Capítulo 2

Vectores, matrices y funciones

En el capítulo anterior hemos considerado sólo operaciones matemáticas con números. Sin embargo en Física aparecen frecuentemente objetos compuestos por conjuntos de números, tales como vectores y matrices. Estos objetos se pueden definir fácilmente en C++. En este capítulo consideraremos sólo los aspectos más sencillos, dejando para un capítulo posterior los aspectos más complejos relacionados con punteros y clases, que son también los que hacen el C++ una herramienta poderosa para el cálculo numérico.

2.1. Vectores

Un vector se define simplemente como una variable del tipo que se desee con la dimensión entre corchetes:

```
double vec[n];
float vec1[10];
int vec2[i];
```

El valor del índice varía de 0 a $dimension - 1$; esto es álgo que se debe tener siempre muy presente, pues es una de las causas más frecuentes de errores.

El programa *vectores.cc* lee la dimensión de dos vectores reales y sus valores desde un fichero y calcula y saca en pantalla su producto escalar:

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main(){
//fichero de entrada
ifstream fin("vec.dat");
//definicion lectura de dimension
int n;
fin>>n;
```

```
//definicion y lectura de vectores
float vec1[n],vec2[n];
for(int i=0;i<n;i++) fin>>vec1[i];
for(int i=0;i<n;i++) fin>>vec2[i];
//definicion y calculo del producto escalar
float producto_escalar=0;
for(int i=0;i<n;i++) producto_escalar=producto_escalar+vec1[i]*vec2[i];
cout<<"Producto escalar = "<<producto_escalar<<endl;
return 0;
}
```

Ejercicio 1: *Hacer un programa que calcule el producto vectorial de dos vectores de dimension 3. Imprimir el resultado en pantalla, junto con los vectores iniciales, que deben leerse desde un fichero.*

Notemos que el valor de *n* se conoce en ejecución, no en compilación, es decir se introduce cuando se ejecuta el prpgrama.. Eso significa que se reserva la memoria necesaria cuando se ejecuta el programa. Esto es imposible en otros lenguajes, como por ejemplo en FORTRAN77, donde hay que dar a *n* el valor máximo para el que se desea utilizar el programa.

Un vector puede inicializarse al declararlo de la siguiente forma:

```
double dvec[3]={1.,3.,0.5};
```

2.2. Matrices

La definición de matrices es igualmente sencilla. La sentencia

```
float mat[n][m];
```

define `mat` como una matriz rectangular de *n* filas y *m* columnas, con elementos de tipo *float*. Análogamente se declararía una matriz compuesta por otro tipo de elementos. Una matriz puede inicializarse en su declaración. Se pueden poner todos sus elementos entre llaves simples:

```
double dmat[3][3]={1.,2.,3.,4.,5.,6.,7.,8.};
```

en cuyo caso los elementos se leen por filas, y si falta alguno se toma como 0. En el caso anterior `dmat[3][3]=0`; Otra posibilidad es poner cada fila entre llaves:

```
double dmat1[3][3]={{1.,2.},{,4.,5.,6.},{7.,8.,9.}};
```

En este caso cada conjunto de elementos entre las segundas llaves se toma como una fila. Si falta alguno en una fila se toma nulo, como en el caso anterior para dmat1[1][3].

El programa `matmult.cc` lee dos matrices desde fichero y las multiplica, imprimiendo el resultado en fichero y pantalla:

```
#include <iostream>
#include <fstream>
//Programa de multiplicacion de dos matrices
using namespace std;
int main(){
//fichero de entrada
```

