Capítulo 4

Punteros y gestión de memoria. Clases.

En el capítulo 2 hemos introducido los vectores y las matrices como arreglos con subíndices. Comentamos en dicho capítulo que un problema de la utilización de arreglos es que, una vez que se han definido, no se puede liberar la zona de memoria que ocupan cuando ya no son necesarias. Comentamos también el hecho de que las matrices, cuando figuran como argumentos de funciones, hay que llamarlas por referencia en vez de por valor, para evitar la copia de grandes cantidades de números en variables locales. En este capítulo introducimos los punteros, que proporcionan una forma mucho más eficiente de trabajar con arreglos y con estructuras complejas, como las clases. Introducimos además un conjunto eficiente de clases de vectores y matrices y de clases de algoritmos lineales, que son adaptaciones de la liberías TNT (*Templates Numerical Toolkit*) y JAMA (*JAva Matrix Algorithms*), respectivamente. Con objeto de poder realizar representaciones gráficas con *Gnuplot* desde los programas, incluimos la librería `gnuplot_i++`. Finalmente, para la realización de cálculos con aritmética de precisión arbitraria introducimos la librería `mpfun`.

4.1. Punteros

Un puntero es una variable o constante cuyo valor es una dirección de memoria, en general asociada a otra variable o función. Podemos pensar en la memoria del ordenador como un conjunto de celdillas de tamaño de 1 byte (8 bits). El valor del puntero es la celdilla de memoria donde empieza la variable. Si se imprime este valor, dicho valor sale por defecto en notación hexadecimal. El valor del puntero da el byte de comienzo de la variable, pero cuando utilizamos el puntero, necesitamos también saber donde acaba dicha variable para leerla enteramente sin invadir la memoria correspondiente a otra variable. Para saber donde acaba una variable, necesitamos saber el tamaño de la misma, lo cual es equivalente a conocer su tipo. Por lo tanto los punteros se definen como apuntando a una variable de un tipo dado. Las sentencias

```
int* ip;
double* dp;
```

definen `ip` como un puntero que apunta a una variable de tipo `int` y `dp` como un puntero que apunta a una variable de tipo `double`. Hay una cierta ambigüedad en la declaración de punteros,

ya que

```
int *p;
```

es equivalente a la declaración anterior, mientras que la sentencia

```
int *p, q;
```

que parece indicar que declara una variable `p` como un puntero a `int` y otra variable `q` como `int`, esta declarando en realidad dos punteros a `int`. Para evitar errores en la lectura, se suele declarar un puntero por línea.

El valor de la variable a la que apunta un puntero se obtiene con el operador * (operador de desreferencia o dirección):

```
int i=20;
int* p=&i;//p apunta a i
int k=*p; // k=20
```

Un puntero con valor 0 no apunta a ningún objeto, por lo que no reserva ninguna cantidad de memoria:

```
p=0; //no apunta a ningún objeto
```

Los punteros permiten cambiar el contenido de una posición de memoria:

```
int j=20;
int *p=&j;
*p=10; //j ahora vale 10
```

La posibilidad de cambiar el contenido de las posiciones de memoria es la fuente de casi todas las aplicaciones de los punteros, y de todos los errores que su uso produce. Es por esto que hay que utilizar los punteros con precaución, y sólo cuando su uso está justificado.

No se puede asignar a un puntero de un tipo dado la dirección de una variable de otro tipo:

```
int* p;
double d=5.;
p=&d; //error &d es una dirreccion de double y p apunta a un int.
```

Cuando declaramos un arreglo (vectores, matrices,..), el nombre del arreglo es un puntero que apunta al primer elemento del arreglo. Todos los elementos del arreglo se encuentran colocados de forma consecutiva en memoria.

```
double matriz[20][20];
```

En esta sentencia, `matriz` es el nombre de un puntero que señala al primer elemento de un arreglo de 400 números del tipo `double`.

Una función puede aceptar punteros como argumentos y devolver valores de punteros como valor:

```
double* f(int*, double); //declaración válida
```

Un puntero toma como valor direcciones de memoria. La dirección de memoria de una variable se obtiene con el operador &

```
int n;
int* p=&n; //p apunta a la variable n
```

Un puntero puede apuntar a otro puntero. En este caso es un puntero doble. Se escribe como:

```
int** pp=&p;
```

Un puntero debe de apuntar a una dirección válida de memoria para poder tomar un valor. Si un puntero se crea sin inicializarlo, toma en general un valor de una dirección de memoria

no válida, es decir no apunta a ningún objeto del tipo que le corresponde, o puede apuntar a una dirección de memoria no accesible. En muchos compiladores el puntero vale 0, es decir no apunta a ningún objeto. Si se le da un valor al contenido de la dirección a la que apunta un puntero no inicializado mediante el operador *, curiosamente no da error de compilación en muchos compiladores, pero sin embargo, se produce en general un error de ejecución:

```
int* kk; // dirección indefinida, kk=0 en muchos compiladores
*kk=5; // probable error de ejecucion, damos un valor a una dirección inexistente de memoria
```

Este es uno de los errores que los compiladores pueden no detectar y a los que hay que prestar especial atención.

Ejercicio 1: Escribir una función que intercambie dos variables de tipo `double`, y que tome como argumentos los punteros a dichas variables (intercambiad los valores de los punteros). Verificad su funcionamiento llamándola desde un programa principal.

