Metodología de la Programación 2

Tema 1.- Introducción

Tipos de Datos
Existen dos tipos de datos: integrales (booleanos, caracteres o enteros) y flotantes (float, double, long double).

Los tipos de datos booleanos solo pueden tener el valor 0 (falso) ó 1 (verdadero) por tanto cualquier valor distinto de 0 será tratado como si fuera verdadero.

```
bool a;
cout << "Introduce 1 valor booleano: ";
cin >> a;
cout << a;
```

Los tipos carácter (char) tienen un valor comprendido entre 0 y 256. Podemos ajustar el rango de este tipo de variables usando la palabra reservada signed y unsigned. Si usamos el tipo signed char el rango de valores estará entre -127 y 127 y en caso de unsigned char el rango de valores estará 0 y 255.

```
unsigned char a = 300;    // Este valor es válido, solo almacenará 45, el resultado de 300%255
```

Los tipos de dato entero disponen de varias palabras reservadas para representarlos. En primer lugar, se pueden definir los tipos short o long, dichas palabras indicarán que el tipo de datos usará 2 ó 4 bytes para representar el tipo de datos entero (lo que implica un rango mayor de números enteros). La segunda palabra reservada es signed o unsigned (usar o no valores negativos). Por ejemplo, si usamos unsigned int, el rango de valores estará entre 0 y 2^{64}-1. En caso de usar signed int, el rango de valores estará entre -2^{32} y 2^{32}.

Salvo los valores signed short int, el resto podemos definirlos al crear una variable, veamos algunos ejemplos:

```
signed long int a = 17;          o bien int a = 17;
unsigned short int a = 17;       o bien int a = 17U;
unsigned long int a = 17;        o bien int a = 17UL;
```

Podemos también definir una variable en sistema hexadecimal u octal precediendo del carácter O ó 0x respectivamente, por ejemplo:

```
int a = O17;      sería 17 en octal, 15 en decimal
int a = 0x17;     sería 17 en hexadecimal,23 en decimal
```

Dentro de los tipos de coma flotante, los valores son almacenados por defecto en double (que es la variable de menor precisión). Puedo definir una variable de tipo flotante añadiendo tras su definición F, es decir:

```
a = 35F;
```

Otros Tipos de Datos
Podemos definir cualquier tipo de dato usando la palabra reservada typedef, veamos el siguiente ejemplo:

```
typedef int miEntero;

miEntero a;
```

O bién podemos definir estructuras más complejas de datos, pudiendo definirse el mismo tipo de dato de dos formas distintas, como en el siguiente ejemplo:

```
struct A {
        char c;
        int d;
};

typedef A miStruct;

// A partir de aquí, podemos usar miStruct así:

miStruct registro2;

// O bien de la siguiente forma:

A registro1;
```

Conversiones de Tipos de Datos
Los tipos de datos pueden realizar dos tipos de conversiones: implícitas y explicitas.

Las conversiones implícitas se realizan en tiempo de ejecución y transforman un tipo de dato para poder operar con él en un momento dado. Suelen ocasionar warnings (advertencias). Por ejemplo:

```
// Sea la siguiente funcion definida que maneja los siguiente tipos de datos: double y float:

void Funcion (double a, float f);

// Si encontramos la siguiente llamada de código:

Funcion(17,3.5);

// Convertimos implícitamente el tipo de dato17 (un tipo int) a double
// Convertimos implícitamente el tipo de dato 3.5 (un tipo double) a float
```

Veamos otro ejemplo:

```
bool par (unsigned char n) {
        return n%2;
}

// Convertimos implícitamente el tipo de dato n (un tipo unsigned char) a int
// Convertimos implícitamente el tipo de dato int (el resultado de la operación n%2) a bool
```

Ahora veamos un ejemplo más práctico:

```
int *p;

if (p)
        cout << "No es nulo/n";
else
        cout << "Es nulo/n";

// Creamos un puntero y comprobamos si tiene memoria asignada (si no tuviese memoria
asignada su valor sería 0), por tanto haciendo una conversión implícita del valor de memoria a
un tipo bool que realiza la operación if(p). Si tenemos asignada cualquier posición de memoria,
el valor de esa comparación será verdadero y ejecutará la sentencia cout << "No es nulo/n"y
en caso contrario, cout << "Es nulo/n";
```

Las conversiones explicitas se realizan en tiempo de compilación y transforman un tipo de dato para poder operar con él en un momento dado. Se conoce como casting (conversión). Por ejemplo:

```
float k;

int a = (int) k;

// Convertimos explícitamente el tipo de dato float k en un int. Este tipo de conversión se conoce como conversión (tipo)expresión

int b = int (k);

// Convertimos explícitamente el tipo de dato float k en un int. Este tipo de conversión se conoce como conversión tipo(expresión) y es propia de C++.

int c = static_const<int> k;

// Convertimos explícitamente el tipo de dato float k en un int. Este tipo de conversión se conoce como conversión static_const<tipo>(expresión)

int d = reinteger_cast<int> k;

// Otra conversión del tipo de datos float a int
```

Veamos otro ejemplo, donde mezclaremos varios tipos de conversión:

```
# include<iostream>

using namespaces std;

unsigned char Negativo(unsigned char x) {
        return 255-x;
}

int main(){
        int g;
        cout << "Introdusca un valor entre 0 y 255: ";
        cin g;
        cout << static_const<int>(Negativo(g));  // Otra conversión del tipo de datos float a int
}
```

Características de Tipos de Datos

Todos los tipos de Datos presentan una serie de características comunes: tamaño, límites numéricos y operadores a nivel de bit.

El tamaño (sizeof), indica el número de bytes que ocupa un tipo de dato. Por ejemplo:

```
int a;

cout << sizeof(int);
cout << sizeof(a);

struct A {
        int d;
        float k;
};

cout << sizeof(A);
```

Se cumplen las siguientes dependencias de tamaño de tipos:

- sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
- 1 <= sizeof(bool) <= sizeof(long)
- sizeof(float) <= sizeof(double) <= sizeof(long double)

Los límites numéricos pueden ser heredados de C, si usamos las librerias climit (para números enteros) o cfloat (para números de coma flotante) o bien propios de C++ si usamos la librería limits. Veamos un ejemplo de uso de climit:

```
#include <climit>

INT_MAX;        // Límite máximo para números enteros
INT_MIN;        // Límite mínimo para números enteros
```

De forma análoga podemos usar cfloat:

```
#include <cfloat>

FLT_MAX;        // Límite máximo para números de coma flotante
FLT_MIN;        // Límite mínimo para números de coma flotante
```

Y para usar limits, debemos utilizarlo con el siguiente prototipo numeric_limit<tipo>::funcion(), veamos como sería:

```
#include <limits>

numeric_limit<int>::max();          // Límite máximo para números enteros
numeric_limit<int>::min();          // Límite mínimo para números enteros
numeric_limit<float>::max();        // Límite máximo para números de coma flotante
numeric_limit<float>::min();        // Límite mínimo para números de coma flotante
```

El siguiente programa, calcula el mayor elemento de un vector, veamos como es su código:

```
#include <iostream>
#include <limits>

int Maximo(int *v, int n) {
        int min = numeric_limit<int>::min();

        for (int i = 0; i<n; i++) {
                if (n<v[i])
                        n=v[i];
        } //endfor

        return n;
}

int main() {
        int v[10] = {-100,100,150,50,25,26,35,7,-7,7,-153};

        cout << Maximo(v,10)
}
```

Para realizar la operación inversa modificaríamos el código fuente de forma que quedaría:

```
#include <iostream>
#include <limits>

int Minimo(int *v, int n) {
        int max = numeric_limit<int>::max();

        for (int i = n; i>0; i--) {
                if (n>v[i])
                        n=v[i];
        } //endfor
        return n;
}

int main() {
        int v[10] = {-100,100,150,50,25,26,35,7,-7,7,-153};

        cout << Minimo(v,10)
}
```

Para comprender mejor los operadores a nivel bit debemos entender como es la estructura interna de los datos. Sea unsigned char a = 10; ha nivel de bits, será almacenado de la siguiente forma:

0	0	0	0	1	0	1	0

Las operaciones a nivel de bits son 6: 4 a nivel lógico y 2 de desplazamiento. Las operaciones de nivel lógico son Y lógico (AND), O lógico (OR), O exclusivo lógico (XOR) y negación lógica (NOT), cuyas salidas representan las siguientes tablas:

Tabla de verdad del operador AND (&)		
Valor 1	**Valor 2**	**Salida**
0	0	0
0	1	0
1	0	0
1	1	1

Tabla de verdad del operador XOR (^)		
Valor 1	**Valor 2**	**Salida**
0	0	0
0	1	1
1	0	1
1	1	0

Tabla de verdad del operador OR (|)		
Valor 1	**Valor 2**	**Salida**
0	0	0
0	1	1
1	0	1
1	1	1

Tabla de verdad del operador NOT (!)	
Valor 1	**Salida**
0	0
0	0
1	0
1	1

Las operaciones de desplazamiento son 2 dos: desplazamiento de bits a la izquierda (<<) o a la derecha (>>). Un desplazamiento a la izquierda implica multiplicar por un número multiplicado por dos elevado al número de desplazamientos y un desplazamiento a la derecha significa ese mismo número divido por 2 elevado al número de bits desplazados en ese sentido. Veamos algunos ejemplos de uso:

```
unsigned char v1=10, v2=8, v3;

v3=v1&v2;       // Realizamos la operación AND
v1=v1<<3;       // Multiplicamos por 2^3 (8) v1
v2=v2>>2;       // Dividimos por 2^2 (4) v2
```

La representación interna de v1 en la primera sentencia es:

0	0	0	0	1	0	1	0

La representación interna de v2 en la primera sentencia es:

0	0	0	0	1	0	0	0

La representación interna de v3 en la primera sentencia es:

0	0	0	0	0	0	0	0

Una vez finalizado el ejemplo, la representación interna de v1 es:

0	1	0	1	0	0	0	0

Vemos que en lugar de almacenar el valor 10, ahora almacena el valor 80.
De forma similar, una vez finalizado el ejemplo, la representación interna de v2 es:

0	0	0	0	0	0	1	0

Vemos que en lugar de almacenar el valor 8, ahora almacena el valor 2.
Finalmente, una vez finalizado el ejemplo, la representación interna de v3 es:

0	0	0	0	1	0	0	0

Vemos que en lugar de almacenar el valor 0, ahora almacena el valor 8.
Vamos a realizar algo más práctico, para ello analizaremos el código fuente de la implementación de un programa que lea BIT a BIT el contenido de un dato.

```
unsigned char ImprimirBITsus (unsigned char x) {
        unsigned char mask = 1;
        // Creamos una variable auxiliar con el BIT menos representativo a 1

        for (int i = 0; i < x; i++) {
                if (x&mask)       // Miramos si el BIT esta activo o no con la operación AND
                        cout << 1;
                else
                        cout << 0;
        mask = mask<<1;           // Comprobamos el siguiente BIT a la izquierda
        } //endfor
}
```

El ejemplo recorre un dato de izquierda a derecha, si deseáramos realizar ese recorrido al revés, el código fuente sería similar al siguiente:

```
unsigned char ImprimirBITsus (unsigned char x) {
        unsigned char mask = 1<<7;
        // Creamos una variable auxiliar con el BIT mas representativo a 1

        for (int i = 0; i < x; i++) {
                if (x&mask)       // Miramos si el BIT esta activo o no con la operación AND
                        cout << 1;
                else
                        cout << 0;
        mask = mask>>1;           // Comprobamos el siguiente BIT a la derecha
        } //endfor
}
```

Otro detalle interesante es que el siguiente fragmento de código:

```
if (x&mask)
        cout << 1;
else
        cout << 0;
```

Puede ser reemplazado por la siguiente sentencia cout << ((x&mask)?1:0);

Veamos una nueva implementación donde el dato que leamos sea unsigned int:

```
unsigned int ImprimirBITsui (unsigned int x) {
        unsigned int mask = 1<<7;
        // Creamos una variable auxiliar con el BIT mas representativo a 1

        for (int i = 0; i < zsizeof(x); i++) {
                cout << ((x&mask)?1:0);
        mask = mask>>1;
        } //endfor
}
```

Para leer una cadena de texto, lo primero que debemos hacer es recorrer la cadena carácter a carácter y detenernos cuando lleguemos al final de la misma, para ello todas las cadenas en C están finalizadas con el carácter especial '/0'. La implementación de la lectura de cadenas sería asi:

```
unsigned char ImprimirBITsuc (unsigned char x) {
        unsigned char mask = 1<<7;
        // Creamos una variable auxiliar con el BIT mas representativo a 1

        for (int i = 0; i < x; i++) {
                cout << ((x&mask)?1:0);
        mask = mask>>1;          // Comprobamos el siguiente BIT a la derecha
        } //endfor
}

void ImprimirBITsCadena (char *cad) {
        int longitud = 0;
        while (cad[longitud]!='/0') {
                longitud++;
        } //endwhile
        for (int i = longitud;  i = 0; i--)
                ImprimirBITsuc(cad[i]);
}
```

En caso de querer consultar un BIT determinado, el siguiente ejemplo nos lo muestra:

```
#include<iostream>

bool Activado (int n, int nbit) {
        int mask = 1 << nbits;

        (n&mask)?true:false);
}

int main () {
```

```
        int numero, bit;

        cout << "Introduzca un número: ";
        cin >> numero;
        cout << "Introduzca el número de BIT: ";
        cin >> nbit;
        cout << Activado(numero,bit);
}
```

Para cambiar el valor de un BIT de una variable, el siguiente ejemplo nos lo muestra:

```
#include<iostream>

void Cambiar (int n, int nbit) {
        int mask = 1 << nbits;

        n = n ^ mask;
}
```

Tipos de Datos Enumerados
Los tipos de enumerados se utilizan para ahorrar el espacio de memoria usado. Simplificando el espacio de variables y por tanto la cantidad de memoria usada. Veamos la diferencia en el siguiente ejemplo:

```
// El siguiente struct es poco eficiente porque usa dos campos de 16 bits.

struct carta  {
        int num;
        int palo; // Asigno un valor numérico a un palo
};

// El siguiente strctu es mas eficiente porque usamos 16 bits de un tipo int y el segundo campo de
almacenamiento ocupará menos de 16 bits

enum Palo {bastos, copas, espadas, oros};

struct carta {
        int num;
        Palo palo;
};
```

El uso de un tipo de dato enumerado una vez definido es muy similar a cualquier struct, veamos el siguiente ejemplo:

```
enum Palo {bastos, copas, espadas, oros};

struct carta {
        int num;
        Palo palo;
};

void EscribePalo (Palo p) {
        switch (p) {
                case bastos:
```

```
                                cout << "bastos" << endl;
                                break;
                        case copas:
                                cout << "bastos" << endl;
                                break;
                        case espadas:
                                cout << "bastos" << endl;
                                break;
                        case oros:
                                cout << "bastos" << endl;
                                break;
                } //endswitch
}
```

```
int main () {
        int a;
        Palo p;

        cout << "Introduce un palo: ";
        cin >> a;
        p = (Palo) a;       // Realizamos una conversión implícita de tipo, de string a Palo
        EscribePalo(p);
}
```

Funciones
La cabecera de una función main (función principal de un programa) puede ser de dos tipos:

- int main () {}: un main sin parámetros.
- Int main (int argc, char * argv []) {}: un main con parámetros donde argc es el número de parámetros incluido el programa y argv el parámetro contenido.

```
int PasaEntero (char * cadena) {
        int resultado = 0;

        for (int i = 0; cadena[i] != '\0'; i++) {
                resultado = 10 * resultado + (cadena[i] – '0');
        } //endfor

        return resultado;
}

int main (int argc, char * argv[]) {
        int a, b;

        if (argc != 3) {
                cout << "Dame dos enteros.";
                return 0;
        } //endif

        a = PasaEntero(argv[1]);
        b = PasaEntero(argv[2]);
        cout << a+b;
}
```

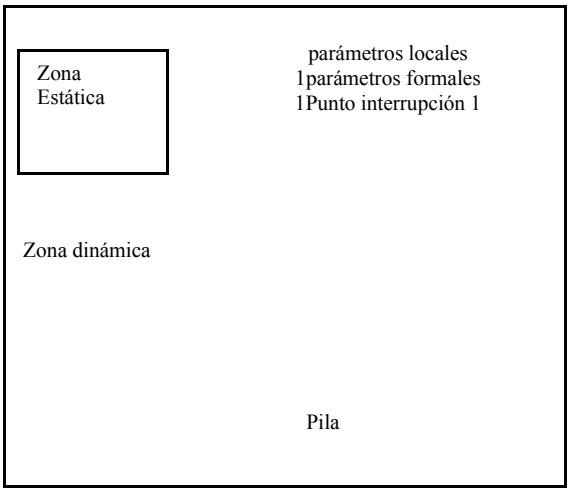
En la librería cstdlib la función atoi realiza la misma función que PasaEntero en el ejemplo anterior y atof permite la conversión de char * a float.

