Capítulo 1

Manual básico

En este capítulo introducimos el lenguaje de programación C++ mediante ejemplos. Cada ejemplo viene acompañado de una breve explicación para hacer énfasi en los comandos nuevos que aparecen en el programa. Estas explicaciones intentan ser, simplemente, útiles, dejando de banda cualquier consideracion de completitud o rigurosidad.

Alguien verdaderamente interesado en aprender a programar deberia completar estas explicaciones con dos cosas: un buen libro de consulta del lenguaje C++, y muchas horas de práctica jugando y modificando los programas de ejemplo o escribiendo nuevos programas.

1.1. Entrada, salida y variables

Nuestro primer programa:

Programa *hola.cc*

```
#include <iostream>

using namespace std;

int main() {
  cout << "Hola mundo!" << endl;
}
```

Salida

```
Hola mundo!
```

Sólo una de las líneas del programa nos interesa ahora: aquella que empieza con `cout`. (Más adelante explicaremos porqué las demás líneas son necesarias.) El comando `cout` sirve para escribir cosas por pantalla. Otro ejemplo:

Programa *calculos1.cc*

```
#include <iostream>

using namespace std;

int main() {
  cout << "Unos calculos:" << endl;
  cout << "2+8*(3+1)=";
  cout << "2+8*" << 3+1 << "=";
  cout << "2+" << 8*(3+1) << "=";
  cout << 2+8*(3+1) << endl;
}
```

Salida

```
Unos calculos:
2+8*(3+1)=2+8*4=2+32=34
```

Todas las instrucciones de nuestros programas acaban con un punto-y-coma (`;`). Podemos escribir varias cosas usando un único `cout`, siempre que las separemos convenientemente por símbolos . Podemos escribir cadenas de texto (también llamadas *strings*) rodeadas por comillas (), que aparecen en pantalla de modo literal; expresiones, como `8*(3+1)`, que se calculan antes de aparecer por pantalla; y `endl`, que quiere decir salto-de-línea.

Programa *calculos2.cc*

```
#include <iostream>

using namespace std;

int main() {
  int x=317;

  cout << "El numero 'x' es " << x << "." << endl;
  cout << "Su cuadrado y su cubo son " << x*x << " y "
       << x*x*x << " respectivamente." << endl;
  cout << "El polinomio 25x^3+12x^2-8x+2 evaluado a 'x' da "
       << 25*x*x*x+12*x*x-8*x+2 << endl;
}
```

Salida

```
El numero 'x' es 317.
Su cuadrado y su cubo son 100489 y 31855013 respectivamente.
El polinomio 25x^3+12x^2-8x+2 evaluado a 'x' da 797578659
```

Podemos romper líneas y añadir espacios en nuestro programa para que resulte más sencillo de leer, siempre que indiquemos el final de las instrucciones con el símbolo `;`. Los espacios sí importan dentro de las cadenas de texto, como por ejemplo en `"i "`.

En este programa, `x` es una variable. Una variable es un trozo de memoria donde el ordenador almacena datos. La línea (`int x=317;`) pide que se reserve un trozo de memoria, llamado a partir de ahora `x`, con suficiente capacidad para almacenar números enteros (`int`, del ingles *integer*), y que contiene inicialmente el número 317. De este punto en adelante, siempre que en el programa aparezca `x`, se consulta el trozo de memoria correspondiente para saber qué entero contiene `x`.

Programa *magia.cc*

```
#include <iostream>

using namespace std;

int main() {
  cout << "Elige un numero 'n'." << endl;
  int n=74;
  cout << "  [elijo el " << n << "]" << endl;

  cout << "Doblalo." << endl;
  n=2*n;
  cout << "  [me da " << n << "]" << endl;

  cout << "Sumale 6." << endl;
  n=n+6;
  cout << "  [obtengo " << n << "]" << endl;

  cout << "Dividelo entre 2 y restale 3." << endl;
```

```
  n=n/2-3;
  cout << "  [sorpresa! obtengo el numero inicial, " << n << "]." << endl;
}
```

Salida

```
Elige un numero 'n'.
  [elijo el 74]
Doblalo.
  [me da 148]
Sumale 6.
  [obtengo 154]
Dividelo entre 2 y restale 3.
  [sorpresa! obtengo el numero inicial, 74].
```

La primera vez que aparece una variable, como `n` en el programa anterior, es necesario *declararla*, como se hace en la línea `int n=74;`. Declarar una variable quiere decir elegir cuál es su tipo (en este caso, `int`, o sea, entero) y darle nombre, para poder hacer referencia a ella más adelante.

El valor almacenado en el trozo de memoria de una variable puede canviar durante la ejecución del programa. Al principio la variable `n` vale 74, pero después vale 2 multiplicado por aquello que valia `n` (o sea, $2 \cdot 74 = 148$), después vale 6 más aquello que valía `n` ($6+148 = 154$), y por último la mitad de `n` menos 3 ($154/2 - 3 = 74$).

Programa *edad.cc*

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

using namespace std;

int main() {
  cout << "Hola, como te llamas?" << endl;
  string nombre;
  cin >> nombre;

  cout << "Hola, " << nombre << ". Cuando naciste?" << endl;
  int nac;
  cin >> nac;

  cout << "Si estamos en el 2007, tu edad es " << 2007-nac-1
       << " o " << 2007-nac << "." << endl;
}
```

Salida (interactiva)

```
Hola, como te llamas?
Pepito
Hola, Pepito. Cuando naciste?
1997
Si estamos en el 2007, tu edad es 9 o 10.
```

En este programa tenemos dos variables, `nombre` y `nac`. Las variables de tipo `string` guardan cadenas de texto. Fijaos que al principio de todo hemos añadido una nueva línea, `#include <string>`. Siempre que utilicemos variables de tipo `string` es conveniente añadir esta línea. El `#include <iostream>` sirve para indicar que queremos usar, entre otras cosas, el `cout` o el `cin` (en nuestro caso, siempre los usaremos). La línea `using namespace std;` también debe escribirse siempre.

En el ejemplo se definen las variables sin inicializarlas a ningún valor, porque se leerán a continuación con `cin`. El comando `cin` funciona de modo parecido a `cout`, excepto que utilizamos $>>$ en vez de $<<$ ya que `cin` pone datos en una variable, en vez de sacarlos.

Por lo tanto, `cout <<` espera recibir *valores* para mostrarlos (por ejemplo, `2007-nac-1`, mientras que `cin >>` espera recibir *variables* donde guardar los datos leídos.

Programa *sumaAritmetica.cc*