4.1.1. Punteros constantes y que apuntan a constantes

Una constante explícita no tiene dirección de memoria:

```
int* p=&20; //error, 20 no tiene dirección de memoria
```

Sin embargo, una constante con nombre sí que tiene dirección, y en este caso, el puntero debe de especificarse como apuntando a un objeto constante:

```
const int i=20;
int* p=&i; // error, p no se ha definido apuntando a una constante
const int* p=&i // correcto
```

Sin embargo, es lícito que puntero constante apunte a una variable:

```
j=10;
p=&j //correcto, aunque j no es constante
```

No se puede cambiar, sin embargo, el valor de un puntero constante:

```
*p=5; //error, p apunta a una constante
```

Si se desea que el puntero en sí mismo sea constante, hay que declararlo como

```
int* const q=&i; //el valor de q es constante
q = &j; // error q no puede apuntar a otra variable distinta que la de la inicialización
```

En este caso lo que permanece constante es la dirección de memoria, es decir la celdilla física, a la cual apunta el puntero, aunque pueda variar su contenido. Notemos la diferente posición del identificador `const` según se especifique la constancia del puntero o de la variable a la que apunta.

Los punteros constantes se deben inicializar en la declaración

```
double* const d; //error, d no inicializado
```

Se puede cambiar el valor de un puntero constante mediante el operador *

```
*q = 3; // correcto, ahora i=3
```

Un puntero se puede declarar como un puntero constante que apunta a una constante:

```
const double* const k=&j;
k=&i;// error k es constante
```

```
*k=i ; // error, el valor de k es constante;
```

Sin embargo, si un puntero constante que apunta a una constante se inicializa con una variable, el valor al cual apunta se puede cambiar cambiando el valor de la variable:

```
int m=50;
const int* const pm=&m; // correcto, *pm=50
m=60; // correcto, ahora *pm=60
```

La constancia de punteros y variables es un instrumento de gran valor para prevenir errores inadvertidos en ejecución. Exigiendo constancia de los valores que no deben cambiar,durante la ejecución del programa, cualquier intento de cambio debido a un error inadvertido de programación, produce un mensaje de error en compilación. Sin embargo, es importante que los punteros constantes que apunten a constantes se inicialicen realmente con constantes, salvo que haya una razón poderosa para que no sea así, pues si se inicializan con variables, el valor al cual apuntan cambiar al cambiar dichas variables, sin que esto produzca nigún error o advertencia.

Ejercicio 2: Escribir un programa en el que se verifiquen las anteriores afirmaciones sobre los punteros.

4.1.2. Punteros y gestión dinámica de memoria

Un puntero se puede crear y destruir en tiempo de ejecución. La sentencia

```
new int* p;
```

crea en ejecución el puntero `p`, reservando la memoria necesaria (el espacio necesario para guardar un entero). La sentencia

```
delete p;
```

destruye el puntero `p` y deja la memoria que ocupa libre, a disposición del sistema operativo. Esto es especialmente importante en el caso de variables que ocupan una gran cantidad de memoria y cuya dimensión no es conocida a priori en tiempo de compilación, o que son necesarias solamente durante una pequeña fracción del tiempo total de ejecución del programa. Es por esta razón que los operadores `new` y `delete` realizan una gestión dinámica de memoria, es decir, en tiempo de ejecución.

4.1.3. Vectores y punteros

La declaración

```
double* vec[20];
```

crea un arreglo de punteros que apuntan a variables de tipo `double` de dimensión 20, mientras que la declaración

```
double (*vec)[20];
```

crea un puntero a un arreglo de números de tipo `double` de dimensión 20. En este último caso, `vec` es un puntero que apunta al primer elemento del arreglo, es decir a `vec[0]`. Si al puntero le sumamos un entero `i`, el resultado es un puntero que apunta a la posición i del arreglo

```
int i=5;
double a=*(vec+i) //a = vec[5];
```

La sentencia

```
double* v = new double [20];
```

reserva la memoria necesaria para un arreglo de 20 números de tipo `double`. Al igual que en el caso anterior, `v` es un puntero que apunta al primer elemento `v[0]`. La declaración y asignación se puede hacer en dos pasos distintos:

```
double* v:
v=new double [n];
```

La sentencia

```
delete [] v;
```

libera la memoria asignada a `v` y elimina el puntero. La memoria queda libre a disposición del sistema operativo.

Cuando tenemos un puntero que apunta a un arreglo, podemos acceder a los elementos del mismo con el operador de subíndice []:

```
for(int i = 0; i <n; i++) cout <<v[i] <<endl;
```

4.1.4. Matrices y punteros

Una matriz se puede definir como un puntero doble o un puntero de punteros. Por ejemplo, una matriz cuadrada de dimensión `n` la podemos definir como:

```
double** dmatriz = new double* [n]; // vector de punteros=filas de matriz
for(int i = 0; i <n; i++) dmatriz[i] = new double [n]
//cada fila es un vector de dimensión n
```

Una vez que hemos definido la matriz, podemos acceder a sus elementos mediante los operadores de subíndices: `dmatriz[i][j]` representa el elemento de la fila `i` y la columna `j`. Todos los elementos de la matriz están almacenados de forma contigua en memoria, una fila detrás de otra (diferente del FORTRAN, donde se almacenan columnas consecutivas, lo cual es importante cuando se compilan subrutinas en FORTRAN con programas en C/C++). Por ejemplo, si queremos rellenar la matriz con los valores de la matriz de Hilbert, `dmatriz[i][j]=1./(i+j+1)` (comenzando en `i=j=0`) escribimos

```
for(int i = 0; i <n; i++)
  for(j = 0; j<n; j++) dmatriz[i][j] = 1./(i+j+1);
```

Una vez que hemos utilizado la matriz y que ya no la necesitamos, devolvemos la memoria al sistema:

```
for (int i = 0; i <n; i++)
  delete [] dmatriz[i];
delete [] dmatriz;
```

Notemos que si no realizamos el primer paso, eliminamos el puntero `dmatriz` pero no liberamos la memoria asignada a cada uno de los punteros de las filas `dmatriz[i]`. Dejamos la eliminación de esta memoria en manos del compilador, y no hay ninguna garantía de que efectivamente lo haga.