Sea el siguiente código fuente:

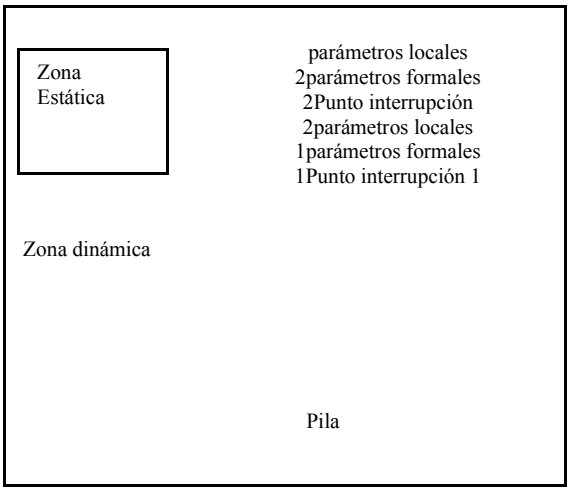
```
void funcion1 (parámetros formales) {
        variables locales;
        funcion2 (parámetros actuales)
}
```

```
void funcion2 (parámetros formales) {
        variables locales;
}
```

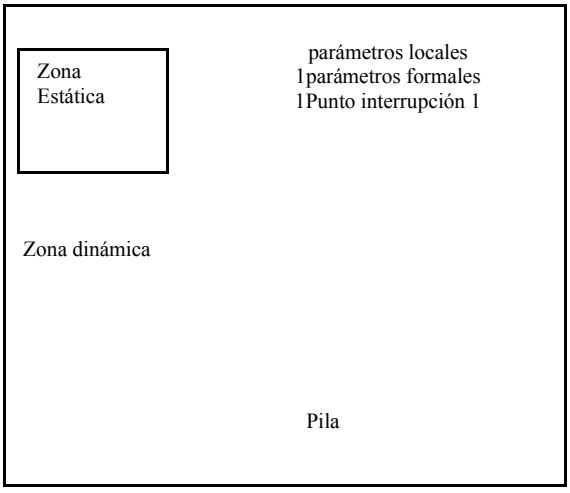
Mientras se ejecuta función1 la situación de la memoria dinámica será la siguiente:



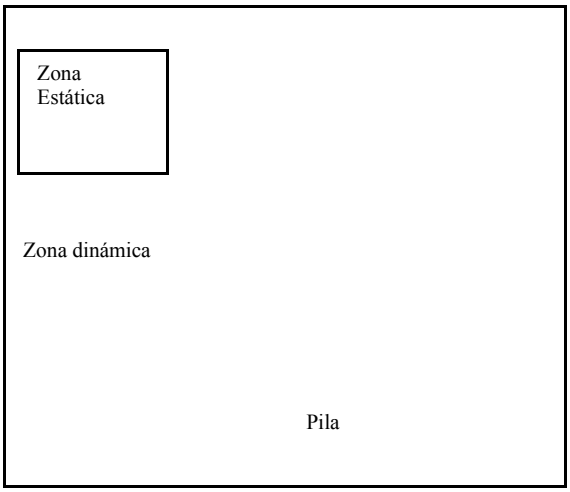
Cuando se ejecuta función2 la situación de la memoria dinámica será la siguiente:



Una vez finalizada la ejecución de función2 la situación de la memoria dinámica será la siguiente:



Cuando se ejecuta función1 la situación de la memoria dinámica será la siguiente:



El paso de parámetros a funciones de C puede ser realizado por valor o por referencia, la diferencia entre ambos pasos es el uso que daremos de la memoria en el momento del uso de los parámetros. Veamos el siguiente ejemplo sobre el uso del paso por valor:

```
int Maximo (int v1, int v2) {
        int r = 0;

        r = ((v1 < v2)?v1:v2);

        return r;
}

int main (int argc, char * argv[]) {
        int a, b, resultado;

        a = 5;
        b = 7;
        resultado = 3*Maximo(a, b);
        cout << resultado;
}
```

En el ejemplo anterior si representásemos el espacio de memoria en uso tendríamos el siguiente estado inicial:

a	**…**	**b**	**…**	**resultado**
5	**…**	7	**…**	?

Al ejecutar la función Maximo aparecerá un nuevo espacio de memoria reservado para las variables de esa función, por tanto nuestra memoria tomaría un aspecto similar a este:

a	**…**	**b**	**…**	**resultado**	**…**	**v1**	**…**	**v2**	**…**	**r**
5	**…**	7	**…**	?	**…**	7	**…**	5	**…**	**?**

Tras finalizar la ejecución de la función Maximo la situación de la memoria será:

a	**…**	**b**	**…**	**resultado**	**…**	**v1**	**…**	**v2**	**…**	**r**
5	**…**	7	**…**	?	**…**	7	**…**	5	**…**	**7**

En ese momento devolvemos el valor de r y operamos con él para obtener resultado para finalizar liberando la memoria ocupada. Por tanto la situación de la memoria antes de finalizar la ejecución del programa principal será similar al siguiente aspecto:

a	**…**	**b**	**…**	**resultado**
5	**…**	7	**…**	21

Veamos ahora el siguiente ejemplo sobre el uso del paso por referencia:

```
int Maximo (int v1, int v2, int *r) {
        *r = ((v1 < v2)?v1:v2);

        return r;
}

int main (int argc, char * argv[]) {
        int a, b, resultado;

        a = 5;
        b = 7;
        Maximo(a, b, resultado);
        cout << 3*resultado;
}
```

En el ejemplo anterior si representásemos el espacio de memoria en uso tendríamos el siguiente estado inicial:

a	**…**	**b**	**…**	**resultado**
5	**…**	7	**…**	?

Al ejecutar la función Maximo aparecerá un nuevo espacio de memoria reservado para las variables de esa función, por tanto nuestra memoria tomaría un aspecto similar a este:

a	**…**	**b**	**…**	**resultado**	**…**	**v1**	**…**	**v2**	**…**	**r**
5	**…**	7	**…**	?	**…**	7	**…**	5	**…**	**&resultado**

Podemos observar que no se creado ningún espacio para r, porque r realmente esta apuntando a la dirección de memoria usada por resultado Tras finalizar la ejecución de la función Maximo la situación de la memoria será:

a	**…**	**b**	**…**	**resultado**	**…**	**v1**	**…**	**v2**	**…**	**r**
5	**…**	7	**…**	7	**…**	7	**…**	5	**…**	**&resultado**

En ese momento modificaremos la dirección de memoria almacenada por r y operamos con la variable resultado. Por tanto la situación de la memoria antes de finalizar la ejecución del programa principal será similar al siguiente aspecto:

a	**…**	**b**	**…**	**resultado**
5	**…**	7	**…**	21

Veamos ahora el siguiente ejemplo sobre el uso del paso por referencia, en el siguiente ejemplo vamos a intercambiar dos índice de un vector sin necesidad de crear un vector auxiliar, de tal manera que ahorraremos memoria:

```
void Intercambiar (int * vector, int indice1, int indice2) {
        int aux;
        aux = valor [indice1];
        vector[indice1] = vector[indice2];
        vector[indice2] = aux;
}

int main () {
        int v[5] = {0, 1, 2, 3, 4};
        int v1 = 3;
        int v2 = 4;

        Intercambiar (v, v1, v2);
}
```

El siguiente ejemplo presenta un desarrollo usando aritmética de punteros:

```
char * PrimerEspacio (char * input) {
        char *p = input;

        while ((*p != '\0') && (*p != ' ')) {
                p++;
        } //endwhile

        if (*p = ' ')
                return p;
        else
                return p--;
}
```

Si lo comparamos con el siguiente ejemplo, que no usa la aritmética de punteros, podremos ver con mayor claridad que realiza el código:

```
char * PrimerEspacio (char * input) {
        int i = 0;

        while ((p[i] != '\0') && (p[i] != ' ')) {
                i++;
        } //endwhile

        if (p[i] = ' ')
                return &p[i];
        else
                return &p[i-1];
}
```

En este caso devolvemos una dirección de memoria porque tengo que dar un char *, por tanto provoco una conversión de los datos contenidos en esa dirección de memoria.

En el siguiente fragmento de código, vamos a usar aritmética de punteros. Después de la ejecución del código, el puntero prueba apuntaría al principio de la cadena “Hola y adiós”, durante la ejecución tendríamos un segundo puntero, imput, que apuntaría a la siguiente dirección “ y adiós”. Por tanto, tenemos dos espacios de memoria destinados a apuntar el espacio de memoria de la cadena prueba, el propio puntero de prueba y otro auxiliar.

```
char * PrimerEspacio (char * input) {
        while ((*input != '\0') && (*input != ' ')) {
                input++;
        } //endwhile

        if (*input = ' ')
                return input;
        else
                return input-1;
}

int main () {
        char *prueba = "Hola y adiós";

        cout << PrimerEspacio(prueba);
}
```

El paso por referencia en C++ es una forma de compartir el mismo valor de una variable con distinto nombre, veamos el siguiente ejemplo:

```
void Maximo(int v1, int v2, int &r) {
        r = ((v1>v2)?v1:v2);
}

int main () {
        int ant, nue, res;

        Maximo(ant, nue, res); // El valor res es asignado por referencia de un valor existente
}
```

El paso por referencia puede hacerse de dos formas:

- tipo <nombre de tipo> ref = a;
- <nombre de tipo> = & <nombre de nueva referencia>

Veamos un ejemplo:

```
int main () {
        int a = 6;
        int d ref = a; // d es una referencia de la variable a.
        a = &t   // t es una referencia de la variable a.
}
```

Si modificamos el valor de la variable a modificamos el valor referenciado por d y t. Y si modificamos el valor de las referencias d o t, modificaremos realmente el valor contenido por la variable a. El siguiente ejemplo vemos es la devolución por referencia:

```
int & Direccion() {
        int i;

        return i;

}

int & Valor(int * v, int pos) {

        return v[pos];

}

int main () {
        int vec[5] = {10 ,20, 30, 40, 50};

// Las siguientes operaciones son correctas porque son nombres alternativos que se dan a un mismo objeto en memoria.

        Valor(v,1) = 3;
        v[2] = 4;
        v[3] = v[0] + Valor(v,4);

// La siguiente línea generará un error de tipo "NULL pointer" porque la variable i deja de existir tras la ejecución de la función Direccion()

        Direccion();
}
```

Veamos un uso práctico del uso de valores referenciados para variables y punteros y como modificarlos:

```
int global = 5;
int * p;

int * Funcion1 () {

        return & global;

}

int & Funcion2 () {

        return global;

}

int main () {
        global = 7;
        * Funcion1() = 7;
        Funcion2() = 7;
        p = & global;
        p = Funcion1();
        p = &Funcion2();
```

```
}
```

En algunas ocasiones, es posible que sea necesario usar parámetros de tipo constante. Dichos parámetros declaran una variable constante para que no sea modificado cuando se pasa por referencia a una función. De esta forma, si deseamos modificar el elemento almacenado, tendremos un error de compilación.

Cuando estamos trabajando con variables de gran tamaño, veamos como se comporta el uso de parámetros constantes.

```
struct Gigante {
};

void Funcion1 (Gigante G) {}

void Funcion2 (Gigante & G) {}

void Funcion3 (Gigante *G) {}

void Funcion4 (const Gigante G) {}

void Funcion5 (const Gigante & G) {}

void Funcion6 (const Gigante *G) {}

int main () {
        Gigante Gi;

        Función1(Gi); // Duplica el espacio ocupado en memoria
        Función2(Gi); // El espacio de memoria es el mismo, los cambios afectarán a Gi
        Función3(&Gi); // Pasamos una dirección de memoria y modificamos Gi por puntero
        Función4(Gi); // Duplica el espacio ocupado en memoria pero el espacio es constante
        Función5(Gi); // La variable G apuntará a Gi pero no podrá modificarlo
        Función3(&Gi); // Pasamos una dirección de memoria y Gi no podrá modficarse
}
```

Finalmente, veamos el siguiente ejemplo:

```
int & Funcion (const int * v, int pos) {

        return v[pos];

}

/* Este código producirá un error de compilación. Nos indica que estamos asignando un valor
constante como no constante, para arreglar este error debemos cambiar el código de la
siguiente forma: */

const int & Funcion (const int * v, int pos) {

        return v[pos];

}
```

Por tanto, podemos hacer constante algo no constante, pero debemos evitar que una variable constante pase a no constante porque provocaremos un error de compilación.

Funciones Inline
Son funciones que tienen poco coste de ejecución. La función inline y su encabezado han de estar en la librería (el archivo .h) para estar listas para el preprocesador.

```
(inline) bool Par (int n) {

        return (n%2 == 0);

}
```

Sobrecarga de Funciones
Podemos tener varias funciones llamadas de igual manera pero diferenciadas por los parámetros de llamada. Veamos el siguiente ejemplo:

```
void Funcion (const char * cad) {
        cout << "Has llamado una cadena constante" << endl;
}

void Funcion (char * cad) {
        cout << "Has llamado una cadena no constante" << endl;
}

int main () {
        const char * cadcte;
        char * cad;

        Funcion ("Prueba"); // Ejecuta void Funcion (const char * cad)
        Funcion (cadcte); // Ejecuta void Funcion (const char * cad)
        Funcion (cad); // Ejecuta void Funcion ( char * cad)
}
```

Parámetros con Valor por Sobrecarga
Al asignar a n un valor por defecto. Existen varias posibilidades de llamada: por parámetros normales o no incluyendo los valores de los parámetros que deseamos incluir por defecto. Por ejemplo:

```
void Imprime (char * cad, int n = 1) {
        for (int i = 0; i < n; i++) {
        cout << cad
        } //endfor
}

int main () {
        Imprime ("Hola",5); // Imprime 5 veces Hola
        Imprime ("Hola"); // Imprime 1 única vez Hola
}
```

Variables Locales Estáticas
Una variable estática (static) es una variable que solo se conoce dentro de su ámbito y permanece con su valor hasta el final de la ejecución de un programa:

```
int Máximo (int n) {
        static int m = 0;

        if (m < n)
                m = n;

        return m;

}

int main () {
        int v;
        do {
                cout << "Introduce un valor de 0 a 10: "
                cin >> v;
                cout << Maximo(v);
        } while (v != 10);
}
```

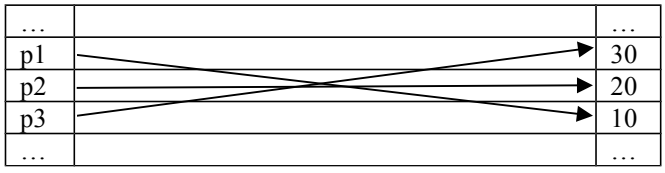
Operaciones sobre Punteros
Existen cuatro tipos de operaciones: asignación (=), composición, dirección (*) e indirección (&).

Para comprender mejor el funcionamiento de los punteros tomemos para el siguiente ejemplo la siguiente distribución de variables y sus respectivos valores dentro de la memoria.

	10		20		30		40		*Posición de memoria*
…	**a**	**…**	**p1**	**…**	**p2**	**…**	**p3**	**…**	*Nombre variable*
	5		10		20		30		*Valor variable*

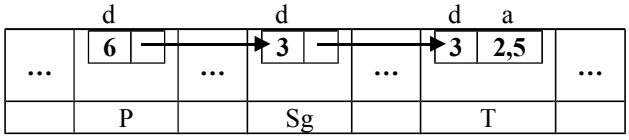
```
int main () {
        int a = 5;
        int *p1 = &a;
        int **p2 = &p1;
        int ***p3 = &p2;

        cout << p3; // La salida de este valor sería 30
        cout << *p3; // La salida de este valor sería 20
        cout << **p3; // La salida de este valor sería 10
        cout << ***p3; // La salida de este valor sería 5
}
```

En el siguiente ejemplo, tenemos un vector de punteros que apuntan a distintos valores de otro vector, vamos a hacer que esos punteros apunten a las posiciones del otro vector de forma que si los leyeramos linealmente los elementos del otro vector apareciesen ordenados de menor a mayor.



```
int main () {
        int*p[3];
        int m[3];

        p[0] = &(m[0]);
        p[1] = &(m[1]);
        p[2] = &(m[2]);

        if ( (*(p[1]) < *(p[0])) && (*(p[1]) < *(p[2])) ) {
                int *aux = p[1];
                p[1] = p[0];
                p[0] = aux;
        } // endif
        else {
                if ( (*(p[2]) < *(p[0])) && (*(p[2]) < *(p[1])) ) {
                        int *aux = p[2];
                        p[2] = p[0];
                        p[0] = aux;
                } //endif
        } // endelse
}
```

Operaciones de Punteros sobre Estructuras
Supongamos la siguiente estructura:



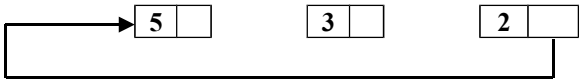
La estructura sería definida de la siguiente forma:

```
struct Tercero {
        int d;
        float a;
};

struct Segundo {
        int d;
        tercero *ptr;
};

struct Primero {
        int d;
        segundo *ptr;
};
```

Veamos ahora como accederíamos a todos sus elementos:

```
int main() {
        Tercero T;
        Segundo Sg;
        Primero P;
        Primero *p;

        p = &P;
        p->d = 5;
        p->ptr = &Sg;
        p->ptr->d = 3;
        p->ptr->ptr = &T;
        p-> ptr->ptr->d = 3;
        p-> ptr->ptr->f = 2.5;
        }
```

En el siguiente ejemplo, supongamos la siguiente estructura:



La estructura sería definida y completada de la siguiente forma:

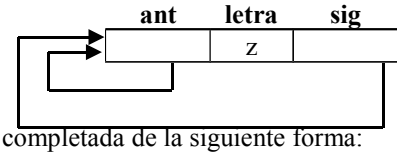
```
struct Celda {
        int d;
        Celda *ptr;
};

int main() {
        Celda vec[3];
        vec[0].d = 5;
        vec[0].ptr = &vec[1];
        vec[1].d = 3;
        vec[1].ptr = &vec[2];
        vec[2].d = 2;
        vec[2].ptr = &vec[0];
        }
```

O bien, podemos definir y completar de esta otra forma:

```
struct Celda {
        int d;
        Celda *ptr;
};

int main() {
        Celda vec[3];

        vec[0].d = 5;
        vec[0].ptr = &vec[1];
        vec[0].ptr->d = 3;
        vec[0].ptr->ptr = &vec[2];
        vec[0].ptr->ptr->d = 2;
        vec[0].ptr->ptr->ptr = &vec[0];
```

```
        }
```

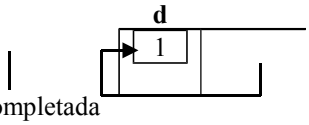
Como podemos observar en los ejemplos, los punteros pueden usarse para apuntar a otros elementos de la estructura, pero también puede hacerlo sobre la misma estructura como en el siguiente ejemplo. Supongamos la siguiente estructura:



La estructura sería definida y completada de la siguiente forma:

```
struct doble {
        doble *ant, *sig;
        char letra;;
};

int main() {
        doble var;

        var.letra = 'z';
        var.ant = &var;
        var.sig = &var;
        }
```

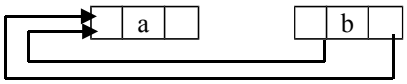
O bien podemos apuntar a un campo específico de la estructura. Por ejemplo, supongamos la siguiente estructura:



La estructura sería definida y completada de la siguiente forma:

```
struct S {
        int d;
        int *pint;
};

int main() {
        S var;

        var.d = 1;
        var.pint = &var.d;
        }
```

Veamos de forma práctica como usar todos los conceptos que hemos desarrollado hasta ahora. Supongamos la siguiente estructura:

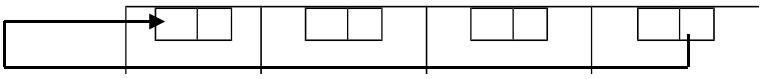


La estructura sería definida y completada de la siguiente forma:

```
struct S {
        char letra;
        S *pant, psig;
};

int main() {
        S var1, var2;

        var1.letra = 'a';
        var1.pant = &var1;
        var1.psig = &var2;
        var2.letra = 'b';
        var1.pant = &var1;
        var1.psig = &var1;
        }
```

El compilador admite definir un puntero a una estructura, dado que se trata de una dirección de memoria lo que debe reservar sin conocer el espacio ocupado por el struct.