```
#include <iostream>

using namespace std;

int main() {
  cout << "Suma de series aritmeticas "
       << "a_0+a_1+...+a_k, on a_i=a+i*c." << endl;
  cout << "Da los valores de:" << endl;

  cout << "a? ";
  double a;
  cin >> a;

  cout << "c? ";
  double c;
  cin >> c;

  cout << "k? ";
  int k;
  cin >> k;

  double resultado=a+(a+k*c); //primero + ultimo
  resultado=resultado*(k+1)/2; //(k+1) sumandos
  cout << "Resultado: " << resultado << endl;
}
```

Salida (interactiva)

```
Suma de series aritmeticas a_0+a_1+...+a_k, on a_i=a+i*c.
Da los valores de:
a? 10
c? 5
k? 3
Resultado: 70
```

El tipo `double` sirve para almacenar números reales[1]. Se pueden hacer cálculos mezclando `int` y `double`. Por ejemplo, `k` es un `int`, pero `c` es un `double`, de modo que la expresion `k*c` es como si fuera de tipo `double`.

Todo aquello que venga a continuación de los símbolos `//` en una línea es ignorado. De este modo podemos escribir comentarios que ayuden a otras personas (o a nosotros mismos) a entender el código escrito. Un programa con pocos comentarios es difícil de entender; un programa con muchos comentarios es pesado de leer.

1.2. Condicionales y bucles

Los programas que hemos hecho hasta ahora son secuenciales, en el sentido que se ejecutan todas las líneas de código en el orden establecido. Esto es claramente insuficiente, porque a menudo querremos hacer una cosa u otra en función de las entradas que recibamos.

Programa *adivina.cc*

```
#include <iostream>
```

[1] En realidad, y para la mayoría de ordenadores actuales, el tipo `double` sólo puede guardar los primeros 15 dígitos significativos de los reales, y el tipo `int` sólo puede guardar enteros entre, aproximadamente, -2000 y 2000 millones.

```
using namespace std;

int main(){
  cout << "Dime el numero x que quieres que piense." << endl;
  int x;
  cin >> x;

  cout << "Adivina el numero x que estoy pensando." << endl;
  int num;
  cin >> num;

  if (num==x) {
    cout << "Muy bien! A la primera!" << endl;
  }
  else {
    cout << "No, no es el " << num << ". Tienes un ultimo intento." << endl;
    cin >> num;
    if (num==x) {
      cout << "Bien hecho." << endl;
    }
    else {
      cout << "Has fallado. El numero era el " << x << "." << endl;
    }
  }
}
```

Salida (interactiva)

```
Dime el numero x que quieres que piense.
10
Adivina el numero x que estoy pensando.
13
No, no es el 13. Tienes un ultimo intento.
10
Bien hecho.
```

El símbolo `==` es una comparación de igualdad, y no es lo mismo que `=`. Escribir `num==x` compara los valores de `num` y `x`; escribir `num=x;` asigna el valor de `x` a la variable `num`.

La construcción `if (...) {...} else {...}`, que se llama instrucción condicional, sirve para ejecutar un código u otro en función de una condición. Si la condición se cumple se ejectan las líneas de código del primer `{...}`; si la condición no se cumple, se ejecuta el `{...}` que hay después del `else`. Por ejemplo, la línea `cout << "Has fallado ...` sólo se ejecutaría si se fallan los dos intentos de adivinar el número.

Es una buena costumbre indentar el programa a la derecha cada vez que se abren unas llaves {}. Los programas mal indentados son difíciles de leer, porque no se ve claramente si una línea se ejecuta siempre, o forma parte de uno o más `it` o `else`. En la página `http://en.wikipedia.org/wiki/Indent_style` podéis encontrar una discusión de los distintos tipos de indentado que se usan en la actualidad.

Programa *segundoGrado.cc*

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
  cout << "Introduce 'a', 'b' i 'c' de la ecuacion 'ax^2+bx+c=0'." << endl;

  double a, b, c;
  cin >> a >> b >> c;
```

```
  double disc=b*b-4*a*c;
  double epsilon=1e-10; //epsilon: numero muy pequenno

  if (disc>epsilon) {
    //discriminante positivo

    cout << "Soluciones reales: "
         << (-b+sqrt(disc))/(2*a) << ","
         << (-b-sqrt(disc))/(2*a) << endl;
  }
  else if (disc<-epsilon) {
    //discriminante negativo

    cout << "Ninguna solucion real." << endl;
  }
  else {
    //discriminante=0

    cout << "Una solucion real (doble): "
         << -b/(2*a) << endl;
  }
}
```

Salida (interactiva)

```
Introduce 'a', 'b' i 'c' de la ecuacion 'ax^2+bx+c=0'.
3
1
-2
Soluciones reales: 0.666667,-1
```

El comando `sqrt( )` calcula raíces cuadradas. Para utilitzarlo hemos de añadir `#include <cmath>`.

En nuestro ejemplo necesitamos utilizar condicionales para distinguir entre discriminantes positivos, negativos y cero. Es práctica habitual encadenar varios `else if` si queremos considerar más de dos alternativas.

Hay que ser muy prudente al hacer comparaciones entre dos `double`: el ordenador sólo guarda la primeras cifras decimales de cada número, de modo que al hacer cálculos se pueden producir pequeños errores de redondeo[2]. Por este motivo es práctica habitual utilizar un valor muy pequeño (en nuestro caso `epsilon`, que vale 10^{-10}) para dar un margen de error a los cálculos.

Programa *pow.cc*

```
#include <iostream>

using namespace std;

int main() {
  cout << "Calculo de x^y." << endl;

  cout << " * Introduce x (diferente de cero!):";
  double x;
  cin >> x;

  cout << " * Introduce y (entero!):";
  int y;
  cin >> y;

  if (y<0) {
    x=1/x;
```

[2]Es parecido a lo que ocurre en las calculadoras cuando se escribe 1/3 y luego se multiplica por 3.

```
    y=-y;
  }

  double res=1;
  while(y>0) {
    res=res*x;
    y=y-1;
  }

  cout << "Resultado: " << res << endl;
}
```

Salida (interactiva)

```
Calculo de x^y.
 * Introduce x (diferente de cero!):3
 * Introduce y (entero!):8
Resultado: 6561
```

El programa anterior calcula potencias (enteras) de un real sin utilizar el comando `pow` del `<cmath>`. Primero, utiliza un `if` para distinguir si el exponente es negativo o positivo (como `y` es un `int` no tenemos los problemas de precisión que podíamos tener con los `double`). Después necesita hacer tantas multiplicaciones como el valor absoluto del exponente.

