UNIDAD 6

Referencias.

1.- **Introducción.**

Una referencia es similar a un apuntador y se utiliza para dar un seudónimo de un objeto.

En otras palabras, una **referencia** es una manera de llamar de otra forma la misma localidad de memoria, es un seudónimo de una variable.

Los usos más frecuentes de las referencias son:

1) La transferencia de grandes objetos a las funciones.
2) El regreso de un objeto de una función.

1.a) Referencia.

Una referencia introduce un sinónimo de un objeto. La notación para definir una referencia es similar a la de los apuntadores.

```
* Notación de apuntador       int *pr;
* Notación de referencia      int &ref;
```

Ejemplo de definición:

```
punto pt1(10,10);
punto &pt2 = pt1;
```

En este caso se esta definiendo a pt2 como una referencia a pt1. Luego de la definición pt1 y pt2 se refieren al mismo objeto, como si fueran la misma variable, es necesario resaltar que una referencia no crea una copia de un objeto, es un sinónimo del objeto.

Una referencia se debe inicializar en cuanto se defina; debe ser el sinónimo de algo, NO SE PUEDE DEFINIR UNA REFERENCIA Y DESPUES INICIALIZARLA.

Es ilegal hacer lo siguiente:

```
punto &pt1;

pt2 = pt1;
```

También es posible inicializar una referencia a una constante. En este caso, se hace una copia de la constante. aunque en algunos compiladores ésto genera una advertencia.

```
int &i = 4;          // Se hace referencia a una copia de 4
```

La seguridad es la razón por la que 4 se vuelve una referencia a una copia de 4, en vez del propio 4.

Veamos un ejemplo.

```
// Programa # 37, introducción a referencias.
#include <iostream.h>

void main()
{
    int a = 10;
    int &b = a;

    cout << "Valor de a = " << a << endl;
    cout << "Valor de b = " << b << endl;

    int& c = 4;          // Se genera una advertencia.

    cout << "Valor de c = " << c << endl;
    ++c;

    int &d = c + 4;      // Se genera una advertencia.

    cout << "Valor de d = " << d << endl;
}
```

OBSERVACIONES.

Es probable que el [4], asignado a la referencia **c** y el [4] sumado a **c** sean el mismo objeto (la mayoria de los compiladores realizan la optimización de constantes y asignan a ambos [4]'s el mismo espacio de memoria). Así aunque esperamos que d = 9, pudiera ser también 10. Veamos la corrida.

La corrida nos resulta:

```
Valor de a = 10
Valor de b = 10
Valor de c = 4
Valor de d = 9
```

Al compilar el programa usando Borland C Ver 3.1, nos genera las siguientes advertencias en las lineas arriba indicadas.

```
temporary used to initialize 'c' en function main()
```

```
temporary used to initialize 'c' en function main()
```

2.- **La característica de Referencia.**

Cuando se construyen bibliotecas de funciones, una buena filosofía de diseño es hacer a las funciones lo más transparente posible al usuario, al cual solo le debe interesar lo que hace la función sin preocuparse como lo hace.

Las funciones en C y C++ que toman direcciones como argumento o devuelven direcciones, obligan al usuario a preocuparse de los mecanismos de implementación. La evolución de los lenguajes de programación ha procurado no obligar al usuario a detenerse en los detalles, sino que se ha orientado a permitir que el usuario dedique su tiempo a pensar en el "Diseño". C++ contiene una característica denominada la **referencia** que le permite al programador pasar direcciones como argumentos a funciones. Esta característica no está disponible en ANSI C o en cualquier otra versión de C.

Si una función toma un argumento tipo puntero:

```
void func(const char *cad)
```

y se le quiere llamar para una vaiable **a**, el programador es el responsable de poner en el argumento la dirección de **a**.

```
func(&a);
```

Si una función regresa un puntero, se puede escribir código para que la llamada de la función sea parte de un valor. Por ejemplo si se declara:

```
class matriz {
     ....
     ....
public:
     float *val(int x, int y);
```

Y se declara:

```
matriz m;
```

Se pueden utilizar elementos de matriz en expresiones.

```
float f = *(m.val(1, 2)) + *(m.val(3, 4));
```

El uso de los **punteros** indica como se produce un valor o como se cambia una variable, haciendo también más difícil de leer el código. Desreferenciando los punteros, se hace más legible el código. Re-escribamos la expresión anterior.

```
float f = m.val(1, 2) + m.val[3, 4);
```

Veamos un ejemplo de uso de punteros.

```
// Programa # 38A, uso de punteros
#include <iostream.h>
#include <string.h>

class matriz {
    enum {tam = 20};
    char a[tam];
public:
    matriz();
    matriz(const char *y) {
    strcpy(a, y);
    }
    char *val(char c);
};

matriz::matriz()
{
    for(int i = 0; i < tam; i++) a[i] = 0;
}


char *matriz::val(char c)
{
    for(int i = 0; a[i]; i++)
        if(a[i] == c)
            return  a + i;
    return a + i;
}


void main()
{
    matriz A("Holga"), B;
    char x;

    x = 'g';
    if(*(A.val(x)))
        cout << "Existe en la cadena la letra: " << x << endl;
    else
        cout << "NO Existe en la cadena la letra: " << x << endl;
    if(*(B.val(x)))
        cout << "Existe en la cadena la letra: " << x << endl;
    else
        cout << "NO Existe en la cadena la letra: " << x << endl;
```

```
}
```

OBSERVACIONES.