```
ifstream fin("mat.dat");
//fichero de salida
ofstream fout("matmult.dat");
//definicion lectura de dimension
int n,m, p,q;
float mat1[m][n],mat2[p][q],mat3[m][q];
//lectura de dimensiones y elementos de matrices y posterior escritura
//en pantalla
fin>>m>>n;
cout<<m<<"   "<<n<<endl;
for(int i=0;i<m;i++){
  for(int j=0; j<n;j++) fin>>mat1[i][j];
  }
for(int i=0;i<m;i++){
 for(int j=0;j<n;j++) cout<<mat1[i][j]<<"  ";
 cout<<endl;
 }
 cout<<endl;
fin>>p>>q;
cout<<p<<"  "<<q<<endl;
for(int i=0;i<p;i++){
  for(int j=0; j<q;j++) fin>>mat2[i][j];
  }
for(int i=0;i<p;i++){
 for(int j=0;j<q;j++) cout<<mat2[i][j]<<"  ";
 cout<<endl;
 }
 cout<<endl;
//Comprobacion de compatibilidad de dimensiones
if(n!=p){
cout<<" Dimensiones incorrectas. No se pueden multiplicar las matrices"<<endl;
return 0;
}
//Multiplicacion de matrices
for(int i=0;i<m;i++){
   for(int j=0;j<q;j++){
     mat3[i][j]=0;
     for(int k=0; k<n;k++)
       mat3[i][j]=mat3[i][j]+mat1[i][k]*mat2[k][j];
     }
   }
//Impresion del resultado
for(int i=0;i<m;i++){
```

```
   for(int j=0;j<q;j++){
     cout<< mat3[i][j]<<"   ";
     }
   cout<<endl;
   }
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;
}
```

Estudiad en detalle cada una de sus partes.

Ejercicio 2: *Escribid un programa que sume matrices, leyéndolas desde fichero y escribiendo el resiultado en fichero y en pantalla.*

Con este método de definir matrices no podemos liberar la memoria ocupada por las mismas una vez que hemos dejado de utilizarlas, es decir, podemos crearlas pero no eliminarlas cuando no nos sean útiles. Veremos en prácticas posteriores que este problema se resuelve mediante la utilización de punteros, y sobre todo utilizando el concepto de clase, en el que radica la capacidad de orientación a objetos del C++.

2.3. Funciones

Hasta ahora hemos escrito todos los programas en una sóla función llamada *main()*. Sin embargo esto no siempre es posible y mucho menos práctico. Normalmente se estructura un programa de forma que *main()* llame a otras funciones que realizan tareas y que a su vez pueden llamar a otras funciones. Las funciones pueden ser funciones matemáticas convencionales o realizar cualquier otra tarea, como imprimir resultados. Las funciones tienen un tipo, que puede ser cualquiera de los tipos de variables del C++. Una función puede ser del tipo *void*, que significa que no devuelve ningún valor. El tipo *void* no se puede aplicar a la función *main()* en el C++ estándar aunque si que se podía en versiones antiguas del compilador. La función *main()* debe ser forzosamente de tipo *int*. Una función que no sea del tipo *void* debe devolver un valor del tipo de la función mediante una sentencia return. Sin embargo, la función *main()* puede no devolver nada, es decir se puede omotir la sentencia return. Sin embargo, se acostrumbra a que *main()* devuelva 0 si todo ha acabado satisfactoriamente y 1 u otro entero no nulo en caso de error. Las funciones del tipo void carecen de sentencia return, es decir no devuelven nada. El programa *radio.cc* listado a continuación lee un valor real por teclado y escribe el volumen de la esfera, superficie de la esfera, área del círculo y longitud de la circunferencia con radio el valor leído, utilizando funciones. Las funciones las hemos definido del tipo `float`.

```
#include <iostream>
#include <cmath>
using namespace std;
const double    pi = 4 * atan(1.);
double
vol(double r)
{
    return 4 * pi / 3 * r * r * r;
} double
sup(double r)
{
    return 4 * pi * r * r;
}
double
circulo(double r)
{
    return pi * r * r;
}
double
circunf(double r)
{
    return 2 * pi * r;
}

int
main()
{
    double          x;
    cout << "Entrar un radio" << endl;
    cin >> x;
    double          volumen = vol(x);
    double          superficie = sup(x);
    double          area = circulo(x);
    cout << "El volumen de una esfera  de radio " << x << " es " << volumen << endl;
    cout << "La superficie de una esfera  de radio " << x << " es " << superficie << e
    cout << "El area de un circulo de radio " << x << " es " << area << endl;
    cout << "La longitud de una circunferencia  de radio " << x << " es " << circunf(x
    return 0;
}
```

Los argumentos de las funciones tienen una validez local, en el sentido de que son copias temporales de los argumentos de la sentencia de llamada a la función, y desaparecen cuando se acaba de ejecutar la función. Esto garantiza que las funciones no puedan alterar los argumentos

de las mismas

