Capítulo 1

Introducción al C++

1.1. Mi primer programa

Vamos a realizar las prácticas utilizando el compilador g++, que está disponible para los sistemas operativos más utilizados: Linux, Windows y MacOSX así como para todos los sistemas UNIX.

Las principales aplicaciones que son necesarias para la creación de los ficheros fuente de C++ son una terminal alfanumérica (una pantalla en la que se puede escribir con el teclado comandos y texto) y un editor de texto ASCII. La particularidad de los editores de texto ASCII es que sólo contienen los caracteres que introducimos sin incluir ninguna información acerca de la fuente, color y otras características. La información de ficheros ASCII se puede transportar fácilmente de un sistema a otro sin que sufra alteraciones (en el peor de los casos los caracteres de cambio de línea puede que no se reconozcan, pero esto se puede arreglar fácilmente). Los ficheros ASCII, debido a su simplicidad pueden sobrevivir sin dificultad al continuo cambio de sistemas operativos y arquitecturas de procesadores. Utilizaremos el editor EMACS, que es rápido y potente y está disponible para los principales sistemas operativos. Otros editores ASCII ampliamente disponibles son el *notepad* de *Windows*, y *pico*, *joe*, *vim* en *Linux* y otros sistemas *Unix*. Hay editores que pueden conmutar entre la opción ASCII y otros formatos que contienen información adicional, como RichText. Si se utiliza uno de estos editores hay que estar seguros de trabajar en modo ASCII. Si el editor que utilizáis no es ASCII, obtendréis un gran número de errores en la primera compilación, pues los comandos no serán reconocidos por el compilador.

Vamos a comenzar la primera sesión con EMACS. Abrid una terminal alfanumérica en vuestro sistema. Teclead en la línea de comandos el comando

```
emacs miprog.cc&
```

Con este comando abrís un nuevo fichero, que se llama *miprog.cc* y que por ahora está vacío. El símbolo & (ampersand) sirve para que EMACS no bloquee la línea de comandos. Si no ponéis este símbolo no pasa nada, salvo que si necesitáis enviar otro comando deberéis abrir una nueva ventana con una terminal alfanumérica o cerrar EMACS para enviar el comando y abrirlo

posteriormente.. Cuando un comando se ejecuta seguido del símbolo & se dice que se ejecuta en segundo plano (o si preferís el término inglés, en el background).

Escribid en la ventana de EMACS, donde aparece el contenido del fichero miprog.cc (que por ahora debe de estar vacío), que consta de las siguientes líneas

```
#include <iostream>
// Mi primer programa
using namespace std;
int main(){
cout<<"!Hola, que tal¡"<<endl;
}
```

Una vez que halláis hecho esto, salvad el fichero y cerrad Emacs. Introducid el siguiente comando en la línea de comandos de la ventana alfanumérica

g++ -o hola miprog.cc

Si todo va bien, se creará un fichero ejecutable que se llama hola. El carácter de ejecutable lo podéis verificar por el color en que aparece su nombre cuando listáis el contenido del directorio (mediante el comando ls o ls –color si ls a secas no produce un listado con colores diferentes para los distintos tipos de ficheros). Lo ejecutáis mediante el comando

./hola

lo que produce el siguiente resultado en la terminal alfanumérica:

```
!Hola, que tal¡
```

Si algo no funciona bien, sin duda es un error de escritura. Prestad atención a los mensajes de error del compilador y verificad en detalle que todos los símbolos y los espacios coinciden con los del texto. En algunas terminales alfanuméricas los símbolos de exclamación pueden dar problemas. Las opciones por defecto de las terminales no suelen tolerar acentos y caracteres no utilizados en inglés y es necesario cambiar la configuración. En ese caso podéis suprimir estos caracteres en este primer ejercicio.

Vamos a explicar un poco lo que hemos hecho. La línea

`#include <iostream>`

significa que hemos incluido la librería de entrada y salida del C++. Sin esta línea no podríamos utilizar la función *cout* para escribir texto en la salida del programa. Notemos la presencia de una almohadilla # al principio de la línea. Eso significa que estas sentencias están tratadas por un programa llamado *preprocesador* antes de por el compilador. Notemos también que las lineas del preprocesador no acaban en ; como las sentencias de C/C++.

La línea

```
// Mi primer programa
```

es un comentario. Los caracteres // tienen como consecuencia que esa línea sea ignorada por el compilador. Nos sirve para documentar el programa. Debemos de incluir todas las líneas de comentario necesarias para que el programa esté claro y bien documentado (O sea, que podáis saber una semana después lo que habéis hecho).

La línea

```
using namespace std;
```

nos da acceso a la librería estándar, que se encuentra en una zona o *namespace* de nombre *std*. Esta sentencia, al igual que todas las que trata el compilador ,deben acabar en ; .

Las líneas

```
int main(){
cout<<"!Hola, que tal¡"<<endl;
}
```

definen la única función del este programa, que es la función *main*. `int main()` define la función `main` como de tipo entero. En versiones antiguas del C++ se podía definir la función `main` como `void` cuando no devolvía ningún argumento, pero ahora es obligatorio definirla como `int`. Como *main* no tiene argumentos en nuestro caso, aparecen los dos paréntesis vacíos (). Las líneas que dan la definición de la función están comprendida entre llaves {}, y en nuestro caso consiste únicamente en la escritura del texto `!Hola, que tal¡` en la salida estándar mediante la función `cout`. Esta función utiliza el comando `<<` para redireccionar texto a la salida estándar, que en nuestro caso es la pantalla. El texto entre dobles comillas se escribe literalmente en la salida estándar. El comando `endl` produce un cambio de línea.