Por ejemplo, supongamos que deseamos definir la siguiente estructura:



La estructura sería definida y completada de la siguiente forma:

```
struct Cada {
        Celda  * p;
};

struct Celda {
        int d;
        Celda  * p;
};
```

Punteros a Funciones
Supongamos que deseo implementar una función que ponga a cero el valor contenido por una variable e implemento de la siguiente forma:

```
void HacerCero(int *p) {
        *p = 0;
}

int main() {
        int a = 5;
        int  * q = &a;

        HacerCero(q);
}
```

Según el ejemplo este sería el estado inicial de la memoria

a		**q**	*Nombre variable*
5	…	10	*Valor variable*

10	20	*Posición de memoria*

En el momento que llamamos a la función HacerCeros(q) la situación es la siguiente:

a		**q**		**p**	*Nombre variable*
5	…	10	…	5	*Valor variable*
10		20		30	*Posición de memoria*

Durante la ejecución de HacerCeros(q), al ejecutar la sentencia **p = 0;* lo que realmente hacemos es lo mismo que decir *a = 0*; por tanto, la situación es la siguiente:

a		**q**		**p**	*Nombre variable*
0	…	10	…	0	*Valor variable*
10		20		30	*Posición de memoria*

Veamos ahora el siguiente ejemplo:

```
struct Puntero {
        int dato;
        int *pint;
};

void HacerCero(Puntero p) {
        p.dato = 0;
        *p.pint = 0;

}

int main() {
        int a = 5;
        Puntero orig = {3,&a};

        HacerCero(orig);
}
```

Según el ejemplo este sería el estado inicial de la memoria

a		**orig.dato**		**orig.pint**	*Nombre variable*
5	…	3	…	10	*Valor variable*
10		20		30	*Posición de memoria*

En el momento que llamamos a la función HacerCeros(orig) la situación es la siguiente:

a		**orig.dato**		**orig.pint**	*Nombre variable*
5	…	3	…	10	*Valor variable*
10		20		30	*Posición de memoria*

Durante la ejecución de HacerCeros(orig), al ejecutar las sentencia *p.dato = 0;* y **p.pint = 0;* la situación es la siguiente:

a		**orig.dato**		**orig.pint**		**p.dato**		**p.pint**	*Nombre variable*
0	…	0	…	10	…	0	…	10	*Valor variable*
10		20		30		200		250	*Posición de memoria*

Veamos ahora el siguiente ejemplo:

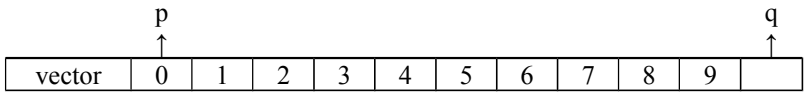
```
int * funcion() {
        int i = 3;
        int * res = &i;

        return res;
}

int main() {
        int a = *funcion();
        cout << a;
}
```

Este ejemplo es una mala práctica de programación, en él devolvemos la dirección de un objeto local, el cual deja de existir al terminar *funcion()*, por tanto no controlamos lo que devolvemos realmente.

Veamos en el siguiente ejemplo, como funciona la aritmética de punteros:

```
int main() {
        int vector[10];
        int *p, *q;

        p = vector;
        q = vector +10;
        while (p != q) {
                *p = p – vector;
                p++;
        } //endwhile
}
```

En este caso veamos que ocurriría si representamos el vector con ese código fuente:



p apuntará a la primera posición del vector, mientras q apuntará a la última posición de vector, mientras p apuntará en cada nueva iteración una casilla adyacente a la derecha.

Punteros Constantes

El uso de punteros constantes es destinado a mantener valores o direcciones de memoria con las que trabajaremos. Los compiladores nos permitirán hacer referencia de algo no constante como constante pero no lo inverso. Veamos unos cuantos ejemplos:

```
int main() {
        int *p;
        *p = 3;
        p = 100;

        // Veamos ahora un uso no correcto de punteros constantes
        const int * const p;
        *p = 3;  // Error de compilación, asigno valor a una constante (valor apuntado por p)
        p = 100;// Error de compilación, cambio una dirección de memoria constante

        // Veamos ahora un uso parcialmente correcto de punteros constantes
```

```
        const int * p;
        *p = 3;  // Error de compilación, asigno valor a una constante (valor apuntado por p)
        p = 100;

        // Veamos ahora un uso parcialmente correcto de punteros constantes
        int * const p;
        *p = 3;
        p = 100; // Error de compilación, cambio una dirección de memoria constante
}
```

Vamos a repasar en los siguientes ejemplos:

```
int main() {
        double f1;
        const double *p = &f1;
        *p = 50; // Error de compilación, el puntero almacena un valor constante

        const double f2 = 5;
        double *p = &f2;
        p = 5.0; // Error de compilación, el puntero almacena un valor constante

        double a = 5.5;
        double * const p = &a;
        *p = 3.2;
        p++; // Error de compilación, el puntero tiene una dirección constante
        const double * const p = &a; // Error de compilación, el puntero tiene una dirección
constante y el valor que contiene es constante
}
```

Vamos a tratar de localizar ahora el error en el siguiente código:

```
void HacerCero(int *p) {
        *p = 0;
}

void EscribirCero(const int *p) {
        cout << *p;
}
int main() {
        const int n = 1;
        int b = 2;
        HacerCero(&a); // Error de compilación, la dirección de a no es constante
        EscribirCero(&a);
        EscribirCero(&b);
}
```

Cadenas de Caracteres

Para trabajar con cadenas de caracteres usaremos la librería cstring cuyas funciones principales son:

- strlen(cadena): Tamaño de cadena
- strcopy(tmp, cadena): Copia cadena en tmp
- strcat(tmp, cadena): Añade cadena a tmp
- strcmp(tmp, cadena): Compara las cadenas tmp y cadena. Da por resultado 0 si las cadenas son iguales y distinto de 0 si son cadenas distintas.

Veamos unos ejemplos prácticos de uso de cadenas:

```
int main() {
        string a, b;

        a = "Hola";
        b = "Adios";
        cout << a.size();
        a = b; // Asignación de cadenas
        a = a + b; // Concatenación de cadenas
        a == b; // Comparación de cadenas
        a = cad;
        strcopy(tmp, a.c_str()) // Copia en tmp una copia de la cadena a.
        Escribir(cad + 6);  // Los *char apuntan a la dirección de memoria del primer elemento
}
```

Veamos un ejemplo práctico:

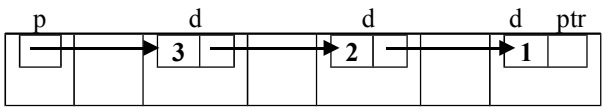
```
#include <cstring>

void Escribir (const char *p) {
        cout << *p;
}

int main() {
        const char cad[11] = {'H','o', 'l', 'a', ' ', 'A', 'd', 'i', 'o', 's', \0'};

        Escribir(cad + 6);  // Los *char apuntan a la dirección de memoria del primer elemento
}
```

Memoria Dinámica

Para trabajar con memoria dinámica podemos definir operaciones inline para reservarla o liberarla. Veamos como funciona:

```
int main() {
        int a = 5;
        int *p1 = &a;
        int new *p2 = new int; // Reserva inline de un puntero p2
        delete p2; // Liberación de la memoria ocupada por p2
}
```

Sea la siguiente estructura, vamos a definirla:




```
struct Celda {
        int d;
        Celda *ptr;
};
```

```
int main() {
        Celda *p;

        p = new Celda;
        p->d = 3;
        p->ptr = new Celda;
        p->ptr->d = 2;
        p->ptr->ptr = new Celda;
        p->ptr->ptr->p = 1;
        p->ptr->ptr->ptr = 0; // Cuando un puntero apunta a NULL ó 0 es lo mismo
        delete p->ptr->ptr;
        delete p->ptr;
        delete p;
}
```

Otra forma, es reservar inline la memoria para vectores es la siguiente:

```
int main() {
        int *p = new int[3]; // Reserva inline el vector p de 4 posiciones
        delete [] p; // Liberación de la memoria ocupada por el vector p
}
```

Veamos un ejemplo donde vamos a trabajar con vectores usando reserva de memoria dinámica:

```
// Contenido de estadistica.h
#ifndef _ESTA_H_
#define _ESTA_H_
        double Media(const int *v, int n);
#endif

// Contenido de estadistica.cpp
#include "estadistica.h"

double Media(const int *v, int n) {
        double suma = 0.0;

        for (int i = 0; i < n; i++) {
                suma += v[i]
        } //endfor

        return suma/n;
}

//Contenido de principal.cpp
#include <iostream>
#include "estadistica.h"

using namespace std;

void LeerDatos(int &*v, int &n){
        cout << "Dime el número de datos: "
        cin >> n;
```

```
        for (int i = 0; i < n; i++) {
                cout << "v[" <<  i << "]: ";
                cin >> v[i];
        } //endfor
}

void unoMasGrande(int &*v, int &n){
        int *aux = new int[n+1];

        for (int i = 0; i < n; i++) {
                aux[i] = v[i];
        } //endfor
        delete [] v;
        v = aux;
        n+1;
}

int main() {
        int num;
        int *vec;

        LeerDatos(vec,num);
        cout << "La media es: " << Media(vec,num);
        delete [] vec;
}
```

Como podemos observar en el ejemplo, debemos pasar por referencia vec y num porque cuando ejecutamos LeerDatos(vec,num) creamos un vector temporal que almacenará los datos, pero cuando finalizamos la ejecución, ese vector creado será marcado como una posición de memoria no usada pero no liberada, es decir, sigue siendo usada por el programa pero no se vuelve a acceder a la memoria. A esa memoria se conoce como basura y la información que obtendríamos sería un vector vacío.

En el siguiente ejemplo, vamos a definir una estructura de datos de tipo lista por celdas enlazadas:

```
// Contenido de lista.h
#ifndef _LISTA_H_
#define _LISTA_H_
        struct Celda {
                int d;
                Celda * ptr;
        };

        void Borrar(Celda *&L);
        void Insertar(Celda *&L, int valor, int pos);
        void Lista_Init(Celda *&L);
        void LeerDatos(Celda *&L);
        int Media(Celda *L);
        int Size(Celda *L);
#endif

// Contenido de lista.cpp
#include "lista.h"

void Borrar(Celda *L){
```

```
        while (L != 0) {
        Celda *p = L;
        L = L->ptr;
        delete p;
        } //endwhile
}

int Get(const Celda * L, int pos) {
        for (int i = 0, const * Celda p = L; i < pos; i++, p = p->ptr);
        return p->d;
}

void Insertar(Celda *&L, int valor, int pos){
        Celda * aux = new Celda;

        aux->d = valor;
        if (pos == 0){
                aux->ptr = L;
                L = aux;
        } //endif
        else {
                Celda *p = L;

                for (int i = 0; i < pos-1; i++) {
                        p = p->ptr;
                } //endfor
                aux->ptr = p->ptr;
                p->ptr = aux;
        } //endelse
}

void Lista_Init(Celda *&L){
        L = 0;
}

void LeerDatos(Celda *&L){
        int n, v;
        Celda * aux = new Celda;
        Celda * p;

        cout << "Dime el número de datos a introducir: ";
        cin >> n;
        for (int i = 0; i < n; i++) {
                cout << "Dime dato " << i << " escrito: ";
                cin >> v;
        } //endfor
        aux->d = v;
        aux->ptr = 0;
        if (L == 0) {
                L = aux;
                p = aux;
        }//endif
        else {
```

```
                p->ptr = aux;
                p = p->ptr;
        } //endelse
}

//Podemos sustituir el código de (Celda *&L) por el siguiente código. Menos eficiente en cada inserción porque recorremos el vector hasta la nueva posición en lugar de una única vez.

void LeerDatos(Celda *&L){
        int n, v;
        Celda * aux = new Celda;
        Celda * p;

        cout << "Dime el número de datos a introducir: ";
        cin >> n;
        for (int i = 0; i < n; i++) {
                cout << "Dime dato " << i << " escrito: ";
                cin >> v;
                Insertar(L,v,i);
        } //endfor
}

// Celda es una variable temporal que cuando finaliza su recorrido por la lista vuelve a apuntar al principio de la misma. const nos indica que los campos contenidos por Celda son constantes pero Celda es una estructura modificable.

int Media(const Celda *L){
        double suma = 0.0;
        int length = Size(L);

        for (int i = 0; i < length; i++){
                suma += Get(L,i);
        } //endfor
        return suma/lenght;
}

int Size(const Celda *L) {
        int num = 0;
        const Celda * p = L;

        while (p != 0) {
                num++;
                p = p->ptr;
        } //endwhile
        return num;
}

//Contenido de principal.cpp
#include <iostream>
#include "lista.h"

using namespace std;
```

```
int main(){
        Celda *L;

        LeerDatos(L);
        cout << "La media es: " << Media(L);
        Borrar(L);
}
```

Representación de Matrices en Memoria Dinámica
Una matriz puede ser representada de varias formas:

- Un único vector de tamaño el número de elementos (filas x columnas).

3	2	1	0
1	1	2	3
7	5	7	1
5	4	3	2

Equivale a:

3	2	1	0	1	1	2	3	7	5	7	1	5	4	3	2
0			c-1	c			2c-1	2c			3c-1	3c-1			4c

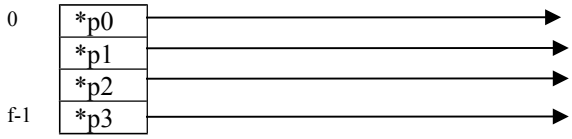
Veamos un ejemplo donde trabajaremos con una matriz siguiendo este tipo de formato:

```
#include <iostream>

using namespace std;

int main(){
        int f, c;

        cout << "Dime las dimensiones: ";
        cin >> f >> c;
        int matriz = new int [f*c];
        for (int i = 0; i < f; i++) {
                for (int j = 0; i < c; j++) {
                        cout << "Introduzca elemento m[" << i << ", " << j << "]: ";
                        cin >> matriz[i*c+j];
                } //endfor
        } //endfor
        delete [] matriz;
}
```

- Un vector de punteros a vectores. Consiste en un vector de punteros donde cada puntero representa una de las filas de la matriz apuntando a un vector que completa los datos de esa fila. Sigamos con el mismo ejemplo para ver su funcionamiento:

3	2	1	0
1	1	2	3
7	5	7	1
5	4	3	2

Equivale a:



3	2	1	0
1	1	2	3
7	5	7	1
5	4	3	2

0 c-1

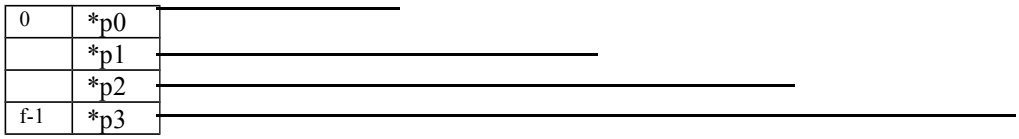
Veamos un ejemplo donde trabajaremos con una matriz siguiendo este tipo de formato:

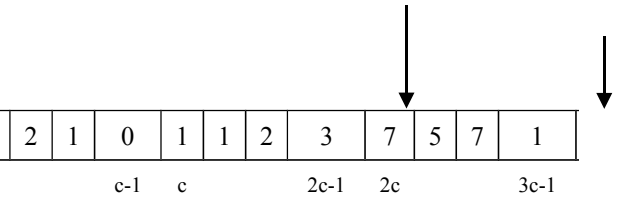
```
#include <iostream>

using namespace std;

int main(){
	int f, c;

	cout << "Dime las dimensiones: ";
	cin >> f >> c;
	if (f>0 && c>0) {
		int **matriz;

		matriz = new int [f];
		for (int i = 0; i < f; i++) {
			matriz[i] = new int[c];
		} //endfor
		for (int i = 0; i < f; i++) {
			for (int j = 0; i < c; j++) {
				cout << "Introduzca elemento m[" << i << ", " << j << "]: ";
				cin >> matriz[i][j];
			} //endfor
		} //endfor
	} //endif
	for (int i = 0; i < f; i++) {
		delete [] matriz[i];
	} //endfor
	delete [] matriz;
}
```