```
double f(double x){
x=2*x;
return x;
}
int main(){
double pi=3.14;
double s=f(pi); // pi sigue valiendo 3.14, en vez de 6.28
return 0;
}
```

2.4. Variables locales y globales. Alcance de las variables

Una variable definida en una función, sólo es accesible y está definida en esa función. En otra función podemos declarar otra variable con el mismo nombre, que puede tener cualquier tipo y que no interferirá en modo alguno con la primera. Una variable declarada dentro de un bucle *for*, no está declarada fuera del mismo. Esto es válido también para el índice del bucle *for*, aunque no lo era en versiones antiguas del C++. Es por esto que siempre especificamos *int i* para el índice del bucle. Sin embargo es posible declarar una variable fuera de las funciones, en el cuerpo del programa. En este caso se dice que la variable es global, y su valor es conocido por todas las funciones del fichero fuente. Este es el caso de la variable `pi` en el programa `radio.cc`.

2.5. Constantes

Las constantes tienen un nombre como las variables, pero se deben inicializar al mismo tiempo que se declaran y no se pueden cambiar posteriormente durante la ejecución del programa; las constantes se deben declarar declarar de tipo *const*. Si se intenta cambiar su valor, se produce un error de compilación, que en general es mucho más facil de detectar que un error de ejecución que suele producir resultados aparentemente correctos. Así, hemos declarado en el programa *radio.cc*

```
const double pi=4.*atan(1.);
```

2.6. Llamada de funciones por valor de los argumentos y por referencia

Cuando llamamos a una función, las variables de los argumentos del programa que ls llama se copian en una copia local de la función. De esta forma, aunque la función cambie el valor de los argumentos, no hay riesgo de que estos cambien en la función que las llama. Este tipo de llamada de una función se denomina *llamada por valor*. Sin embargo, el que los argumentos permanezcan inalterados no es siempre lo deseado, pues frecuentemente es necesario escribir una función que

intercambie variables, por ejemplo para ordenar una lista. Por otro lado la copia en variables locales consume memoria y tiempo de cálculo, lo cual puede ser especialmente importante si el argumento es una matriz de grandes dimensiones. Una alternativa a la llamada por valor es la llamada por referencia. En este caso, l se especifica como argumento una *referencia*, lo que es equivalente a proporcionar la posición de memoria en la que está almacenada la variable. El operador & produce una *referencia a una variable*. Podemos definir una referencia como un sinónimo de la variable que se refiere a la misma posición de memoria en la que está almacenada la variable y no a una copia temporal de la misma almacenada en otra posición de memoria.

```
double x=3.;
double& y =x; // y es sinonimo de x. y=3
y=2*y;// x=6 ya que x e y son sinónimos
```

Consideremos el programa `swap_ref.cc` en el que hay dos funciones de intercambio, `swap(int, int)` y `swap_ref(int&, int&)`. La primera no produce ningun efecto en las variables de los argumentos mientrs que la segunda intercambia los argumentos

```
#include <iostream>
using namespace std;
void swap_valor(int k, int l)
{
  int temp;
  temp=k;
  k=l;
  l=temp;
}

void swap_ref(int& k, int& l)
{
  int temp;
  temp=k;
  k=l;
  l=temp;
}

int
main()
{
    int i=5,j=7;
    cout<<"valores originales"<<endl;
    cout<<"i= "<<i<<" j= "<<j<<endl;
    cout<<"intercambio, llamada por valor"<<endl;
    swap_valor(i,j);
    cout<<"i= "<<i<<" j= "<<j<<endl; //los valores no se intercambian
    swap_ref(i,j);
```

```
    cout<<"intercambio, llamada  por referencias"<<endl;
    cout<<"i= "<<i<<" j= "<<j<<endl;//correcto, los valores se intercambian
    return 0;
}
```

Si deseamos garantizar que una función no pueda cambiar un argumento que se llama por referencia por referencia, podemos definir dicho argumento como constante en la llamada a la función mediante el prefijo `const:`