Para conseguirlo utlitzamos el comando `while (...) {...}`. El código de dentro de las llaves {} se ejecutará una y otra vez *mientras* la condición de los paréntesis sea cierta. En nuestro ejemplo, se ejecutará tantas veces como el valor que `y` tenía originalmente: la primera vez `y` tendrá el valor original, después `y` valdrá uno menos, etc., hasta que `y` valga cero, momento a partir del cual la condición `y>0` será falsa y se saldrá del `while`.

Programa *adivina2.cc*

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>

using namespace std;

int main() {
  srand(time(0));
  bool continua=true;

  while (continua) {
    int x=rand()%100+1;

    cout << "Adivina el numero (1-100) que estoy pensando." << endl;

    int num=-1;
    int intentos=0;
    while (num!=x) {
      cout << "Di:";
      cin >> num;
      ++intentos;

      if (num>x) cout << "Demasiado grande!" << endl;
      else if (num<x) cout << "Demasiado pequenno!" << endl;
      else cout << "Correcto! Has tardado " << intentos << " intentos." << endl;
    }

    string resp="";
    while (resp!="si" and resp!="no") {
      cout << endl << "Quieres seguir jugando? (si/no)" << endl;
      cin >> resp;
```

```
    }

    if (resp=="no") continua=false;
  }
}
```

Salida (interactiva)

```
Adivina el numero (1-100) que estoy pensando.
Di:50
Demasiado pequenno!
Di:75
Demasiado pequenno!
Di:88
Demasiado pequenno!
Di:94
Demasiado grande!
Di:91
Correcto! Has tardado 5 intentos.

Quieres seguir jugando? (si/no)
no
```

No es demasiado vistoso, pero es el primer juego que podemos programar. Aparecen varios comandos nuevos. El tipo `bool` es un nuevo tipo de datos, como `int`, `string` o `double`. Es un tipo muy sencillo: una variable `bool` sólo puede almacenar dos valores: el valor `false` y el valor `true`. En nuestro ejemplo usamos la variable `continua` en la instrucción `while (continua)` para que se vayan jugando partidas mientras el usuario quiera continuar.

En este juego necesitamos obtener nombres aleatorios. El comando `rand()` nos devuelve un natural aleatorio más grande o igual que 0. El valor máximo que puede devolver `rand()` depende del ordenador y del compilador[3]. La operación `%` sirve para calcular el residuo de una división. Por lo tanto, `rand()%100+1` quiere decir: genera un número aleatorio; divídelo por 100 pero quédate con el residuo (un número aleatorio entre 0 y 99); súmale 1. Siempre que queramos utilizar `rand()` tenemos que añadir el comando `srand(time(0))` al principio del programa (sirve para inicializar el generador de números aleatorios) y incluir `<cstdlib>` y `<ctime>`. Las otras novedades son que el símbolo $\neq$ se escribe `!=`, que podemos escribir condiciones compuestas en los `while` y en los `if`, utilizando las palabras `and` y `or`, y que `++intentos` incrementa `intentos` en 1 unidad (es equivalente a escribir `intents=intents+1`).

Finalmente, podemos prescindir de poner las llaves {} en un `if` (o en un `while`) si el bloque de código que queremos ejecutar, caso de cumplirse la condición, es de una sola instrucción, como ocurre en todos los `if`s que aparecen en el programa. El objetivo es conseguir un programa más compacto y fácil de leer. Hacerlo o no hacerlo es una cuestión de gustos[4].

Programa *primeros1.cc*

```
#include <iostream>

using namespace std;

int main() {
  int minimo=1000;
  int maximo=1030;

  cout << "Primeros entre " << minimo << " y " << maximo << ":" << endl;
```

[3]Suele ser un poco más de 2000 millones.

[4]Hay programadores que prefieren poner siempre los símbolos {}, ya que si más tarde se modifica el programa y se añaden instrucciones en el `if`, uno podría olvidarse de incluir los {}.

```
  for (int i=minimo;i<=maximo;++i) {
    //miramos si i es primero

    int div=2; //posible divisor de i
    bool descartado=false;

    //si i no es primero, tiene un divisor <= que su raiz cuadrada

    while(div*div<=i and not descartado) {
      if (i%div==0) descartado=true; //confirmado: no es primer
      else ++div;
    }

    if (not descartado) cout << i << " es primero" << endl;
  }
}
```

Salida

```
Primeros entre 1000 y 1030:
1009 es primero
1013 es primero
1019 es primero
1021 es primero
```

En este programa aparece la instrucción `for`, la cual es una especialización del `while`: escribir `for(A;B;C) {...}` es basicamente equivalente a escribir `A; while(B) {...; C;}`. Es costumbre utilitzar el `for` cuando queremos un `while` que itere por un cierto rango de números. Por ejemplo, si quisiéramos repetir un código 10 veces, podríemos escribir `for(int i=0;i<10;++i) {...}`, y el código de dentro de las llaves se ejecutaría una vez por cada posible valor de `i`, desde 0 hasta 9.

También podemos ver que los símbolos $\leq$ y $\geq$ se escriben `<=` y `>=`, y que `not` sirve para negar un valor booleano. La condición `div*div<=i` nos permite probar únicamente los posibles divisores imprescindibles.

Programa *xcubo.cc*

```
// Dibuja la funcion y = x^3.

#include <iostream>

using namespace std;

int main() {
  double xmin=-1, xmax=1;
  double ymin=-1, ymax=1;
  int xres=40, yres=20;
  char c='X';

  for(int iy=0;iy<yres;++iy) {
    for(int ix=0;ix<xres;++ix) {
      double x=xmin+ix*(xmax-xmin)/(xres-1);
      double y=ymax-iy*(ymax-ymin)/(yres-1);

      if (y>x*x*x) cout << ' '; else cout << c;
    }
    cout << endl;
  }
}
```

Salida

```
                                        X
                                        X
                                       XX
                                      XXX
                                     XXXX
                                    XXXXX
                                   XXXXXX
                                 XXXXXXXX
                                XXXXXXXXX
                             XXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
        XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Aqui aparece un nuevo tipo de datos, `char`, que sirve para almacenar caracteres. La diferencia entre `char` y `string` es que un `char` siempre contiene una única letra o símbolo, mientras que un `string` contiene una secuencia de 0, 1 o más caracteres. Para distinguir entre ellos escribimos los `strings` con los símbolos , y los caracteres con los símbolos `''`.

Como se puede ver en este ejemplo, es conveniente poner los números importantes de un programa en variables declaradas al inicio. De este modo no es necesario rebuscar dentro el cógigo si nunca queremos cambiar estos valores (por ejemplo, la parte de la función que queremos dibujar —el recuadro definido por $(-1,-1)$ y $(1,1)$, o la resolución del dibujo—40 caracteres de ancho por 20 caracteres de alto).