Todas las sentencias de una función acaban en punto y coma (;). Uno de los errores más frecuentes es olvidar un punto y coma, lo que produce en general un gran número de errores del compilador. De hecho se pueden escribir varias sentencias en la misma línea, separándolas por ; , pero no se recomienda hacerlo por claridad, salvo en sentencias muy cortas y relacionadas. Lo usual es no hacerlo nunca. Cada sentencia se escribe en su línea.

Probad a suprimir un ; en el programa anterior y compilar de nuevo para ver el efecto que produce el olvido del ;.

La línea `return 0;` devuelve el valor 0 para la función `main`. Es obligatoria ya que main es de tipo *int*. En el caso de funciones de tipo void se puede suprimir.

1.2. Un poco de matemáticas

1.2.1. Variables y tipos

El primer concepto importante es el de variable. Una variable es un símbolo que puede tomar valores. El símbolo lo representamos con caracteres alfanuméricos, siendo el primero alfabético. En C++ se diferencia entre mayúsculas y minúsculas. Normalmente elegimos como símbolo un nombre que nos de información sobre la variable. Esta es una buena práctica de programación y ayuda a la legibilidad de los programas. Las variables tienen un tipo. Por ahora vamos a utilizar el tipo entero (*int)* y los tipos reales en precisión simple (*float*) y doble (*double*). Hay varios

tipos más que describiremos posteriormente. La sentencia en la que se especifica el tipo de una variable se denomina declaración de dicha variable. Una variable debe ser declarada antes de usarla. En caso contrario el compilador da un mensaje de error. La declaración se puede realizar en cualquier punto del programa (esta es una diferencia fundamental entre C y C++ pues en C las variables hay que declararlas al principio del programa). Muchos programadores declaran las variables justo antes de usarlas mientras que otros prefieren declarar todas las variables al principio del programa al estilo del C. Se puede asignar un valor a una variable en el momento de declararla. Esta asignación se denomina inicialización de la variable. A continuación se dan algunos ejemplos de declaración e inicialización:

```
double x; //declaracion de x del tipo double
int m, float z, double y; //declaracion de m como int,
//z como float e y como double
int k=4,double x=3.02, float yy=5.5; //declaración e inicialización
//de k, x e yy
```

1.2.2. Aritmética

Un concepto fundamental es el significado del símbolo = llamado operador de asignación. Este símbolo significa que asignamos a la variable a su izquierda el valor del término de la derecha.

```
a=6; // le damos a la variable a el valor 6
x=y; //le damos a la variable x el valor de la variable y
```

Los operadores aritméticos fundamentales son +,-,*, /, representan la suma, resta, multiplicación y división, respectivamente. La multiplicación y división se realizan antes que la suma y la resta, es decir tienen precedencia. Las operaciones de las expresiones incluídas entre paréntesis se realizan en primer lugar. Mediante la introducción de paréntesis se logra que las operaciones que se realicen sean las que nosotros realmente deseamos. En caso de duda, la regla es introducir un paréntesis. El uso de un estilo de programación correcto ayuda a introducir los paréntesis correctamente, sin dejarse ninguno ni a derecha ni a izquierda, lo que es una de las causas más frecuentes de errores tanto en compilación como en ejecución (estos últimos cuando se restaura el número de parentesis en posiciones incorrectas, ya que si hay más paréntesis a un lado que a otro el programa no compila).

En C++ existen las llamadas operaciones compuestas: +=, -=. *= y /=. Su significado es el realizar la primera operación entre los miembros de la derecha y la izquierda y asignarle este valor al de la izquierda.

```
a+=b; //a=a+b
a-=b; //a=a-b
a*=b; // a=a*b
a/=b; // a=a/b
```

También existen y se utilizan frecuentemente los operadores de incremento

```
i++; // i=i+1 después de utilizar i
i--; //i=i-1 después de utilizar i
++i; // i=i+1 antes de utilizar i
```

```
--i; //i=i-1 antes de utilizar i
```

Hay que tener precaución a la hora de utilizar dichos operadores, pues es muy fácil producir con ellos sentencias que son sintácticamente correctas, es decir compilan sin problemas, pero cuyo resultado no está definido según el ISO del C++, por lo que pueden dar valores diferentes en ordenadores diferentes. Un ejemplo es

```
i=5;
k=i-- + ++i;
```

El resultado depende de si la suma se calcula evaluando primero el primer o el segundo sumando. Si se evalúa en primer lugar el segundo término el resultado es 12 (k=6+(i=6), i=5) mientras que si se evalúa antes el primer término el resultado es 10 (k=5+ (i=4, i=5)). Cualquiera de los dos resultados son posibles.

1.2.3. Funciones matemáticas

Todas las operaciones matemáticas distintas de las operaciones aritméticas se realizan con las funciones de la librería matemática, heredada del C, que están declaradas en la cabecera *cmath o math.h* (Ambas son prácticamente idénticas *s*alvo escasas diferencias).