```
double fun(const double& x);
```

Las llamadas a las funciones por referencia son muy frecuentes en C++ debido a que este tipo de llamada ahorra el tiempo de procesador y la memoria necesarios para copiar las variables en otras variables temporales cuando se llama a la función. Esto es particularmente importante cuando los argumentos son grandes vectores y matrices, o *clases de objeto*s, definidas en la práctica 4. Siempre que deseemos garantizar que el objeto que aparece en el argumento permanece inalterado, se utiliza la declaración *conts*.

2.7. Funciones con vectores y matrices como argumento

Consideremos el programa anterior de multiplicación de matrices. Se puede estructurar de forma que la multiplicación de matrices se realice en una función llamada `matmult`, como se muestra en el programa `matmultfun.cc`:

```
#include <iostream>
#include <fstream>
// Programa de multiplicacion de dos matrices
using namespace std;
const int       n = 3, q = 2, m = 4, p = 3;

void
matmult(float mat1[][n], float mat2[][q], float mat3[][q])
{
    //Comprobacion de compatibilidad de dimensiones
    if (n != p) {
        cout << " Dimensiones incorrectas. No se pueden multiplicar las matrices" << e
        exit(1);
    }
    //Multiplicacion de matrices
        for (int i = 0; i < m; i++)
        for (int j = 0; j < q; j++)
            mat3[i][j] = 0.;
    for (int i = 0; i < m; i++)
        for (int j = 0; j < q; j++)
```

```
            for (int k = 0; k < n; k++)
                mat3[i][j] += mat1[i][k] * mat2[k][j];
}

int
main()
{
    //fichero de entrada
    ifstream fin("mat1.dat");
    //fichero de salida
        ofstream fout("matmultfun.dat");
    //definicion de dimensiones y matrices
        float           mat1[m][n], mat2[p][q], mat3[m][q];
    //lectura de dimensiones y elementos de matrices y posterior escritura
        // en pantalla
        for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            fin >> mat1[i][j];

    cout << "Primera Matriz" << endl;
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)
            cout << mat1[i][j] << "  ";
        cout << endl;
    }
    cout << endl;

    for (int i = 0; i < p; i++)
        for (int j = 0; j < q; j++)
            fin >> mat2[i][j];

    cout << "Segunda matriz" << endl;
    for (int i = 0; i < p; i++) {
        for (int j = 0; j < q; j++)
            cout << mat2[i][j] << "  ";
        cout << endl;
    }
    cout << endl;
    //Llamada a la funcion de multiplicacion
        matmult(mat1, mat2, mat3);
    //Impresion del resultado en pantalla y fichero
        cout << "Matriz producto" << endl;
    for (int i = 0; i < m; i++) {
```

```
        for (int j = 0; j < q; j++)
            cout << mat3[i][j] << "   ";
        cout << endl;
    }
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < q; j++)
            fout << mat3[i][j] << "   ";
        fout << endl;
    }
    //cierre de las unidades de entrada y salida
        fin.close();
    fout.close();
    return 0;
}
```

Conviene estudiar en detalle este programa. Vemos que las matrices `mat1, mat2 ,mat3` figuran como argumentos de la función pero que la primera dimensión no se especifica en la función. Además, las dimensiones se especifican fuera de las funciones como del tipo *const int* . Eso significa que son constantes enteras, es decir que no se pueden cambiar. Las constantes hay que inicializarlas en el programa, es decir, no se pueden leer en la ejecución. Además la definición se realiza fuera de las funciones, es decir, son *variables globales*, cuyo valor es conocido para todas las funciones incluídas en el fichero de código fuente. La función *matmult* es de tipo *void*, es decir no devuelve nada. Podríamos pensar en principio que la función matmult podría definirse de forma que devuelva una matriz, sin embargo una función no pueden devolver una matriz de la forma que estamos utilizando aquí, es decir como objetos con subíndice. Sin embargo, pueden devolver un *puntero*, que es una posición de memoria donde se encuentra la matriz. El concepto de puntero lo veremos posteriormente. Hemos diseñado la función *matmult* de forma que el producto de las dos matrices se almacene en el tercer argumento. Como los argumentos de una función no se transmiten al programa que las llama cuando la llamada es por valor, hemos utilizado la llamada por referencia