En cuanto a los nombres de las variables, una buena regla es utilizar los nombres mas cortos que dejen claro el significado. Por ejemplo, las variables `i, j`, tal y como ocurre en matemáticas, se suelen usar para índices; está claro, pues, que `ix, iy` son dos índices para recorrer las posiciones (x,y) del dibujo. Pensad, sin embargo, que aquí estamos escribiendo programas cortos –en general, los programas largos y complejos tienen muchas variables, y se hace necesario usar nombres más descriptivos.

1.3. Funciones

Para ver que es una función reescribiremos el programa 1.2.

Programa *xcubo2.cc*

```
// Dibuja la funcion y = x^3.

#include <iostream>

using namespace std;

double f(double x) {
  return x*x*x;
}

void dibuja(double xmin, double xmax, double ymin, double ymax,
            int xres, int yres, char c) {

  for(int iy=0;iy<yres;++iy) {
    for(int ix=0;ix<xres;++ix) {
      double x=xmin+ix*(xmax-xmin)/(xres-1);
      double y=ymax-iy*(ymax-ymin)/(yres-1);
```

```
      if (y>f(x)) cout << " ";
      else cout << c;
    }
    cout << endl;
  }
}

int main() {
  dibuja(-1,1, -1,1, 40,20, 'X');
}
```

Salida

```
                                        X
                                        X
                                       XX
                                      XXX
                                     XXXX
                                    XXXXX
                                   XXXXXX
                                 XXXXXXXX
                                XXXXXXXXX
                              XXXXXXXXXXXX
             XXXXXXXXXXXXXXXXXXXXXXXXXXXX
          XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
         XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
       XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
      XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
     XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Este programa es mas largo, utiliza cosas que no hemos visto todavía, pero da exactamente la misma salida que el anterior programa. ¿Qué hemos ganado? Claridad.

Leemos este programa empezando por el final, el `main()`, que es aquello que primero ejecuta el ordenador. Aparentemente hay una única instrucción, la (sospechosa) `dibuja(...)`. Esta instrucción es inventada: no forma parte del lenguaje de programación, sino que nosotros la hemos creado. Es como una instrucción que recibiera muchos datos (cuatro `double`s, dos `int`s, un `char`) y que, una vez se ejecute, hara aquello que se indica en su definición (los dos `for`s, uno dentro del otro, que nos permiten dibujar la gráfica de x^3). Pero también podem ver que, en vez de `(y>x*x*x)` como antes, ahora aparece `(y>f(x))`. La idea es la misma: `f` es una función inventada por nosotros que, en este caso, recibe un `double x` y devuelve otro `double`, utilizando el comando `return`.

Es común llamar *funciones* a aquellos trozos de programa que devuelven valores (como por ejemplo `f()`, que devuelve un `double`) y *acciones* o `procedimientos` a aquellos trozos del programa que no devuelven nada, pero que provocan algún efecto (por ejemplo, `dibuja()` muestra la gráfica por pantalla, pero no devuelve nada; esto se marca con `void`, el tipo vacío. En el lenguaje de programación C++ no se hace una distinción explícita entre funciones y acciones, ya que una acción no es más que una función que no devuelve nada.

La claridad que hemos ganado también nos ayuda a poder modificar el programa más fácilmente. Por ejemplo:

1. Si queremos que el usuario nos introduzca los márgenes del dibujo, sólo tenemos que modificar el bloque de código del `main()`.

2. Si queremos cambiar el modo en que la gráfica se dibuja, sólo tenemos que modificar el bloque de código de `dibuixa()`.

3. Si queremoos cambiar la función a dibujar, sólo tenemos que modificar el bloque de código de `f()`.

Ahora que sabemos qué son las funciones, podemos entender mejor que es el `int main()`: es una función que no recibe nada y que devuelve un entero, y que es llamada inicialmente cuando ejecutamos nuestro programa[5].

Programa *grafica.cc*

```
// Dibuja la funcion y = sin(x).

#include <iostream>
#include <cmath>

using namespace std;

double f(double x) {
  return sin(x);
}

void dibuixa(double xmin, double xmax, double ymin, double ymax,
             int xres, int yres, char c) {
  double incx=(xmax-xmin)/(xres-1);
  double incy=(ymax-ymin)/(yres-1);

  for(int iy=0;iy<yres;++iy) {
    for(int ix=0;ix<xres;++ix) {
      double x=xmin+ix*incx;
      double y=ymax-iy*incy;

      if ((f(x)+incy>y) and (y>f(x))) cout << c;
      else cout << ' ';
    }
    cout << endl;
  }
}

int main() {
  double pi=acos(0)*2;
  dibuixa(-pi,pi, -1,1, 40,20, 'X');
}
```

Salida

```
                          XXXXXX
                         X      X
                        X        X
                       X          X
                      X
                     X             X
                                    X
                    X                X
                   X
X                                     X
                  X
 X                X
  X
   X             X
                X
```

[5]Siempre que un programa finaliza, tiene que devolver un número llamado *código de error* al sistema operativo, para indicarle si se ha ejecutado correctamente o si ha habido algún problema. Este es el número que se espera que devuelva la función `int main`. No devolver ningún número, como ocurre con todos nuestros programas, es equivalente a devolver el número 0, que quiere decir "todo ha ido correctamente".

```
    X           X
     X         X
      X       X
       XXXXXX
```

Hemos modificado el programa anterior para que dibuje la gràfica de la función $\sin(x)$ en $[-\pi, \pi]$. Modificando el código de `f()` hemos conseguido que dibuje $\sin(x)$ en vez de x^3; modificando el código de `dibuja()` hemos cambiado el modo en que se dibuja la gràfica; y modificando el código del `main()` hemos cambiado el rango del dibujo. Uno de los (muchos) modos de utilizar π sin tener que escribir sus decimales es recordar que $\cos(\pi/2) = 0$.

Programa *parametros.cc*

```
#include <iostream>

using namespace std;

int incrementa(int x1, const int &x2, int &x3) {
  x1=x1+1;
  // x2=x2+1 daria error de compilacio, por el const
  x3=x3+1;
  return x2+1;
}

int main() {
  int y, y1=7, y2=7, y3=7;
  y=incrementa(y1, y2, y3);
  cout << "y=" << y << "; y1=" << y1
       << "; y2=" << y2 << "; y3=" << y3 << endl;
}
```

Salida

```
y=8; y1=7; y2=7; y3=8
```

Las funciones pueden recibir los parámetros de tres modos: por *copia de valor* (`int`), por *referencia constante* (`const int &`) o por *referencia* (`int &`). Los dos primeros modos son de *entrada*, porque dentro de la función podemos consultar los parámetros recibidos, pero no podemos modificarlos (en el caso de la copia por valor nos está permitido modificar el parámetro, pero las modificaciones sólo tienen efecto dentro de la función; en el caso de referencia constante, no está permitido modificar el parámetro). El tercer modo, la referencia, es de *entrada-salida*: la función puede modificar el parámetro, y las modificaciones se mantienen cuando se sale de la función.

Programa *angulos.cc*

