int() { return i; }
    char lee_char() { return c; }
};

void main()
{
    uno X(65), Y('B');

    cout << "Valor del entero: " << X.lee_int() << endl;
    cout << "Valor del char: " << X.lee_char() << endl;

    Y.i = 107;
    cout << "Valor del char: " << Y.lee_char() << endl;
    cout << "Valor del entero: " << Y.i << endl;
}
```

OBSERVACIONES.

En este caso las variables **i** y **c** comparten la misma área de memoria, pueden ser accesadas tanto por las funciones miembro como de manera directa, son públicas. Note ésto en la sentencia Y.i = 107 y en la sentencia:

```
cout << "Valor del entero: " << Y.i << endl;
```

Los constructores me permiten inicializar los objetos del tipo **uno** y el destructor los regresa a CERO al salir del ámbito de definición.

La corrida nos resulta:

```
Valor del entero: 65
Valor del char: A
Valor del char: k
```

```
Valor del entero: 107
```

En el primer caso el objeto **X** se inicializa con 65 y el objeto **Y** con el caracter 'B', sin embargo éste es sobre-escrito con el valor 107, y es el que se muestra en las dos últimas sentencias de la salida, primero como caracter, usando una función miembro, luego como número, de manera directa.

7.d.1) Aclarando la union con **enum**.

- Enumeraciones sin marca.
- Uniones anónimas.

Una manera de prevenir que el usuario seleccione el elemento equivocado, una vez que la **union** es inicializada, es encapsulandola en una clase y auxiliandose de una enumeración, veamos el siguiente ejemplo:

```
// Programa # 31, Encapsulando la union
#include <iostream.h>

class super_var {
    enum { caracter,
           entero,
           real
    } tipovar;
    union {
        char c;
        int i;
        float f;
    };
public:
    super_var(char ch) {
        tipovar = caracter;
        c = ch;
    }
    super_var(int ii) {
        tipovar = entero;
        i = ii;
    }
    super_var(float ff) {
        tipovar = real;
        f = ff;
    }
    void imprime();
};


void super_var::imprime()
{
```

```
    switch(tipovar) {
        case caracter :
                cout << "Caracter: " << c << endl;
                break;
        case entero :
                cout << "Entero: " << i << endl;
                break;
        case real :
                cout << "Real: " << f << endl;
    }
}


void main()
{
    super_var  A('c'), B(12), C(3.1459F);
    A.imprime();
    B.imprime();
    C.imprime();
}
```

OBSERVACIONES.

La corrida nos resulta:

```
Caracter: c
Entero: 12
Real: 3.1459
```

Los valores almacenados en la **union** se definen en la declaración del objeto, de acuerdo al tipo, es seleccionado el constructor.

Cada objeto almacena el valor y además guarda un indicador de que tipo de valor fué almacenado en la union, entonces al extraer el valor mediante la función miembro imprime, se verifica primero el indicador y luego se manda a la pantalla usando el tipo adecuado.

NOTA: Es necesario el cast a constante (**F**) en la declaración del objeto C, de lo contrario se genera un error, ya que por default el real en "C" es del tipo double.

En el código anterior la enumeración no tiene un nombre de tipo, a esta forma de enumeración se le llama **enumeración sin marca**, este tipo de definición es permitido si se definen inmediatamente después instancias de la enumeración, como se hizo en este caso (usando tipovar).

En el código anterior la **union** no tiene un nombre de tipo ni nombre de variables del tipo union. A este tipo de uniones se les llama: **uniones anónimas**. No es necesario asociar la union con un nombre de variable, tampoco se requiere el (**.**). Veamos un ejemplo:

```
// Programa # 31B, uniones anónimas.
#include <iostream.h>
void main()
{
union {
    long i;
    float f;
    };

    f = 15.56;

    cout << "\nEl real = " << f << endl;
    cout << "El entero = " << i << endl;
}
```

OBSERVACIONES.

La corrida nos resulta;