En la función miembro char ***val(char)** regresamos la dirección del caracter de la cadena del dato miembro que coincide con el caracter pasado como argumento a la función. Si no hay coincidencia se regresa CERO. Note en el if la forma de expresar el retorno de la dirección y la referencia a su contenido. Veamos la corrida.

```
Existe en la cadena la letra: g
NO Existe en la cadena la letra: g
```

3.- **Sintaxis de referencias.**

En C++ el único lugar donde se manejan referencias es en la lista de argumentos de las funciones. Es posible pasar argumentos por referencia y regresar referencias a una variable. En cualquier lugar de la función el argumento aparece como una variable, ésto es, como si se hubiera pasado por valor, de hecho uno de los beneficios más sobresalientes en el manejo de referencias es que se pueden combinar pase por valor y pase por referencia, con solo cambiar los argumentos. A continuación se muestra un ejemplo de pase y retorno por valor, pase y retorno por puntero y pase y retorno por referencia.

```
// Programa # 39

// Pase de argumentos y retorno de valores por
// valor, puntero y por referencia.
#include <iostream.h>

class uno {
    int i, j, k;
public:
    uno(int u = 0, int v = 0, int w = 0) {
        i = u;
        j = v;
        k = w;
    }
    uno por_valor(uno X);         // Pase y retorno por valor
    uno *por_puntero(uno *X);     // Pase y retorno por puntero
    uno &por_referencia(uno &X); // Pase y retorno por referencia
    void imprime(char *msg = "") {
        cout << msg << ": i = " << i << " j = " << j
            << " k = " << k << endl;
    }
};
```

```
uno uno::por_valor(uno X)          // Pase por valor.
{
    X.i = X.j = X.k = 700;
    return X;                      // Retorno por valor.
}


uno *uno::por_puntero(uno *X)      // Pase por puntero.
{
    X ->i = X ->j = X ->k = 800;
    return X;                      // Retorno por puntero.
}

uno &uno::por_referencia(uno & X)  // Pase por referencia.
{
    X.i = X.j = X.k = 900;
    return X;                      // Retorno por referencia.
}

void main()
{
    uno A, B;
    A.por_valor(B).imprime("Resultado de por valor");
    A.por_puntero(&B)->imprime("Resultado de por puntero");
    A.por_referencia(B).imprime("Resultado de por referencia");
}
```

OBSERVACIONES.

En el pase por valor, el objeto en el argumento es copiado en el parámetro X, siendo X local de la función. El valor de retorno es regresado en: **A.por_valor(B)**, en este caso. Note que el objeto A solo sirve para cumplir cin la sintaxis.

En las siguientes 2 funciones; por puntero y por referencia, se regresa el argumento modificado y el valor de retorno es un puntero, es por eso que se puede usar el operador flecha, en el caso de regreso por puntero.

Analizando el tercer caso, pase por referencia, es idéntico al pase por valor, con la sola diferencia en la declaración del parámetro. Al regresar una referencia, ésta es tomada como una variable, de allí el uso del punto, como en el caso del regreso por valor.

En los tres casos la función miembro imprime() es asociada al valor de retorno de la función, él cual, en el primero y tercer caso es un objeto y en el segundo es un puntero a un objeto.

Insistimos, note que se usa la misma sintaxis cuando se usan valores que cuando se usan referencias, lo que le dá mayor claridad al programa.

4.- **¿Porqué las refrencias son esenciales?.**

Las referencias causan a menudo bastante confusión, y no son consideradas de la misma manera por todas las personas. Aquellas que no tienen una base profunda de C, utilizan las referencias muy a menudo, excepto donde sean esenciales los punteros. Normalmente los programadores de C que aprendieron C++, las evitan, ya que piensan que son otra forma de pasar las direcciones, y los punteros pueden desempeñar la misma función.

Hay una situación, sin embargo, donde las referencias son esenciales y por ello fueron introducidas en C++; es el concepto de sobrecarga de operadores.