```
#include <iostream>
#include <cmath>

using namespace std;

void posxy(double grados, double radio, double &x, double &y) {
  double conversion=2*acos(0)/180;
  x=radio*cos(grados*conversion);
  y=radio*sin(grados*conversion);
}

void escribe(double grados, double radio) {
  double x,y;
  posxy(grados, radio, x, y);
  cout << "El punto de angulo " << grados << " a distancia " << radio
       << " es el (" << x << "," << y << ")" << endl;
}
```

```
int main() {
  escribe(0, 2);
  escribe(45, 3);
  escribe(90, 4);
  escribe(180, 5);
}
```

Salida

```
El punto de angulo 0 a distancia 2 es el (2,0)
El punto de angulo 45 a distancia 3 es el (2.12132,2.12132)
El punto de angulo 90 a distancia 4 es el (2.44921e-16,4)
El punto de angulo 180 a distancia 5 es el (-5,6.12303e-16)
```

Como podemos apreciar en la acción `posxy()` anterior, si queremos acciones que "devuelvan" más de un dato necesitamos utilizar parámetros de salida, para los cuales usamos el paso de parámetros por referencia.

1.4. Strings

Hasta ahora hemos tratado los strings como si fuesen un tipo de datos básico: los hemos leído por la entrada, escrito por la salida, y comparado con otros strings con los operadores `==` y `!=`. Ahora veremos otras cosas que podemos hacer con ellos.

Programa *palindromo.cc*

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

using namespace std;

bool espalindromo(const string &st) {
  int n=st.size();

  for (int i=0;i<n/2;++i) {
    if (st[i]!=st[n-1-i]) return false;
  }
  return true;
}

int main() {
  string nombre="";

  while (nombre!="FIN") {
    cin >> nombre;
    if (nombre!="FIN") {
      if (espalindromo(nombre))
        cout << nombre << " es palindromo de "
             << nombre.size() << " letras." << endl;
    }
  }
}
```

Entrada

```
palindromo 2001 2002
nosappason H anna mannana FIN
```

Salida

```
2002 es palindromo de 4 letras.
nosappason es palindromo de 10 letras.
H es palindromo de 1 letras.
anna es palindromo de 4 letras.
```

Cuando pasamos strings a un procedimiento es preferible no utilizar el paso de parámetros por copia de valor, ya que los strings pueden ser grandes y por tanto lento de copiar.

La llamada `st.size()` nos devuelve un entero con el tamaño (el número de caracteres) de `st`. La construcción `st[i]` nos devuelve el caracter (`char`) i-ésimo de `st`. Los caracteres de un string están numerados del 0 al $n-1$, donde n es el tamaño del string. Intentar acceder al carácter de un string con un índice que no esté entre el 0 y el $n-1$ es un acceso ilegal, y tiene consecuencias indefinidas: puede ser que la llamada devuelva un valor sin sentido, o que el programa acabe bruscamente dando un error de ejecución, por haber accedido a memoria que no tenía asignada. Por ejemplo, `st[st.size()]` siempre es ilegal, mientras que el código `st[st.size()-1]` accede al último elemento del string, el ekemento $n-1$, excepto en el caso que el string fuera vacío, ya que entonces sería un acceso ilegal.

Programa *gira.cc*

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

using namespace std;

string gira(const string &st) {
  int n=st.size();
  string stgirado=string(n, ' ');

  for (int i=0;i<n;++i)
    stgirado[i]=st[n-1-i];

  return stgirado;
}

int main() {
  string st="";
  while (st!="FIN") {
    getline(cin, st);
    if (st!="FIN") {
      cout << st << endl;
      cout << "[" << gira(st) << "]" << endl;
    }
  }
}
```

Entrada

```
Hola, soy una frase. Girame, por favor.
Hola, soy otra frase. Yo tambien quiero.
FIN
```

Salida

```
Hola, soy una frase. Girame, por favor.
[.rovaf rop ,emariG .esarf anu yos ,aloH]
Hola, soy otra frase. Yo tambien quiero.
[.oreiuq neibmat oY .esarf arto yos ,aloH]
```

La llamada `string(n,c)` nos devuelve un string de n caracteres inicialmente con valor `c` (espacios, en nuestro ejemplo). La instrucción `getline(cin,st)` sirve para leer texto de la entrada hasta encontrarnos con un caracter final de línea (se diferencia de la instrucción `cin >> st` porque ésta lee la entrada hasta encontrarse un final de línea o un espacio).

Programa *palabrasGrandes.cc*

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

using namespace std;

bool esletra(char c) {
  return (c>='A' and c<='Z') or (c>='a' and c<='z');
}

//incorpora la palabra st[i0]..st[i1-1] en resultado
void incorpora(const string &st, int i0, int i1, string &resultado) {
  resultado = resultado + "[" + st.substr(i0, i1-i0) + "]";
}

string palabrasGrandes(const string &st, int tammin) {
  string resultado;
  int n=st.size();
  int i=0;

  while(i<n) {
    //buscamos primera letra
    while (i<n and (not esletra(st[i])))
      ++i;

    //buscamos ultima letra
    if (i<n) {
      int i0=i;
      while(i<n and (esletra(st[i])))
        ++i;

      if (i-i0>=tammin) incorpora(st, i0, i, resultado);
    }
  }
  return resultado;
}

int main() {
  string st;
  getline(cin, st);

  cout << palabrasGrandes(st, 5) << endl;
}
```

Entrada

```
Hola! Solo queremos palabras con *muchas* l e t r a s (grandes!).
```

Salida

```
[queremos][palabras][muchas][grandes]
```

El ordenador guarda los caracteres en un código llamado ASCII, pero para conocer si un `char` está entre (pongamos) la 'A' y la 'Z' no hace falta conocer cuáles son los códigos correspondientes, tenemos bastante sabiendo que las letras tienen códigos consecutivos. Podemos concatenar strings utilizando el símbolo de suma `+`. Esto funciona con strings pero no con `char`s: por este motivo hemos escrito `"["` en vez de `'['`. Por último, `st.substr(i,len)` nos devuelve un string de tamaño `len` formado por los carateres del `st[i]` al `st[i+len-1]`.

1.5. Vectores

A menudo se tiene la necesidad de almacenar muchos datos en un programa. Un `vector` de tamaño n es una secuencia de n variables del mismo tipo, numeradas del 0 al $n-1$. Un string es, en cierto modo, un `vector` con variables de tipo `char`.