```
El real = 15.56
El entero = 1098446275
```

Podemos observar en la declaración de la **union** que no cuenta con nombre de tipo ni con variables del tipo **union** declaradas en seguida, las que no son necesarias tampoco, es posible accesar de manera directa los elementos de la **union**.

La única diferencia con otras declaraciones es que ambas variables ocupan el mismo espacio, por lo tanto el equivalente del 15.56 en decimal, sacado de la misma localidad de memoria es el 1098446275.

7.e) Funciones miembro, static

Una función miembro **static** es análoga a un dato miembro **static** en que "pertenece a una clase" más que pertenecer a una instancia en particular de esa clase. Una función miembro **static** puede llamarse para la clase como un conjunto, es decir sin especificar un objeto en particular. Puede unicamente afectar a los datos que son comunes para todos los objetos de la clase (datos miembro **static**). No ocupa espacio en cada objeto, por lo que el tamaño del objeto puede ser reducido.

Las funciones miembro ordinarias (no **static**) siempre tienen la dirección inicial (**this**) de la estructura del objeto para la cual fueron llamadas; **this**, normalmente se pasa "secretamente" por el compilador como un argumento de una función miembro. Las funciones miembro **static** no tienen **this**.

Las funciones miembro **static** se encuentran en el mismo ámbito que las funciones miembro no **static**, por lo que tienen los mismos accesos. Como la función

miembro **static** no tiene **this**, no puede llamar a las funciones miembro no **static** ni manipular los datos miembro no **static**.

Una función miembro static puede ser llamada, de la forma ordinaria, usando el (.) o la (->), asociandola con el objeto. Sin embargo una función miembro static, también puede ser llamada sin hacer referencia a un objeto en particular, usando solo el operador de resolución de ámbito.

La función miembro static va a tener un tiempo de llamada rápido, como el de una función global, pero su nombre solo va a ser visible dentro de la clase, de tal manera que no interfiere con nombres de funciones globales. Ejemplo:

```
// Programa PLCP32, llamada a funciones miembro static
#include <iostream.h>

class estatica {
    static i;
public:
    static void pone(int ii) { i = ii; }
    static void imprime() { cout << i; }
};

int estatica::i = 0;

void pone(int x);

void main()
{
    estatica A;
    int y = 40;

    A.pone(20);
    pone(y);
    cout << "\nValor puesto en el miembro estatic: ";
    estatica::imprime();
}

void pone(int x)
{
    cout << "Pone en la pantalla: " << x << endl;
}
```

OBSERVACIONES.

El programa anterior nos muestra la forma de definir funciones miembro **static** y la manera de llamarlas, se pueden llamar como una función miembro común, o

haciendo la referencia a la clase unicamente, usando para ello, el operador de resolución de ámbito (**::**).

Podemos observar también el uso de una función global del mismo nombre y la manera de utilizar ambas.

7.e.1) La palabra reservada **this**.

La sentencia **this** es un apuntador autorreferenciado, implicitamente declarado por el C++. El apuntador **this** apunta a la dirección inicial del objeto para el cual la función miembro es invocada. Veamos el siguiente ejemplo:

```
// Programa 32A, Uso de la palabra this
#include <iostream.h>

class ver_this {
    char c;
public:
    ver_this(char ch) {c = ch; }
    char ver_char() { return this -> c; }
};

void main()
{
  ver_this A('X'), B(65);

  cout << "Caracter de inicialización: " << A.ver_char() << endl;
  cout << "Caracter de inicialización: " << B.ver_char() << endl;
}
```

OBSERVACIONES.

La salida del programa resulta:

```
Caracter de inicialización: X
Caracter de inicialización: A
```

Note que no fué necesario definir la dirección del objeto para el cual se llamó la función, **this** es la dirección de dicho objeto, del cual se regresa el contenido de su elemento **c**. En el primer caso sería la dirección de **A** y el el segundo, la dirección de **B**.

A continuación presentamos otro ejemplo del uso de **this.**