- Un vector de punteros a un único vector. Consiste en un vector de punteros donde cada puntero representa la posición de inicio de cada fila de la matriz. Sigamos con el mismo ejemplo para ver su funcionamiento:

3	2	1	0
1	1	2	3
7	5	7	1
5	4	3	2

Equivale a:


0
*p0
*p1
*p2
f-1
*p3


2
1
0
1
1
2
3
7
5
7
1
c-1
c
2c-1
2c
3c-1

Veamos un ejemplo donde trabajaremos con una matriz siguiendo este tipo de formato:

```
#include <iostream>

using namespace std;

int main(){
	int f, c;

	cout << "Dime las dimensiones: ";
	cin >> f >> c;
	if (f>0 && c>0) {
		int **matriz;

		matriz = new int *vector;
		matriz[vector] = new int[f*c];
		for (int i = 0; i < f; i++) {
			matriz[i] = m[i-1] + c;
			matriz[i] = m[0] + c*i;	// Otra alternativa
			matriz[i] =&( m[0] [i*c]);  // Otra alternativa
		} //endfor
		for (int i = 0; i < f; i++) {
			for (int j = 0; i < c; j++) {
				cout << "Introduzca elemento m[" << i << ", " << j <<
"]: ";
				cin >> matriz[i][j];
			} //endfor
		} //endfor
	delete [] matriz[0];
	delete [] matriz;
}
```

En el siguiente ejemplo, vamos a definir una matriz y vamos a trabajar con ella:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

struct Matriz {
	int **matriz;
	int f,c;
};

void Matriz_Init(Matriz &M);
void Matriz_Borrar (Matriz &M);
void Matriz_Escribir (Matriz &M);
void Matriz_Leer (Matriz &M);
void Matriz_Reserva (Matriz &M, int f, int c);
int Matriz_Get (Matriz &M, int f, int c);
inline void Matriz_Set (Matriz &M, int f, int c, int valor);
inline void Matriz_Fila (Matriz &M);
inline void Matriz_Columna (Matriz &M);
```

```
inline void Matriz_Set (Matriz &M, int f, int c, int valor) {
        M.matriz[f][c] = valor;
}

inline void Matriz_Fila (const Matriz &M){

        return M.f;

}

inline void Matriz_Columna (const Matriz &M){

        return M.c;

}

#endif
```

Como podemos ver en el código de la librería, las funciones inline están definidas también dentro de la librería. Para elegir una función como inline, debemos comprobar que su cuerpo sea sencillo.

Continuemos con el ejemplo y veamos como es el código fuente del archivo Matriz.cpp:

```
// Matriz.cpp

void Matriz_Init(Matriz &M){
        M.matriz = 0;
        M.f = M.c = 0;
}

void Matriz_Borrar (Matriz &M) {
        for (int i = 0; i < M.f; i++) {
                delete [] matriz[i];
        } //endfor
        delete [] matriz;
}

void Matriz_Escribir (const Matriz &M) {
        for (int i = 0; i < M.f; i++) {
                for (int j = 0; j < M.c; j++) {
                        cout << Matriz_Get(M, i, j);
                } //endfor
        } //endfor
}

void Matriz_Leer (Matriz &M) {
        int f, c, valor;

        Matriz_Init(M);
        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                Matriz_Reserva(M,f,c);
                for (int i = 0; i < f; i++) {
```

```
                        for (int j = 0; j < c; j++) {
                                cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                cin >> valor;
                                Matriz_Set(M,i,j,valor);
                        } //endfor
                } //endfor
        } //endif
}

void Matriz_Reserva (Matriz &M, int f, int c) {
        M.f = f;
        M.c = c;
        M.matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                M.matriz[i] = new int [c];
        } //endfor
}

// En esta función, si no documentamos los parámetros de entrada, tendremos que depurar la
entrada. El dato al que deseamos acceder  debe estar dentro de los limites de la matriz.

int Matriz_Get (const Matriz &M, int f, int c) {
        if (f > 0 && f < M.f && c > 0 && c < M.c)

                return M.matriz[f][c];

}
```

Veamos el uso de esta estructura de forma práctica en una aplicación que calcule la matriz transpuesta de una dada:

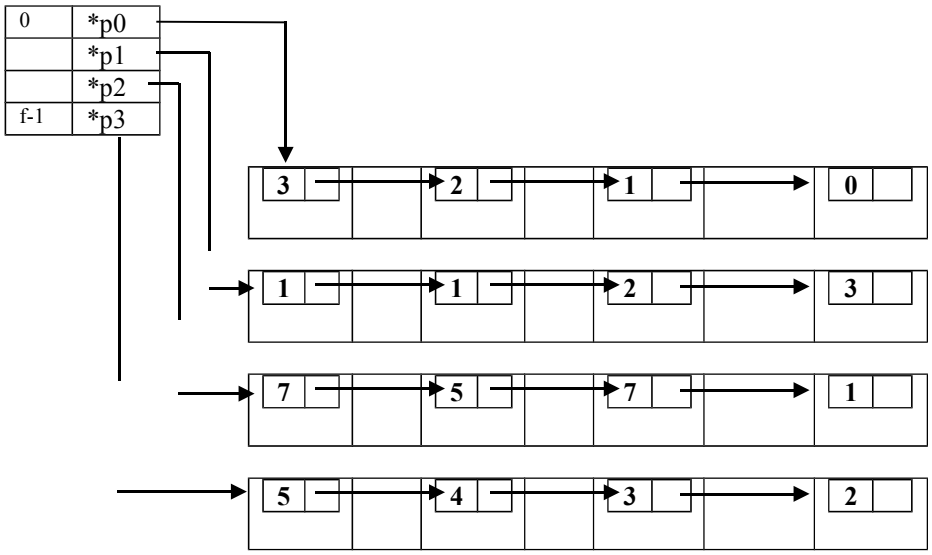
```
// Transpuesta.cpp

#include <iostream>
#include "Matriz.h"

using namespace std;

int main () {
        Matriz Entrada, Salida;
        int f, c;

        Matriz_Init(Entrada);
        Matriz_Init(Salida);
        Matriz_Leer(Entrada);
        f = Matriz_Fila(Entrada);
        c = Matriz_Columna (Entrada);
        Matriz_Reserva(Salida,c,f);
        for (int i = 0; i < f; i++)  {
                for (int j = 0; j < c; j++) {
                        int valor;
                        valor = Matriz_Get(Entrada, i, j);
                        Matriz_Set(Salida,j,i,valor)
```

```
                } //endfor
        } //endfor
        cout << "Matriz Origen:" << endln;
        Matriz_Escribir(Entrada);
        cout << "Matriz Transpuesta:" << endln;
        Matriz_Escribir(Salida);
        Matriz_Borrar(Entrada);
        Matriz_Borrar(Salida);
}
```

- Un vector de punteros celdas. Consiste en un vector de punteros donde cada puntero representa la posición de inicio de cada fila de la matriz. Pero los elementos de la matriz están alineados por una lista de celdas que nos permite recorrer la matriz por completo Sigamos con el mismo ejemplo para ver su funcionamiento:

3	2	1	0
1	1	2	3
7	5	7	1
5	4	3	2

Equivale a:



En el siguiente ejemplo, vamos a definir una matriz y vamos a trabajar con ella:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

struct Celda {
        int d;
        Celda *next;
};

struct Matriz {
        Celda *matriz;
        int f,c;
```

```
};

void Matriz_Init(Matriz &M);
void Matriz_Borrar (Matriz &M);
void Matriz_Escribir (Matriz &M);
void Matriz_Leer (Matriz &M);
void Matriz_Reserva (Matriz &M, int f, int c);
int Matriz_Get (Matriz &M, int f, int c);
void Matriz_Set (Matriz &M, int f, int c, int valor);
inline void Matriz_Fila (Matriz &M);
inline void Matriz_Columna (Matriz &M);

inline void Matriz_Fila (const Matriz &M){

        return M.f;

}

inline void Matriz_Columna (const Matriz &M){

        return M.c;

}

#endif
```

Continuemos con el ejemplo y veamos como es el código fuente del archivo Matriz.cpp:

```
// Matriz.cpp

void Matriz_Init(Matriz &M){
        M.matriz = 0;
        M.f = M.c = 0;
}

void Matriz_Borrar (Matriz &M) {
        while(M,atriz != 0) {
                Celda *p = M.matriz;
                M.matriz = M.matriz->next;
                delete p;
        } //endwhile
}

void Matriz_Escribir (const Matriz &M) {
        for (int i = 0; i < M.f; i++) {
                for (int j = 0; j < M.c; j++) {
                        cout << Matriz_Get(M, i, j);
                } //endfor
        } //endfor
}

void Matriz_Leer (Matriz &M) {
        int f, c;
```

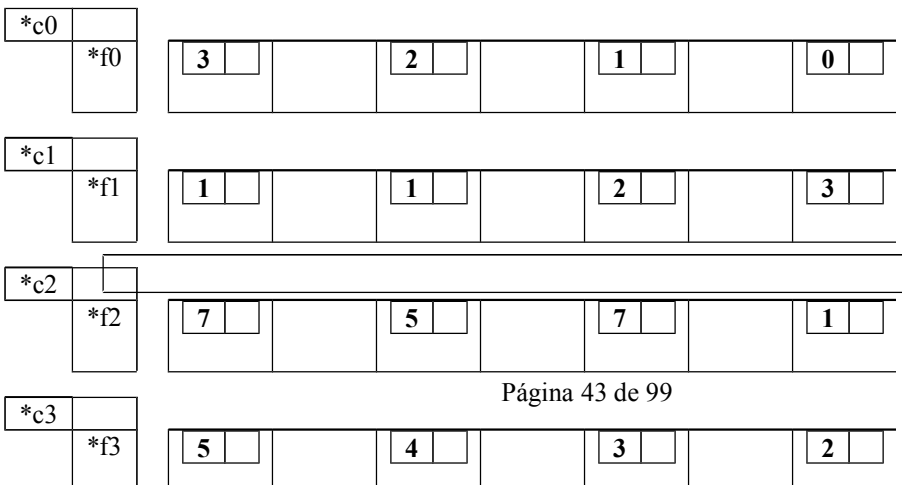
```
        Matriz_Init(M);
        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                Matriz_Reserva(M,f,c);
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                cin >> valor;
                                Matriz_Set(M,i,j,valor);
                        } //endfor
                } //endfor
        } //endif
}

//En esta función no respetamos el orden de entrada, los elementos son insertados y
almacenados en orden inverso a la entrada.

void Matriz_Reserva (Matriz &M, int f, int c) {
        Celda *p;

        M.f = f;
        M.c = c;
        M.matriz = 0;
        for (int i = 0; i < f*c; i++) {
                Celda *aux = new Celda;

                aux->next = &M.matriz;
                M.matriz = aux;
        } //endfor
}

//En esta función respetamos el orden de entrada, los elementos son insertados y almacenados
en orden a la entrada. Para ello, necesitamos un puntero que apunte a la última posición del
elemento insertado.

void Matriz_Reserva (Matriz &M, int f, int c) {
        Celda *p;

        M.f = f;
        M.c = c;
        M.matriz = 0;
        for (int i=0; i<f; i++) {
                for (int j=0; j<c; j++) {
                        Celda *aux = new Celda;

                        aux->next = 0;
                        if (M.matriz == 0) {
                                M.matriz = aux;
                                p = aux;
                        } //endif
```

```
                        else {
                                p->next = aux;
                                p = p->next;
                        } //endelse
                } //endfor
        } //endfor
}

// En esta función, si no documentamos los parámetros de entrada, tendremos que depurar la entrada. El dato al que deseamos acceder  debe estar dentro de los limites de la matriz.

int Matriz_Get (const Matriz &M, int f, int c) {
        Celda *p = M.matriz;
        int desplazamiento = M.c * M.f + c;
        for (int i = 0; i < desplazamiento; i++) {
                p = p->next;
        } //endfor

        return p->d; // Otra opción es return  *p.d

}

// En esta función, si no documentamos los parámetros de entrada, tendremos que depurar la entrada. El dato al que deseamos acceder  debe estar dentro de los limites de la matriz.

int Matriz_Set (const Matriz &M, int f, int c, int valor) {
        Celda *p = M.matriz;
        int desplazamiento = M.c * M.f + c;
        for (int i = 0; i < desplazamiento; i++) {
                p = p->next;
        } //endfor
        p->d = valor;
}
```

- Una lista enlazada de celdas de punteros donde cada puntero representa la posición de inicio de cada fila de la matriz. Pero los elementos de la matriz están alineados por una lista de celdas que nos permite recorrer la matriz por completo Sigamos con el mismo ejemplo para ver su funcionamiento:

3	2	1	0
1	1	2	3
7	5	7	1
5	4	3	2

Equivale a:



En el siguiente ejemplo, vamos a definir una matriz y vamos a trabajar con ella:

```
                                                                // Matriz.h
```

```
#ifndef _MATRIZ_H_
#define _MATRIZ_H_

struct Celda1 {
        int d;
        Celda1 *next;
};

struct Celda2 {
        Celda1 *first;
        Celda2 *next;
};

struct Matriz {
        Celda2 *matriz;
        int f,c;
};

void Matriz_Init(Matriz &M);
void Matriz_Borrar (Matriz &M);
void Matriz_Escribir (Matriz &M);
void Matriz_Leer (Matriz &M);
void Matriz_Reserva (Matriz &M, int f, int c);
int Matriz_Get (Matriz &M, int f, int c);
void Matriz_Set (Matriz &M, int f, int c, int valor);
inline void Matriz_Fila (Matriz &M);
inline void Matriz_Columna (Matriz &M);

inline void Matriz_Fila (const Matriz &M){

        return M.f;

}

inline void Matriz_Columna (const Matriz &M){

        return M.c;

}

#endif
```

Continuemos con el ejemplo y veamos como es el código fuente del archivo Matriz.cpp:

```
// Matriz.cpp

void Matriz_Init(Matriz &M){
        M.matriz = 0;
        M.f = M.c = 0;
}

void Matriz_Borrar(Matriz &M) {
         while (M.matriz!=0) {
                Celda2 * p = M.matriz;

                //ahora destruimos la fila
                while (p->first!=0){
                        Celda1 *q = p->first;
```

```
                        p->first= p->first->next;
                        delete q;
                } //endwhile
                M.matriz = M.matriz->next;
                delete p;
        } //endwhile
}

void Matriz_Escribir (const Matriz &M) {
        if (f > 0 && c > 0) {
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                cout << Matriz_Get(M,i,j,valor);
                        } //endfor
                } //endfor
        } //endif
}

void Matriz_Leer (Matriz &M) {
        int f, c;

        Matriz_Init(M);
        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                Matriz_Reserva(M,f,c);
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                cin >> valor;
                                Matriz_Set(M,i,j,valor);
                        } //endfor
                } //endfor
        } //endif
}

//En esta función no respetamos el orden de entrada, los elementos son insertados y
almacenados en orden inverso a la entrada.

void Matriz_Reserva (Matriz &M, int f, int c) {
        Celda2 *aux = new Celda2;

        M.f = f;
        M.c = c;
        M.matriz = 0;
        for (int i = 0; i < f; i++) {
                Celda2 *aux = new Celda2;
                Celda1 *q = new Celda1;

                if (M.matriz == 0) {
                        M.matriz = aux;
                        p = aux;
                } //endif
                else {
                        p->next = aux;
                        p = p->next;
                } //endelse
                for (int j = 0; i < c; j++) {
```

```
                        Celda1 *nueva = new Celda1;

                        nueva->next = 0;
                        if (p->frst == 0) {
                                p->first = nueva;
                                q = nueva;
                        } //endelse
                        else {
                                q->next = nueva;
                                q = q->next;
                        } //endelse
                } //endfor
        } //endfor
}

// En esta función, si no documentamos los parámetros de entrada, tendremos que depurar la entrada. El dato al que deseamos acceder  debe estar dentro de los limites de la matriz.

int Matriz_Get(Matriz &M,int f,int c) {
        Celda2 *p =M.matriz;

        //nos situamos en la fila
         for (int i=0;i<f;i++) {
                p= p->next;
        }
 //endfor
        Celda1*q = p->first;

        //nos situamos en la columna
        for (int j=0;j<c;j++) {
                q = q->next;
        } //endfor

        return q->d;

}

// En esta función, si no documentamos los parámetros de entrada, tendremos que depurar la entrada. El dato al que deseamos acceder  debe estar dentro de los limites de la matriz.

void  Matriz_Set (Matriz &M, int f, int c, int valor) {
        Celda2 *p =M.matriz;

        //nos situamos en la fila
         for (int i=0;i<f;i++) {
                p= p->next;
        }
 //endfor
        Celda1*q = p->first;

        //nos situamos en la columna
        for (int j=0;j<c;j++) {
                q = q->next;
        } //endfor
        q->d = valor;
}
```