```
matmult(mat1,mat2,mat3);
```

De esta forma el valor de `mat3` se transmite a `main()`. El nombre de una matriz o vector es de hecho una referencia a la posición de memoria en la que se almacena dicho vector; es por esto que no hemos necesitado utilizar el símbolo de referencia &. La llamada por referencia es mucho más eficiente, en el sentido de que se evita la copia de las matrices en variables temporales locales de la función, lo cual puede consumir una cantidad importante de tiempo de cálculo en el caso de matrices grandes.

En general, interesa crear funciones para todas aquellas operaciones que realizamos frecuentemente en el programa. De esta manera evitamos tener que escribir muchas líneas de código y el programa queda mucho más claro y legible. Si realizamos frecuentemente el cubo de una variable del tipo double podemos definir la función como

```
double cubo(double x){
return x*x*x;
```

```
}
```

El problema del uso de funciones es que su llamada cuesta tiempo de procesador, ya que se almacenan en memoria en un lugar distinto del programa que las llama. Una forma de evitar el consumo de tiempo adicional es declararlas como *inline*. Por ejemplo, en el caso anterior pondríamos

```
inline double cubo(double x){
return x*x*x;
}
```

Una llamada a una función de tipo *inline* expande el código en el punto de llamada, sustituyendo la sentencia de llamada por las líneas de código de la función, como si la hubiésemos reescrito. Sin embargo, la declaración de una función como *inline* es sólo una sugerencia al compilador y no tenemos ninguna garantía que que el compilador la cumpla. Sólo la considerará como *inline* si la función es suficientemente sencilla. Hay que tener especial cuidado con el uso de funciones en el interior de los bucles, que es donde se consume el 90 % del tiempo de cálculo en un programa científico típico.

Las funciones deben de aparecer en el fichero del código fuente antes de que se las llame, pues en caso contrario el compilador da un mensaje de error. No siempre es esto posible ni práctico. Una forma de resolver el problema es declararlas al principio del programa, antes de la primera función y después de las cabeceras, en una sentencia en la que aparecen los tipos de la función y de sus argumentos, como se hace en el programa `cubodecl.cc`:

```
#include <iostream>
#include <cmath>
using namespace std;
double cubo(double);
main(){
double x;
cout<<"Entrar un numero"<<endl;
cin>>x;
double c=cubo(x);
cout<<"El cubo de "<<x<<" es "<<c<<endl;
return 0;
}

double cubo(double y){
return y*y*y;
}
```

Ejercicio 3: *Suprimid la declaración de la funcion, compilad y leer el mensaje de error.*

Se pueden incluir todas las declaraciones de funciones en una cabecera. En este caso hay que poner la extension .h:

```
#include "micabecera.h" //incluye declaraciones de funciones
```

2.8. Recursividad de la llamada de funciones

Un aspecto importante de las funciones en C/C++ es la recursividad, que significa que una función puede llamarse a sí misma. Vamos a verlo con un ejemplo de cálculo de los números de Fibonacci. Estos números se definen como $f_0 = 0$, $f_1 = 1$, $f_n = f_{n-1} + f_{n-2}$. Crecen extraordinariamente deprisa y el cociente f_n/f_{n-1} converge rápidamente a la media áurea $M = (1+\sqrt{5})/2 \simeq 1{,}618$, que muchos han considerado como el cociente ideal que deben satisfacer ciertas proporciones en arte y anatomía. El programa `fibonnacci.cc` calcula dichos números:

```
#include <iostream>
#include <fstream>
#include <cmath>
#include<iomanip>
using namespace std;
long int fibonacci(int);
int main(){
cout<<"Entrar un numero entero"<<endl;
int n;
cin>>n;
long int fb=fibonacci(n);
cout<<" El numero de Fibonacci "<<n<<"-esimo es "<<fb<<endl;
}
long int fibonacci(int m){
if (m==0) return 0L;
 else if (m==1) return 1L;
 else return fibonacci(m-1)+fibonacci(m-2);
}
```

Ejercicio 4: *Modificar el programa anterior de cálculo de los números de Fibonacci para verificar que el cociente de dos números consecutivos se aproxima a la media áurea.*

2.9. Sobrecarga de las funciones

Se entiende como sobrecarga de funciones la definición de dos funciones distintas con el mismo nombre y que se diferencian en el tipo de sus argumentos. Esta es una posibilidad del C++ que tampoco existe en otros lenguajes, en particular C y FORTRAN77. Como una aplicación, consideremos el caso de la función factorial. Si bien su definición es sencilla para números enteros, para números reales es una función trascendente que hay que calcular mediante aproximaciones polinomiales. Desearíamos por lo tanto tener una función factorial para algumentos enteros y otra para reales, con el mismo nombre. En el programa `factorial.cc` se programan estas dos funciones. El cálculo del factorial de un número real se realiza mediante la aproximación de Stirling en tercer orden. Sólo es válida para valores grandes del argumento, pero nos vale como ejemplo. El argumento de entrada se ha tomado de tipo *double,* y se verifica posteriormente si es entero

o no. Se ha utilizado el tipo *long double* para el valor de la función factorial, para extender en la medida de lo posible el dominio de validez, aunque necesariamente esta función da *overflow* muy rápidamente. En la práctica, lo que se hace es diseñar una función que calcule el logaritmo del factorial, que es un número ordinario, como se hace en el programa `factorial.cc`:

```
#include <iostream>
#include <math.h>
#include <iomanip>
using namespace std;
const double    pi = 4 * atan(1.);
long double
fact(int m)
{
  long double fac=m;
  for(int i=1;i<m;i++) fac=fac*(m-i);
  return fac;
}
long double
fact(double x)
{
  //Calculo de x! por la formula de Stirling en tercer orden
  return (long double) sqrt(2*pi*x)*pow(x/exp(1.),x)*
  (1.+1./(12*x)+1./(288*x*x)-139./(51840*x*x*x)) ;
}

int
main()
{
    double          x;
    cout << "Entrar un numero" << endl;
    cin >> x;
    int k= int(x);
    if((k-x)==0)  cout << "El factorial de  " << k << " es "<<setw(20)
     <<setprecision(18)<<fact(k) << endl;
    else   cout << "El factorial de   " << x << " es "
    <<setw(20)<<setprecision(18)<<fact(x)<< endl;

    return 0;
}
```

2.10. Funciones de patrones

Muchas veces deseamos escribir un programa que sirva para argumentos de varios tipos como *float*, *double* o más generales, como números complejos, matrices o clases. Una forma de realizarlo es con funciones cuyo tipo es un parámetro. La función se programa como un patrón (template) en el que puede cambiar el parámetro. El siguiente ejemplo, se programa una función patrón que calcula un polinomio mediante el algoritmo de Horner:

```
template<class T>T pol(const Vector<T>& a, const T& x)
{int n=a.size();T tmp=a(n);
for (int i=n-1;i>=1;i--) tmp=tmp*x + a(i);
return tmp;
```

La primera sentencia indica que `pol` es una función de tipo T, donde T es un parámetro, es decir puede ser cualquier tipo (`int, double, complex`, u otros definidos más adelate como `Matrix`). `template<class T>`indica que lo que sique es una función de un parámetro llamado `T` y que por lo tanto esta función es un patrón (*template*) que depende de ese parámetro. Esta función funciona para cualquier tipo de vectores para los que la suma y el producto (*) estén definidos. Como un ejemplo más sencillo, el programa *cubotemplate.cc* a continuación contiene la programación de la función *cubo* como un template.

```
#include <iostream>
#include <cmath>
using namespace std;
template<class T> T cubo(T);
main(){
double x;
cout<<"Entrar un numero real"<<endl;
cin>>x;
double cx=cubo(x);
cout<<"El cubo de "<<x<<" es "<<cx<<endl;

int y;
cout<<"Entrar un numero entero"<<endl;
cin>>y;
int cy=cubo(y);
cout<<"El cubo de "<<y<<" es "<<cy<<endl;

char z;
cout<<"Entrar una letra"<<endl;
cin>>z;
char cz=cubo(z);
cout<<"El cubo de "<<z<<" es "<<cz<<endl;
//cout<< "int("<<z<<")= "<<int(z) <<
//" int("<<cz<<")= "<<int(cz)<<endl;
```