A diferencia de las matrices definidas directamente, estudiadas en el Capítulo 2, aquí no necesitamos saber la dimensión de la matriz en compilación, sólo en ejecución. Una vez que hemos acabado de utilizar la matriz, devolvemos la memoria al sistema. Notemos que `dmatriz[i]` es

un vector que representa la fila `i+1` de la matriz y que `dmatriz[i][j]` representa el elemento `(i+1, j+1)`.

La notación de punteros para las matrices es especialmente útil en el caso de matrices que tienen una gran parte de sus elementos nulos. Este tipo de matrices aparecen frecuentemente en Cálculo Numérico. Un caso especialmente frecuente es el de las matrices tridiagonales. En este caso, sólo hay 3 elementos no nulos por fila. Una forma de definir una matriz de este tipo, economizando memoria, es

```
double** dmatriz = new double* [n];// vector de n punteros=filas de matriz
for(int i = 0; i <n; i++)
  dmatriz = new int [3];
//cada fila es un vector de dimension 3
```

De la forma anterior ahorramos una gran cantidad de memoria si `n` es mucho mayor que 3.

4.1.5. Funciones y punteros

Una función puede devolver un puntero, y puede tener punteros como argumentos. Cuando una función tiene un puntero como argumento, el efecto de la llamada a la función es el de una llamada por referencia a la variable a la que apunta el puntero, ya que si dos punteros tienen el mismo valor, apuntan a la misma dirección física de memoria. Una función que toma un puntero como argumento, puede cambiar la variable a la que apunta el puntero, salvo que especifiquemos lo contrario mediante una sentencia `const`:

```
double fun(const double* const px); // ni px ni *px pueden cambiar
```

Si sólo especificamos que el puntero es constante, no podemos hacer que apunte a otra variable pero el valor de la variable puede cambiar. En particular, si el puntero es el nombre de un arreglo, los valores de los elementos del mismo pueden cambiar.

Un ejemplo típico de función que devuelve un puntero, es la que calcula el producto de dos matrices. En el programa `matmult_punt.cpp` se reescribe el programa de multiplicación de matrices con punteros

```
#include <iostream>
#include <fstream>
// Programa de multiplicacion de dos matrices
using namespace std;

float **
matmult(float **const m1, float **const m2, int nn, int nm, int nq)
{
  float          ** matprod = new float *[nm];
  for (int i = 0; i < nm; i++)
    matprod[i] = new float[nq];
  for (int i = 0; i < nm; i++) {
    for (int j = 0; j < nq; j++) {
      matprod[i][j] = 0;
```

```
      for (int k = 0; k < nn; k++)
        matprod[i][j] += m1[i][k] * m2[k][j];
    }
  }
  return matprod;
}

int
main()
{
  // fichero de entrada
  ifstream fin("mat1.dat");
  // fichero de salida
  ofstream fout("matmult1.dat");
  //definicion  de dimensiones
  int             n, m, p, q;
  // lectura de dimensiones y elementos de matrices y posterior escritura
  // en pantalla
  fin >> m >> n;
  float         **const mat1 = new float *[m]; //mat1 apunta a direccion constante
    for (int i = 0; i < m; i++)
    mat1[i] = new float[n];
  for (int i = 0; i < m; i++)
    for (int j = 0; j < n; j++)
      fin >> mat1[i][j];
  fin >> p >> q;
  float         **const mat2 = new float *[p];
  // mat2 apunta a direccion constante
    for (int i = 0; i < p; i++)
    mat2[i] = new float[q];
  for (int i = 0; i < p; i++) {
    for (int j = 0; j < q; j++)
      fin >> mat2[i][j];
  }
    float         **mat3 = new float *[m];
  for (int i = 0; i < m; i++)
    mat3[i] = new float[q];
  // Comprobacion de compatibilidad de dimensiones
    if (n != p) {
    cout << " Dimensiones incorrectas. No se pueden multiplicar las matrices" << endl;
    return 0;
  }
  // Multiplicacion de matrices
```

```
  mat3 = matmult(mat1, mat2, n, m, q);
  // Impresion del resultado en fichero
  for (int i = 0; i < m; i++) {
    for (int j = 0; j < q; j++) {
      fout << mat3[i][j] << "   ";
    }
    fout << endl;
  }
  fin.close();
  fout.close();
  return 0;
}
```

Una función no debe devolver un puntero a una variable local de dicha función, como es el caso de las variables temporales, puesto que dichas variables son liberadas al acabar la llamada a la función, y su valor puede ser utilizado en principio por el OS. Sin embargo, en general los compiladores no dan un mensaje error en este caso.

También podemos definir un puntero a una función, ya que una función es un objeto que reside en memoria. Los punteros a funciones se pueden utilizar como argumentos de funciones.

4.2. Clases

Las clases del C++ son esencialmente una mejora de las estructuras del C. La capacidad del C++ de programar clases constituye la característica que lo convierte en un lenguaje orientado a objetos y de alto nivel, en contraposición al C. Una clase es esencialmente una estructura que contiene datos y funciones. Una característica de las clases es que se pueden crear cuando hagan falta y destruir con facilidad cuando ya no son necesarias, lo que constituye una gran comodidad en la gestión dinámica de la memoria. En este capítulo vamos a explicar las clases desde el punto de vista del usuario de las mismas, sin entrar en los detalles de la programación de clases. Ya hemos utilizado la clase `complex` en el tema anterior. Construíamos un número complejo como

```
complex<double>c=(3.,0.);
```