Tema 2.- Clases

Introducción. Conversión de struct a class

Hasta ahora hemos utilizado un tipo de dato abstracto, con un tipo de valores y unas funciones asociadas para trabajar con los datos abstractos defifinidos. Para ello, lo primero que hacemos es definir las operaciones dentro del struct, para ello, el "*};*" de cierre de definición del struct quedaría después de las funciones. Por tanto Matriz.h tendría un aspecto similar a este:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

struct Matriz {
        int **matriz;
        int f,c;

        void Matriz_Init(Matriz &M);
        void Matriz_Borrar (Matriz &M);
        void Matriz_Columna (Matriz &M);
        void Matriz_Escribir (Matriz &M);
        int Matriz_Get (Matriz &M, int f, int c);
        void Matriz_Fila (Matriz &M);
        void Matriz_Leer (Matriz &M);
        void Matriz_Reserva (Matriz &M, int f, int c);
        void Matriz_Set (Matriz &M, int f, int c, int valor);
};

#endif
```

Para utilizar dichas funciones por tanto deberíamos escribir sentencias del tipo:

```
Matriz M;
M.Reserva(M,5,2);
```

Si nos fijamos con atención, tenemos una rebundancia de datos, esta se produce porque pasamos un objeto Matriz (el objeto M que usa la función) y además debemos pasar el mismo objeto por memoria (El M que hay de parametro en la función). Para eliminar esa redundancia, quitamos la llamada a la matriz en tódos los cuerpos de las funciones. Además, como estan dentro del struct, las operaciones son renombradas quitando el prefijo Matriz (son operaciones internas del tipo de dato abstracto) y el atributo const. Por tanto nuestro nuevo Matriz.h tendrá el siguiente aspecto:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

struct Matriz {
        int **matriz;
        int f,c;

        void Init();
        void Borrar ();
        void Columna () const;
        void Escribir () const;
        int Get (int f, int c) const;
        void Fila () const;
        void Leer () const;
        void Reserva (int f, int c);
```

```
        void Set (int f, int c, int valor);
};

#endif
```

Y estos cambiamos tambien se realizarán dentro del Matriz.cpp, teniendo el siguiente código:

```
// Matriz.cpp

void Matriz::Init(){
        matriz = 0;
        f = c = 0;
}

void Matriz::Borrar () {
        for (int i = 0; i < M.f; i++) {
                delete [] matriz[i];
        } //endfor
        delete [] matriz;
}

int Matriz::Columna () const {

        return c;

}

void Matriz::Escribir () const {
        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        cout << Get(i, j);
                } //endfor
        } //endfor
}

int Matriz::Fila () const {

        return f;

}

int Matriz::Get (int f, int c) const {
        if (f > 0 && f < this.f && c > 0 && c < this.c)

                return matriz[f][c];

}

void Matriz::Leer () {
        int f, c, valor;

        this.Init();
        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                this.Reserva(f,c);
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
```

```
                                        cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                        cin >> valor;
                                        this.Set(i,j,valor);
                                } //endfor
                        } //endfor
                } //endif
}

void Matriz::Reserva (int f, int c) {
        this.f = f;
        this.c = c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
        } //endfor
}

int Matriz::Set (int f, int c, int valor) {
        if (f > 0 && f < this.f && c > 0 && c < this.c)

                matriz[f][c] = valor;

}
```

Observamos dos detalles: el primero de ellos es el uso del puntero this para poder diferenciar entre una variable c y un objeto de Matriz con el mismo nombre.

En segundo lugar, observamos que todos los objetos que usaban la matriz const, ahora lo indican añadiendo al final de la definición de la función la palabra const.

A continuación, debemos definir que campos son de acceso público (accesibles por cualquier función de un modulo ajeno) o privados (solo accesibles por las funciones que definen el tipo de dato abstracto). Despues de definir la accesibilidad, solo debemos sustituir la palabra struct por class. Veamos las modificaciones de Matriz.h:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

class Matriz {
        private:
                int **matriz;
                int f,c;

        public:
                void Init();
                void Borrar ();
                void Columna ();
                void Escribir ();
                int Get (int f, int c);
                void Fila ();
                void Leer ();
                void Reserva (int f, int c);
                void Set (int f, int c, int valor);
};

#endif
```

La diferencia entre un struct y class es que por defecto todos lo contenido por struct es publico mientras todo lo contenido por class es privado.

Constructores y destructores
Los constructores son funciones de reserva y liberación de resrerva automática. En el momento de declaración de la variable del tipo clase, el compilador crea un objeto con valore válidos y cuando finaliza su uso, libera los recursos de la variable.

Vamos a continuar con el ejemplo de Matriz.h, en dicho ejemplo, sustituiremos la función Init por lo que se conoce por un constructor por defecto mientras Borrar equivale al destructor, siempre lo ejecutará al finalizar el proceso main. Observaremos que no tienen parámetros a devolver en su declaración:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

class Matriz {
        private:
                int **matriz;
                int f,c;

        public:
                Matriz();
                ~Matriz();
                void Columna ();
                void Escribir ();
                int Get (int f, int c);
                void Fila ();
                void Leer ();
                void Reserva (int f, int c);
                void Set (int f, int c, int valor);
};

#endif
```

Y dentro de Matriz.cpp tendremos lo siguiente:

```
// Matriz.cpp

Matriz::Matriz (){
        matriz = 0;
        f = c = 0;
}

Matriz::~Matrz () {
        for (int i = 0; i < M.f; i++) {
                delete [] matriz[i];
        } //endfor
        delete [] matriz;
        f = 0;
        c = 0;
}

int Matriz::Columna () const {

        return c;
```

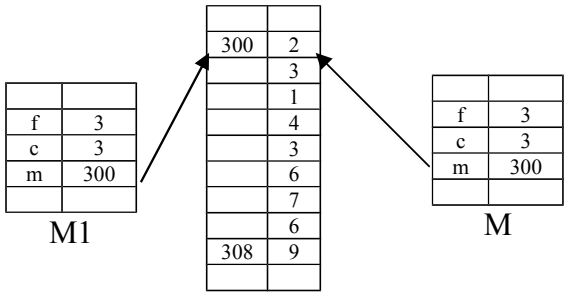
```
}

void Matriz::Escribir () const {
        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        cout << Get(i, j);
                } //endfor
        } //endfor
}

int Matriz::Fila () const {

        return f;

}

int Matriz::Get (int f, int c) const {
        if (f > 0 && f < this.f && c > 0 && c < this.c)

                return matriz[f][c];

}

void Matriz::Leer () {
        int f, c, valor;

        this.Init();
        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                this.Reserva(f,c);
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                cin >> valor;
                                this.Set(i,j,valor);
                        } //endfor
                } //endfor
        } //endif
}

void Matriz::Reserva (int f, int c) {
        this.f = f;
        this.c = c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
        } //endfor
}

int Matriz::Set (int f, int c, int valor) {
        if (f > 0 && f < this.f && c > 0 && c < this.c)

                matriz[f][c] = valor;

}
```

También tenemos lo que se conoce el constructor por parámetros y el constructor por copia. En el caso del constructor por parámetros, los usamos commop si realizasemos las operaciones Matriz y reserva al

mismo tiempo. Veamos ahora como funciona el constructor por copia. Supongamos ahora el siguiente código fuente:

```
void funcion (Matriz &M) {
        M.Columna();
}

int main () {
        Matriz M1(5,2); // Usamos el constructor por parámetros para crear este objeto Matriz
        funcion(M1);
        M1.Columna(); // No podríamos realizar esta consulta por esrror de memoria.
}
```

El mapa de memoria durante la ejecución de la funcion "*funcion (Matriz &M)*" sería por ejemplo:



Tenemos que una matriz definida se copia por parámetros pero cuando finaliza la ejecución de la funcion "*funcion (Matriz &M)*" la matriz a la que esta apuntando queda marcada para ser eliminada por el destructor. Por tanto nos encontramos que un objeto que inicialmente estaba siendo usado en memoria ya no lo esta cuando finaliza la ekjjecucion de la función y tenemos un error de memoria. Por eso es necesario usar un constructor por copia. Veamos como quedaria modificado Matriz.h:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

class Matriz {
        private:
                int **matriz;
                int f,c;

        public:
                Matriz ();
                Matriz (int f, int c);
                Matriz (const Matriz &M);
                ~Matriz ();
                void Escribir ();
                void Leer ();
                void Reserva (int f, int c);
                void Columna ();
                void Fila ();
                int Get (int f, int c);
                void Set (int f, int c, int valor);
};

#endif
```

Y dentro de Matriz.cpp tendremos lo siguiente:

```
// Matriz.cpp

Matriz::Matriz () {
        matriz = 0;
        f = c = 0;
}

Matriz::Matriz (int f, int c) {
        this.f = f;
        this.c = c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
        } //endfor
}

Matriz::Matriz (const Matriz &M) {
        f = M.f;
        c = M.c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
                for (int j = 0, j < c; j++) {
                        matriz[i][j] = M.matriz[i][j];
                } //endfor
        } //endfor
}

Matriz::~Matriz () {
        for (int i = 0; i < M.f; i++) {
                delete [] matriz[i];
        } //endfor
        delete [] matriz;
        f = 0;
        c = 0;
}

void Matriz::Escribir () const {
        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        cout << Get(i, j);
                } //endfor
        } //endfor
}

void Matriz::Leer () {
        int f, c, valor;

        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                this.Reserva(f,c);
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                cin >> valor;
                                this.Set(i,j,valor);
                        } //endfor
```

```
                } //endfor
        } //endif
}

void Matriz::Reserva (int f, int c) {
        this.f = f;
        this.c = c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
        } //endfor
}

// Las funciones inline de una clase se implmentan de esta forma
int Matriz::Columna () const { return c;}

int Matriz::Fila () const { return f;}

int Matriz::Get (int f, int c) const { return matriz[f][c];}

int Matriz::Set (int f, int c, int valor) { matriz[f][c] = valor;}
```

Operadores
Supongamos ahora el siguiente código fuente:

```
int main () {
        Matriz M2 = Matriz(5,2);
 }
```

El compilador nos permitirá realizar esta operación. Por defecto creará Matriz (5,2) y al finalizar la ejecución de creación destruirá el objeto creado. Por tanto, el operador = esta igualando a un elemento de valor NULL. Para evitar este error, es necesario sobrecargar el operador = para evitar la perdida de esos datos y almacenarlos en M2. Veamos como sería el código en Matriz.cpp:

```
//Matriz.cpp Debe tener su correspondiente cabecera definida en Matriz.h

Matriz & Matriz ::operator = (const Matriz & M) {

// Comprobamos que no asigna el mismo objeto
        if (this != M.matriz) {

        // Eliminamos el objeto sobre el que vamos a copiar
                for (int i = 0; i < f; i++) {
                        delete [] m[i];
                } //endfor
                delete [] m;

        // Iniciamos la copia de informacion sobre el objeto
                f = M.f;
                c = M.c;
                matriz = new int * [f]
                for (int i = 0; i < f; i++) {
                        matriz[i] = new int [c];
                        for (int j = 0; j < c; j++) {
                                matriz[i][j] = M.matriz[i][j];
                        } //endfor
                } //endfor
        } //endif
```

```
        return *this;

}
```

La razon por la que devolvemos *this es la siguiente: sea por ejemplo la siguiente sentencia de código: "*M3 = M2 = M1*" el compilador la traducirá como M3.operator=(M2.operator=(M1)) si alguna de las operaciones devuelve un valor NULL, no tendremos mnada que asignar a M3.

Con la devolución *this el compilador manda al constructor por copia cada matriz donde crea un objeto temporal y lo devuelve. Por tanto, por eficiencia, devolvemos Matriz & en lugar de Matriz. Veamos en el siguiente ejemplo mejor los que esta ocurriendo:

```
Matriz Mglobal;

Matriz funcion1 (Matriz Mpar) {

        return Mpar;

}

Matriz funcion2 (const Matriz &Mt) {

        return Mt;

}

int main() {

        // Constructor por defecto
        Matriz M1;

        // No llama a ningun constructor, es una dirección de memoria
        Matriz *Mptr;
        Matriz &Mref = M1;
        Mptr = &Mref;

        // Lo siguiente es un bloque, todo lo que se cree entre estas llaves existira hay
        {

        /*
        1.- Constructor por defecto para Mbloque
        2.- Constructor por parámetros para temporal Matriz (7,5)
        3.- Asignamos con noperador = temporal a Mbloque
        4.- Destruimos temporal
        */
        Matriz Mbloque = Matriz(7,5);
        }

        /*
        1.- Constructor por defecto para M1
        2.- funcion1 crea un constructor por copia de M1, temporal.
        3.- Como es retorno por valor, se usa el constructor por copia de la matriz temporal
        4.- Destruimos Mpar
        5.- Aplicamos el operador = copiando en M1 el contenido de temporal.
        6.- Destruimos temporal
        */
        Matriz M2 = funcion1 (M1);

        /*
```

```
        Solo aplicamos el operador = copiando en M1 el contenido de temporal.
        */
        Matriz M3 = funcion2 (M1);

} //Destruimos M1, M2, M3 y Mglobal
```

Por tanto, una clase minima debe contener los siguientes elementos:

- Constructor por defecto
- Constructor por parametros
- Constructor por copia y no lo usaremos si el tipo de dato abstracto no usa memoria dinámica
- Destructor
- Operador de asignación

Veamos como quedaria Matriz.h cumpliendo estos requisitos:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

class Matriz {
        private:
                int **matriz;
                int f,c;

        public:
                Matriz ();
                Matriz (int f, int c);
                Matriz (const Matriz &M);
                ~Matriz ();
                void Escribir ();
                void Leer ();
                void Reserva (int f, int c);
                void Columna ();
                void Fila ();
                int Get (int f, int c);
                void Set (int f, int c, int valor);
                Matriz & operator = (const Matriz & M);
                Matriz & operator + (const Matriz & M);
                bool operator == (const Matriz & M);
                bool operator != (const Matriz & M);
};

#endif
```

Y dentro de Matriz.cpp tendremos lo siguiente:

```
// Matriz.cpp

Matriz::Matriz () {
        matriz = 0;
        f = c = 0;
}

Matriz::Matriz (int f, int c) {
        this.f = f;
        this.c = c;
```

```
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
        } //endfor
}

Matriz::Matriz (const Matriz &M) {
        f = M.f;
        c = M.c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
                for (int j = 0, j < c; j++) {
                        matriz[i][j] = M.matriz[i][j];
                } //endfor
        } //endfor
}

Matriz::~Matriz () {
        for (int i = 0; i < M.f; i++) {
                delete [] matriz[i];
        } //endfor
        delete [] matriz;
        f = 0;
        c = 0;
}

void Matriz::Escribir () const {
        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        cout << Get(i, j);
                } //endfor
        } //endfor
}

void Matriz::Leer () {
        int f, c, valor;

        cout << "Dime la dimensión de la matriz";
        cin >> f >> c;
        if (f > 0 && c > 0) {
                this.Reserva(f,c);
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                cout << "Dime el valor de la posición m[" << i << ","<< j
<< "]:";
                                cin >> valor;
                                this.Set(i,j,valor);
                        } //endfor
                } //endfor
        } //endif
}

void Matriz::Reserva (int f, int c) {
        this.f = f;
        this.c = c;
        matriz = new int *[f];
        for (int i = 0; i < f; i++) {
                matriz[i] = new int [c];
        } //endfor
```

```
}

// Las funciones inline de una clase se implmentan de esta forma
int Matriz::Columna () const { return c;}

int Matriz::Fila () const { return f;}

int Matriz::Get (int f, int c) const { return matriz[f][c];}

int Matriz::Set (int f, int c, int valor) { matriz[f][c] = valor;}

Matriz & Matriz ::operator = (const Matriz & M) {

// Comprobamos que no asigna el mismo objeto
        if (this != M.matriz) {

        // Eliminamos el objeto sobre el que vamos a copiar
                for (int i = 0; i < f; i++) {
                        delete [] m[i];
                } //endfor
                delete [] m;

        // Iniciamos la copia de informacion sobre el objeto
                 f = M.f;
                 c = M.c;
                 matriz = new int * [f]
                 for (int i = 0; i < f; i++) {
                        matriz[i] = new int [c];
                        for (int j = 0; j < c; j++) {
                                matriz[i][j] = M.matriz[i][j];
                        } //endfor
                 } //endfor
        } //endif

        return *this;

}

Matriz & Matriz ::operator + (const Matriz & M) {
        Matriz Mlocal(*this); // o bien escribimos: Mlocal(M);

        if (f==M.f && c == M.c) {
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                Mlocal.matriz[i][j] = M.matriz[i][j];
                        } //endfor
                 } //endfor
        } //endif

        return Mlocal;

}

bool Matriz ::operator == (const Matriz & M) {

// Comprobamos las dimensiones
        if (f  ==  M.f && c == M.c) {
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                if (matriz[i][j] = M.matriz[i][j]) {
```

```
                                        return false;

                                    } //endif
                                } //endfor
                        } //endfor

                        return true;

                } //endif
                else {

                        return false;

                } //endelse
}

// Podemos implementar este operador de dos formas. La primera es asi:
bool Matriz ::operator != (const Matriz & M) {

// Comprobamos las dimensiones
                if (f  ==  M.f && c == M.c) {
                        for (int i = 0; i < f; i++) {
                                for (int j = 0; j < c; j++) {
                                        if (matriz[i][j] = M.matriz[i][j]) {

                                                return true;

                                        } //endif
                                } //endfor
                        } //endfor

                        return false;

                } //endif
                else {

                        return true;

                } //endelse
}

// O aprovechamos la implementación de operator == y lo usamos  de la siguiente forma:

bool Matriz ::operator != (const Matriz & M) {

                return !(*this == M)

}
```

Si observamos este ejemplo, operator + funciona de la siguiente forma:

1. Se llama un constructor por copia que crea una matriz temporal.
2. Asignamos esa matriz temporal a nuestra variable Mlocal.
3. Operamos y devolvemos Mlocal.
4. Destruimos la matriz temporal creada por el constructor por copia.