```
// Programa 32B, Uso de la palabra this

#include <iostream.h>
```

```
class es_this {
    char c;
    int i;
    long li;
    float f;
public:
    es_this(char cc = 0, int ii = 0, long ll = 0, float ff = 0)
    {
        c = cc;
        i = ii;
        li = ll;
        f = ff;
    }
    void prueba()
    {
        void *pr = this;
        cout << "c =  " << *((char *)pr) << endl;
        ((char *)pr)++;
        cout << "i =  " << *((int *)pr) << endl;
        ((int *)pr)++;
        cout << "li = " << *((long *)pr) << endl;
        ((long *)pr)++;
        cout << "f =  " << *((float *)pr) << endl;
    }
};

void main()
{
    es_this  A('k', 30000, 300000, 45.789F);
    es_this  B('F', 9000, 100000, 67.90F);

    cout << "Mostrando el valor de A: " << endl;
    A.prueba();
    cout << "\nMostrando el valor de B: " << endl;
    B.prueba();
}
```

OBSERVACIONES.

En la función miembro prueba, inicializamos el puntero definido como **pr** con el valor de **this**, ésto es, la dirección inicial del objeto para el cual se llama la función. Si hacemos avanzar el puntero **pr** de acuerdo a su base, haciendo un cast cada vez de acuerdo al tipo de dato manejado, el puntero **pr** avanzará de elemento en elemento de la clase, con lo cual comprobamos una vez más que **this** apunta al inicio del objeto referenciado, esta dirección es asignada automanticamente por el compilador. Veamos la corrida.

Mostrando el valor de A:

```
c =  k
i =  30000
li = 300000
f =  45.789001

Mostrando el valor de B:
c =  F
i =  9000
li = 100000
f =  67.900002
```

Se vemos la memoria, observamos lo siguiente:

```
          Objeto "A"                    Objeto "B"

          FFF4      42                  FFE8      42
          FFF3      37                  FFE7      87
          FFF2      27                  FFE6      CC
          FFF1      F0    45.789        FFE5      CD    67.90
          FFF0      00                  FFE4      00
          FFEF      04                  FFE3      01
          FFEE      93                  FFE2      86
          FFED      0E    300000        FFE1      40    100000
          FFEC      75                  FFE0      23
          FFEB      30    30000         FFDF      28    9000
pr ->     FFEA      6B    K     pr ->   FFDE      46    F
```

Como vemos en el programa es posible hacer un cast de un puntero definido como **void *pr** a un tipo específico, no se debe modificar **pr** usando la misma notación, ésto es, por ejemplo; decir ((int *)pr++. El problema estriba en que mientras el cast puede producir un apuntador con el valor y tipo del lado derecho, no es un lvalue modificable puede ser una copia del puntero original incrementando la copia y no el puntero original. Para incrementar el mismo puntero como si se tratase de un tipo diferente, no solo se deberá hacer un cast al puntero del tipo deseado, sino también a una referencia a un puntero de ese mismo tipo, ésto es, ((int *&)pr)++. El valor producido por este cast es una referencia, la cual puede ser usada como un lvalue, Esto nos asegura que **pr** va a ser incrementado y no una copia de él, además si se intenta incrementar ((int *)pr)++ en algún compilador que realice una estricta verificación, marcará un error.

Cuando se definen datos dentro de una sección (private, public o protected) el orden de definición es en el que son almacenados en memoria.

En el caso de definir varios elementos dentro de la clase en diferentes secciones (public, private o protected), hay que tener cuidado ya que el compilador puede alterar el orden.

Puesto que **this** es un puntero al inicio de un objeto de la clase, es posible seleccionar elementos de la clase usando el operador flecha (->), como se vió en el priner ejemplo, de hecho no es necesario hacer ésto, ya que dentro de una función miembro, es posible referenciar directamente un dato miembro Recuerde que a **this** solo tiene acceso dentro de una función miembro.

Si se quiere hacer referencia a toda la estructura del objeto se usa ***this**. Recuerde que el operador (*) nos dá el contenido de la dirección almacenada en el apuntador. Puesto que **this** apunta a toda la estructura ***this** es la estructura y si escribimos **return *this** regresa una copia de la estructura.

El operador **this** es más frecuentemente usado para regresar el resultado de una operación, cuando un operador es sobrecargado. lo cual se verá más adelante.

8.- Funciones miembro **const** y **volatile**.

C++ nos permite restringir el uso de una función miembro, lo que nos asegura que solo se emplee de manera adecuada. Este trabajo se lleva a cabo mediante las palabras reservadas **const** y **volatile**. Normalmente se usa con más frecuencia funciones miembro **const** que **volatile**.

8.a) Objetos **const**.

La palabra const le dice al compilador que la variable no va a cambiar durante su tiempo de vida. Esto se aplica a variables de tipos predefinidos. Cuando el compilador ve un **const**, almacena el valor en su tabla de símbolos, insertandolo directamente, luego realiza verificación de tipos (recuerde que ésto es una propiedad de C++, actuando de manera diferente que en "C"), previniendo que el valor vaya a ser cambiado. La razón de declarar un objeto **const** es prevenir que vaya a ser cambiado.

8.b) Uso de **const** con punteros.

C++ de la misma manera que "C", nos permite el uso de constantes, para este propósito se utiliza la palabra reservada **const**, mediante la cual podemos declarar un identificador y asegurar que su valor NO cambie durante la ejecución del programa.