Cuando se sobrecarga un operador se crea una función la cual es exactamente igual que otra función, excepto por el nombre. El nombre de la función siempre consiste de la palabra reservada **operator** seguida por el ó los caracteres usados por el operador. Por ejemplo, la suma para una clase llamada **mas** se presenta como: **mas operator+(mas y)**. El resto de la función es la misma (aunque hay limitaciones en el número de argumentos, se añade una cosa a la vez). El compilador llama a la función sobrecargada cuando ve que los tipos de datos son los apropiados para ser utilizados con este operador, a continuación se muestra un ejemplo.

```
// Programa # 40, Uso del operador [+]
#include <iostream.h>

class mas {
    int i;
public:
    mas(int x = 0) { i = x; }
    mas operator+(mas y) { return mas(i + y.i); }
    void imprime(char *msg) {
        cout << msg << ": i = " << i << endl;
    }
};

void main()
{
    mas A(20), B(50);
    mas C;

    C = A + B;
    C.imprime("C = A + B");
}
```

OBSERVACIONES.

El **operator +()** lleva un único argumento, representado por el valor a la derecha del signo [**+**]. Recuerde que esta función es llamada por el objeto a la izquierda del signo [**+**]. La función **operator +()** crea un nuevo objeto **mas**, llamando al constructor **mas()**, el argumento al constructor **mas**, es la suma de las partes de datos privados de 2 objetos. El constructor **mas()** crea un nuevo objeto (temporal) que se copia fuera de la función como el valor de retorno del operador.

Veamos la corrida.

```
C = A + B: i = 70
```

En el ejemplo anterior no hay ningún problema, en el operador sobrecargado, los objetos se pasan y se devuelven por valor. Si embargo cuando sea necesario pasar las direcciones de un objeto (objetos muy grandes), nos encontraremos con algunos inconvenientes. Veamos el siguiente ejemplo.

```
// Programa # 40A, Uso del operador [+], pase por puntero.
// Retorno por valor.
#include <iostream.h>

class mas {
    int i;
public:
    mas(int x = 0) { i = x; }
    mas operator+(mas *y) { return mas(i + y ->i); }
    void imprime(char *msg) {
        cout << msg << ": i = " << i << endl;
    }
};

void main()
{
    mas A(20), B(50);
    mas C;

    C = A + &B;
    C.imprime("C = A + B");
}
```

OBSERVACIONES.

El programa sintacticamente es correcto, sin embargo, la manera de expresar la suma es bastante incoherente; C = A + &B, se podría pensar en una sintaxis

homogenea, tal vez; &A + &B, sin embargo al hacer ésto, el compilador supone que se estan sumando 2 punteros.

Las referencias nos permiten pasar direcciones a funciones de operador sobrecargadas, y mantienen la notación clara de **A + B** acomodemos de esta manera el ejemplo anterior.

```
// Programa # 40, Uso del operador [+], pase por referencia.
#include <iostream.h>

class mas {
    int i;
public:
    mas(int x = 0) { i = x; }
    mas operator+(mas &y) { return mas(i + y.i); }
    void imprime(char *msg) {
        cout << msg << ": i = " << i << endl;
    }
};

void main()
{
    mas A(20), B(50);
    mas C;

    C = A + B;
    C.imprime("C = A + B");
}
```

OBSERVACIONES.

Con las referencias se puede sobrecargar operadores, y pasar direcciones cuando se utilizan objetos demasiado grandes.

Cuando se usa el pase por referencia en la sobrecarga de un operador, se nota una mejoría en la sintaxis de los argumentos de la función.

5.- **Referencias independientes.**

Las referencias fueron diseñadas en C++ para pasar argumentos y regresar valores, sin embargo como el lenguaje es consistente, aunque no sea una práctica recomendada, es posible crear referencias independientes, no asociadas con argumentos de ninguna función.

A diferencia de un puntero, el cual no necesariamente tiene que apuntar a una dirección válida. Cuando se crea una referencia, debe inicializarse en ese mismo lugar.

Las referencias independientes pueden inicializarse solo a una variable existente del tipo correcto, o a una constante. La sintaxis, nos resulta:

```
int a;
int &ar = a;
```

En este caso la dirección de **a** es tomada por **ar**, ahora, tenemos dos maneras de referirnos a la misma variable, con **a** y con **ar**, lo cual puede parecer confuso, a primera vista.

Se pueden cuestionar aspectos como, el cambio de una variable de un tipo de dato a una referencia de otro tipo de dato.

```
float k;
int &x = (int&)k;
```

No se pueden crear arreglos de referencias, lo cual reafirma lo dicho anteriormente, el hecho de evitar el uso de referencias independientes, utilizando en estos casos punteros.

Una referencia es diferente a un puntero.

* Un puntero, es una variable, la cual puede contener cualquier dirección del tipo base.

* Una referencia debe inicializarse cuando se crea y no puede ser referida a otra variable.

Veamos el siguiente ejemplo:

```
int x = 10, y = 100;
int &xr = x;
xr = y;
```

Recuerde que **xr** es un sinónimo de **x**, por lo tanto, la expresión **xr = y**, copia el contenido de **y** en **x**.