Vamos a realizar ahora un programa ligeramente más complicado que el anterior, que consistirá en calcular ambos lados de algunas igualdades bien conocidas. Examinad el fichero mat0.cc

```
sin(pi)=1.22465e-16 cos(pi)=-1 sin(pi)^2+cospi^2=1
log[exp(3.)*exp(4.)]= 7
```

Si examinamos el fichero vemos que se incluye una nueva cabecera

```
#include <cmath>.
```

Esta cabecera tiene como finalidad el incluir la librería matemática del C. En las siguientes líneas definimos las variables `pi, s, c, h` y `b` como del tipo *double*. Esto quiere decir que se utilizan dos palabras de ordenador para cada una de estas variables, lo que tiene como resultado el que se expresen con unas 16 cifras decimales exactas. En las siguientes líneas de la función *main* se utilizan las funciones arcotangente (*atan*), seno (*sin*), coseno (*cos*), potencia (*pow*), exponencial (*exp*) y logaritmo neperiano (*log*). El programa no hace otra cosa que verificar las igualdades $\sin^2(\pi)+\cos^2(\pi)=1$, y $\ln(\exp(3)*\exp(4))=7$. Si lo compilamos con

```
g++ -o mat0 mat0.cc
```

y ejecutamos el programa resultante *mat0* mediante el comando

```
./mat0
```

obtenemos el siguiente resultado

```
sin(pi)=1.22465e-16 cos(pi)=-1 sin(pi)^2+cospi^2=1
log[exp(3.)*exp(4.)]= 7
```

Vemos que el seno de π no sale 0 sino 1.22465e-16. Esto es debido a que los números se representan con una precisión limitada (y en sistema binario), precisión que corresponde a dos

Tabla 1.1: Principales funciones de la librería matemática

Función	Tipo argumento	Tipo función	Significado matemático	Dominio argumentos
abs(x)	double	double	|x|	$-\infty < x < \infty$
fabs(x)	double	double	|x|	$-\infty < x < \infty$
sqrt(x)	double	double	$\sqrt{x}$	$x \geq 0$
pow(x,y)	double,double	double	x^y	$x \geq 0; x = 0, y > 0$
pow(x,i)	double,int	double	x^i	$-\infty < x, i < \infty$
exp(x)	double	double	e^x	$-\infty < x < \infty$
log(x)	double	double	logaritmo natural	$x > 0$
log10(x)	double	double	logaritmo decimal	$x > 0$
sin(x)	double	double	seno	$-\infty < x < \infty$
cos(x)	double	double	coseno	$-\infty < x < \infty$
tan(x)	double	double	tangente	$-\infty < x < \infty$
asin(x)	double	double	arcoseno	$-1 \leq x \leq 1$
acos(x)	double	double	arcocoseno	$-1 \leq x \leq 1$
atan(x)	double	double	arcotangente	$-\infty < x < \infty$
atan2(x,y)	double,double	double	arcotangente(x/y)	$-\infty < x, y < \infty$
sinh(x)	double	double	seno hiperbólico	$-\infty < x < \infty$
cosh(x)	double	double	coseno hiperbólico	$-\infty < x < \infty$
tanh(x)	double	double	tangente hiperbólica	$-\infty < x < \infty$
asinh(x)	double	double	arcoseno hiperbólico	$-\infty < x < \infty$
acosh(x)	double	double	arcocoseno hiperbólico	$-\infty < x < \infty$
atanh(x)	double	double	arcotangente hiperbólica	$-\infty < x < \infty$

palabras de ordenador para los números de tipo `double`. Los números reales se pueden definir del tipo *float* (una palabra de ordenador por número), *double* (dos palabras de ordenador) y *long double* (no tiene una definición en el documento ISO del C++, pero una definición usual en las versiones de Linux y Unix más recientes es la de cuatro palabras de ordenador, de hecho dos double con exponentes diferentes, ajustados para dar un mayor número de cifras significativas de la mantisa).

Ejercicio: Repetir la compilación y ejecución de mat0, cambiando todos los *double* a *float* primero y a *long double* posteriormente. Comentad los resultados obtenidos.

En la tabla 1 se dan las funciones matemáticas más usuales del C++. Un listado completo se encuentra en cualquiera de los manuales de referencias de la bibliografía.

1.3. Leyendo y escribiendo en ficheros

Con las funciones *cin* y *cout* se lee y escribe desde el teclado y en la pantalla del ordenador, respectivamente. En muchas circunstancias es más cómodo e incluso necesario leer desde

ficheros y escribir la salida del programa en ficheros. Para realizar esto es necesario introducir la cabecera *fstream*. Esta cabecera permite declarar ficheros como unidades de entrada y salida. Veamos el siguiente ejemplo:

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
double          pi = 4 * atan(1.);
ifstream        fin("datos.dat");
ofstream        fout("resultados.dat");
double          grados;
fin >> grados;
double          radianes = grados * pi / 180.;
cout << grados << " grados corresponden a " << radianes << " radianes" << endl;
fout << "grados" << "\t" << "radianes" << endl;
fout << grados << "\t" << radianes << endl;
fout.close();
fin.close();
return 0;
}
```

Notemos en primer lugar la presencia de la cabera

```
#include <fstream>
```

que permite el definir ficheros como unidades de entrada y salida. Las líneas

```
ifstream fin("datos.dat");
ofstream fout("resultados.dat");
```

definen los ficheros *datos.dat* y *resultados.dat* como unidades de entrada y salida que hemos denominado *fin* y *fout* por analogía a *cin* y *cou*t, pero que podíamos haber llamado de cualquier otra forma, como *salida* y *entrada* por ejemplo. Recompilad el programa *fstream.cc* cambiando los nombres de las unidades de salida y entrada, para verificar lo que acabamos de decir. Notemos que los nombres de los ficheros están entre dobles comillas " ". Estos nombres forman parte de un tipo de variables denominado *string,* distinto del tipo *char;* Las variables de este último tipo están encerrados entre comillas sencillas ' ' . Discutiremos más adelante las diferencias entre los tipos *string* y *char.*

Notemos también la presencia de las líneas