En la línea anterior, para crear un número complejo hemos utilizado el llamado constructor estándar. Otra forma de construir un número complejo es copiándolo de otro ya existente:

```
complex<double>b=c;
```

que es lo que se llama el constructor de copia. Si deseamos conocer las partes real e imaginaria de un número complejo, las podemos obtener mediante

```
double re = real(c);
double im = imag(c);
```

Las funciones `real()` e `imag()` son *funciones de acceso*, es decir funciones que permiten obtener cada uno de los componentes elementales de la clase. Podemos hacer cálculos con números complejos mediante las f*unciones públicas* de la clase, que son funciones definidas en la clase que admiten un argumento complejo, como por ejemplo

`complex<double>d = sin(c);`

También podemos utilizar los operadores aritméticos:

`complex<double>c1 = c * d + d / cos(d);`

Estas operaciones son posibles porque en la programación de la clase se ha indicado que los operadores +, -, * y / se pueden aplicar a números complejos, y se han programado sus reglas de operación. Se dice que estos operadores han sido *sobrecargados*, es decir, cuando los operandos son números complejos se les ha dado una definición distinta de la usual para los tipos nativos. Notemos que si `c1 = re1 + im1i` y `c2 = r2 + im2 i`, `c1 * c2 = re1 * re2 - im1 * im2 + (re1 * im2 + re2 * im1) i`, que es una definición completamente distinta de la multiplicación de números reales. La capacidad de sobrecargar operadores es muy conveniente para el cálculo científico, ya que proporciona claridad en la notación. Pero no todo son ventajas, ya que cada vez que se realiza una operación con un operador sobrecargado, el programa tiene que acceder a la definición de ese operador, lo que consume tiempo de cálculo. En cualquier caso, la sobrecarga es una posibilidad que proporciona elegancia a la programación, con un bajo precio a pagar en término de recursos de tiempo y memoria, si se utiliza correctamente.

De lo dicho anteriormente, se desprende que si deseamos utilizar una clase, debemos de conocer la siguiente información: los constructores de la clase, los operadores sobrecargados y las funciones públicas de acceso. Utilizando correctamente estos datos, una clase no es más que una extensión del concepto de tipo. Por ejemplo, la utilización del tipo `complex`, que no es un tipo nativo sino una clase, no es muy diferente de la de los tipos `int` o `double`.

4.2.1. La clase *string*

La librería estándar del C++ incluye la clase `string`, que permite manipular cadenas de caracteres. Las cadenas de caracteres se pueden unir mediante el operador +. Las funciones `size()`, `erase()`, `find()`, `replace()`, permiten manipular y editar cadenas de caracteres. La función `c_str()` convierte las cadenas de la clase string en cadenas de C. Esta función es necesaria, por ejemplo, si deseamos convertir una string en el nombre de un fichero de la clase `fstream`. Una string de C no es más que una variable del tipo `char*`, mientras que las strings de la clase `string` son estructuras completamente distintas. El programa `string.cpp`, a continuación, muestra como se utilizan las funciones de la clase string. La función `getline` permite rellenar una string leída desde teclado.

```
//compilar con
//g++
#include <iostream>
#include <iomanip>
#include <string>
#include <fstream>
using namespace std;

int main(){
```

```
  //Uso de la clase string
  string s1="En un lugar de la mancha";
  string s2=" de cuyo nombre no quiero acordarme";
  //suma de string
  string s3=s1+s2;
  string s4=" ";
  //funciones size() y lentgh()
  int i= s1.size();
  int j= s2.size();
  int k=s3.length();
  cout<<s3<<endl;
  cout<< i<<s4<<j<<s4<<k<<s4<<endl;
  //funciones substr, replace, find, erase e insert
  string s5=s1.substr(6,5);
  cout<<s5<<endl;
  s1.replace(18,6,"Mancha");
  cout<<s1<<endl;
  int pos = s3.find("mancha");
  cout<<pos<<endl;
  s3.erase(pos,6);
  cout<<s3<<endl;
  s3.insert(pos,"Alcarria");
  cout<<s3<<endl;
  // conversion a C strings
  ofstream fout(s5.c_str());
  fout<<s3<<endl;
  fout.close();
  cout<<"Entrar una linea"<<endl;
  //lectura de strings a partir del teclado
  getline(cin,s1);
  cout<<s1+s2<<endl;
  return 0;
}
```

4.2.2. Las clases de entrada - salida: ostream, istream, ofstream y ifstream

Las operaciones de entrada y salida mediante los operadores $<<$ y $>>$ se realizan invocando las clases `ostream` e `istream` para la salida y entrada estándar, y `ofstream` e `ifstream` para la salida y entrada mediante ficheros.

4.3. Clases de vectores y matrices. Librería TNT.

Vamos a utilizar en estas prácticas la librería TNT (Template Numerical Toolkit, Roland del Pozo), adaptada para las necesidades específicas de este curso. Para acceder a las funciones de esta librería hemos creado el namespace CALNUM. Esta librería proporciona una serie de clases, pensadas para el cálculo numérico intensivo. En particular vamos a utilizar las clases Vector y Matrix. Son clases del tipo de patrones ("templates") que dependen de parámetros. La librería no se compila, sino que se introducen, tanto las funciones y operadores como los algoritmos, mediante cabeceras. Esta librería permite realizar operaciones con vectores y matrices, escribiendo el código de una forma análoga a la que se utiliza en un texto de álgebra. El tipo de los elementos de los vectores y matrices es el parámetro del template y puede ser int, double o long double. También funciona para otras clases, con tal que las operaciones definidas sean válidas. Para números complejos, las funciones de la librería no son válidas para aquellas operaciones que precisen el complejo conjugado, como el producto escalar de dos vectores complejos, o la matriz transpuesta, que sería la matriz hermítica conjugada. Sin embargo, no es complicado escribir una especialización de las clases Vector y Matrix y de las funciones correspondientes para argumentos de tipo `complex`. Los operadores sobrecargados son los siguientes:

Vectores

Sean dos vectores `v1` y `v2` de la misma dimensión, y un escalar `c`. Las siguientes operaciones están definidas

1. `v1 = v2`: asignación a v1 de v2.
2. `Vector v2(v1)`. Creación ce una copia v2 de v1. Es ell lamado *copy constructor*.
3. `Vector v2 = v1`: Creación de v2 con el contenido de v1. Es equivalente a hacer una copia.
4. `Vector v1(n, c)`: Creación de un vector de dimensión n con todos los elementos iguales a c.
   `Vector v1(n)` reserva un vector de n componentes. Notad los paréntesis redondeados, en vez de cuadrados.
5. `v1 = c`: Creación de un vector con todos los elementos de v1 iguales a c.
6. `v1 + v2`: Suma de vectores.
7. `v1 += v2`: v1 =v1+v2.
8. `v1 - v2`: Resta d e vectores.
9. `v1 -= v2` : v1 = v1-v2.
10. `v1 * v2` : Producto escalar de vectores (En la versión original de TNT es un vector con elementos el producto elemento a elemento de v1 y v2).

11. `c * v1, v1*c`: producto de un escalar por un vector.
12. `v*=v: v=c*v` (Añadido a la vesión original de TNT).
13. `v=v/c: v=v*(1/c)` (Añadido a la versión original de TNT).
14. `v/=c: v=v*(1/c)` (Añadido a la versión original de TNT).
15. `cout<<v` : Impresión de un vector, primero la dimensión y luego sus elementos.
16. `cin>>v`: Lectura del vector, primero la dimensión y luego los elementos.

Las siguientes funciones están definidas:

1. `dot_prod(v1,v2)` : v1*v2.
2. `v.destroy()`: Destructor del vector.
3. `v.dim()`: Devuelve un valor entero igual a la dimensión del vector.
4. `v.size()`: Devuelve un valor entero igual a la dimensión del vector.

Matrices

Sean tres matrices `m`, `m1` y `m2`, un vector `v` y un escalar `c`. Las siguientes operaciones están definidas:

1. `Matrix m(p,q, c)`: Creación de una matriz de p filas y q columnas con todos los elementos iguales a c. Si c no se especifica, se toma como 0.
2. `Matrix m = c`: Se ponen todos los elementos de m iguales a c.
3. `Matrix m1 = m`: Creación de m1 como una copia de m.
4. `Matrix m2(m)` : Copy constructor. La matriz m2 es una copia de m.
5. `m = m1` : Asignación a m del contenido de m1.
6. `m1 + m2` : Suma ordinaria de matrices.
7. `m1 - m2` : Diferencia ordinaria de matrices.
8. `m1 * m2` : Producto de matrices.
9. `m += m1`: m = m + m1 (Añadido a la versión original de TNT).
10. `m -= m1`: m = m - m1 (Añadido a la versión original de TNT).
11. `m *= m1`: m = m*m1 (Añadido a la versión original de TNT).

12. `c*m, m*c`: producto de un escalar por una matriz. Añadido. Hay que convertir `c` al tipo `T` del argumento de `Vector<T>`. En particular, si `T = double`, basta poner el punto decimal para realizar la conversión (`5*v` da error pero no `5.*v`).

13. `m/c` : m*(1/c) (Añadido a la versión original de TNT).

14. `m /= c`: m = m *(1/c) (Añadido a la versión original de TNT).

15. `m(i,j)`: elemento (i,j) ; i,j comienzan en 1.

16. `m[i][j]`: elemento (i+1,j+1); i,j comienzan en 0.

17. <<: Impresión de una matriz, primero sus dimensiones y después sus elementos, ordenados por filas.

Las siguientes funciones de matrices están definidas:

1. `transpose(m)`: Devuelve ls transpuesta de m.

2. `m.num_rows()`: Devuelve el número de filas de m.

3. `m.num_cols()`: Devuelve el número de columnas de m.

4. `m.size()`: Número entero igual al producto del número de filas por el número columnas.

5. `matmult(m1,m2)`: Devuelve m1*m2.

6. `matmult(m,v)`: Devuelve m*v.

7. `mult_element(m1,m2)`: Devuelve el producto de matrices elemento a elemento.

Además existe la clase `stopwatch` que permite crear cronómetros, con la finalidade de conocer el tiempo que tarda en ejecutarse un grupo de instrucciones. Tiene las siguientes funciones:

1. `stopwatch Q;` : Crea un cronómetro Q.

2. `Q.start();` : Inicia el cronómetro. Se lamma al comienzo del grupo de instrucciones cuyo tiempo de ejecución queremos conocer.

3. `Q.stop();` : Para el cronómetro. La llamada a esta funnción se coloca al final del grupo de instrucciones.

4. `Q.resume();` : Vuelve a poner en marcha un cronómetro parado.

5. `double time = Q.read()`: lee el valor del tiempo y lo deposita en la variable `time`.