En el caso de asignar una constante a una referencia.

```
int &cr = 50;
```

El compilador crea una variable temporal, copia el valor constante en ella, utilizando la dirección de la variable temporal, la cual es asignada a la referencia.

La discusión anterior nos lleva a pensar que cualquier función que pase un argumento por referencia, también puede tener un argumento constante, (CUIDADO,

los resultados pueden ser impredecibles si se modifica el valor del argumento dentro de la función.

El siguiente ejemplo nos ilustra lo planteado con anterioridad.

```
// Programa # 41, referencias independientes.
#include <stdio.h>
void main()
{
    int a = 10, b = 100;
    int &ar = a;

    printf("\nValor de ar  = %d", ar);
    printf("\nValor de a   = %d", a);
    printf("\nValor de b   = %d\n", b);
    ar = b;
    printf("\nValor de ar  = %d", ar);
    printf("\nValor de a   = %d", a);
    printf("\nValor de b   = %d", b);

    float k = 1;
    unsigned long &m = (unsigned long &)k;

    printf("\nValor de k (float)         = %f", k);
    printf("\nValor de m (unsigned long) = %lx\n", m);
}
```

OBSERVACIONES.

La corrida nos resulta.

```
Valor de ar  = 10
Valor de a   = 10
Valor de b   = 100

Valor de ar  = 100
Valor de a   = 100
Valor de b   = 100
Valor de k (float)         = 1.000000
Valor de m (unsigned long) = 3f800000
```

En el primer grupo de salida, **ar** toma el valor de **a**, al asignarle **b** a **ar**, ahora **ar** = 100, el cual es el valor de **b**, al hacer un cast a refrencia unsigned long int a la variable **k**, la dirección de **k** es tomada por **m**, al imprimir **m**, nos muestra el contenido de la variable **k**, pero expresado como entero en hexadecimal. Esto es, **m** y **k**, hacen referencia a la misma localidad de memoria.

6.- **El significado de regresar una dirección.**

Al regresar una dirección de una función ya sea por referencia o por puntero, se esta regresando una dirección de memoria, se puede leer el valor de la dirección de memoria, y si no se declaró el puntero **const**, también es posible sobre-escribir la dirección. si este es un dato miembro declarado como **private**, este es un punto de discusión importante.

6.a) Lo que no se debe hacer.

Cuando se regresan punteros de funciones debe tenerse particular cuidado en no regresar la dirección de una **variable local**. Sintacticamente, es perfectamente aceptable y no generará ningún error ni advertencia. Veamos un ejemplo, él cual fué compilado en el paquete de QuickC Ver 2.0, (no en todos los paquetes da el mismo resultado).

```
/* Programa # 42, */
/*Regresando la dirección de una variable local. */
/* Programa compilado con QuickC Ver 2.0 */

#include <stdio.h>

int *tere()
{
    int a = 100;
    return &a;
}

int *ana()
{
    int b = 200;
    return &b;
}


void main()
{
    int *x = tere();
    ana();
    printf("Regreso de tere(): %d", *x);
}
```

OBSERVACIONES.

Corramos el programa.

```
Resgreso de tere(): 200
```

En esta prueba, primero se llama a tere(), si observaramos el valor en este punto, ***x** valdría 100, sin embargo al llamar a ana(), el valor es regresado a la misma dirección y ahora ***x** vale 200, como se puede observar la salida del programa, aunque en ana() el valor regresado es aparentemente perdido.

Analicemos lo que sucede. Cuando tere() es llamada, se le asigna un espacio de memoria sobre el stack para almacenar la variable **a** en el cual guarda el valor de 100, al terminar tere(), regresa la dirección de su variable local **a**, si otra función es llamada, y ésta tiene una variable local, la variable local nueva, en este caso la de ana(), va a sobre-escribir el área de memoria que anteriormente fué asignada a **a**, de tere(), con lo que el valor de 100 va a ser cambiado, ahora por el de **b** = 200. Puesto que la función ana() es idéntica a tere(), van a coincidir las 2 variables locales, en el mismo espacio de memoria.

El hecho anterior no se da de la misma manera en todos los compiladores, depende del orden y anidamiento de llamado de funciones, por lo que no se debe tomar como una regla y quererla utilizar en algún programa, lo que puede causar problemas.

En el siguiente ejemplo, a primera vista, nos da la impresión de que las funciones estan regresando direcciones de variables locales. En ambos casos las variables son static, con lo que tienen visibilidad local, y un tiempo de vida global. Esto es seguro y aceptado, pudiendo regresar direcciones sin ningún problema.

```
// Programa # 43, Funciones que regresan punteros seguros.

#include <stdio.h>

char *martha()
{
    char *ts = "Esto parece una variable local";

    return ts;
}

int &cristina()
{
    static int x = 400;
    return x;
}


void main()
{
    char *y = martha();
```

```
    int z = cristina();

    printf("\nValor de martha(): %s", y);
    printf("\nValor de crist():  %d", z);
}
```

OBSERVACIONES.