```
fout.close();
fin.close();
```

Estas líneas sirven para cerrar los ficheros una vez utilizados y evitar cambios en los mismos cuando se cierra el programa (Aunque si no se cierran los ficheros no suele pasar nada, es mejor ponerlas para asegurar la integridad de los resultados).

1.4. Repitiendo una y otra vez

En el programa anterior hemos convertido de grados a radianes un sólo ángulo. Sería mas interesante hacer una tabla, con la conversión de todos los valores enteros entre 0 y 180 grados. Para ello hay que utilizar alguna de las sentencias del C++ que nos permiten hacer iteraciones. En el siguiente programa esto se realiza mediante la sentencia *for:*

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
double          pi = 4 * atan(1.);
ofstream        fout("tabla.dat");
double          grados, radianes;
fout << "grados" << "\t" << "radianes" << endl;
for (int i = 0; i < 180; i++) {
grados = i + 1;
radianes = grados * pi / 180.;
cout << grados << " grados corresponden a " << radianes << " radianes" << endl;
fout << grados << "\t" << radianes << endl;
}
fout.close();
return 0;
}
```

Notemos en primer lugar la presencia del bloque

```
for(int i=0;i<180;i++){
  grados=i+1;
  radianes=grados*pi/180.;
  cout<<grados<<" grados corresponden a "<<radianes<<"radianes"<<endl;
  fout<<grados<<"\t"<<radianes<<endl;
  }
```

Este bloque es de la forma:

```
for(int i=0;i<180;i++) {sentencia compuesta}
```

En este caso todo lo comprendido entre las dos llaves {} constituye una sentencia compuesta del bloque *for*, que se repite 180 veces en este caso. La repetición viene controlada por la variable *i* que es del tipo *int,* mediante el que se representan variables enteras. La sentencia *for* encierra tres campos entre paréntesis ordinarios. El primero es una variable que se toma usualmente entera. El segundo es una condición, que cuando no se cumple finaliza el bucle for. La tercera es una acción que se repite en cada iteración. i++ es una construcción del C++ que significa que transformamos i en i+1 después de ejecutar todas las sentencias comprendidas en el bucle for. También es posible

poner ++i que significa que pasamos i a i+1 antes de ejecutar las sentencias del bucle for. Las construcciones i– y –i también son válidas, y significan disminución en una unidad. Notemos también que hemos comenzado en i=0 y puesto la condición de i<180. Esto es debido a que en C++ los vectores comienzan en el índice 0, y por lo tanto el último índice de un vector de dimensión 180 es 179.

La sentencia for puede ir seguida de una sentencia simple, en cuyo caso no hacen falta las llaves:

```
for(int i=0;i<n;i++) sentencia;
```

Cuando va seguida de varias sentencias como en el caso anterior, hay que agruparlas con las llaves {}; Esta agrupación se denomina una sentencia compuesta. Cada una de las sentencias individuales debe de ir seguida de un ; .

El tipo del índice del for no tiene porqué ser entero. Una sentencia for de la forma

```
for(double i=1;i<=180;i++){
  grados=i;
  radianes=grados*pi/180.;
  cout<<grados<<" grados corresponden a "<<radianes<<"radianes"<<endl;
  fout<<grados<<"\t"<<radianes<<endl;
  }
```

es perfectamente válida, en el sentido que no da error de compilación. Sin embargo, debido a los errores de redondeo, nunca podemos estar seguros si la última condición se cumple en la iteración 180, ya que `i` podría resultar 180.000000000000001 debido a la precisión finita de los números reales y al redondeo que se realiza para que el valor digital (expresado con el número correspondiente de dígitos) sea lo más próximo al valor real. El que funcione en vuestro ordenador (en el mío de hecho funciona), no significa que funcione en cualquier ordenador.

Ejercicio 2: *Modificar lo mínimo posible el programa* `for.cc` *para que se utilice* `++i, --i,` *y* `i--` *en vez d*e `i++` .

Otra forma de realizar iteraciones es mediante la sentencia *while*. El bucle anterior se podría escribir utilizando esta sentencia como

```
int i=0;
while(i<180){
  grados=i+1;
  radianes=grados*pi/180.;
  cout<<grados<<" grados corresponden a "<<radianes<<"radianes"<<endl;
  fout<<grados<<"\t"<<radianes<<endl;
 i++;
  }
```

El bucle *while* lleva entre paréntesis ordinarios una condición, que cuando deja de cumplirse produce la interrupción del bucle. Cuando se desea iterar un número N bien definido de veces se utiliza siempre la sentencia *for*. El número de iteraciones está bien definido por el índice y no hay riesgo de hacer una de más o de menos. Cuando se desea iterar mientras que una condición relativamente compleja se cumple, el bucle *while* es mucho más conveniente. Hay todavía una versión del bucle *while* (la sentencia *while do*) de la que solo diremos aquí que su empleo no es

recomendable. Consideremos como aplicación del *bucle while* el cálculo de e mediante serie de potencias

$$e = \sum_{i=0}^{\infty} \frac{1}{i!}$$

Podemos sumar 1000 o un millón de términos para garantizar una determinada precisión, pero lo que realmente nos interesa es parar la suma cuando el término correspondiente sea menor que una determinada cantidad ε. El siguiente programa realiza este cálculo con una precisión $\varepsilon = 10^{-6}$:

```
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;
int main()
{
double          e = 0., t = 1., epsilon = 1.e-6;
int             i = 1;
while (t >= epsilon) {
e = e + t;
t = t / i;
i++;
}
cout << "Convergencia con i=" << i << " terminos" << endl;
cout << "Termino final= " << t << endl;
cout << "e=" << e << endl;
return 0;
}
```

Ejercicio 3: *Intentad ejecutar el programa co*n $\varepsilon = 10^{-12}$ *¿Cuál es el problema*?

