antes, y las que contienen caracteres especiales en posiciones erróneas. Normalmente, es consecuencia de haber usado programas que usan esta codificación[12].

Incompatibilidad de representaciones

Después de lo expuesto, es lógico considerar que las representaciones son incompatibles. Aún así, es recomendable que el lector se pare un instante para insistir en ello.

Por ejemplo, imagine que a partir de la posición de memoria *100*, almaceno la secuencia de 7 caracteres *"123.456"*. Así, en la posición 100, encuentro el carácter '1' (ASCII 49, en binario *00110001*), hasta la posición 106, en la que encuentro el carácter '6' (ASCII 54, en binario *00110110*). En la figura 1.4 se presenta gráficamente este ejemplo.

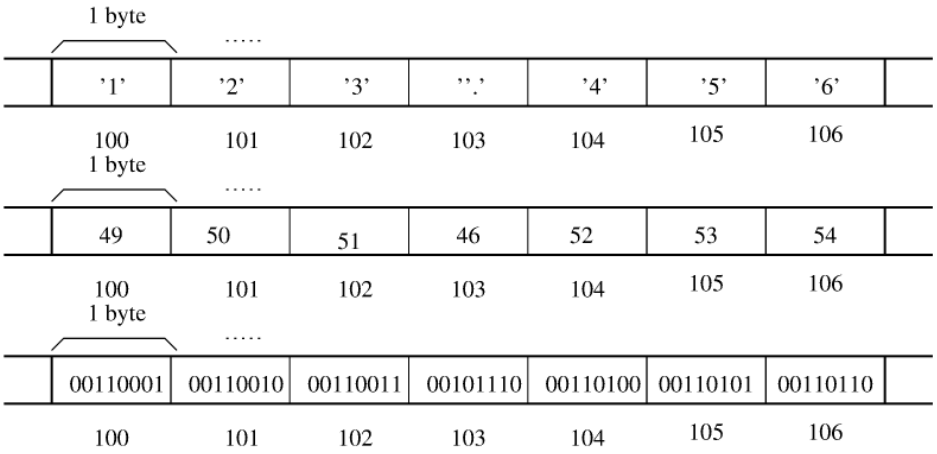

Figura 1.4
Representación de la secuencia de caracteres *"123.456"*.

Esta codificación es distinta a la del número real, ya que para éste, por ejemplo, se podrían usar los 32 bits iniciales (por ejemplo, las posiciones 100 a 103). En la figura 1.5 se presenta gráficamente como la decodificación de una secuencia de caracteres como un número real da lugar a un resultado erróneo.

De igual forma, es incompatible la codificación de un número entero y un real, a pesar de que puedan ocupar el mismo espacio (por ejemplo, 32 bits). Por lo tanto, el ordenador debe conocer exactamente la codificación que usa para sus datos, de forma que si en una posición almacenó un dato de un tipo, cuando quiera interpretar el contenido de ésta, deberá usar esa misma interpretación.

1.3. Lenguajes de programación

Seguro que el lector conoce programas que realizan complejas tareas, y que seguramente, se componen de millones de instrucciones. Lógicamente, el problema de desarrollar la

[12]No es extraño encontrarlos, ya que es muy simple, y es válida para el lenguaje inglés que no contiene esos caracteres especiales.

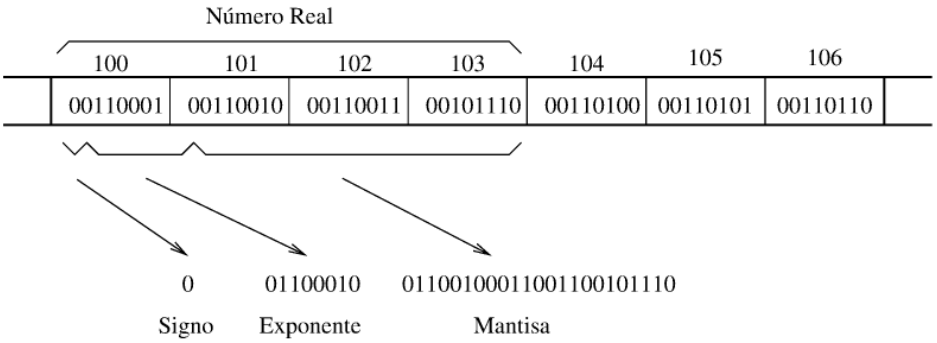

Figura 1.5
Error de decodificación de una secuencia como número real.

secuencia de instrucciones que incluye uno de estos programas parece algo difícil de llevar a cabo, si no imposible, por una persona.