Programa *evalua.cc*

```
#include <iostream>
#include <vector>

using namespace std;

//polinomio: coefs[i] es el coeficiente de x^i
double evalua(const vector<double> &coefs, double x) {
  double res=0;
  double z=1; // en cada iteracion, z vale x^i
  for(int i=0;i<coefs.size();++i) {
    res+=z*coefs[i];
    z*=x;
  }
  return res;
}

//entrada: numero n de coeficients,
//         los n coeficientes c[n-1], c[n-2], ..., c[0]
//         el numero x
//salida:  evaluacion del polinomio c[n-1]x^{n-1}+...c[1]x+c[0]
int main() {
  int n;
  cin >> n;
  vector<double> coefs(n);

  int i=n-1;
  while (i>=0) {
    cin >> coefs[i];
    --i;
  }

  double x;
  cin >> x;

  cout << evalua(coefs, x) << endl;
}
```

Entrada

```
5
1 0 1 0 1
2
```

Salida

```
21
```

Cuando usemos vectores tenemos que hacer un `#include <vector>`. Los vectores están asociados a algún tipo, por ejemplo, vectores de doubles, y esto lo escribimos `vector<double>`. Cuando creamos un vector tenemos que decir su tamaño, como en `vector<double>coefs(n)`, ya que de otro modo tendríamos un vector de tamaño 0. Cuando pasemos vectores a un procedimiento es conveniente hacerlo siempre por referencia (no constante o constante, según si queremos modificarlo o no), para evitar que se hagan copias innecesarias de los (muchos) datos que puede tener.

Programa *diagrama.cc*

```
#include <iostream>
#include <vector>

using namespace std;

int maximo(const vector<int> &nums) {
  int max=nums[0];
  for (int i=1;i<nums.size();++i)
    if (max<nums[i]) max=nums[i];
  return max;
}

//dibuja un diagrama de barras de los numeros leidos
void dibuja(const vector<int> &nums) {
  int m=maximo(nums);
  for(int i=1;i<=m;++i) {
    for(int j=0;j<nums.size();++j)
      if (nums[j]>m-i) cout << "*";
        else cout << " ";
    cout << endl;
  }
}

//lleemos numeros hasta encontrar un negativo
int main() {
  vector<int> nums;
  int num;
  cin >> num;
  while(num>=0) {
    nums.push_back(num);
    cin >> num;
  }
  dibuja(nums);
}
```

Entrada

```
1 1 2 6 5 2 2 4 5 5 3 1 2 1 -1
```

Salida

```
   *
   **   **
   **  ***
   **  ****
  ********* *
**************
```

A menudo desconocemos cuantos elementos tendrá nuestro vector. En este caso podemos utilizar la función `v.push_back(e)` de los vectores, que primero incrementa el `v.size()` en 1, y después hace que el nuevo elemento `v[v.size()-1]` pase a valer `e`. Intentar acceder fuera de los márgenes del vector es, al igual que con los strings, ilegal.

Programa *transpuesta.cc*

```
#include <iostream>
#include <vector>

using namespace std;

typedef vector<double> fila;
typedef vector<fila> matriz;
```

```
matriz lee_matriz() {
  int filas, cols;
  cin >> filas >> cols;

  matriz M(filas, cols);
  for(int f=0;f<filas;++f)
    for(int c=0;c<cols;++c)
      cin >> M[f][c];
  return M;
}

void escribe_matriz(const matriz &M) {
  int filas=M.size();
  int cols=M[0].size();
  for(int f=0;f<filas;++f) {
    cout << M[f][0];
    for(int c=1;c<cols;++c)
      cout << " " << M[f][c];
    cout << endl;
  }
}

matriz transpuesta(const matriz &M) {
  int filasM=M.size();
  int colsM=M[0].size(); //num columnas=tam de fila

  int filasMt=colsM;
  int colsMt=filasM;
  matriz M_t(filasMt, colsMt);

  for(int ft=0;ft<filasMt;++ft)
    for(int ct=0;ct<colsMt;++ct)
      M_t[ft][ct]=M[ct][ft];
  return M_t;
}

//transponemos una matriz
int main() {
  matriz M=lee_matriz();
  matriz M_t=transpuesta(M);
  escribe_matriz(M_t);
}
```

Entrada

```
5 6
1 2 3 4 5 6
7 8 9 0 1 2
3 4 5 6 7 8
9 0 1 2 3 4
5 6 7 8 9 0
```

Salida

```
1 7 3 9 5
2 8 4 0 6
3 9 5 1 7
4 0 6 2 8
5 1 7 3 9
6 2 8 4 0
```

En este ejemplo trabajamos con vectores de `vector<double>`. Hubiéramos podido declararlos directamente con `vector<vector<double> >`[6], pero con el comando `typedef` conseguimos un programa más legible: por ejemplo, `typedef vector<double> fila` indica que, siempre que escribamos `fila` queremos ecir `vector<double>`.

En principio, no hay nada especial en tener un vector de vectores: si `M` es un vector de vectores de double, `M[i]` es un vector de double (la fila i-ésima), y `M[i][j]` es un double (el elemento j-ésimo de la fila i-ésima). En este ejemplo todas las filas tienen el mismo tamaño, pero nada impediría que filas distintas tuvieran tamaños distintos.

La declaración `matriz M(filas, cols);` es un modo abreviado de escribir `matriz M(filas, fila(cols));`, es decir: `M` se declara como una `matriz` con `filas` filas, cada una de las cuales es una `fila` con `cols` doubles.

Programa *lexicografico.cc*

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

using namespace std;

int columnas = 17; //ancho para escribir la salida.

//Funcion que compara los strings por longitud.
//(devuelve cierto cuando se cumple a<b segun nuestro criterio)
bool compara_por_long(const string &a, const string &b) {
  if (a.size()!=b.size()) return a.size()<b.size();
  else return a<b;  //en caso de empate, el orden normal
}

void escribe4columnas(const vector<string> &v) {
  for(int i=0;i<v.size();++i) {
    cout << string(columnas-v[i].size(), ' ') << v[i];
    if (i%4==3) cout << endl; //4 columnas
  }
}

int main() {
  vector<string> v;
  string st;

  cin >> st;
  while (st!="FIN") {
    v.push_back(st);
    cin >> st;
  }

  sort(v.begin(), v.end());
  escribe4columnas(v);
  cout << endl << endl;

  sort(v.begin(), v.end(), compara_por_long);
  escribe4columnas(v);
}
```

Entrada

```
Por favor, ordename por orden alfabetico (lexicografico) primero, y
despues ordename por longitud de palabras, y escribeme en 4
columnas. Las
mayusculas van antes que las minusculas, porque los codigos ASCII de
```

[6] Es necesario poner un espacio entre los símbolos `> >`, de otro modo el ordenador se confundiría con el `>>` del `cout`.