Aparte de las clases mencionadas hay otras varias que podéis consultar en la documentación en formato html de la librería. El siguiente programa ilustra el empleo de la librería TNT:

```
// Ejemplos con TNT
// compilar con  g++ - I./templates -o tnt_ejemplos  tnt_ejemplos.cpp
#include <cmath>
#include <iostream>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;

int main()
{
  int             n; //dimension, que puede ser leida en ejecucion
  cout << "Entrar dimension" << endl;
  cin >> n;

  //Ejemplos con vectores

  Vector < double >a(n); //Construccion de vectores
  Vector < double >b(n), c(n);
  for (int i = 0; i < n; i++)
    a[i] = i; //Relleno de vector a
  b = 2.;  //inicializacion:todos los elemtos = 2
  cout << "a= " << a << "b= " << b << endl;
  c = a + b; //suma de vectores
  cout<<"c=a+b= "<<c<<endl;
  c+=a;    //sobtrcarga +=
  cout<<"c+=a= "<<c<<endl;
  c-=a;
  cout<<"c-=a= "<<c<<endl;
  c*=n;
  cout<<"c*=n = "<<c<<endl;
  c/=n;
  cout<<"c/=n = "<<c<<endl;
  c=a%b;
  cout<<"c=a%b="<<c<<endl;

  cout<< "norma de c = "<<norm (c) <<endl; //norma del vector

  double          s = a * b;  //producto escalar de a y b
  double          d = dot_prod(a, b); //otra forma de producto escalar

  cout << "producto escalar= " << s << endl;
  cout << d << endl; //impresion de vectores
  double          x = 5.;
```

```
 //productos de un escalar por un vector a la derecha y a la izquierda
 cout << "c= " << c << endl; //impresion vector c
 c = c * x;
 cout << "c*x =" << c << endl;
 c = x * c;
 cout << "c= " << c << endl;

 //Ejempos con matrices y vectores

 //Construccion de tres matrices de double
 Matrix < double >A(n, n), B(n, n), C(n, n);
 for(int i = 0; i < n; i++)
   A[i][i] = double (i); //relleno de la matriz A
 B = 1.; //todos los elemtos de B = 1.

 //Sobrecarga += para matrices
 A += B;
 cout<< "A+=B "<< A<<endl;


 //Sobrecarga -= para matrices
 A -= B;
 cout<< "A-=B ="<<A<<endl;


 //Sobrecarca *= para matrices
 A *= B;
 cout<<"A*=B ="<<A<<endl;

 //Sobrecarga *= Matrices y escalares
 A*=x;
 cout<<"A*=x ="<<A<<endl;

 //Sobrecarga /= Matrices y escalares
 A/=x;
 cout<<"A/=x ="<<A<<endl;

//Sobrecarga * para matrices
C = A * B; //multiplicacion de matrices
cout << C << endl; //impresion matriz C

 C = C + A - B; //Suma y resta de matrices
 A = transpose(C); //transpuesta de una matriz
```

```
  c = B * b; //producto de una matriz por un vector
  cout << "B= " << B << " b= " << b << " c=B*b= " << c << endl;

  C = B * x;//Producto de una matriz por un escalar
  cout<<"C= B*x= "<<C<<endl;

  B= C/x; // Division de una matriz por un escalar
  cout<<"C/x= "<<B<<endl;
  double tr=spur(B);
  cout<< "Traza de B=C/x= "<< tr <<endl; //traza

  cout << "A= " << A << "c= " << c << "x= " << x << endl;
  c = x * A * c * x; //producto de un escalar por matriz por vector por escalar
  cout << " c=x*A*c*x= " << c << endl;
}
```

4.4. Clases de objetos de Gnuplot

El programa `gnuplot_i++` proporcione una interfaz de un programa con *Gnuplot*. Es necesario introducir la cabecera `gnuplot_i.h` en el programa que va a utilizar *Gnuplot*, y compilar dicho programa conjuntamente con el fichero `gnuplot_i.cpp`.

Se crea un objeto de *Gnuplot* con la sentencia

```
Gnuplot g1=Gnuplot();
```

donde g1 es el nombre del objeto, a nuestra elección. Este es el constructor estándar. Las sentencias

```
g1.set_style("lines");
g1.plot_xy(xx,yy," ");
```

dibujan dos vectores xx e yy, yy en función de xx, con estilo `lines`, es decir con líneas. La sentencia

```
g1.cmd("replot \"pol.gpl\" with errorbars");
```

dibuja el fichero `pol.gpl` con barras de errores. La función `cmd()` envía un comando de *Gnuplot* como una cadena de caracteres. Notemos que las comillas se representan por \" en el interior de esta cadena de caracteres.

Para poder contemplar la figura durante un intervalo de tiempo, hay que realizar una llamada a la función `sleep()`, con argumento el número de segundos que queremos que permanezca la figura en la pantalla. Alternativamente, podemos poner una llamada a la función `pause()`, con lo que la figura permanece en la pantalla hasta que interrumpamos la ejecución del programa con un Ctrl-C.

El programa `gnuploti_ejemplo.cpp` es un ejemplo del uso de `gnuplot_i++`:

```
// Programa ejemplo del uso de gnuplot_i++
```

```
// Compilar con
// g++ -I./templates -o gnuploti_ejemplo gnuploti_ejemplo.cpp gnuplot_i_calnum.cpp
#include "gnuplot_i_calnum.h"
#include <iostream>
#include <cmath>
#include <iomanip>
#include <fstream>
#include "algebralineal.h"
using namespace std;
using namespace CALNUM;
#define SLEEP_LGTH 100 //la figura permanece 100 segundos en pantalla

//Ejemplo de uso de gnuplot_i++

int main()
{
//Leer xmin y xmax
double          xmin, xmax;
cout << "Entrar xmin" << endl;
cin >> xmin;
cout << "Entrar xmax" << endl;
cin >> xmax;
int n = 200;
//dibujamos 200 puntos entre xmin y xmax
//Declaracion de los vectores de puntos
Vector < double >xx(n), yy(n);

//Relleno de los vectores
        double h = (xmax - xmin) / n;
xx[0] = xmin;
yy[0] = sin(xx[0]);
for (int i = 1; i < n; i++)
xx[i] = xx[i - 1] + h;
for (int i = 1; i < n; i++)
yy[i] = sin(xx[i]);

//Dibujo de puntos yy versus xx
Gnuplot g1 = Gnuplot();
g1.set_style("lines");
g1.plot_xy(xx, yy, "funcion seno ");

//Algunas sentencias alternativas
        // g1.set_style("points");
```

```
//g1.set_style("errorbars");
//g1.plot_xy(xx, error, "");
//g1.cmd("replot \"fichero.gpl\" with errorbars");
//g1.cmd("replot \"pol.gpl\" with errorbars");

sleep(SLEEP_LGTH);//la figura permanece SLEEP_LGTH segundos

        // Con las dos sentencias abajo la figura permanece hasta recibir un Ctrl - C
// const sigset_t * s;
//sigsuspend(s);

return 0;
}
```