En el caso de la cadena, donde aparentemente **ts** es local, ya que no le fué antepuesta la etiqueta static, recuerde que una constante de cadena, crea una variable static, aunque no sea declarada explicitamente.

La función martha() regresa un puntero y la función cristina() regresa una referencia, en ambos casos son direcciones de memoria.

6.b) Peligros al devolver referencias y punteros.

Cuando se regresa un puntero de una función, esta dirección se puede asignar a una variable puntero y entonces tener acceso a los datos privados de la clase, sin verificar permiso de acceso a a dichos datos, lo cual también incluye la posibilidad de indexar un arreglo e inclusive sobrepasar los limites de los arreglos, como se puede ver en el siguiente ejemplo.

```
// Programa # 44, Problemas con punteros de retorno de funciones.
#include <stdio.h>
#include <stdlib.h>

const tam = 100;

class vector {
    int  v[tam];
    void error(char *msg) {
        fprintf(stderr, "\nError en vec: %s", msg);
        exit(1);
    }
public:
    int *operator[] (int indice) {
        if(indice < 0 || indice >= tam)
            error("Indice Fuera de rango");
        return &v[indice];
    }
};

void main()
```

```
{
    vector vec;

    *vec[5] = 88;
    printf("\nValor de *vec[5]: %d", *vec[5]);
    int *x = vec[0];
    x++;
    *x = 50;
    printf("\nValor de *vec[1]: %d", *vec[1]);
     x[10] = 4;
     printf("\nValor de *vec[11]: %d", *vec[11]);
     x[100] = 8;
     printf("\nValor de *vec[101]: %d", *vec[101]);
}
```

OBSERVACIONES.

En la clase **vector** la función **operator[]()** de sobrecarga del operador **[]** nos permite manejar el arreglo **v[]** de forma indexada, la función **operator[]()** nos regresa un puntero al elemento referenciado, inclusive nos permite refrenciar el arreglo aun fuera de su rango, como se puede ver en la penúltima sentencia del main(), aunque para evitar catastrofes, el rango es verificado dentro de la función de sobrecarga de **[]**.

Veamos la corrida.

```
Valor de *vec[5]: 88
Valor de *vec[1]: 50
Valor de *vec[11]: 4
Error en vec: Indice Fuera de rango
```

El valor de 50 se asignó a un dato miembro, sin la intervención de una función miembro, simplemente, se usó la dirección del dato miembro, después de haberse incrementado su dirección.

Luego al asignarsele a x[10] = 4, en realidad esta posición corresponde a la posición *vec[11] ya que el puntero **x** fué incrementado con anterioridad en **x++;**

Note que la salida de error, siempre es enviada a la pantalla.

Puesto que una referencia es una dirección al momento de su creación y la dirección no puede ser cambiada, es más difícil obtener el control de datos privados. Por ello el regreso por referencia es el método preferido cuando es necesario regresar direcciones de una función.

Es posible tener acceso a los datos privados, cuando se regresa una referencia, como se muestra en el siguiente ejemplo.

El siguiente ejemplo es muy similar al anterior, la diferencia es que **operator[]()** regresa una referencia en vez de un puntero, con lo cual se consigue una sintaxis más legible. En lugar de escribir:

```
     *vector[7] = 88;
```

escribimos:

```
     vector[7] = 88;

// Programa # 44A, Devolución de una referencia
// desde una función.

#include <stdio.h>
#include <stdlib.h>

const tam = 20;

class vec2 {
    int v[tam];
    void error(char * msg) {
        fprintf(stderr, "\nError en vec2: %s\n", msg);
        exit(1);
    }
public:
    vec2(int ival = 0);
    int &operator[](int indice) {
        if(indice < 0 || indice >= tam)
            error("Indice fuera de rango");
        return v[indice];
    }
    void imprime(char *msg = "");
};

vec2::vec2(int ival) {
    for(int i = 0; i < tam; i++)
        v[i] = ival;
}

void vec2::imprime(char *msg) {
    if(*msg) printf("%s\n", msg);
    for(int i = 0; i < tam; i++) printf("%d,", v[i]);
    printf("\n");
}

void main()
{
    vec2 vector(7);
    vector[7] = 88;

    int &x = vector[0];
    x = 100;
```

```
    vector.imprime("Después vector[0] = 100");
    *(&x + 1) = 99;
    vector.imprime("Después vector *(&x + 1) = 99");
    int &y = vector[20];
    y = 200;
    vector.imprime("Después vector[20] = 200");
}
```

OBSERVACIONES.