```
las letras cumplen 'A'<'Z'<'a'<'z'. FIN
```

Salida

```
 'A'<'Z'<'a'<'z'.  (lexicografico)                4             ASCII
              Las               Por       alfabetico            antes
          codigos        columnas.          cumplen               de
               de          despues               en         escribeme
           favor,             las              las           letras
         longitud             los       mayusculas       minusculas,
            orden         ordename         ordename         palabras,
              por              por           porque          primero,
              que              van                y                 y


               4                y                y               de
              de               en              Las              Por
             las              las             los              por
             por              que             van            ASCII
           antes            orden         favor,           letras
          porque         codigos          cumplen          despues
        longitud         ordename         ordename         primero,
        columnas.        escribeme        palabras,       alfabetico
       mayusculas       minusculas,  (lexicografico) 'A'<'Z'<'a'<'z'.
```

Para ordenar los elementos de un vector usamos el procedimiento `sort`, el cual puede ser llamado de diversos modos. Para ordenar un vector `v` entero de menor a mayor, la opción más usual, debemos hacer la llamada `sort(v.begin(), v.end());`. Es necesario, sin embargo, que los tipos sean comparables los unos con los otros: podemos hacerlo con `vector<string>`, porque los strings son comparables (lexicográficamente), pero también con enteros, con reales, con vectores, y en general, con cualquier tipo predefinido.

Cuando no queramos utilizar la función de comparación por defecto, o cuando el tipo que tengamos no sea comparable, podemos llamar al `sort` pasándole un parámetro extra, nuestra propia función de compración: una función que recibe dos parámetros del tipo que queremos comparar, y devuelve un booleano que indica si el primer valor es menor que el segundo.

1.6. Maps

Las variables de un vector de tamaño n están indexadas por números del 0 al $n-1$. Un `map` es parecido a un `vector` pero sin ésta limitación: somos libres de elegir cualquier tipo de datos y cualquier rango para indexar las variables que contiene el mapa.

Programa *deudas.cc*

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

using namespace std;

typedef map<string, int> cuentas;

void lee_cuentas(cuentas &c) {
  string da;
  cin >> da;
  while(da!="FI") {
    int q;
    string recibe;
    cin >> q >> recibe;
```

```
    c[da]-=q;
    c[recibe]+=q;
    cin >> da;
  }
}

int main() {
  cuentas c;
  lee_cuentas(c);

  string nombre;
  cin >> nombre;
  while (nombre!="FI") {
    cout << " * " << nombre << ": " << c[nombre] << endl;
    cin >> nombre;
  }
}
```

Entrada

```
Pepito 100 Marta
Maria 50 Pepito
Pepito 10 Marta
Maria 12 Pepito
Juan 13 Pepito
Pepito 10 Marta
FI
Pepito
Marta
JuanAndres
FI
```

Salida

```
 * Pepito: -45
 * Marta: 120
 * JuanAndres: 0
```

En este ejemplo creamos un `map<string,int>`, que es un `map` que almacena `int`s (las cuentas de la gente) y las variables del cual están indexadas por `string` (los nombres de la gente). Tenemos que hacer un `#include <map>` siempre que queramos utilizar mapas.

Podemos usar de índice de un mapa cualquier tipo comparable (strings, enteros, vectors, ...). Además, en un `map` no hay accesos ilegales. Si usamos un índice que no haya aparecido antes, el `map` nos crea una variable nueva inicializada con el índice y el valor por defecto del segundo tipo (0, en el caso de enteros).

Programa *dnis.cc*

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

using namespace std;

//entrada:
//  listado de pares nombre -- dni
//  acaba con un nombre con dni=0
//salida:
//  los nombres ordenados por dni

typedef map<int, string> dni_nombre;

int main() {
```

```
  dni_nombre dn;

  int dni;
  string nom;

  cin >> nom >> dni;
  while (dni!=0) {
    dn[dni]=nom;

    cin >> nom >> dni;
  }

  for(dni_nombre::iterator it=dn.begin(); it!=dn.end(); ++it) {
    dni=it->first;
    nom=it->second;
    cout << nom << " (" << dni << ")" << endl;
  }
}
```

Entrada

```
Johnny 12345678
Roger 87654321
Striker 45454545
Pepet 66666666
FIN 0
```

Salida

```
Johnny (12345678)
Striker (45454545)
Pepet (66666666)
Roger (87654321)
```

El `map` almacena los datos ordenados por sus respectivos índices. Es posible hacer un recorrido sobre todos los índices de un `map`, pero para hacerlo necesitamos utilizar `iterator`s. El bucle `for` del programa hace este recorrido. Si al nombre del tipo `dni_nombre` (un `map` de enteros a strings), le añadimos un `::iterator`, nos define un nuevo tipo de datos, un "iterador a `dni_nombre`", que sirve para recorrer los datos del map. El método `dn.begin()` nos devuelve un iterador que apunta a la posición inicial del mapa `dn`, mientras que el método `dn.end()` nos devuelve un iterador que apunta una posición más allá de la posición final de `dn`[7]. Escribir `++it` hace que el iterador `it` apunte al elemento siguiente del que apuntaba antes; `it->first` es el índice del elemento apuntado, `it->second` es el valor.

Programa *repeticiones.cc*

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

using namespace std;

typedef map<string,int> repeticiones;

bool existe(const repeticiones &r, const string &st) {
  return (r.find(st)!=r.end());
}

int main() {
  repeticiones r;
```

[7] `begin()` y `end()` es como el 0 y el N en un `vector v` de tamaño N: `0` es la posición inicial, N es una posición más allá de la posición final.

```
  string st;

  cin >> st;
  while (st!="FIN") {
    if (not existe(r, st)) {
      r[st]=0; //empieza con 0 repeticiones
    }
    else ++r[st];

    cin >> st;
  }

  cout << "Han aparecido " << r.size() << " palabras distintas." << endl;

  if (r.size()>0) {
    repeticiones::iterator it=r.begin();
    ++it;

    repeticiones::iterator itmasrep=it, itmenosrep=it;

    for(;it!=r.end();++it) {
      if (it->second<itmenosrep->second) itmenosrep=it;
      if (it->second>itmasrep->second) itmasrep=it;
    }

    cout << "Una de les palabras mas repetidas es '" << itmasrep->first
         << "' (" << itmasrep->second << " repeticiones)" << endl;
    cout << "Una de les palabras menos repetidas es '" << itmenosrep->first
         << "' (" << itmenosrep->second << " repeticiones)" << endl;
  }
}
```