```
const float pi = 3.14159;
```

El tipo **const** congela el valor de un identificador dentro de su ámbito, se pueden definir constantes de cualquier tipo.

Es posible utilizar **const** con punteros, en este caso, se debe tener cuidado en declarar quien será constante; el valor al cual apunta el puntero, el puntero en si mismo o ambos.

a) El valor es constante más no el apuntador.

```
// Programa # 33A, uso de const
#include <stdio.h>


void main()
{
    const char *a = "Verónica";

    printf("\nNombre: %s", a);
    a = "Patricia";
    printf("\nNombre: %s", a);
}
```

OBSERVACIONES.

En este caso la declaración será:

const char *a = "Verónica";

No es posible cambiar el dato almacenado, pero si es posible asignar el apuntador a otra dirección. Si se tratara de cambiar el dato sería:

```
a[0] = 'G';    instrucción ilegal.
```

b) Apuntador constante, más no el dato almacenado.

```
// Programa # 33B, uso de const
#include <stdio.h>


void main()
{
    char *const a = "Verónica";

    printf("\nNombre: %s", a);
    strcpy(a, "Patricia");  // Es legal.
    // a = "Teresa";        // Es ilegal
    a[3] = 'O';
    printf("\nNombre: %s", a);
}
```

```
// Programa # 33C, constante puntero y constante contenido
#include <stdio.h>
void main()
{
    const char *const a = "Hola muchachas";

    printf("\nNombre: %s", a);
}
```

OBSERVACIONES.

En este caso la declaración será:

const char *const a = "Hola muchachas";

No es posible cambiar ni el dato almacenado, ni asignar el apuntador a otra dirección. Es ilegal:

```
a = Teresa";
```

como:

```
a[3] = 'O';
```

El uso de constantes nos permite generar un código más eficiente ya que el compilador puede validar inmediatamente cuando las declaraciones son correctas. Por ejemplo; con la directiva del preprocesador **#define**, el compilador se da cuenta de un error hasta que una instrucción utiliza la constante simbólica. Por otro lado el uso de **const**, facilita la lectura del programa.

Es posible declarar como **const** un objeto de un tipo definido por el usuario. Aunque es conceptualmente posible que el compilador almacene tal objeto en su tabla de símbolos y genere en tiempo de compilación llamadas a sus funciones miembro, en la practica no es cierto, sin embargo se aprovecha el otro aspecto de **const** el cual lo hace invariante durante su tiempo de vida.

8.c) Funciones miembro **const**.

El concepto de **const**ante puede aplicarse a un objeto de un tipo definido por el usuario y significa lo mismo, el estado de un objeto declarado como **const** no puede ser cambiado. Esto es cierto si existe una manera de asegurar que todas las operaciones realizadas sobre un objeto **const** no lo van a alterar. C++ nos brinda una sintaxis especial para decirle al compilador que una función miembro no deberá alterar al objeto.

La palabra **const** colocada después de la lista de argumentos de una función miembro, le va a indicar al compilador que solamente esta función miembro es capaz de leer datos miembros, pero no puede modificarlos (escribirlos).