1.5. Formato de datos

Si habéis realizado el último ejercicio, os habréis encontrado con el problema de que el número significativo de cifras con que se sacan los resultados son siempre 6 cifras significativas. Esto supone un inconveniente si se realizan cálculos con 12 cifras significativas. Para controlar la precisión de salida de los números existe otra cabecera de nombre *iomanip*. En la sentencia *cout* el comando *setw(n)* define como *n* la anchura el campo del dígito que se escribe inmediatamente después. Sólo vale una vez, es decir hay que escribir tantos *setw* como variables se escriben. El comando *setprecision(m)* define como *m* el número de cifras decimales que se imprime. Este comando es válido hasta que se incluya otro comando *setprecision* que cambie el valor previo. Si queremos imprimir *e* con 20 cifras decimales en el programa anterior haremos los siguientes cambios:

Tabla 1.2: Actuación de los operadores lógicos

a	b	$a\&\&b$	$a\|\|b$	$!a$
V	V	V	V	F
V	F	F	V	F
F	V	F	V	V
F	F	F	F	V

```
#include <iomanip>
⋮
cout<<setw(22)<<setprecision(20)<<"e="<<e<<endl;
```

Ejercicio 4: *Escribir el programa de conversión de grados a radianes de forma que se escriban los valores inicial y convertido de la tabla con 12 cifras decimales.*

1.6. Expresiones lógicas

Una expresión lógica es esencialmente una variable numérica entera que es falsa (F) cuando vale 0 y verdadera (V) en caso contrario. Las expresiones lógicas se pueden construir con los operadores de relación: `==, <, <=, >, >=, !=`. Estos signos se leen, respectivamente como, igual, mayor, mayor igual, menor, menor igual y distinto y corresponden a los símbolos matemáticos $=, <, \leq, > \geq$ y $\neq$. Si `a=5` y `b=7`, las expresiones `a==b, a>b, a>=b` son falsas y `a<b, a<=b a!=b` son verdaderas. Podemos componer expresiones lógicas con los operadores lógicos && (and), || (or) y ! (not) . La tabla de verdad de dos variables lógicas compuestas con dichos operadores se dan en la Tabla 1.2. Una variable lógica es una variable que vale 0 (falso) o 1 (verdad). Una variable entera se puede poner en lugar de una variable lógica, y se toma como falsa si vale 0 y verdadera en caso contrario.

1.7. Tomando decisiones

Un requerimiento indispensable en programación es el poder tomar decisiones. La sentencia fundamental para la toma de decisiones en C/C++ es la sentencia *if*. La estructura de esta sentencia es

if(expresión lógica) sentencia;

o

if(expresión lógica) {sentencia compuesta}

Por ejemplo, si en el programa anterior queremos introducir una sentencia para que interrumpa el bucle *while* si se sobrepasan las 1000 iteraciones, incluiríamos

```
if (i>1000) break;
```

La sentencia *break* lo que hace es justamente salir del bucle.

Ejercicio 5: *Comprobad el funcionamiento de break.*

La línea

```
if(3) sentencia;
```

dará como resultado que siempre se ejecute la sentencia (3 es siempre verdad).

Un error frecuente es confundir el signo igual (=) de asignación con del operador de relación de igualdad (==). La línea

```
if (b=10) sentencia;
```

lo que hace es darle a *b* el valor 10 y por lo tanto *b* es verdadero, produciendo la ejecución de *sentencia* siempre.

La sentencia se puede escribir de forma que se tomen varias decisiones alternativas

```
if(a) sentencia1;
else if(b) sentencia2;
else sentencia3;
```

La sentencia *else if* puede aparecer varias veces. Hay que tener cuidado en que todos los casos sean mutuamente exclusivos. El siguiente programa lee un número por el teclado (entrada estándar) e imprime su valor absoluto por pantalla (salida estándar):

```
#include <iostream>
using namespace std;
int main()
{
float           a, absa;
cout << "Entrar un numero" << endl;
cin >> a;
if (a > 0)
absa = a;
else if (a < 0)
absa = -a;
else
absa = 0;
cout << "El valor absoluto de " << a << " es " << absa << endl;
return 0;
}
```

1.8. Representación gráfica

Los lenguajes C y C++ carecen de posibilidades de representación gráfica en su definición estándar. Sin embargo existe un gran número de librerías gráficas tanto públicas como comerciales que permiten la salida gráfica tanto en C como C++. En este curso nos limitaremos a utilizar el programa público *gnuplot* como utilidad de representación gráfica, realizando el proceso en dos etapas_: nuestro programa imprime los resultados en ficheros de datos y posteriormente leemos esos ficheros de datos con *gnuplot* para representarlos gráficamente. *gnuplot* puede exportar las gráficas en un gran número de formatos gráficos, que podéis utilizar para confeccionar posteriormente la memoria de la práctica en el sistema operativo de vuestra elección, pues estos formatos

son universales. *gnuplot* se puede ejecutar directamente desde vuestro programa en C++ a través *una cabecera,* como veremos en el siguiente tema. En la última sección de esta práctica presento una breve descripción de *gnuplot*.

1.9. Algo más sobre tipos

Los tipos de enteros admitidos en C++ son los siguientes*: short int, int, long int, unsigned int*, mientras que los tipos de reales son: *float, double* y l*ong double*.

Tipos adicionales son *boolean* para las variables lógicas y *char* para cadenas de caracteres. El número de bytes de una variable de un tipo dado se puede conocer con la función *sizeof* descrita más adelante.