Entrada

```
In quodam loco Manicae regionis, cuius nominis nolo meminisse, nuper vivebat
quidam fidalgus ex eis de quibus fertur eos habere lanceam in repositorio,
scutum antiquum, equum macrum canemque venaticum cursorem...

Domi autem erat ei hera domus plus quam quadraginta annos nata et neptis, quae
viginti annos nondum adimpleverat, et etiam famulus agri, qui eodem modo
cingebat sellam equo vel utebatur forfice ad arbores putandas. Aetas huius
fidalgi erat circa quicuaginta annos. Complexio eius erat fortis, sed
macillentus corpore et gracilis vultu, promptus ad surgendum mane ex lecto atque
aliquanto fruebatur venatione...
FIN
```

Salida

```
Han aparecido 84 palabras distintas.
Una de les palabras mas repetidas es 'erat' (2 repeticiones)
Una de les palabras menos repetidas es 'Complexio' (0 repeticiones)
```

Escribir `(r.find(st)!=r.end())` sirve para saber si el map `r` tiene una variable indexada por `st`. No podemos escribir, por ejemplo, `(r[st]!=0)`, ya que, por una lado, podría darse el caso que `r[st]` existiera y ya valiera `0` (por lo que pensaríamos que la variable no existía); y por otro lado, si el `map` no tuviera ninguna variable indexada por `st`, el mero hecho de escribir `r[st]`, incluso dentro de la instrucción `if (r[st]!=0) ...`, nos crearía una variable con valor 0, cosa que seguramente no deseamos.

1.7. Cosas diversas

- El comando `break` sirve para salir de un `while` o de un `for`. Gracias a esto podemos escribir la condición de acabamiento un bucle en el lugar que nos resulte más conveniente[8].

```
int dni;
string nom;

cin >> nom >> dni;
while (dni!=0) {
  dn[dni]=nom;

  cin >> nom >> dni;
}
```

```
int dni;
string nom;

while(1) { //bucle infinito
  cin >> nom >> dni;
  if (dni==0) break;

  dn[dni]=nom;
}
```

- El código `string st; cin >> st;` lee la entrada hasta que encuentra un carácter no espacio o no salto de línea; entonces empieza a añadir caracteres al string `st`, y acaba cuando encuentra un espacio o un salto de línea, que no añade a `st`, y que no cuenta como leído. En particular, los strings así leídos nunca podrán ser vacíos y nunca contendrán caracteres espacio o salto de línea.

- El código `string st; getline(cin, st);` es parecido a `string st; cin >> st;`, excepto que deja de leer cuando encuentra un salto de línea, que no añade. En particular, los strings leídos pueden ser vacíos y pueden contener caracteres espacio, pero nunca contendrán caracteres salto de línea.

- Hay que ir con cuidado si se hace la lectura de la entrada combinando `cin >>` y `getline(cin,...);`, ya que `cin >>` no leerá el caracter espacio o salto de línea separador, mientras que un `getline(cin,...);` ejecutado justo a continuación sí que lo leerá (si era un espacio, el string que devuelva el `getline` empezará con espacio; si era un salto de línea, el string será vacío). En estos casos es necesario introduir algun `char ignora; cin >> ignora;` o `string ignora; getline(cin, ignora);` adicionales para leer y ignorar los caracteres separadores no deseados.

- Tanto `cin >>` como `getline(cin, ...)` esperan encontrarse datos el leer la entrada, y esperan a que el usuario teclee los datos. Sin embargo, si saben que no encontrarán los datos que buscan (por ejemplo, porque buscaban un número y han encontrado letras; o porque l'entrada no era de teclado, sino que venía de un fichero y ya lo han leído entero; o perque el usuario ha escrito el carácter control+Z en Windows o control+D en Linux), dejarán de esperar y el programa continuará funcionando. En tal caso, no modificarán el valor de la variable que deberían leer.

  Esta situación es muy arriesgada: si el programador confiaba que le entrarían datos por la entrada, pero no hay más datos, el programa puede hacer cosas no previstas. Si puede saber si el `cin` ha agotado los datos usándolo como si fuera una variable booleana: devolverá `true` si la anterior lectura ha tenido éxito, y `false` si ha ocurrido cualquier situación que le ha impedido leer la entrada. De este modo no es necesario utilizar marcas de final de entrada como el `"FIN"` que usábamos en los ejemplos.

- A veces es necesario pasar un número de `int` a `string`, o a la inversa. Es fácil hacer una pequeña función que realice la traducción en un sentido o en otro, pero hay un

[8] Hay quien considera que utilizar `breaks` es una mala práctica de programación, porque entre otras cosas obliga a una persona a leer todo el código del bucle para conocer cuál o cuáles son sus condiciones de acabamiento.

```
string nombre;

cin >> nombre;
while (nombre!="FIN") {
  cout << "He leido " << nombre << endl;
  cin >> nombre;
}
```

```
string nombre;

//lee mientras puedas
while (cin >> nombre) {
cout << "He leido " << nombre << endl;
}
```

modo más fácil y versátil de hacerlo: utilizar un `istringstream` o un `ostringstream`. Un `istringstream` es como un `cin` que, en vez de leer de la entrada, lee de un `string` que li hayamos pasado; un `ostringstream` es como un `cout` que, en vez de escribir en la salida, escribe a un `string` que después podemos consultar.

```
#include <string>

(...)

int st2int(const string &st) {
  int num=0;
  int t=1; //para calcular las potencias de 10
  for (int i=st.size()-1; i>=0; --i) {
    num+=(st[i]-'0')*t;
    t*=10;
  }
  return num;
}

//solo funciona para n>0
string int2st(int n) {
  string st="";
  while (1) {
    st=string(1,(n%10)+'0')+st;
    if (n==0) break;
    n=n/10;
  }
}
```

```
#include <string>
#include <sstream> //para (i/o)stringstream

(...)

int st2int(const string &st) {
  istringstream iss(st);
  int num;
  iss >> num;
  return num;
}

string int2st(int n) {
  ostringstream oss;
  oss << n;
  return oss.str();
}
```

- Haciendo `#include <cmath>` tendremos acceso a las operaciones matemáticas más comunes: `sqrt` (raíz cuadrada), `abs` (valor absoluto), `sin`, `cos`, `tan` (seno, coseno y tangente de un ángulo en radianes), `asin`, `acos`, `atan` (los correspondientes arcoseno, arcocoseno y arcotangente), `exp`, `log` (exponencial o logaritmo en base e), `pow(x,y)` (x elevado a y) y `floor( )` (parte entera de un nombre real positivo).