```
class AB
     int i;
public::
     int func1() const;
};
```

De esta manera func1() puede ser llamada por cualquier objeto declarado como **const** y el compilador sabe que esta función no va a alterar ningún dato miembro. Será código inválido la siguiente definición de la función func1().

int AB::func1() const {return i++; }

Note que la palabra **const** deberá ir tanto en la declaración como en la definición de la función.

Una función miembro **const** podrá ser utilizada sobre un objeto declarado como **const** porque demanda seguridad y el compilador maneja la definición de acuerdo a esta demanda. Veamos el siguiente ejemplo:

```
// Programa # 34, funciones miembro const
#include <iostream.h>

class consmem {
    int x;
public:
    consmem(int X) { x = X; }
    int X() const { return x; }
    int XX() { return x; }
};

void main()
{
    consmem A(1);
    const consmem B(2);

    cout << "Regresa A.X() : " << A.X() << endl;
    cout << "Regresa A.XX(): " << A.XX() << endl;
    cout << "Regresa B.X() : " << B.X() << endl;
//  cout << "Regresa B.XX() : " << B.XX() << endl;
}
```

```
// Warning NO const función consmem::XX() llamada por objeto
const
```

OBSERVACIONES.

Note la advertencia generada al compilar el programa, ya que estamos llamando una función miembro NO declarada como **const** desde un objeto declarado como **const**. Si embargo, si se ejecuta la función miembro.

Marcaría un error el compilador si trataramos de modificar el valor de un dato miembro desde una función declarada como **const.**

```
int X() const {return X++; }
```

Funciones miembro declaradas como **const** pueden ser llamadas desde un objeto no declarado **const**, pero, funciones miembro no declaradas como **const** NO deben ser llamadas desde un objeto declarado como **const**.

La corrida del programa nos resulta:

```
Regresa A.X() : 1
Regresa A.XX(): 1
Regresa B.X() : 2
Regresa B.XX() : 2
```

8.d) Modificación de objetos declarados como **const**. cast al estado **const**

En algunos casos, aunque muy raros, se requiere modificar ciertos miembros de un objeto, aunque el objeto haya sido declarado como **const**, ésto se puede resolver mediante el uso de un cast. Recuerde que el cast le indica al compilador que suspenda la suposición actual y permite sobre-escribir un nuevo tipo.

Para cambiar el contenido de un dato miembro de un objeto declarado como **const**, se selecciona un miembro con el apuntador **this**. Puesto que **this** es la dirección del objeto referenciado en la llamada de la función, aunque paresca redundante, anteponiendo a **this** un cast a el mismo, se remueve de manera implicita el **const** (ya que **const** es parte del cast). Veamos un ejemplo:

```
// Programa 35, cast a const
#include <iostream.h>

class uno {
    int a, b;
public:
    uno() { a = b = 2; }
    void a_cast() const {
```

```
        (((uno *)this) -> a)++;
    }
    void b_cast() const {
        (((uno *)this) -> b)++;
    }
    int leeA() const { return a; }
    int leeB() const { return b; }
};

void main()
{
    const uno fp;

    cout << "Valor de a: " << fp.leeA() << endl;
    cout << "Valor de b: " << fp.leeB() << endl;
    fp.a_cast();
    fp.b_cast();
    cout << "Valor de a: " << fp.leeA() << endl;
    cout << "Valor de b: " << fp.leeB() << endl;
}
```

OBSERVACIONES.

```
void a_cast() const
{
   (((uno *)this) -> a)++;
}
```

Note la forma de realizar el cast, sin el **const** elimina el **const** de **a**, mientras **b** es un **const**, esta forma de alterar un **const**, no es una manera limpia de hacer las cosas, y se esta brincando los mecanismos de seguridad en tipos, es bueno saber como hacerlo, pero deberá evitarse en lo posible.

Veamos la corrida.