Así, la distancia entre el problema y la solución parece muy grande (véase figura 1.6). Es necesario establecer métodos para poder pasar de uno a otro.

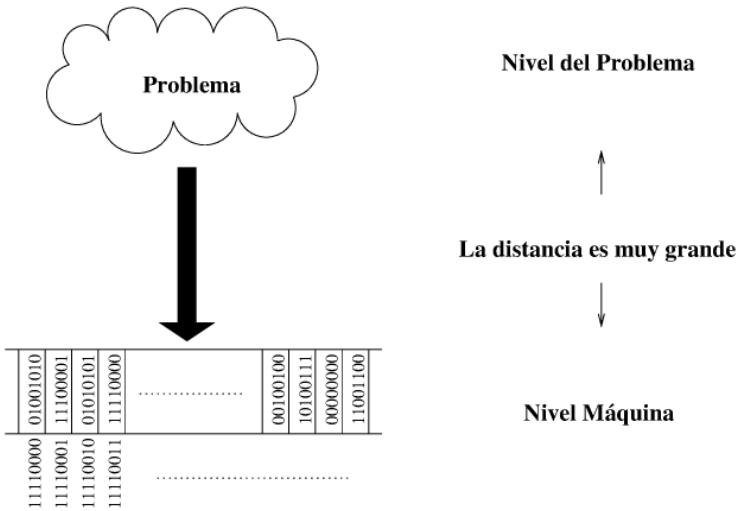

Figura 1.6
Del problema al programa.

Definición 1.5 (Lenguaje máquina) *Denominamos* lenguaje máquina *al código (conjunto de instrucciones) que entiende, directamente, un procesador.*

Por ejemplo, cuando mostrábamos el funcionamiento de un computador, el número 17 era una instrucción para llevar un dato de la memoria al procesador, el número 35 indicaba sumar, o el 19 llevar desde la CPU a la memoria. Estas 3, son instrucciones que entiende el procesador, y por tanto, son parte del lenguaje máquina. Este lenguaje tiene serios inconvenientes:

- El repertorio de instrucciones es *limitado y simple*. Cualquier tarea, independiente de su dificultad, se debe descomponer en un conjunto de instrucciones de este repertorio. Imagine una persona que, usando un lápiz, sólo sabe obedecer cuatro instrucciones:

1. *Levanta lápiz.*
2. *Baja lápiz.*
3. *Avanza*. Se mueve 1 milímetro en la dirección actual.
4. *Gira*. Cambia la dirección actual girando 1 grado a la derecha.

Y usted quiere darle las instrucciones necesarias para realizar cierto retrato. Sin duda, determinar todas estas instrucciones es una tarea difícil y laboriosa.

- El repertorio de instrucciones *depende del procesador*. Distintos fabricantes diseñan procesadores que difieren en las instrucciones y codifican las instrucciones de distinta forma. Por ejemplo, imagine que en el ejemplo anterior, usamos un procesador que entiende el 17 como realizar una suma, en lugar de entender que se quiere llevar un dato al procesador. O incluso, entiende el 35 como una división, que no era parte del conjunto de instrucciones del primer procesador.

Debido a la dificultad de este lenguaje, debemos crear otros métodos para obtener las instrucciones máquina que necesita un ordenador para realizar una tarea. La solución está en los *lenguajes de alto nivel*. Son lenguajes más potentes, ya que contienen más instrucciones y más complejas. Por ejemplo, imagine que para el problema del dibujante, creamos un lenguaje de mayor nivel, incluyendo:

1. Linea *x*. Pinta una línea de longitud *x* milímetros.
2. Girar *x*. Cambia la dirección actual girando *x* grados.

Ahora resulta mucho más fácil realizar dibujos con líneas rectas. Por ejemplo, podemos dibujar un cuadrado con lado 10 con:

1. Linea 10. Girar 90.
2. Linea 10. Girar 90.
3. Linea 10. Girar 90.
4. Linea 10. Girar 90.

Ahora bien, este conjunto de instrucciones no las entiende directamente el dibujante (no está en su repertorio). Antes de dárselo al dibujante, tenemos que "traducirlo". Así, la instrucción *Linea x* se traduce en *x+2* instrucciones de su repertorio: *Bajar Lápiz*, *Avanza* (x veces), *Levantar Lápiz*.