La corrida nos resulta.

```
Después vector[0] = 100
100,7,7,7,7,7,7,88,7,7,7,7,7,7,7,7,7,7,7,7,
Después vector *(&x + 1) = 99
100,99,7,7,7,7,7,88,7,7,7,7,7,7,7,7,7,7,7,7,
Error en vec2: Indice fuera de rango
```

Podemos notar como el valor de 100, se asignó a la referencia **x**, y fué agregado al arreglo en la posición del subindice 0, lo que significa que estamos haciendo acceso a miembros privados, sin la intervención de funciones miembro. El caso se repite para el valor 99, y se repetiria de nuevo para la referencia **y**, si no se hubiera traspasado el limite verificado en la función **operator[]()**.

Es posible prevenir que el programador modifique el contenido de una referencia, mediante la devolución de una referencia a **const**, con lo cual solo es posible leer el valor almacenado en dicha dirección, pero no modificarlo. Al compilar el programa e intentar modificar la referencia **const**, el compilador nos marcaría un error.

```
"No se puede modificar un objeto const en función main()"
```

6.c) Punteros y referencias a objetos.

Cuando se pasa un puntero o referencia a un objeto como un argumento de función, es necesario seleccionar los miembros (datos o funciones) por medio de la estructura del operador puntero para punteros y la estructura del operador miembro para referencias, como en el siguiente ejemplo.

```
// Programa # 45, Seleccionando miembro para
// argumento de dirección.
#include <iostream.h>

class impe {
    char *frase;
```

```
public:
    impe(char *msg) {
        frase = msg;
    }
    void imprime(char *msg2) {
        cout << msg2 << ": " << frase << endl;
    }
};

void por_puntero(impe *p) {
    p -> imprime("Entra por puntero");
}         // La flecha selecciona la función desde el puntero.

void por_referencia(impe &p) {
    p.imprime("Entra por referencia");
}         // El punto selecciona la función desde la referencia.


void main()
{
    impe X("Es la frase UNO"), Y("Es la frase DOS");

    por_puntero(&X);
    por_referencia(Y);
}
```

OBSERVACIONES.

La corrida arroja:

```
Entra por puntero: Es la frase UNO
Entra por referencia: Es la frase DOS
```

Las funciones por_puntero() Y por_referencia() solamente pueden manipular datos o elementos **public** de la clase **impe**. Para seleccionar elementos **private** las funciones deberán ser declaradas como funciones **friend** dentro de la definición de la clase.

La siguiente definición de la clase **impe** fué cambiada para por medio de las funciones por_puntero() y por_referencia(), declaradas como friend, tener acceso a la función imprime(), la cual para no alterar la lógica del programa se declaró como **private**, pudiendo entonces solo ser llamada por funciones miembro o friend.

```
class impe {
    char *frase;
    void imprime(char *msg2) {
        cout << msg2 << ": " << frase << endl;
    }
```

```
public:
    impe(char *msg) {
        frase = msg;
    }
    friend void por_puntero(impe *p);
    friend void por_referencia(impe &p);
};
```

6.d) Miembros de clase declarados como referencia.

Hay solo dos situaciones donde se puede mencionar una referencia sin inicializarla.

1) Cuando se declara una variable extern.

```
extern int &n;
```

2) Es posible usar referencias (en vez de punteros) en datos miembro, en este caso se deberán inicializar en el constructor.
Veamos un ejemplo:

```
// Programa # 46, Uso de Referencia a Datos miembros de una clase
#include <iostream.h>



class cecy {
    int &refmem;
public:
    cecy(int *j) : refmem(*j) {}
    cecy(int &j) : refmem(j) {}
    void pone(int nval) { refmem = nval; }
};

void main()
{
    int x = 100, y = 200;

    cout << "x = " << x << ", y = " << y << endl;
    cecy W1(&x);
    cecy W2(y);
    cout << "x = " << x << ", y = " << y << endl;
    W1.pone(50);
    W2.pone(80);
    cout << "x = " << x << ", y = " << y << endl;
}
```

OBSERVACIONES.

```
x = 100, y = 200
x = 100, y = 200
x = 50, y = 80
```

1) Declaramos 2 enteros inicializandolos; x = 100, y = 200, se imprimen dichos valores.

2) Declaramos 2 objetos del tipo cecy inicializandolos en el constructor **x** por puntero, **y** por referencia. Note la sintaxis de estos constructores un poco especial a lo conocido hasta ahora. Al imprimir de nuevo los valores de **x** y de **y**, siguen conservando su valor anterior, éste no ha sido cambiado.

3) La función miembro pone(), actualiza el dato miembro, note que el nuevo valor es pasado por valor y asignado de la misma forma al dato miembro declarado por referencia. Al llamar a la función miembro pone() referida a W1, su dato miembro tiene la dirección de **x**, y la referida a W2, su dato miembro tiene la dirección de **y**, por lo que estas variables son modificadas al cambiar el valor del dato miembro de cecy de los objetos W1 y W2.

7.- **Cuando utilizar referencias.**

En "C" el uso de punteros es claro, se usan éstos cuando es necesario pasar a una función una dirección o regresar una dirección de una función.