Además no podriamos realizar operaciones como las del siguiente ejemplo:

```
int main(){
        Matriz M1, M2;

        M2 = M1 + M2; // Se ejecutaria sin problemas
        M2 = M2 + M1; //No podríamos ejecutarlo.
}
```

Funciones friend (funciones amigas)
Para aumentar la versatilidad de una clase, usamos las funciones amigas. Estas ffunciones nos permiten acceder a la parte privada de una clase. Cuando las usamos, debemos recordar que no es posible usar *this, ya que nos encontramos trabajando fuera de la clase. Veamos un ejemplo de uso:

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

class Matriz {
        private:
                int **matriz;
                int f,c;

        public:
                Matriz ();
                Matriz (int f, int c);
                Matriz (const Matriz &M);
                ~Matriz ();
                void Escribir ();
                void Leer ();
                void Reserva (int f, int c);
                void Columna ();
                void Fila ();
                int Get (int f, int c);
                void Set (int f, int c, int valor);
                Matriz & operator = (const Matriz & M);
                Matriz & operator + (const Matriz & M);
                bool operator == (const Matriz & M);
                bool operator != (const Matriz & M);
                friend Matriz & Transpuesta (const Matriz & M);
};

#endif
```

Y dentro de Matriz.cpp tendremos lo siguiente:

```
Matriz Transpuesta (const Matriz &M) {
        Matriz Mlocal(M.f, M.c);

        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        Mlocal.nmatriz[j][i] = M.matriz[i][j];
                } //endfor
        } //endfor

        return Mlocal;

}
```

A continuación vamos a ver que ocurre si creamos un operator friend:

```
Matriz  operator + (const Matriz & M1, const Matriz M2) {
        Matriz Mlocal(M1);

        if (M1.f==M2.f && M1.c == M2.c) {
                for (int i = 0; i < f; i++) {
                        for (int j = 0; j < c; j++) {
                                Mlocal.matriz[i][j] += M2.matriz[i][j];
                        } //endfor
                } //endfor
        } //endif

        return Mlocal;

}
```

Si observamos este ejemplo, operator + funciona de la siguiente forma:

1. Se llama un constructor por copia que crea una matriz temporal, M1.
2. Se llama un constructor por copia que crea una matriz temporal, M2.
3. Asignamos esa matriz temporal a nuestra variable Mlocal.
4. Operamos y devolvemos Mlocal.
5. Destruimos las matrices temporales creadas por el constructor por copia.

Y ahora podriamos realizar operaciones como las del siguiente ejemplo:

```
int main(){
        Matriz M1, M2;

        M2 = M1 + M2; // Se ejecutaria sin problemas
        M2 = M2 + M1; // Se ejecutaria sin problemas
}
```

Veamos un nuevo ejemplo de uso de friend con operator. Sea la siguiente definición de operator siuma de un entero a una matriz:

```
Matriz & Matriz::operator + (int b) {
        Matriz Mlocal(*this);

        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        Mlocal.matriz[i][j] += b;
                } //endfor
        } //endfor

        return Mlocal; // Al finalizar ejecucion, Mlocal es copiado y destruido

}
```

Podemos definir la siguiente friend operator:

```
// Podemos implementar esto de varias formas, veamos la primera de ellas
Matriz  operator + (int b, Matriz & M) {
        Matriz Mlocal(M);

        for (int i = 0; i < f; i++) {
                for (int j = 0; j < c; j++) {
                        Mlocal.matriz[i][j] += b;
                } //endfor
```

```
            } //endfor

            return Mlocal; // Al finalizar ejecucion, Mlocal es copiado y destruido

}

//Otra solución seria:
Matriz  operator + (int b, Matriz & M) {
            Matriz Mlocal(M);

            Mlocal = Mlocal+b;

            return Mlocal;

}

// Finalmente, podemos también escribirlo de la siguiente forma:
Matriz  operator + (int b, Matriz & M) { return (M+b); }
```

Veamos en último lugar algunos operadores de consulta:

```
// Este operator permite realizar este tipo de operaciones:
//cout <<  matriz(3,3);

int  operator () (int f, int c) { return m[i][j]; }

// Este operator permite realizar este tipo de operaciones:
// matriz(3,3) = 2;

int &  operator () (int f, int c) { return m[i][j]; }

// Este operator permite realizar este tipo de operaciones:
// cout << matriz[3][3];

const int * operator [] (int pos) { return m[i]; }

// Este operator permite realizar este tipo de operaciones:
//  matriz[3][3] = 2;

int * operator [] (int pos) { return m[i]; }

// Este operator permite realizar este tipo de operaciones:
// matriz[3] = 2;

int * const operator [] (int pos) { return m[i]; }
```

Las funciones de entrada y salida de una clase consisten en sobrecargar los operadores de entrada y salidas de cadenas >> y <<. Para ello, ademas de definir las operaciones debemos incluir la libreria stdio.h y el namespaces std para asegurarnos que dicha sobrecarga tiene efecto.

Las funciones de entrada y salida suelen ser de tipo friend, puesto que no tiene que acceder ni modificar partes privadas de la clase.

```
// Matriz.h

#ifndef _MATRIZ_H_
#define _MATRIZ_H_

#include <iostream>
```

```
using namespace std;

class Matriz {
	private:
		int **matriz;
		int f,c;

	public:
		Matriz ();
		Matriz (int f, int c);
		Matriz (const Matriz &M);
		~Matriz ();
		void Escribir ();
		void Leer ();
		void Reserva (int f, int c);
		void Columna ();
		void Fila ();
		int Get (int f, int c);
		void Set (int f, int c, int valor);
		Matriz & operator = (const Matriz & M);
		Matriz & operator + (const Matriz & M);
		bool operator == (const Matriz & M);
		bool operator != (const Matriz & M);
		friend ostream & operator << (ostream &os, Matriz &M);
		friend istream & operator >> (istream &is, Matriz &M);
};

#endif
```

Y dentro de Matriz.cpp tendremos lo siguiente:

```
ostream & operator << (ostream &is, Matriz &M)  {
	for (int i = 0; i < M.f; i++) {
		for (int j = 0; j < M.c; j++) {
			os << M.matriz[i][j];
		} //endfor
	} //endfor
	os << endl;

	return os;

}

istream & operator >> (istream &is, Matriz &M)  {
	int nf, nc;

	is >> nf << nc;
	Matriz Mlocal(nf, nc);
	for (int i = 0; i < M.f; i++) {
		for (int j = 0; j < M.c; j++) {
			is >> Mlocal.matriz[i][j];
		} //endfor
	} //endfor

	return is;

}
```

La clase vector dinámico

La clase vector dinámico consiste en un vector que dispone de un espacio reservado de memoria para almacenar datos. Veamos como sería la definición de este tipo de dato en vd.h

```
//VD.h

#ifndef _VD _H_
#define _VD _H_

class VD {
        private:
                int nReservado; // Espacio reservado de memoria
                int nElementos; // Numero de elementos, donde nElementos < nReservado
                int * vec;
                void Ampliar () {
                        nReservado = 2*nReservado;
                        int aux = new int [2*nReservado];

                        for (int i = 0; i < nElementos; i++) {
                                aux[i] = vec[i];
                        } //endfor
                        delete [] vec;
                        vec = aux;
                }

        public:
                VD ();
                VD (int tam);
                VD (const VD & v);
                ~VD ();
                VD & operator = (const VD & v);
                void  Add (int a);
                void Borrar(int pos);
                int Size () { return nElementos; }
                bool operator == (const VD & v);
                bool operator != (const VD & v);
                VD operator + (const VD & v);
                VD operator + (int a);
                friend VD operator + (int a, const VD & v);
                void operator += (const VD & v);
                int & operator [] (int i) { return vec[i]; }
};

/*
Si declaramos fuera las funciones de lectura y escritura, tenemos que usar los metodos publicos
de la clase VD. En el momento que necesitemos tocar alguna parte privada de la clase, debemos
incluirlas en la clase como un metodo friend.
*/

ostream & operator << (ostream & os, const VD & v);
istream & operator >> (istream & os, VD & v);
```

Y dentro de VD.cpp tendremos lo siguiente:

```
//VD.cpp

#include "VD.h"

#define MAX 10;  // Espacio mínimo de nReservado
```

```
VD () {
        nReservado = MAX;
        nElementos = 0;
        vec = new int [nReservado];
}

VD (int tam) {
        nReservado = tam;
        nElementos = 0;
        vec = new int [nReservado];
}

VD (const VD & v) {
        nReservado = v.nReservado;
        nElementos = v.nElementos;
        vec = new int [nReservado];
        for (int i = 0; i < nElementos; i ++) {
                vec[i] = v.vec[i];
        } //endfor
}

~VD () {
        if (vec != 0) {
                delete [] vec;
        } //endif
}

VD & VD::operator = (const VD & v) {
        if (*this != v) {
                delete [] vec;
                nReservado = v.nReservado;
                nElementos = v.nElementos;
                vec = new int [nReservado];
                for (int i = 0; i < nElementos; i ++) {
                        vec[i] = v.vec[i];
                } //endfor
        } //endfor

        return *this;

}

void  VD::Add (int a) {
        if (nElementos  >= nReservado) {
                Ampliar();
        } //endif
        vec[nelementos] = a;
        nElementos++;
}

void VD::Borrar (int pos) {
        for (int i = 0; i < nElementos; i ++) {
                vec[i-1] = vec[i];
        } //endfor
        nElementos --;
}

bool VD::operator == (const VD & v) {
        if (nElementos == v.nElementos) {
```

```
                    for (int i = 0; i < nElementos; i++) {
                            if (vec[i] != v.vec[i]) {
                                    return false;
                            } //endif
                    } //endfor

                    return true;

            } //endif
            else {

                    return false;

            } //endelse
}
bool VD::operator != (const VD & v) {

                    return !(this == v);

}

VD VD::operator+(const VD & v) {
            VD Vlocal(nElementos + v.nElementos);

            for (int i = 0; i < nElementos; i++) {
                    Vlocal.Add(vec.[i]);
            } //endfor
            for (int i = 0; i < v.nElementos; i++) {
                    Vlocal.Add(v.vec.[i]);
            } //endfor

            return Vlocal;

}

VD VD::operator+(int a) {
            VD Vlocal(nElementos + 1);

            for (int i = 0; i < nElementos; i++) {
                    Vlocal.Add(vec.[i]);
            } //endfor
            Vlocal.Add(a);

            return Vlocal;

}

VD VD::operator+(int a, const VD & v) {
            VD Vlocal(nElementos + 1);

            Vlocal.Add(a)
            for (int i = 0; i < nElementos; i++) {
                    Vlocal.Add(vec.[i]);
            } //endfor

            return Vlocal;

}

VD VD::operator+=(const VD & v) {
```

```
        for (int i = 0; i < nElementos; i++) {
                v.vec.Add(vec.[i]);
        } //endfor
}

ostream & operator<< (ostream & os, const VD & v) {
        os << v.Size() << ' ';
        for (int i = 0; i < nElementos; i++) {
                os << v.Get (i) << ' ';
        } //endfor

        return os;

}

istream & operator>>  (istream & is, VD & v) {
        int n;
        is >> n;
        VD Vlocal(n);
        for (int i = 0; i < n; i++) {
                int aux;

                is >> aux;
                Vlocal.Add(aux);
        } //endfor
        v = Vlocal;

        return is;

}
```

Vamos a definir el tipo de dato conjunto (una lista donde debemos comprobar en cada nueva inserción que un elemento esta o no incluida dentro de el) usando ahora vector dinámico:

```
//Conjunto.h

#ifndef _CONJUNTO _H_
#define _CONJUNTO _H_

#include "VD.h"
#include <iosrtream>

using namesspace std;

class Conjunto {
        private:
                VD vd;
        public:
                Conjunto ();
                Conjunto (const Conjunto & c) { vd = c.vd; };

/*
La siguiente llamada es mas eficiente porque indicamos que por defecto use el constructor por copia de vector dinámicomientras en la anterior llamada realliza primero una llamada al constructor por defecto y a continuación realiza la llamada al constructor por copia
*/
                Conjunto (const Conjunto & c)::vd(c.vd) { vd = c.vd; };
                ~ Conjunto ();
                void  Add (int a);
```

```
                int Cardinal() { return vd.Size(); };
                bool Esta(int a);
                bool NoEsta(int a);
                Conjunto & operator =  (const Conjunto & c);
                Conjunto & operator +  (const Conjunto & c);
                Conjunto & operator +  (int a);
                void operator +=  (const Conjunto & c);
                bool operator ==  (const Conjunto & c);
                bool operator !=  (const Conjunto & c) { return !(*this == c); }
                friend Conjunto & operator +  (int a, const Conjunto & c);
};

friend ostream & operator << (ostream & os, const Conjunto & c);
friend istream & operator >> (istream & os, Conjunto & c);
```

Y dentro de Conjunto.cpp tendremos lo siguiente:

```
//Conjunto.cpp

#include "Conjunto.h"

void  Add (int a) {
        if (NoEsta(a)) {
                vd.Add(a);
        } //endif
}

bool Esta(int a) {
        for (int i = 0; i < vd.Size(); i++) {
                if (vd[i] == a) {

                        return true;

                }
        } //endfor

        return false;
}

bool NoEsta(int a) {

        return !(Esta(a));

}

Conjunto & Conjunto::operator =  (const Conjunto & c) {
        vd = c.vd;

        return *this;

}

Conjunto & Conjunto::operator +  (const Conjunto & c) {
        Conjunto Cnuevo(*this);

        for (int i = 0; i < c.Size(); i++) {
                int a = c.vd.Get(i);

                Cnuevo.Add(a);
```

```
        } //endfor

        return Cnuevo;

}

Conjunto & Conjunto::operator +  (int a) {
        Conjunto Cnuevo(*this);

        Cnuevo.Add(a);

        return Cnuevo;

}

/* Como es un metodo friend, no es necesario preceder escribir el metodo asi:
Conjunto & Conjunto::operator + (int a, const Conjunto & c) */

Conjunto & operator + (int a, const Conjunto & c) {
        Conjunto Cnuevo(*this);

        Cnuevo.Add(a);

        return Cnuevo;

}

// Podemos simplificar mas aun esta función si escribimos este otro codigo:

Conjunto & operator + (int a, const Conjunto & c) {

        return c + a;

}

void Conjunto::operator +=  (const Conjunto & c) {

        return *this + c;

}

bool Conjunto::operator ==  (const Conjunto & c) {
        if (Size() == c.Size()) {
                for (int i = 0; i < Size(); i++) {
                        bool encontrado = false;
                        int a  = vd.Get(i);
                        int b = vd.Get(i);

                        if (a == b) {
                                encontrado = true;
                        } //endif
                } //endfor
                if (!encontrado) {

                        return true;

                } //endif
                else {

                        return true;
```

```
                } //endelse
        } //endif
}

ostream & operator << (ostream & os, const Conjunto & c) {
        os << vd << ' ';

        return os;

}

istream & operator >> (istream & os, Conjunto & c) {
        is >> vd;

        return is;

}
```

La clase lista

La clase lista consiste en un conjunto de celdas enlazadas. Veamos como sería la definición de este tipo de dato en lista.h:

```
//Lista.h

#ifndef _LISTA _H_
#define _LISTA _H_

#include <iosrtream>

using namesspace std;

struct Celda {
        int n;
        Celda *next;
};

class Lista {
        private:
                Celda first;

                void Borrar();
                void Copiar (const Lista & l);
                void Reserva (int tam);
        public:
                Lista () { first = 0; }
                Lista (int tam) { Reserva(tam); }
                Lista (const Lista & l);
                ~ Lista () { Borrar(); }
                Lista & operator =  (const Lista & l);
                Lista & operator +  (int a);
                int Get (int pos);
                void Set (int pos, int a);
                int Size () const;
};

ostream & operator << (ostream & os, const Lista & l);
istream & operator >> (istream & os, Lista & l);
```

Y dentro de Lista.cpp tendremos lo siguiente:

```
//Lista.cpp

#include "Lista.h"
#include <cassert>

void Lista::Borrar () {
        while (first != 0) {
                Celda *p = first;

                first = first->next;
                delete p;
        } //endwhile
}

void Lista::Copiar (const Lsita & l) {
        Celda *p = first;
        Celda *q = l.first;

        while (p != 0 && q != 0) {
                p->n = q->n;
                p = p->next;
                q = q->next;
        } //endwhile
}

Lista::Reserva (int tam) {
        first = 0;
        Celda *p;

        for (int = 0; i < tam; i++) {
                Celda *aux = new Celda;

                aux->next = 0;
                if (first == 0) {
                        first = aux;
                        p= aux;
                } //endif
                else {
                        p->next = aux;
                        p = p->next;
                } //endelse
        } //endfor
}

Lista::Lista (const Lista & l) {
        int tam = l.Size();

        Reservar(tam);
        Copiar(l);
}


Lista & Lista::operator =  (const Lista & l) {
        if (*this != l) {
                Borrar();
                Reserva(l.Size());
                Copiar(l);
        } //endif
```

```
        return *this;

}

/*
La inserccion de nuevos elementos en una lista pueden realizarse de dos formas:
        - Insertando por la cola y asi respetando el orden de entrada de datos
        - Insertando por la cabeza y asi almacenando los datos en orden inverso a la entrada
Veamos como es la inserción por cola
*/

Lista & Lista::operator +  (int a) {
        Celda *aux = new Celda;
        Lista Lnueva(*this);

        aux->n = a;
        aux ->next = 0;
        while p->next != 0) {
                p = p->next;
        } //enwhile
        p->next = aux;

        return Lnueva;

}

// Y ahora veamos como es la inserción por cabeza:

Lista & Lista::operator +  (int a) {
        Celda *aux = new Celda;
        Lista Lnueva(*this);

        aux->n = a;
        aux ->next = Lnueva.first;
        Lnueva.first = aux;

        return Lnueva;

}

int Lista::Get(int pos) const {
        Celda *p = first;

        assert (pos > 0 && pos < Size() );
        for (int i = 0; i <pos; i++) {
                p = p->next;
        } //endfor

        return p->n;

}

int Lista::Set(int pos, int valor) {
        Celda *p = first;

        assert (pos > 0 && pos < Size() );
        for (int i = 0; i <pos; i++) {
                p = p->next;
        } //endfor
```

```
        p->n = valor;
}

int Lista::Size () {
        Celda *p = first;
        int contador = 0;

        while (p != 0) {
                p = p->next;
                contador++;
        } //endwhile

        return contador;
}

ostream & operator << (ostream & os, const Lista & l) {
        for (int i = 0; i < l.Size(); i++) {
                os <<  l.Get(i) << ' ';
        } //endfor

        return os;

}

istream & operator >> (istream & os, Lista & l) {
        Lista Llocal;
        int a;

        while (is >> a) {
                Llocal = Llocal + a;
        } //endwhile
        L = Llocal;

        return is;

}
```

Vamos a implementar el tipo de dato abstracto pila usando nuestra lista, una pila es un contenedor que almacena los datos apilandolos y donde solo se puede tomar el último dato insertado, es decir, el último dato que se encuentre en la posición de tope.