Muchas veces podemos desear escribir un programa con un tipo general, para decidir en una aplicación particular el tipo nativo concreto al que debe corresponder. Esto se puede conseguir con la sentencia *typedef* que sirve para declarar un tipo como sinónimo de otro. Por ejemplo, podemos declarar todos los reales en nuestro programa como *Real* e introducir al principio del programa la sentencia

```
typedef Real double
```

que nos dice que Real es sinónimo de *double*. Si en un momento determinado queremos utilizar reales del tipo *float* en vez de *double* sólo tenemos que cambiar una sentencia en todo el programa en vez de rectificar todas las apariciones de *double*.

1.10. Caracteres y cadenas de caracteres

La sentencia

```
char a;
```

indica que a es una variable que toma caracteres como valores. Los valores hay que expresarlos entre comillas. Por ejemplo

```
char a='B';
```

inicializa la variable `a` al valor `B`. En C++ existen cadenas de caracteres que pertenecen a un tipo llamado *string* que en realidad es un tipo complejo llamado *clase* estudiado en el capítulo 4. Las cadenas de caracteres se expresan entre dobles comillas

```
string a="!Hola, que tal¡"
```

indica que *a* es una variable del tipo *string* que toma el valor `!Hola, que tal!`. Para poder utilizar el tipo *string* hace falta incluir la cabecera <*string*>. Las cadenas de caracteres las utilizamos implícitamente al imprimir con el operador *cout*.

Existe una diferencia fundamental entre los vectores de caracteres del C y los strings del C++.

Dejamos para el capítulo 4 una discusión más detallada del tipo (clase) *string*.

1.11. Tamaño de variables: función sizeof

Se puede conocer el número de bytes que requiere cada tipo de variable mediante la función *sizeof*. El programa *sizeof.cc* produce como salida el tamaño en bytes de los tipos nativos:

```
#include <iostream>
using namespace std;
int main()
{
short int       si = 1;
int             i = 2;
long int        li = 3;
float           fl = 3.14;
double          db = 3.14;
long double     ld = 3.14;
char            c = 'c';
cout << "El numero de bytes de un short int es " << sizeof(si) << endl;
cout << "El numero de bytes de un int es " << sizeof(i) << endl;
cout << "El numero de bytes de un long int es " << sizeof(li) << endl;
cout << "El numero de bytes de un float es " << sizeof(fl) << endl;
cout << "El numero de bytes de un double es " << sizeof(db) << endl;
cout << "El numero de bytes de un long double es " << sizeof(ld) << endl;
cout << "El numero de bytes de un char es " << sizeof(c) << endl;
}
```

Ejercicio 6: *Compilad y ejecutad el programa sizeof.cc. Estudiad la salida.*

1.12. Overflows y underflows

Cada tipo de variable puede almacenar valores un valor máximo y un valor mínimo que dependen del tipo, de la versión del compilador y del sistema operativo. Por ejemplo, en mi ordenador (PPC con MacOSX 10.4) el tipo *int* puede representar valores enteros hasta 2147483647. Por lo tanto las siguientes sentencias

```
int a, b, c;
a=b=2147483647;
c=a+b;
```

darán para `c` el valor 2147483647. Cualquier cantidad positiva que sumemos a la variable *a* la dejará inalterada. Cuando una operación intenta que una variable sobrepase el valor del límite máximo correspondiente a su tipo decimos que se ha producido un *overflow* y cuando intenta que tome un valor inferior al mínimo (en valor absoluto) decimos que se ha producido un *underflow*. Los overflows y underflows pueden producir resultados numéricos completamente erróneos. Dependiendo de los sistemas operativos y compiladores, los overflows y underflows se tratan de forma distinta. Es frecuente que los underflows se traten poniendo el valor de la variable igual

a 0, emitiendo quizás un mensaje del sistema como advertencia. Los overflows se suelen tratar de forma más severa, produciendo siempre mensajes del sistema y quizás una parada automática del programa después de un número determinado de overflows. Las variables que dan overflow toman un valor no numérico que se representa por *inf* que representa infinito y que no se trata como un número ordinario. Se debe realizar la programación de problemas numéricos de forma que se eviten tanto overflows como underflows.

Ejercicio 7: *Escribid un programa que realice operaciones que produzcan overflows y underflows. Estudiad el resultado y el comportamiento del programa. Encontrad cual es el máximo entero que se puede representar en vuestro sistema.*

1.13. La cabecera <limits>

Aparte de los valores máximos y mínimos de los tipos nativos, hay una serie de límites que es necesario conocer. Cuando se programan problemas numéricos, conocer los límites numéricos del ordenador con el que se trabaja es esencial, pues gran parte de los problemas de funcionamiento de programas numéricos se deben a una programación que no tiene en cuenta estos límites.

En el caso de números reales existe el limite *epsilon*, que es la cantidad más pequeña en valor absoluto que sumada a 1 lo deja inalterado:

```
cout<<1.+ epsilon<<endl;
```

dará como resultado 1. El valor de *epsilon* es distinto para los tipos *float, double* y *long double*.