En "C++" se puede elegir como pasar o devolver direcciones.

Aunque en "C++" es posible utilizar referencias independientes, no es una buena práctica y no se consideraran como alternativa.

7.a) Ventajas de usar referencias.

1) Las referencias nos proporcionan una notación clara. Cuando se esta tratando de entender una parte del código, no se tiene que pensar en como son menejados los parámetros. Esto significa que el usuario puede concentrarse en el problema y no en la forma en que es implementado.

2) Las referencias dejan la responsabilidad del pase de argumentos sobre el que escribió la función, no sobre los programadores que la usan. Lo que significa que el lenguaje hace mayor trabajo que el usuario, lo que trae como consecuencia una reducción en el número de errores.

3) Las referencias son un complemento necesario para manejar sobrecarga.

7.b) Problemas con referencias.

Aunque las referencias son notaciones más claras y dejan la responsabilidad del pase de parámetros al programador, hay situaciones donde pueden ocultar errores. Por ejemplo, la función istream get() es llamada con un argumento de caracter: cin.get(c); Como las referencias ocultan el tipo de pase del argumento, no se puede decir al ver la función si **c** pasa por valor o pasa por referencia. Si se esta tratando de encontrar un problema asociado a la variable **c**, se puede pensar que **c** es pasado por valor, y no es modificada durante el llamado de la función, cuando de hecho sí lo es.

7.c) Lineamientos para el pase de argumentos.

1) Cuando se usan tipos de datos predefinidos, si no se va a alterar su valor al ser pasados como argumento a una función, es más conveniente pasarlos por valor que por referencia.

2) Cuando se usan tipos de datos predefinidos, si se va a alterar el valor del argumento, use el pase por puntero, es más claro.

3) Si una función modifica un argumento, el cual es de un tipo definido por el usuario, pase una referencia. Cualquier función que modifique los datos privados debe ser una función miembro o friend. Esto significa que la clase tiene control sobre las funciones que pueden modificar sus datos privados. Entonces, cualquier alteración de los datos miembro, el origen de la alteración se localiza en la misma clase.

4) Para pasar objetos de regular tamaño, que no van a ser alterados, use referencias a **const**.

En el siguiente ejemplo, se muestran el pase de varios tipos de parámetros, de acuerdo a las sugerencias descritas.

```
// Programa # 47, Lineamientos de pase de argumentos.
#include <iostream.h>

#include <conio.h>

const tam1 = 10;

// tipos predefinidos.

int funo(int x);    // No se modifica variable.
```

```
int fdos(int *x);   // Modifica variable.

// tipos pequeños, definidos por el usuario.

class pclase {
    int i;
public:
    pclase(int x) {i = x; }
    int fun_miem() {
        i = 2*i;
        return i;
    }
    friend int fcuatro(pclase &arg);   // Permiso explicito para
                                       // modificar datos privados
    void imprime() {
        cout << "Valor de i = " << i << endl;
    }
};


int ftres(pclase arg) // No modifica la variable externa.
{
    return arg.fun_miem();
}


int fcuatro(pclase &arg)    // Modifica la variable externa
{                           // por manipulación
    int k = 4*(arg.i);      // directa de sus datos miembro
    arg.i = k;
    return k;
}


void fcinco(pclase &arg)     // Modifica la variable externa
{                            // llamando a una función miembro.
    arg.fun_miem();
}


// tipos grandes, definidos por el usuario.

class gclase {
    enum { tam = 10 };
    int dato[tam][tam];
public:
    gclase(int x);
    void fun_miem();
    int fun_mod() const { return dato[5][5]; }
```

```
    friend void fsiete(gclase &);   // Permiso explicito para
                                    // modificar datos privados.
    void imprime();
};


gclase::gclase(int x)
{
        for(int i = 0; i < tam; i++)
            for(int j = 0; j < tam; j++)
                dato[i][j] = x;
}


void gclase::fun_miem()
{
        for(int j = 0; j < tam; j++)
            dato[9][j] = 200;
}



void gclase::imprime() {
        for(int i = 0; i < tam; i++) {
            for(int j = 0; j < tam; j++) {
                cout.width(5);
                cout << dato[i][j];
            }
            cout << endl;
        }
}


int fseis(const gclase &arg)        // No modifica la variable
{                                   // externa.
    int k;

    k = 3*arg.fun_mod();
    return k;
}


void fsiete(gclase &arg)          // modifica la variable externa
{                                 // por manipulación directa de
    for(int i = 0; i < tam1; i++)   // sus datos privados.
        *arg.dato[i] = 100;
}

void focho(gclase &arg)         // Modifica su variable llamando
```

```
{                                   // a una función miembro.
    arg.fun_miem();
}

void main()
{
    char c;
    int a = 5, b = 10;
    pclase A(1);
    gclase B(2);

    cout << endl;
    cout << "Valor de a =            " << a << endl;
    cout << "Retorno de funo(a =     " << funo(a) << endl;
    cout << "Valor de a =            " << a << endl;
    cout << endl;
    cout << "Valor de b =            " << b << endl;
    cout << "Retorno de fdos(&b) =   " << fdos(&b) << endl;
    cout << "Valor de b =            " << b << endl;
    cout << endl;
    cout << "Inicia análisis de clase pequeña (pclase)" << endl;
    A.imprime();
    cout << "Regreso de ftres(A) =   " << ftres(A) << endl;
    A.imprime();
    cout << "Regreso de fcuatro(A) = " << fcuatro(A) << endl;
    A.imprime();
    cout << "Llamada a fcinco(A)" << endl;
    fcinco(A);
    A.imprime();
    cout << "\nOprima [Enter] para continuar...";
    cin.get(c);
    clrscr();
    cout << "Inicia análisis de clase grande (gclase)" << endl;
    cout << "Regreso de fseis() =    " << fseis(B) << endl;
    cout << "Llamada a fsiete(B)" << endl;
    fsiete(B);
    B.imprime();
    cout << "Llamada a focho()" << endl;
    focho(B);
    B.imprime();
}


int funo(int x)
{
    return 2+x;
}

int fdos(int *x)
```

```
{
    *x = 4 + *x;
    return *x;
}
```