4.5. Clases de números con precisión arbitraria

Mediante el empleo de clases, es posible realizar cálculos en aritmética de precisión arbitraria como la realizamos con números ordinarios. Un número en precisión arbitraria está codificado como una cadena de caracteres lo suficientemente larga como para contener el número deseado. Como la tablas de caracteres tiene 128 elementos estos números están codificados en base 128, lo que permite representar números muy elevados en base 10 con relativamente pocos dígitos. Por supuesto, el uso de cadenas de caracteres para representar números hace que las operaciones con números de precisión arbitraria sean mucho más lentas que las operaciones con números normales, por lo que dichos números en precisión arbitraria se deberían utilizar sólo para aquellas operaciones donde la pérdida de precisión es un grave riesgo, como la inversión de matrices o solución de sistemas de ecuaciones mal condicionados.

Existen muchas librerías que permiten trabajar con números de precisión arbitraria. Hemos incluido en esta práctica la librería *mpfun++* que es sencilla y poderosa, aunque algo antigua. Deberéis de instalar y compilar esta libería en vuestro sistema. Crearéis el directorio `mpfun`, donde descomprimiréis el archivo `mpfun++.tar.gz` con

```
tar -zxvf mpfun++.tar.gz
```

Bajáis al directorio *mpfun* y hacéis `make`. Se creará la librería `libmp.a` en `mpfun/lib`. Podéis instalar esta librería en `/usr/local/lib` y los ficheros .h que se encuentran en `mpfun/include` en `/usr/local/include` (si podéis trabajar como superusuarios en Linux o Macosx), o en `/usr/lib` y `/usr/include` en `cygwin`. Si no sois superusuarios, podéis simplemente colocar `mpfun` en el directorio donde compiléis vuestro programa. En este caso, el comando de compilación de un fichero (fich.cc) que utilice la librería `libmp.a` será

```
g++ -I./mpfun/include -L./mpfun/lib fich.cc -lmp
```

Vamos a ver ahora algunos ejemplos prácticos de utilización de esta librería. El fichero fuente debe de incluir la cabecera

```
#include "mpint.h"
```

si sólo trabajamos con números enteros,

```
#include "mpreal.h"
```

si sólo trabajamos con números reales (incluyendo enteros) o

```
#include "mpcomplex.h"
```

si trabajamos con números complejos. Sólo es necesario incluir una de estas cabeceras. Se definen tres clases: `mp_int, mp_real y mp_complex`. La librería tiene los siguientes parámetros: `mpipl`, que es la máxima precisión inicial en dígitos y vale 1000 por defecto; `mpiou`, que es la precisión de salida de los números y vale 56 por defecto; y el número más pequeño que es posible representar con todos sus dígitos, que vale 10^{-900} . Estos parámetros se definen en `mpfun/src/mpfun.cpp` y se pueden cambiar, aunque en este caso hay que recompilar la librería.

Constructores

Construimos los números de precisión arbitria de la siguiente forma:

```
mp_int ia=-6;
mp_real p=5.0;
mp_real x="1.234567890 1234567890 1234567890 D-100"
```

En el último ejemplo hemos inicializado `x` mediante una cadena de caracteres. Los blancos no cuentan.

Sobrecarga de operadores

Los siguientes operadores están sobrecargados: `+, -` (unario y binario), `*, /, ++, --` (ambos pre y post), `+=, -=, *=, /=, ==, !=, <=, >=, <, >, <<, >>`.

Funciones públicas

Cada una de las clases tiene definidas una serie de funciones análogas a las de la librería `cmath`, como `power` (no pow), `log`, `log10`, funciones trigonométricas y funciones trascendentes. Una lista completa de las funciones disponibles se encuentra en las cabeceras `mpint_friends.h`, `mpreal_friends.h` y `mpcomplex_friends.h`.

Ejemplos de uso

El siguiente ejemplo muestra el fichero `mpej1.cpp` en el que se realiza el cálculo de π y e y de sus funciones

```
//compilar con
//g++   -I./mpfun/include -L./mpfun/lib -o mpej1 mpej1.cpp -lmp
#include <iostream>
#include <iomanip>
#include "mpreal.h"
using namespace std;
//definimos Double como sinonimode mp_real
```

```
typedef mp_real Double;
int main()
{
//caculamos pi
    Double pi=4*atan(Double(1.));
    cout<< "El numero de bytes de un mp_real es "<<sizeof(mp_real)<<endl;
//Calculamos la tangente de pi
    Double xxx=tan(pi/4);
    xxx=power(xxx,5);
//imprimimos el resultado con mpiou (56) decimales
    cout<<"pi="<<pi<<endl;
//calculamos el seno de pi/2
    cout<<"sin(pi/2)="<<xxx<<endl;
//calculamos e
    Double ee=exp(Double(1.));
//imprimimos e
    cout<<"e= "<<ee<<endl;
//imprimimos log(e)
    cout<<"log(e)= "<<log(ee)<<endl;
    return 0;
}
```