Otros límites importantes son los exponentes máximo y mínimo tanto binarios como decimales con los que se puede representar números de un tipo dado. También es fundamental el número máximo de dígitos de la mantisa, tanto en representación binaria como decimal. Es igualmente necesario conocer el máximo error de redondeo (es decir, si hay redondeo o truncado). El siguiente fichero indica como utilizar la librería *numeric_limits* para obtener los diferente límites para cada tipo nativo. Para poder utilizar esta librería es necesario incluir la cabecera <*limits*>mediante la directiva del preprocesador

```
#include <limits>
```

En el fichero *limits.cc,* listado a continuación, se indica como obtener cada uno de los límites relevantes para cada tipo nativo.

```
#include <iostream>
#include <limits>
using namespace std;
//namespace is discussed later

int main()
{


cout << "int type" << endl;
```

```
cout << "largest int = "
<< numeric_limits < int >::max() << endl;
cout << "smallest int = "
<< numeric_limits < int >::min() << endl;

cout << endl << "float type" << endl;

cout << "smallest float = "
<< numeric_limits < float >::min() << endl;
cout << "largest float = "
<< numeric_limits < float >::max() << endl;
cout << "min exponent in binary = "
<< numeric_limits < float >::min_exponent << endl;
cout << "min exponent in decimal = "
<< numeric_limits < float >::min_exponent10 << endl;
cout << "max exponent in binary = "
<< numeric_limits < float >::max_exponent << endl;
cout << "max exponent in decimal = "
<< numeric_limits < float >::max_exponent10 << endl;
cout << "# of binary digits in mantissa: "
<< numeric_limits < float >::digits << endl;
cout << "# of decimal digits in mantissa: "
<< numeric_limits < float >::digits10 << endl;
cout << "base of exponent in float: "
<< numeric_limits < float >::radix << endl;
cout << "infinity in float: "
<< numeric_limits < float >::infinity() << endl;
cout << "float epsilon = "
<< numeric_limits < float >::epsilon() << endl;
cout << "float rounding error = "
<< numeric_limits < float >::round_error() << endl;
cout << "float rounding style = "
<< numeric_limits < float >::round_style << endl;


cout << endl << "double type" << endl;
cout << "smallest double = "
<< numeric_limits < double >::min() << endl;
cout << "largest double = "
<< numeric_limits < double >::max() << endl;
cout << "min exponent in binary = "
<< numeric_limits < double >::min_exponent << endl;
cout << "min exponent in decimal = "
```

```
<< numeric_limits < double >::min_exponent10 << endl;
cout << "max exponent in binary = "
<< numeric_limits < double >::max_exponent << endl;
cout << "max exponent in decimal = "
<< numeric_limits < double >::max_exponent10 << endl;
cout << "# of binary digits in mantissa: "
<< numeric_limits < double >::digits << endl;
cout << "# of decimal digits in double mantissa: "
<< numeric_limits < double >::digits10 << endl;
cout << "base of exponent in  double: "
<< numeric_limits < double >::radix << endl;
cout << "infinity in  double: "
<< numeric_limits < double >::infinity() << endl;
cout << " double epsilon = "
<< numeric_limits < double >::epsilon() << endl;
cout << "double rounding error = "
<< numeric_limits < double >::round_error() << endl;
cout << "double rounding style = "
<< numeric_limits < double >::round_style << endl;

cout << endl << "long double type" << endl;
cout << "smallest long  double = "
<< numeric_limits < long double >::min() << endl;
cout << "largest long double = "
<< numeric_limits < long double >::max() << endl;
cout << "min exponent in binary = "
<< numeric_limits < long double >::min_exponent << endl;
cout << "min exponent in decimal = "
<< numeric_limits < long double >::min_exponent10 << endl;
cout << "max exponent in binary = "
<< numeric_limits < long double >::max_exponent << endl;
cout << "max exponent in decimal = "
<< numeric_limits < long double >::max_exponent10 << endl;
cout << "# of binary digits in long double mantissa: "
<< numeric_limits < long double >::digits << endl;
cout << "# of decimal digits in long double mantissa: "
<< numeric_limits < long double >::digits10 << endl;
cout << "base of exponent in long double: "
<< numeric_limits < long double >::radix << endl;
cout << "infinity in long double: "
<< numeric_limits < long double >::infinity() << endl;
cout << "long double epsilon = "
<< numeric_limits < long double >::epsilon() << endl;
```

```
cout << "long double rounding error = "
<< numeric_limits < long double >::round_error() << endl;
cout << "long double rounding style = "
<< numeric_limits < long double >::round_style << endl;

cout << endl << "char type" << endl;
cout << "number of digits in char: "
<< numeric_limits < char >::digits << endl;
cout << "char is singed or not: "
<< numeric_limits < char >::is_signed << endl;
cout << "smallest char: "
<< numeric_limits < char >::min() << endl;
cout << "biggest char: "
<< numeric_limits < char >::max() << endl;
cout << "is char an integral type: "
<< numeric_limits < char >::is_integer << endl;
        return 0;
}
```

La cabecera `<limits>` no está definida en versiones del compilador anteriores a la `g++-3.1`. Podéis conocer la versión de vuestro compilador con el comando:

```
g++ --version
```

Ejercicio 8: *Compilad y ejecutad el fichero limits.cc en vuestro sistema. Estudiad la salida producida.*

1.14. Gráficas con Gnuplot

En este curso utilizaremos *gnuplot* para representar gráficas. *gnuplot* es un software público, que está bien mantenido y que está disponible para todos los sistemas operativos. En la página web del curso de cálculo numérico doy varios cursos y manuales de gnuplot, tanto en inglés como en español. En este apartado sólo doy las nociones más elementales. Escribid *gnuplot* en la línea de comandos, o haced click en el icono de gnuplot, dependiendo del sistema operativo con el que trabajéis. El comando

```
plot sin(x)
```

dibuja la función sin(x) entre -10 y 10. El comando

```
plot [0:10] sin(x)
```

dibuja la función *sin(x)* entre 0 y 10. Si tenéis un fichero de datos llamado *misdatos.dat,* el comando

```
plot 'misdatos.dat'
```

dibuja los puntos. El fichero *misdatos.dat* debe de tener en la primera columna la variable *x* y en la segunda la variable *y.*

El comando

```
replot cos(x)
```

dibuja la función cos(x) en la misma ventana que está abierta, superponiendo la gráfica sobre las gráficas que ya están dibujadas. El comando *plot* dibuja la gráfica borrando lo que previamente se había dibujado. Si queremos dibujar puntos experimentales con barras de error, los ponemos en un fichero donde la primera columna es la variable independiente, la segunda la ordenada y la tercera el error. El comando

```
plot 'misdatos.dat' with errorbars
```

dibuja los puntos contenidos en el fichero *misdatos.dat* con barras de error. Las barras de error vienen dadas por la tercera columna del fichero.