OBSERVACIONES.

Pase parámetros de tipos predefinidos.

```
funo(int x)    Pase por valor.
fdos(int *x)   Pase por referencia.
```

Pase de parámetros de tipos pequeños, definidos por el usuario.

```
ftres()        Función general. Pase de una clase por valor.
fcuatro()      Función amiga. Pase por referencia.
fcinco()       Función general. Pase por referencia.
```

Pase de parámetros de tipos grandes, definidos por el usuario.

```
fseis()        Función general. Pase por referencia,
               simulando un pase por valor.
fsiete()       Función amiga. Pase por referencia.
focho()        Función general. Pase por referencia.
```

En la corrida del programa podemos ir viendo cuando es modificado el dato miembro y cuando no lo es. Note que las funciones generales no pueden accesar los datos privados, ésto se puede hacer solo mediante las funciones miembro o funciones friend.

En la función fseis() se pasa un objeto como **const** y de igual forma se debe declarar la función referida por el objeto **const**, por ello se tuvo que declarar como función **const** a la función fun_mod().

Las funciones; función constructor, gclase(), funciones miembro, fun_miem() e imprime() no pueden ser manejadas como funciones **inline**, por su tamaño, y deben ser definidas fuera de la clase, como funciones normales.

Veamos la corrida.

```
Valor de a =            5          // Primer grupo.
```

```
Retorno de funo(a) =     7
Valor de a =             5

Valor de b =             10         // Segundo grupo.
Retorno de fdos(&b) =    14
Valor de b =             14

Inicia análisis de clase pequeña (pclase)    // Tercer grupo.
Valor de i = 1
Regreso de ftres(A) =    2
Valor de i = 1
Regreso de fcuatro(A) = 4
Valor de i = 4
Llamada a fcinco(A)
Valor de i = 8
```

Oprima [Enter] para continuar...

```
Inicia análisis de clase grande (gclase)     // Cuarto grupo.
Regreso de fseis() =    6
Llamada a fsiete(B)
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
Llamada a focho()
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  100    2    2    2    2    2    2    2    2    2
  200  200  200  200  200  200  200  200  200  200
```

En el primer grupo, se manejan tipos predefinidos, pase por valor, la variable no es alterada.

En el segundo grupo se manejan tipos predefinidos, pase por puntero, se altera la variable.

En el tercer grupo, se maneja un tipo de dato definido por el usuario, una clase pequeña, la función ftres(), recibe el objeto por valor, y no altera el contenido del dato privado, lo cual se puede apreciar en el tercer renglón de este grupo.

La función fcuatro(), es una función friend, tiene la capacidad de alterar el dato privado, lo hace multiplicandolo por 4. recibe el objeto por referencia.

La función fcinco(), es una función general recibe el objeto por referencia, usando la función miembro fun_miem() para modificar el dato privado. Lo modifica multiplicando su contenido por 2. Se puede ver el valor del dato miembro privado después de la llamada a la función fcinco().

En el cuarto grupo, se maneja un tipo de dato definido por el usuario, una clase grande, las tres funciones reciben el objeto por referencia. La función fseis(), se comporta como la función ftres(), aunque recibe el objeto por referencia, no puede alterar su contenido.

La función fsiete(), es una función friend, tiene el permiso para escribir sobre el dato privado, lo hace llenando la primer columna (columna 0), con el valor 100.

La función focho(), es una función general, solo puede modificar el dato privado usando una función miembro, lo hace, por medio de fun_miem(), miembro de la clase gclase. El resultado de esta modificación, es el llenado del decimo renglón (renglón 9), con el valor de 200.