```
//Pila.h

#ifndef _PILA _H_
#define _PILA _H_

#include "Lista.h"

class Pila {
        private:
                Lista l;

        public:
                Pila () { }
                Pila (const Pila & p):l (P.l) { }
                ~ Pila () { }
                Pila & operator =  (const Pila & l);
                void Push (int a);
```

```
                int Pop ();
};

friend ostream & operator << (ostream & os, const Pila & p);
friend istream & operator >> (istream & os, Pila & p);
```

Y dentro de Pila.cpp tendremos lo siguiente:

```
//Pila.cpp

#include "Pila.h"

Pila & Pilla::operator =  (const Pila & l) {
        l = P.l;

        return *this;

}

void Pila::Push (int a) {
        l = l + a;
}

int Pila::Pop () {
        int a = l.Get(0);
        Lista Lnueva;

        for (int i = 1; i < l.Size(); i++) {
                Lnueva = Lnueva + l.Get(i);
        } //endfor

        l = Lnueva;

        return a;

}

friend ostream & operator << (ostream & os, const Pila & p) {
        os << P.l

        return os;
}

friend istream & operator >> (istream & os, Pila & p){
        is >> P.l

        return is;
}
```

Tema 3.- Recursividad

Definiendo la recursividad
Las funciones recursivas se definen siguiendo las siguientes reglas:

- Todas las funciones recursivas presentan un caso base y un caso general. Los casos generales tienen al menos una llamada recursiva sobre un conjunto más pequeño y una serie de operaciones.
- Las llamadas recursivas tienen que converger a un caso base. Es decir, en algún momento se debe de llegar al caso base.
- La cabecera sirve como llamada de partida o recursiva.

Veamos por ejemplo distintas soluciones para el problema del cálculo de factoriales:

```
#include <iostream>

using namespace std;

// Esta es la función factorial que realiza el cálculo lineal del factorial de un número
int Factorial (int n) {
	int res = 1;

	for (int i = 2; i < n; i++) {
		res *= i;
	} //endfor

	return res;

}

// Esta es la función factorial que realiza el cálculo recursivo del factorial de un número
int Factorial (int n) {
	// Caso base
	if (n == 0) {

		return 1;

	} // endif
	else {

		return (n*Factorial(n-1));

	} //endelse
}
```

Veamos por ejemplo distintas soluciones para el cálculo de la sumatoria de elementos de un vector:

```
#include <iostream>

using namespace std;

// Esta es la función factorial que realiza el cálculo lineal del factorial de un número
int Sumatoria (const int *v, int n) {
	int res = 0;

	for (int i = 0; i < n; i++) {
		res += v[i];
	} //endfor
```

```
        return res;

}

/*
Esta función factorial realiza el cálculo recursivo de factorial de un número. Tomamos el primer elemento del vector y le sumamos el resultado de calcular el vector de menor tamaño
*/

int SumatoriaR1 (const int *v, int n) {
        // Caso base
        if (n == 0) {

                return v[0];

        } // endif
        else {

                return (v[0]+SumatoriaR1 (v+1, n-1) );

        } //endelse
}

/*
Esta función factorial realiza el cálculo recursivo de factorial de un número. Tomamos el ultimo elemento del vector y le sumamos el resultado de calcular el vector de menor tamaño
*/

int SumatoriaR2 (const int *v, int n) {
        // Caso base
        if (n == 0) {

                return v[0];

        } // endif
        else {

                return (v[n]+SumatoriaR2 (v, n-1) );

        } //endelse
}


/*
Esta función factorial realiza el cálculo recursivo de factorial de un número. Dividimos el vector en dos partes y calculamos la suma de cada vector
*/

int SumatoriaR3 (const int *v, int n) {
        // Caso base
        if (n <= 2) {
                int suma = 0;

                for (in i = 0; i < 2; i++) {
                        suma += v[i];
                } //endfor

                return suma;

        } // endif
```

```
        else {

                return ( SumatoriaR3(v, n/2) +  SumatoriaR3( (v+n/2), (n- n/2) ) );

        } //endelse
}

/*
Esta función factorial realiza el cálculo recursivo de factorial de un número de la misma forma
que el primer ejemplo, pero en luar de devolver un valor, sobrescribe un parámetro. Es
importantetener en cuenta que la función debe ser inicializadadesde una llamada exterior
*/

int SumatoriaR3 (const int *v, int n, int & res) {
        // Caso base
        if (n == 0) {
                res =  v[0];
        } // endif
        else {
                SumatoriaR4 (v+1, n-1, res) );
                res+= v[0];
        } //endelse
}
```

En el siguiente ejemplo vamos a desarrollar una función que calcule el elemento de mayor tamaño de un vector, veamos dos formas de realizarlo recursivamente:

```
int Maximo (const int *v, int n) {
        if (n == 0) {
                return v[0];
        } //endif
        else {
                int aux = Maximo(v+1,n-1);

                if (v[0] < aux) {

                        return aux;

                } //endif
                else {

                        return v[0];

                } //endelse
        } //endelse
}

//Otra forma de calcularlo recursivamente

int Maximo (const int *v, int n, int &res) {
        if (n == 0) {
                res = v[0];
        } //endif
        else {
                Maximo(v+1,n-1,res);
                if (v[0] < res) {
                        res = v[0];
                } //endif
        } //endelse
```

```
}
```

Veamos los siguientes ejemplos para ver como trabajar con las llamadas recursivas y las funciones.

```
// Calculo de a*b de forma recursiva

int funcion (int a, int b) {
        if (b >= 1) {

                return a + funcion(a,b-1);

        } //endif
        else {

                return 0;

        } //endelse
}

// Calculo de a/b de forma recursiva

int funcion (int a, int b) {
        if (a > b) {

                return 1 + funcion(a-b, b);

        } //endif
        else {

                return 0;

        } //endelse
}

// Calculo de a^b de forma recursiva

int funcion (int a, int b) {
        if (b >= 1) {

                return a * funcion(a,b-1);

        } //endif
        else {

                return 1;

        } //endelse
}

// Calculo de log_b a de forma recursiva

int funcion (int a, int b) {
        if (a > b) {

                return 1 + funcion(a/b,b);

        } //endif
        else {
```

```
                return 0;

        } //endelse
}

// Maximo comun divisor de forma recursiva

int funcion (int a, int b) {
        if (a%b = 0) {
                cout << b << endl;
                funcion (a ,b)
        } //endif
        else {
                funcion (a ,b+1)
        } //endelse
}

// Descomposición factorial de forma recursiva

int funcion (int a, int b) {
        if (b == 0) {

                return a;

        } //endif
        else {

                return funcion (b, a%b);

        } //endelse
}
```

Algoritmo de búsqueda binaria
El algoritmo toma un vector ordenado y lo divide por la mitad, si ese elemento es mayor que el elemento que buscamos, tomaremos la mitad superior y en caso contrario la miad inferior:

```
bool BúsquedaBinaria(const int *v, int inicio, int fin, int valor) {
        if (inicio < = fin) {
                int mitad = (inicio+fin) /2;

                if (v[mitad] == x) {

                        return true;

                } //endif
                else {
                        if (v[mitad] > x) {

                                return BusquedaBinaria(v,mitad+1,fin,valor);

                        } //endif
                        else {

                                return BusquedaBinaria(v,mitad+1,fin,valor);

                        } //endelse
                } //endelse
        } //endif
```

```
        else {

                return false;

        } //endelse
}
```

Veamos otra forma de realizar la búsqueda binaria sin índices:

```
bool BúsquedaBinaria(const int *v, int n, int valor) {
        if (n >= 1) {
                int mitad = (inicio+fin) /2;

                if (v[mitad] == x) {

                        return true;

                } //endif
                else {
                        if (v[mitad] > x) {

                                return BusquedaBinaria(v,mitad,valor);

                        } //endif
                        else {

                                return BusquedaBinaria(v+(mitad+1),n-(mitad-1),valor);

                        } //endelse
                } //endelse
        } //endif
        else {

                return false;

        } //endelse
}
```

Algoritmo de cambio de base
El algoritmo toma una base y divide el cociente por la base mientras el cociente de resultado sea mayor. Después, comenzando por el último cociente y finalizando en el primer resto escribiremos el número en la nueva base. Veamos el siguiente ejemplo:

```
10|2
10 5|2          Por tanto 10 en base 10 es 1010 en base 2
 0 4 2|2
   1 2 1
     0
```

Y ahora veamos como realizar el algoritmo:

```
void CambioBase (int n, int b) {
        if (n < b) {
                cout << n;
        } //endif
        else {
                CambioBase(n/b, b);
                Cout << n%b;
        } //endelse
```

```
}
```

Podemos mejorar el algoritmo depurando el formato para hexadecimal

```
void CambioBase (int n, int b) {
        const char lista [6] = {'A', 'B', 'C', 'D', 'F'};

        if (n < b) {
                if (n < 16) {
                        cout << n;
                } //endif
                else {
                        cout << lista[n-10];
                } //endelse
        else {
                CambioBase(n/b, b);
                int resultado = n%b;

                if (resultado < 16) {
                        cout << r;
                } //endif
                else {
                        cout << lista[n-10];
                } //endelse
        } //endelse
}
```

Algoritmo Mergesort (Ordenación por mezcla)
El algoritmo toma un vector y lo divide por la mitad hasta alcanzar el caso base de ordenación. A continuación, con los dos subvectores ordenados, vamos almacenando los valores ordenados en el en el vector de la llamada. Finalizadas todas las ejecuciones, tendremos el vector ordenado:

```
void Mergesort ( int *v, int n) {
        if (n <= 2) {

                //Caso base, un vector de dos casillas y hacemos un intercambio de casillas
                if (v[1] < v[0]) {
                        int aux;

                        aux = v[0];
                        v[0] = v[1];
                        v[1] = aux;
                } //endif
        } //endif
        else {

                //Dividimos el vector en dos partes y aplicamos la recursividad
                int * izquierda = new int [n/2];
                int * derecha = new int [n-(n/2)];

                for (int i = 0; i < n/2; i++) {
                        izq[i] = aux[i];
                } //endfor
                for (int i = 0; i < (n/2)-1; i++) {
                        der[i] = v[i+(n/2)];
                } //endfor
                Mergesort(izq, n/2);
                Mergesort(der, n-(n/2));
```

```
                //Reconstruimos el vector superior con los subvectores ordenados
                int i_v = 0;
                int i_izq = 0;
                int i_der = 0;

                while ( (i_izq < n/2) && (i_der < (n-n/2)) ) {
                        if (izq[i_izq] < der[i_der]) {
                                v[i_v] = izq[i_izq];
                                i_izq++;
                        } //endif
                        else {
                                v[i_v] = der[i_der];
                                i_der++;
                        } //endelse
                        i_v++;
                } //endwhile
                delete [] izq;
                delete [] der;
        } //endelse
}

// Aunque podemos optimizar el codigo de la siguiente forma:

void Mergesort ( int *v, int n) {
        if (n <= 2) {

                //Caso base, un vector de dos casillas y hacemos un intercambio de casillas
                if (v[1] < v[0]) {
                        int aux;

                        aux = v[0];
                        v[0] = v[1];
                        v[1] = aux;
                } //endif
        } //endif
        else {

                //Dividimos el vector en dos partes y aplicamos la recursividad
                int * izquierda = new int [n/2];
                int * derecha = new int [n-(n/2)];

                for (int i = 0; i < n/2; i++) {
                        izq[i] = aux[i];
                } //endfor
                for (int i = 0; i < (n/2)-1; i++) {
                        der[i] = v[i+(n/2)];
                } //endfor
                Mergesort(izq, n/2);
                Mergesort(der, n-(n/2));

                //Reconstruimos el vector superior con los subvectores ordenados
                int i_v = 0;
                int i_izq = 0;
                int i_der = 0;

                while (i_izq < n/2) {
                        v[i_v] = izq[i_izq];
                        i_izq++;
                        i_v++;
                } //endwhile
```

```
                while (i_der < n–(n/2)) {
                        v[i_v] = der[i_der];
                        i_der++;
                        i_v++;
                } //endwhile
                delete [] izq;
                delete [] der;
        } //endelse
}
```

Las Torres de Hanoi

Este problema consiste en resolver el juego de las Torres de Hanoi. En este juego se dispone inicialmente de 3 postes separados equidistantemente el uno del otro. En uno de los postes laterales encontraremos m discos situados en un poste y ordenados de mayor a menor tamaño, de tal forma que los discos de menor tamaño descansan sobre los de mayor tamaño. El objetivo del juego es trasladar los discos de una columna a su opuesta, usando para ello la columna central y evitando que una pieza de mayor tamaño descanse sobre una menor. Por tanto el algoritmo seria de la siguiente forma:

```
void Hanoi (int m, int i, int j) {
        if (m>0) {

                // 6-i-j Calcula la unica columna que puede quedar de destino
                Hanoi(m-1, i, 6-i-j);
                cout << "Pasamos de " << i << " a " << j << endl;
                Hanoi(m-1,6-i-j, j);
        } //endif
}
```

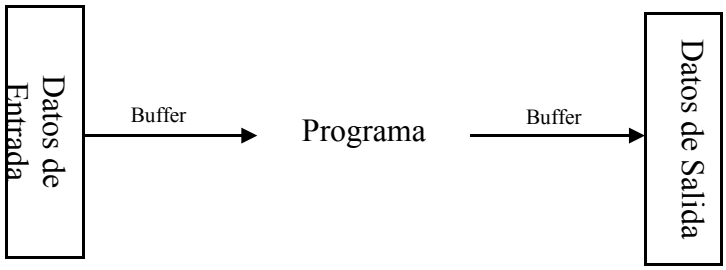
Tema 4.- Flujo de Entrada/Salida. Ficheros

Introducción
Los flujos son secuencias de bytes que pueden darse en los procesos de entrada o en los de salida. Si el flujo es de entrada este es enviado por el dispositivo al programa mientras su se trata de un flujo de salida, el programa envia el flujo de datos al dispositivo.

Los flujos son controlados por la librería iostream, la cual tenemos como istream para el control de flujos de entrada y ostream para flujos de salida. Veamos algunos ejemplos:

```
ostream(cerr) // Salida de errores
ostream(clog) // Salida de eventos
```

El funcionamiento de todo flujo puede resumirse en el siguiente esquema:



Por ejemplo un flujo puede funcionar de esta forma:

// Código previo

cout << "Pasa por aqui" << ende;
cout.flush()

// Código posterior

Hasta ahora, nuestros programas han tomado flujos de datos desde el teclado, como en el siguiente ejemplo:

```
#include <iostream>

using namespace std;

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

        cout << "Introduzca el primer entero: ";
        cin >> a;
        cout << "Introducido el primer entero: " << a << endl;;
        cout << "Introduzca el segundo entero: ";
        cin >> b;
        cout << "Introducido el segundo entero: " << b << endl;
        cout << "Introduzca el tercer entero: ";
        cin >> c;
        cout << "Introducido el tercer entero: " << a << endl;;
        cout << "Introduzca el cuarto entero: ";
        cin >> d;
        cout << "Introducido el cuarto entero: " << a << endl;;
}
```

Pero tambien nos podemos encontrar con un archivo, por ejemplo, con el siguiente formato:

□□1\t□\n9□□□\n7**t35**
Estos datos deberiamos poder procesarlos con una llamada del estilo ./main <datos.txt> salida.txt

Operaciones básicas de flujo
Todos los flujos tienen una marca de fin de secuencia, EOF, destinada a indicar el fin de la lectura de datos. Para la lectura o escritura de datos de un flujo usaremos las funciones int istream::get() y ostream & ostream::put(char c). Veamos un pequeño programa de ejemplo:

```
#include <iostream>

using namespace std;

int main() {
        int c;

        while ((c=cin.get()) != EOF) {
                cout.put(c);
        } //endwhile
}
```

De esta forma, se almacenará los datos que se encuentren hasta que aparezca un carácter de retorno de carro (\n) o se llegue al final del archivo. Por tanto, la entrada □□1\t□\n9□□□\n7\t35 es procesada como una entrada secuencial de valores 1, 9 y 7. Si lo comparamos con el siguiente código:

```
#include <iostream>

using namespace std;

int main() {
        char c;

        cin >> c;
        cout << c;
}
```

Podemos ver que los espacios no son reconocidos.

Estados de los flujos (Banderas, flags)
Un flag es un bit que se activa en cierta situación. Existen 4 tipos:

- bad(): Error fatal, por ejemplo, si trabajamos en un fichero, se activa cuando accedemos a una posición fuera de ese fichero.
- eof(): Marca el fin de un flujo.
- fail(): No podemos hacer una operación de entrada o salida, por ejemplo, buffer de entrada vacio, eof, se esperaba otro tipo de dato…
- good(): Todas las operaciones realizadas han sido correctas.

Veamos un ejemplo de uso:

```
#include <iostream>

using namespace std;

int main() {
        int dato;

        cin >> dato;
        if (cin.fail()) {
                cerr << "Error en la lectura de enteros";
```

```
        } //endif
        else {
                cout << "Se ha leido " << dato << endl;
        } //endelse

        // /Vamos a leer ahora varios enteros
        while (cin >> dato) {
                cout << dato << endl;
        } //endwhile
}
```

Por último, veamos el siguiente código:

```
#include <iostream>

using namespace std;

int main() {
        int dato;

        while (!cin.eof()) {
                cin >> dato;
                cout << dato << endl;
        } //endwhile
}
```

Si intrducimo el valor 10 y luego 10\n el valor introducido será 1010. En el siguiente ejemplo, veremos como se modifica el estado de un flujo:

```
#include <iostream>

using namespace std;

int main() {
        int dato[3];

        for (int i = 0; i < 3; i++) {
                cout << "Introduzca un entero: " << i << ": ";
                do {
                        cin >> dato[i];
                        if (cin.fail()) {

        // Borramos el estado de flujo y la unica bandera activa es good.
                                cin.clear();
                        } //endif
                }while (!cin);
        } //endfor
}
```

Veamos una forma más avanzada, usando la sobrecarga de operadores:

```
//Complejo.h

#ifndef _COMPLEJO_H_
#define _ COMPLEJO_H_

class Complejo {
        private:
                double r, i;
```

```
        public:
                Complejo () {r=rr; i =ii; }
                ~ Pila () { }
                friend ostream & operator << (ostream & os, const Complejo& C);
                istream &istream::ignore(int n = 1, char del='\n')
                istream & istream::setstate(flag);
                istream & istream::pullback(char c);
                char istream::peek();
};

//Complejo.cpp

istream & operador>>(istream & entradca, Complejo &C) {
        char car;
        double r, i;

        if (car == '(') {
                entrada >> r >> car;
                if (car == ',') {
                        entrada >> r >> car;
                        if (car == ')') {
                                c = Complejo (r, i)
                        } //endif
                        else {
                                entrada.setstate(ios::failbit);
                        } //endelse
                } //endif
                else {
                        entrada.setstate(ios::failbit);
                } //endelse
        } //endif
        else {
                entrada.setstate(ios::failbit);
        } //endelse

        return entrada;

}

// Si deseamos solo obtener la parte real, modificaremos el codigo de la siguiente forma:

istream & operador>>(istream & entradca, Complejo &C) {
        char car;
        double r, i;

        if (car == '(') {
                entrada >> r >> car;
                if (car == ',') {
                        entrada >> r >> car;
                        if (car == ')') {
                                C = Complejo (r, i)
                        } //endif
                        else {
                                entrada.setstate(ios::failbit);
                        } //endelse
                } //endif
                else {
                        entrada.setstate(ios::failbit);
                } //endelse
```

```
        } //endif
        else {
                entrada.pullback(car);
                entrada >> r;
                C = Complejo (r, 0);
        } //endelse

        return entrada;

}

// Para obtener la parte imaginaria, modificaremos el codigo de la siguiente forma:

istream & operador>>(istream & entradca, Complejo &C) {
        char car;
        double r, i;

        if (car == '(') {
                entrada >> r >> car;
                if (car == ',') {
                        entrada >> r >> car;
                        if (car == ')') {
                                C = Complejo (r, i);
                        } //endif
                        else {
                                entrada.setstate(ios::failbit);
                        } //endelse
                } //endif
                else {
                        entrada.setstate(ios::failbit);
                } //endelse
        } //endif
        else {
                entrada.pullback(car);
                entrada >> r;
                if (entrada.peek() == 'i') {
                        C = Complejo (r, 0);
                        entrada.ignore();
                } //endif
                else {
                        C = Complejo (r, 0);
                } //endelse
        } //endelse

        return entrada;

}
```

En el siguiente ejemplo, vamos a ver un proceso de entrada y salida carácter a carácter:

```
#include <iostream>

using namespace std;

void Eco(istream & i; ostream &o) {
        char c;

        while (i.get(c)) {
                o.put(c);
```

```
        } //endwhile
}

int main() {
        Eco(cin, cout);
}
```

Vamos a aumentar el grado de dificultad, estudiaremos un proceso de entrada y salida de cadenas de carácter. Para ello, usaremos las funciones istream & istream::getline(char * v, int n, char del='\n')

```
#include <iostream>

using namespace std;

int main() {
        char linea[1000];
        int  = 1i;

        while (i.getline(linea, 1000)) {
                cout << i << linea << endl;
                i++;
        } //endwhile
}
```

Podemos observar que get almacena el carácter \n en el buffer mientras getline no. Veamos ahora un par de ejemplos donde usaremos strings:

```
#include <iostream>
#include <string>

using namespace std;

int main() {
        string linea;
        int i = 1;

        while (i.getline(cin, linea)) {
                cout << i << linea << endl;
                i++;
        } //endwhile
}

// Un ejemplo un poco más complejo:

#include <iostream>
#include <string>

using namespace std;

int main() {
        string dni, linea;

        while (i.getline(cin >> dni)) {
                cout << dni << endl;
                getline(cin, linea);
        } //endwhile
}
```

Entrada/Salida de caracteres sin formato

Para trabajar la entrada y salida de flujos usaremos las funciones istream & istream::read(char *v, int n) para entrada y ostream &ostream::write (const char *v, int n) para las salidas

```
#include <iostream>

using namespace std;

int main() {
        int TAM_BLQ = 1024;
        char v[TAM_BLQ];

        // Lee y comprueba el estado del flujo
        while (cin.read(v, TAM_BLQ))  {
                cout.write(v, TAM_BLQ);
        } //endwhile

        // Escribe la cola de 500
        cout.write(v, cin.gcount());
}
```

Flujos asociados a ficheros
El flujo asociado a ficheros se puede utilizar la librería fstream, esta librería contiene la librería ifstream para lectura de fichero y ofstream para salida de ficheros. Los principales operadiores que usaremos ara trabajar con ficheros seran:

- Apertura: conectamos el flujo con el fichero en disco usando la operación open().
- Usamos los operadores de entrada y salida de archivos
- Cerrar: desconectamos el flujo del fichero en disco usando la operación close().

Veamos un programa muy basco de uso de este tipo de operadores y los flujos con un fichero:

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char * argv[]) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif

        // Declaramos el flujo
        ifstream f;

        /*
        Pasamos por argumento el nombre del fichero. Podemos omitir open() declarando:
        ifstream f(argv[i])
        */

        f.open(argv[1]);

        // Comprobamos si la apertura ha sido correcta
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;
```

```
	} //endif

	char c;

	// Leemos y escribimos
	while (f.get(c)) {
		cout.put(c);
	} //endwhile

	/*
	Cerramos el fichero. Podemos omitir close() porque esta operacion se realiza por
	defecto al terminar el programa, ya que ejecutara el destructor de f
	*/

	f.close();
}
```

En el siguiente ejemplo, vamos a implementar un programa que copie el contenido de un archivo de entrada en otro de salida:

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char * argv[]) {
	if (argc != 3) {
		cerr << "Numero de parámetros incorrectos"

		return 1;

	} //endif

	// Declaramos el flujo
	ifstream fi(argv[1]);

	// Comprobamos si la apertura ha sido correcta
	if (!fi) {
		cerr << "No puedo abrir el fichero " << argv[1];

		return 1;

	} //endif

	ofstream fo(argv[2]);

	// Comprobamos si la apertura ha sido correcta
	if (!fo) {
		cerr << "No puedo crear el fichero " << argv[2];

		return 1;

	} //endif

	//Leemos desde fi y escribimos en fo
	char c;

	while (fi.get(c)) {
```

```
                fo.put(c);
        } //endwhile

        // Miramos EOF o si el disco esta lleno
        if (!fi.eof() || !fo) {
                cerr << "No hemos podido hacer la copia"

                return 1;
        } //endif
}
```

En el siguiente ejemplo, no sabemos cuantos archivos nos dan, pero hay que listarlos

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char * argv[]) {

        // Declaramos el flujo
        ifstream fi(argv[1]);
        for (int i = 1; i < argc; i++) {

                // Comprobamos si la apertura ha sido correcta
                f.open(argv[1]);
                if (f) {
                        char c;
                        while (¨f.get(c)) {
                                cout.put(c);
                        } //endwhile
                        f.close();

                        //Bajamos banderas porque se ha activado el eofbit
                        f.clear();
                } //endif
                else {
                        f.clear();
                } //endelse
        } //endfor
}
```

El siguiente ejemplo, saca por pantalla el número de lineas que tiene el fichero:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char * argv[]) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif

        // Comprobamos si la apertura ha sido correcta
```

```
        ifstream f(argv[1]);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif

        int contador = 0;
        *string a;

        /*
        Leemos la linea de 3 formas:
                get(istream)
                getline(istream)
                getline((istream,  string)
        */
        while (getline(f ,a)) {
                contador++;
        } //endwhile
        cout << "Numero de lineas: " << contador << endl;
}
```

El siguiente ejemplo, mostraremos por pantalla hasta que encontremos en el comienzo de linea el carácter # en el fichero:

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(int argc, char * argv[]) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif

        // Comprobamos si la apertura ha sido correcta
        ifstream f(argv[1]);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
        string a;

        while (getline(f ,a)) {
                if (a[0] != '#') {
                        cout << a;
                } //endif
        } //endwhile
}
```

Modos de apertura de un fichero

Cuando usamos el operador ifstream fichero (<nombre_fichero>, modo_apertura) y ofstream fichero (<nombre_fichero>, modo_apertura) disponemos de varios modos de apertura de archivos:

- ios::in modo de apertura solo lectura, por defecto esta activado si usamos ifstream
- ios::out modo de apertura solo escritura, por defecto esta activado si usamos ofstream
- ios::binary modo binario, indica que las operaciones de entrada y salida no exigen formato y estan en binario.
- ios::atc modo adjunto (atachment), abre el archivo y se situa al final del mismo
- ios::app modo añadido (append), escribe al final del archivo este donde este stuado
- ios::trunc modo truncado (truncate), elimina la información previa en el fichero.

Estos modos de apertura pueden ser combinados, veamos algunos ejemplos:

```
// Apertura de un archivo para lectura y escritura
ifstream f("fichero", ios::in|ios::out)

// Apertura de un archivo para entrada de datos binarios
ifstream f("fichero", ios::in|ios::binary)

// Apertura de un archivo para salida de datos binarios
ifstream f("fichero", ios::out|ios::binary)

// Apertura de un archivo para entrada de datos truncados
ifstream f("fichero", ios::in|ios::trunc)

// Apertura de un archivo para añadir datos al final del mismo
ifstream f("fichero", ios::out|ios::app)
```

Las operaciones de entrada y salida en modo binario son dos: lectura (istream & stream::read(char * c, int c)) y escritura (ostream & stream::write(const char * c, int c)). Si nos fijamos en el segundo operador, n, indicamos el número de caracteres a escribir sin tener en cuenta el formato.

```
#include <iostream>
#include <fstream>

using namespace std;

int main(int argc, char * argv[]) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif

        // Leemos 100 enteros en formato binario
        int lista [100];
        for (int i = 0; i < 100; i++) {
                cin >> lista[i];
        } //endfor

        // Comprobamos si la apertura ha sido correcta
        ofstream f(argv[1], ios::out|ios::binary);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
```

```
        // Escribimos 100 enteros en formato binario en un archivo de salida
        f.write( integer_cast<const char * c>(lista), 100*sizeof(int) );

        // Otra menos eficiente, porque realizamos 100 entradas en lugar de 1
        for (int i = 0; i < 100; i++) {
                f.write( integer_cast<const char * c>(lista), sizeof(int) );
        } //endfor

        // Ahora mostramos el contenido por consola
        for (int i = 0; i < 100; i++) {
                cout << lista [i] <<endl;
        } //endfor
}
```

La clase fstream tiene la capacidad de comportarse com ifstream u ofstream, permitiendo las operaciones de entrada o salida según sea necesario

```
#include <iostream>
#include <fstream>

using namespace std;

void Leer (istream &i) {
        int datos[100];

        i.read( integer_cast<const char * c>(lista), 100*sizeof(int));
        for (int i = 0; i < 100; i++) {
                cout << datos[i] << endl;
        } //endfor
}

void Escribir (ostream &o) {
        int datos[100];

        for (int i = 0; i < 100; i++) {
                cin >> datos[i];
        } //endfor
        o.write( integer_cast<const char * c>(datos), 100*sizeof(int) );
}

int main(int argc, char * argv[]) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif

        fstream f(argv[1], ios::in|ios::binary);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
        Escribir(f);

        fstream f(argv[1], ios::out|ios::binary);
```

```
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
        Leer(f);
}
```

Veamos como usar una estructura persona para escribir en un fichero:

```
struct Persona {
        char DNI[9];
        char nombre[20];
        char apellidos[30];
        char direccion[50];
        char tlf[9];
};

istream & operador >> (istream &i, Persona &P);

istream & operador >> (istream &i, Persona &P) {
        i >> P.DNI;
        i >> P.nombre;
        i >> P.apellidos;
        i >> P.direccion;
        i >> P.tlf;

        return i;

}

// Podemos reescribir el codigo de la siguiente forma:

istream & operador >> (istream &i, Persona &P) {
        i.read(reinterpred_cast<char *>(&DNI), 9);
        i.read(reinterpred_cast<char *>(&nombre), 20);
        i.read(reinterpred_cast<char *>(&apellidos), 30);
        i.read(reinterpred_cast<char *>(&direccion), 50);
        i.read(reinterpred_cast<char *>(&tlf), 9);

        return i;

}

// O bien reescribirlo en una única linea de el codigo:

istream & operador >> (istream &i, Persona &P) {
        i.read(reinterpred_cast<char *>(&P), sizeof(Persona));

        return i;

}
```

Continuando con la estructura del ejemplo anterior. Vamos a desarrollar los metodos para añadir un nuevo objeto persona y para buscar una persona con un determinado dni:

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

struct Persona {
        char DNI[9];
        char nombre[20];
        char apellidos[30];
        char direccion[50];
        char tlf[9];
};

int main (int argc, char * argv []) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif

        fstream f(argv[1], ios::in|ios::app|ios::binary);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
        f.write(reinterpred_cast<char *>(&P), sizeof(Persona));
}

bool VerPersona(const char * name, const char * dni, Persona &Pout) {
        fstream f(argv[1], ios::in|ios::binary);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return false;

        } //endif
        Persona P;
        bool encontrado = false;
        while (!encontrado && f.read(reinterpred_cast<char *>(&P), sizeof(Persona))) {
                if (strcmp(dni, P.DNI) == 0) {
                        encontrado = trae;
                } //endif
                if (encontrado == true) {
                        Pout = P;
                } //endif
        } //endwhile

        return encontrado;

}
```

Ficheros de acceso aleatorio

Existen 3 operaciones que usaremos dependiendo de si se trata de un archivo de entrada (ifstream) o salida (ofstream):

- El operador que indica la posición actual: tellp() o tellg()
- El operador que indica la posición absoluta en un fichero: seekp(posición absoluta) o seekg(posición absoluta)

- El operador que indica la posición absoluta en un fichero respecto a una base: seekp(desplazamiento, base) o seekg(desplazamiento, base). La base puede ser de 3 tipos:
    - ios::beg (beggin), al comienzo del archivo
    - ios::cur (current), respecto a la posición actual
    - ios::end (end), al final del archivo

Es facil asociar las operaciones, las acabadas en g (get, obtener) estan asociadas a los ficheros de entrada mientras que las asociadas a las p (put, poner) a salida. Veamos un ejemplo de uso, en el siguiente ejemplo vamos a acceder a una direccion concreta de un tipo de dato double:

```
#include <iostream>
#include <fstream>

using namespace std;

double GetRead (istream & entrada, int i);

int main (int argc, char * argv []) {
        if (argc != 2) {
                cerr << "Numero de parámetros incorrectos"

                return 1;

        } //endif
        int pos;

        cin >> pos;
        fstream f(argv[1], ios::in|ios::binary);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
        cout << GetRead(f, pos);
}

double GetRead (istream & entrada, int i) {
        entrada.seekg(i*sizeof(double));
        if (entrada) {
                double d;

                entrada.read(reinterpred_cast<char *>(&d), sizeof(double));

                return d;

        } //endif
        else {

                return 0.0;

        } //endelse
}
```

En este otro ejemplo, calculamos el tamaño de un archivo:

```
int Size (const char * nombre) {
```

```
        fstream f (nombre, ios::in);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return false;

        } //endif
        f.seekg(0,ios::end);
        int contador = f.tellg();

        return contador;

}
```

O bien podemos recorrer un fichero de texto en orden inverso:

```
int Inverso (const char * nombre) {
        fstream f (nombre, ios::in);
        if (!f) {
                cerr << "No puedo abrir el fichero " << argv[1];

                return 1;

        } //endif
        f.seekg(-1,ios::end);
        int posicion = f.tellg();

        while (posicion >= 0) {
                char c;

                f.get(c);
                cout.put(c);
                posicion--;

                // Si llegamos a 0 y avanzo una casilla y obtendre un error
                if (posicion >= 0) {
                        f.seekg(posicion, ios::end);
                } //endif
        } //endwhile
}
```