```
return 0;
}

template<class T> T cubo(T y){
return y*y*y;
}
```

Compilad y ejecutad este programa. Vemos que la función cubo se puede llamar con diversos tipos de argumentos. La última llamada a cubo, calcula el cubo de un carácter, lo cual no tiene sentido para nosotros, como podéis comprobar descomentando las últimas dos líneas antes del return, pero como la multiplicación de carácteres es legal en C++, la operación está definida.

Las funciones de patrones se utilizan muy frecuentemente en C++, y de hecho, la posibilidad de utilizarlas es una de las mayores ventajas del C++, que contribuye a hacerlo más rápido que el FORTRAN en algunas circunstancias. Sin embargo, para el óptimo aprovechamiento de las cualidades de las funciones de patrones, éstas no se deben de incluir en el cuerpo del programa que se compila, como hemos hecho en el ejemplo anterior, sino como cabeceras que se incluyen con el preprocesador (#). En el ejemplo anterior, pódiamos haber incluído el template en una cabecera llamada `mistemplates.h` por ejemplo, e incluir al principio del programa la línea

```
#include "mistemplates.h"
```

Cuando se utilizan las *funciones de templates* como cabeceras, se produce una optimización del programa en todas las llamadas a estas funciones, es como si estuvieran declaradas como *inline*. De esta forma se evitan todas las copias de argumentos a variables temporales locales y el tiempo de procesador que necesita la llamada a una función. Sin embargo, las funciones de patrones pueden ser mucho más complejas que las funciones usualmente aceptadas como *inline* por el compilador. Esta es la razón por la cual el uso de funciones de patrones aumenta la velocidad de ejecución de los programas en el caso de grandes programas que consisten de muchos ficheros fuente independientes, muchos de los cuales llaman a una misma función (como por ejemplo la función cubo o la multiplicación de matrices).

En el programa anterior, el tipo de la función cubo se deduce del argumento de la función. A veces ésto no es posible. En este caso hay que especificar el tipo entre paréntesis angulares. Por ejemplo, lo más correcto es invocar

```
int cy=cubo<int>(y);
```

aunque no es necesario en este caso, puesto que el tipo de la función `cubo` es el mismo que el de y, y el compilador lo deduce sin necesidad de especificarlo. Sin embargo, cuando introduzcamos el tipo `complex,` como un número complejo puede estar compuesto por pares de números de distintos tipos, deberemos especificar el tipo, invocando por ejemplo `complex<double>`.

2.11. Ejercicios a presentar como memoria

1. Escribid un programa que calcule el producto vectorial de dos vectores de dimension 3. Imprimir el resultado en pantalla y en fichero, junto con los vectores iniciales, que deben leerse desde un fichero. Si los vectores tienen dimensión distinta de 3 debe de generar un mensaje de error y pararse el programa.

2. Escribir un programa no recursivo para el cálculo de los números de Fibonacci y sus cocientes Si lo ejecutais para n=50 ¿Que diferencias observais con el programa recursivo? ¿Que conclusiones extraéis? Declarad el cociente *long double* y los números *long int*. Utilizad las funciones de formato de números para imprimir al menos 25 decimales del cociente. Comparad con el valor exacto de la media áurea.

3. Escribir un programa que calcule la función factorial de forma recursiva, compuesto por el programa principal y una función que llamaréis fact.

4. Escribid un programa que lea una serie de valores desde un fichero como un vector (la dimension del vector debe leerse en primer lugar) y calcule el valor medio y desviación típica de la serie de números compuesta por los elementos del vector.

5. Escribid un programa que calcule el producto de una matriz por un vector, ambos de tipo *double*. Se deben leer los datos desde fichero y escribir el resultado en fichero y pantalla.

6. Escribid una función de patrones para calcular el producto escalar de dos vectores. Debe de parecerse lo más posible a la dada en este capítulo. El programa principal debe de calcular el producto escalar de vectores de tipo int, float, double y char, cuya dimensión y valores son leídos desde fichero. Compilad y ejecutad vuestro programa.

7. Incluid la función de producto escalar en una cabecera independiente, incluída en el programa principal. Compilad y ejecutad el programa.