```
Valor de a: 2
Valor de b: 2
Valor de a: 3
Valor de b: 3
```

Vemos que aunque **fp** fué declarada como **const** se pudo alterar los elementos de la estructura utilizando la técnica anteriormente descrita.

8.e) volatile: Declaración de Objetos y funciones miembro.

Mientras la palabra **const** le dice al compilador, "Esto nunca cambia", (lo que le permite al compilador desarrollar optimizaciones adicionales), la palabra reservada **volatile** indica al compilador "No se puede saber cuando va a cambiar", evitando el desarrollo por parte del compilador de cualquier optimización. Esta palabra reservada se diseñó para ser utilizada en la lectura de registros o partes de hardware, en donde su alteración no depende del programa.

Los objetos que pueden ser candidatos a ser volatiles son aquellos que son involucrados en actividades concurrentes como por ejemplo, objetos modificados por la señal de un manejador.

La sintaxis para funciones miembro **const** y **volatile** es idéntica, adicionalmente objetos y funciones miembro pueden ser llamadas simultaneamente **const volatile**, al hacer ésto se le indica al compilador que el programa no va a intentar cambiar el objeto, y si sucede, genera un error. Los datos miembro del objeto solo pueden ser cambiados de manera externa, mientras el programa se esta ejecutando.

Veamos un ejemplo:

```
// Programa # 36, Uso de const y volatile
#include <iostream.h>
#include <dos.h>
#include <conio.h>
#include <stdlib.h>
void interrupt cuenta(void);

class constvol {
    unsigned of, svseg, svof;
    struct SREGS rseg;
    void getvec(int intr, unsigned *segment, unsigned *offset)
    const volatile;
    void setvec(int intr, unsigned segment, unsigned offset)
    const volatile;
public:
    constvol() {
        segread(&rseg);
        of = FP_OFF(cuenta);
        getvec(0x1C, &svseg, &svof);
        setvec(0x1C, rseg.cs, of);
    }
    ~constvol() {
        setvec(0x1C, svseg, svof);
    }
};

void constvol::getvec(int intr, unsigned *segment, unsigned
*offset)
const volatile
{
```

```
    union REGS a;
    struct SREGS b;

    a.h.ah = 0x35;
    a.h.al = intr;
    intdosx(&a, &a, &b);
    *segment = b.es;

    *offset = a.x.bx;
}

void constvol::setvec(int intr, unsigned segment, unsigned
offset)
const volatile
{
    union REGS a;
    struct SREGS b;

    a.h.ah = 0x25;
    a.h.al = intr;
    b.ds = segment;
    a.x.dx = offset;
    intdosx(&a, &a, &b);
}

void main()
{
    clrscr();
    const volatile constvol A;
    getch();
}


void interrupt cuenta(void)
{
    static int a = 1, c = 0;

    c++;
    if(c > 100) {
        gotoxy(1, 10);
        if(a == 1) {
            cout << "hola, esta es una prueba interesante...";
            a = 0;
        } else {
            clreol();
            a = 1;
        }
        c = 0;
    }
```

```
}
```

OBSERVACIONES.

Este programa lo que hace es asignar la dirección de la función cuenta() a la interrupción 0x1C, ésta es llamada por la interrupción del timer de la PC, la cual se llama 18 veces por segundo.

La función cuenta(), cada vez que es llamada incrementa la variable **c**, pudiendo entrar a la función hasta que el valor de esta variable supera el valor de 100, la primera vez que entra al primer if de esta función, muestra el letrero, la segunda, lo borra.

Esta función es llamada de manera automática por la computadora 18 veces por segundo, por lo que no es necesario llamarla desde el programa.

Este tipo de funciones no pueden ser manejadas como funciones miembro de una clase, ya que las funciones miembro deben ser manejadas mediante un objeto perteneciente a una clase, lo que no sucede con estas funciones.

La palabra interrupt, antes del nombre de la función, propicia que sean almacenados en la pila los registros del micro, antes de llamar a la función, y restaurados al salir de ella. lo que nos asegura la continuidad de las actividades del procesador.

La función getvec(), almacena la dirección de la interrupción 0x1C, para su restauración al salir del programa mediante el destructor.

La función setvec(), asigna una nueva dirección a la interrupción 0x1C, la cual en este caso es la dirección de entrada a la función cuenta().

Las funciones miembro declaradas como **const** y **volatile** se deberán usar con objetos declarados **const** y/o **volatile**. Sin embargo, en la práctica, **volatile** es menos usada que **const**.