Al compilarlo y ejecutarlo da como resultado

```
pi=10 ^         0 x  3.14159265358979323846264338327950288419716939937510582097,

sin(pi/2)=10 ^         0 x  1.,

e= 10 ^         0 x  2.71828182845904523536028747135266249775724709369995957496,

log(e)= 10 ^         0 x  1.,
```

En el siguiente ejemplo, más complicado, consideramos la inversión de una matriz de Hilbert de orden N, utilizando las librerías TNT y JAMA (namespace CALNUM).

```
//Ejemplo de mpfun con  JAMA y TNT
//Valores y vectores propios de una matriz de Hilbert de orden N
//compilar con
//g++ -I./templates -I./mpfun/include  -o mpfun_tnt  mpfun_tnt.cpp -L./mpfun/lib  -lmp
#include <iostream>
#include <iomanip>
#include "mpreal.h"
#include "algebralineal.h"
```

```
using namespace CALNUM;
using namespace std;

//Double sinonimo de mp_real

typedef mp_real Double;

//typedef long double Double;

int main(){


//Definicion matriz de Hilbert de orden 100
int N;

cout<<" Entrar la dimension de la matriz de Hilbert"<<endl;
cin>>N;

//Calculo del tiempo de ejecucion. Definicion cronometro Q
Stopwatch Q;

//Comienzo cronometro
Q.start();


CALNUM::Matrix<Double> A(N,N);
for (int i=0;i<N;i++)for (int j=0;j<N;j++)
{A[i][j]=Double(1./(i+j+2));}

//descomposicion LU
CALNUM::LU<Double>  lu(A);
Matrix<Double> L(N,N);
L=lu.getL();
//cout<<"L="<<L<<endl;

Matrix<Double> U=lu.getU();
//cout<<U<<endl;

//Vector<int> p=lu.getPivot();
//cout<<"Pivote="<<p<<endl;

// Multiplicacion de L*U
```

```
Matrix<Double>NM(N,N);
NM=matmult(L,U);
//Salida de L*U-A
cout<<"L*U-A= "<<NM-A<<endl;

//Calculo del  determinamte de L*U
Double det=lu.det();
cout<<"determinante(L*U)= "<<det<<endl;


//Calculo de valores propios
Eigenvalue<Double> eig(A);
Vector<Double> aa(N);

//poner los valores propios reales en aa
eig.getRealEigenvalues(aa);

//vectores propios, se escriben en la matriz L
eig.getV(L);

//matriz diagonal, se escribe eb H
Matrix<Double> H(N,N);
H=0.;
eig.getD(H);

cout<<"Valores Propios= "<<aa<<endl;
cout<<"Vectores Propios= "<<L<<endl;
cout<<"Matriz diagonal= "<<H<<endl;


//calculo de la inversa
//Definicion de la matriz identidad XX
//inicializacion de XX a la matriz nula
Matrix<Double> XX(N,N,0.);
for (int i=0;i<N;i++) XX[i][i]=Double(1.);

// la inversa ZZ es la solucion de A*ZZ=XX;

Matrix<Double> ZZ=lu.solve(XX);

//cout<<ZZ<<endl;

//Comprobacion Inv(A)*A=I
```

```
Matrix<Double> QQ=matmult(ZZ,A);// QQ debe ser la matriz identidad.

//Incluso con long double falla miserablemente para N=20
//con mpfun el error esta en el decimal 963

cout<<"Matriz identidad="<<QQ<<endl;

//Salida del tiempo de calculo. Parada cronometro y lectura.
Q.stop();
Double time=Q.read();
cout<<"time="<<time<<endl;

return 0;

}
```

Con N=50, en un ordenador con procesador PowerPC de 1.63 Ghz tarda 140 s y el error en la matriz identidad, impresa al final, está en el decimal 970. Con `long double` el programa falla miserablemente incluso para N=20.

4.6. Ejercicios a presentar como memoria

1. Escribid un programa que defina dos variables `a=10` y `b=20,` y una función `intercambia(&a, &b)` que las intercambie utilizando referencias, es decir, después de la llamada de la función, `a=20` y `b=10`. Escribid otra función, `intercambia2(pa,pb)` que intercambie `a` y `b` butilizando los punteros `pa` y `pb` de `a` y `b,` respectivamente.

2. Escribid un programa que lea la dimensión de dos vectores de la misma dimensión, y los cree utilizando el operador `new`. Los valores de los vectores se pueden leer desde la entrada estándar o desde fichero, a elección. Seguidamente, debéis calcular el producto escalar de ambos vectores e imprimirlo por la salida estándar (se puede utilizar una función para calcular el producto escalar o hacerlo directamente en `main()`). Finalmente, los vectores se deben destruir con `delete` antes de finalizar el programa.

3. Escribid un programa que lea tres `string` en la entrada estándar. Debe de crear un fichero con el nombre de la primera `string` y que contenga la segunda y la tercera string unidas. Debe de imprimir, además, la longitud de esta `string` total.

4. Sean los vectores $v1 = (1,2,3,4)$ y $v2 = (-1,0,1,0)$, y las matrices $A = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix}$

y $B = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$. Escribid un programa, utilizando la librería TNT, que calcule

$$C = A * A^T + B * B^T$$

$$C += B$$

$$v3 = 5 * v1 + (v1 * v2) * v2$$

$$v4 = A * v3 + B * v2$$

Definid cada variable del tipo que coresponda (ej. Vector v3). Imprimid los vectores y matrices resultantes en fichero.

5. Utilizando la librería `mpfun`, calcular un programa que calcule e imprima el factorial de 200 y el número de Fibonnaci 200 (de forma no recursiva).

6. Haced un programa que produzca una ventana con la gráfica de `tan(x)` entre -10 y 10, utilizando la librería `gnuplot_i++`. La gráfica debe de permanecer 10 segundos en pantalla.