Si tenéis una función calculada numéricamente en un conjunto de puntos contenidos en un fichero *mifun.dat* y queréis unirlos con una línea continua, escribís

```
plot 'mifun.dat' with lines
```

Un conjunto de comandos de *gnuplot* se puede escribir en un fichero (*mifich.gpl* por ejemplo), que se puede cargar en *gnuplot* mediante el comando

```
load 'mifich.gpl'
```

Entre los ficheros de la práctica 1 está fichero *ejemplo.gpl* como fichero de comandos. Estudiadlo con detenimiento.

Para utilizar las figuras realizadas con *gnuplot* en documentos hace falta guardarlas en un formato adecuado para ser insertado en documentos. Se utiliza el comando *set output* para redirigir la salida de *gnuplot* a un fichero. Si escribís

```
set terminal postscript
set output 'mifig.ps'
plot 'miplot.dat'
set terminal X11
```

se dibuja el contenido de *miplot.dat* en el fichero *mifig.ps*. La última línea restablece el valor de terminal a su valor por defecto, que es X11. El comando

```
set terminal
```

lista los tipos de terminales disponibles (depende de la versión de *gnuplot* y del sistema operativo). Un tipo muy utilizado es el *postscript*

```
set terminal postscript
```

que dibuja la figura en un fichero de tipo *postscript,* que es un formato gráfico vectorial similar al *pdf* que se puede introducir directamente en ficheros de LaTeX y otros editores de texto y que se puede convertir con casi todos los conversores gráficos a los formatos *jpg*, *png* y otros formatos gráficos aceptados por los programas que no reconocen el formato *postscript*. Además, el terminal *postscript* está disponible en todas las versiones de *gnuplot*. Otro terminal importante es el *fig*, que realiza la salida en un fichero nativo del programa de dibujo *xfig*. Esto permite editar la gráfica con *xfig* y salvarla en uno de los muchos formatos en que *xfig* puede realizarlo.

Ejercicio 9: *Iniciad gnuplot y verificar los diferentes puntos descritos en este apartado. Cargad 'ejemplo.gpl'.*

1.15. Ejercicios a presentar como memoria

1. Escribir un programa para calcular $\cos(x)$ con 12 cifras decimales, a partir de su serie de potencias. Escribid en fichero los resultados, de 30 a 150 grados de 30 en 30 grados. Imponer el límite de términos de la serie en 1000. El programa también debe de imprimir en pantalla los resultados junto con el número de iteraciones necesarios para calcular cada valor. Nota: Utilizad el programa `while.cc` como punto de partida donde introduciréis la cabecera $< iomanip >$ para la salida de datos formateada. Notad que la serie del coseno es alterna y solo tiene términos pares. Una forma de actualizar el término general es $t = -t * \left(\frac{x}{i}\right) * \left(\frac{x}{i+1}\right)$ con la que se evitan overflows y se tiene en cuenta el cambio de signo. El índice hay que actualizarlo como $i+=2$.

2. Realizar un programa para resolver una ecuación de segundo grado $ax^2+bx+c=0$. Se deben de considerar cada uno de los casos del discriminante $\Delta = b^2-4ac$. La precisión con la que se calcula la raíz debe ser 10^{-6}. Definir todos los reales del tipo `float`. Imprimir los resultados con seis cifras decimales.

3. Realizar un programa que diga si un punto (x,y) que se lee mediante teclado está comprendido en la zona limitada por los ejes coordenados y las líneas y=x e y=4.

4. Verificad que la suma de `N` enteros es `N(N+1)/2`. ¿Hasta que número podéis verificarla declarando los enteros como a) `int` y b) `long int`? Tendréis que averiguar cuando el ordenador comienza a hacer tonterías.

5. Escribid un programa para calcular la función factorial. ¿Hasta que número funciona con `int` y con `long int`? ¿Se os ocurre una idea para extender su funcionamiento a números más grandes?

6. Dibujad con `gnuplot` la función sin(x) entre 0 y 1 junto con un conjunto de 10 puntos igualmente espaciados de la función x*sin(x). Para ello escribid un programa que escriba en un ficheros los pares `(x, x*sin(x))`, en forma de columnas que puedan ser leídas por gnuplot.

7. Escribid un programa para dibujar una línea recta $y = ax+b$ con `gnuplot`. El programa debe pedir la pendiente y la ordenada en el origen, el intervalo (x_{min},x_{max}) en el que se desea dibujar la recta y el número de puntos en los que se desea calcular los valores de la recta. Debe igualmente preguntar el nombre del fichero en el que se deben de escribir los valores (se puede dar un nombre por defecto en caso de que no se introduzca nada, sólo Ret) y debe de escribir los valores tanto en pantalla cómo en el fichero. El fichero lo utilizaréis para dibujar las rectas con `gnuplot`.

8. Repetid el ejercicio anterior para el caso de una parábola $y = ax^2+bx+c$ y dibujad las ecuaciones del ejercicio 2.