Así, en la programación de ordenadores, se crean lenguajes de alto nivel, más potentes, junto con programas *traductores*. Así, el programador escribe en alto nivel, y un programa traductor lo transforma del lenguaje de alto nivel, al lenguaje máquina, de forma que pueda ser ejecutado.

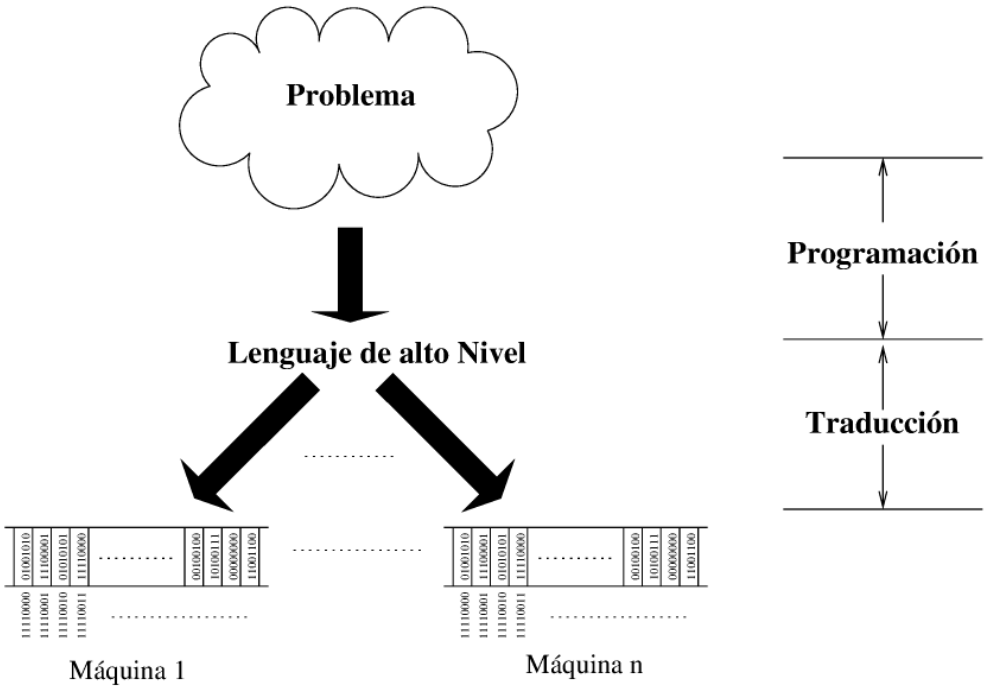

Figura 1.7
Lenguaje de Alto nivel.

La distancia entre el problema y la solución (programa en lenguaje de alto nivel) es menor, ya que ahora se dispone de más instrucciones y de mayor potencia.

Además, como muestra la figura 1.7, es posible disponer de la solución en diferentes máquinas. Note que si disponemos de un programa traductor en dos sistemas totalmente distintos (por ejemplo, tienen dos lenguajes máquina diferentes), podemos desarrollar una única solución en el lenguaje de alto nivel, que se traducirá en distintas versiones dependiendo del sistema destino.

Definición 1.6 (Portabilidad) *Un sistema software es* portable *si es válido (se puede traducir y ejecutar) en distintos sistemas, sin que sea necesaria ninguna modificación.*

El lenguaje C++

La historia de los lenguajes de alto nivel es muy larga, ya que tiene más de medio siglo. Durante este tiempo han aparecido múltiples lenguajes, incorporando nuevas cualidades y especializándose para los problemas que iban surgiendo.

En 1972, se crea el lenguaje C, como el lenguaje para el desarrollo de programas en los sistemas *UNIX*. Debido a que varios fabricantes empezaron a desarrollar versiones de C que empezaban a sufrir distintas variaciones, se determinó la necesidad de llegar a un acuerdo. En 1989 se crea el estándar ANSI de C (conocido por C89). El hecho de que haya un estándar es fundamental, ya que fabricantes de computadoras que obtienen sistemas muy diferentes pueden ofrecer la posibilidad de usar este estándar. Así, los programadores pueden escribir programas que se pueden llevar a muchos computadores, a pesar de sus diferencias.

A principios de los 80, Bjarne Stroustrup creó el lenguaje C++ como una extensión de C y evolucionó hasta el estándar de 1998 (conocido como C++99). Posteriormente, han ido apareciendo algunas actualizaciones y correcciones hasta llegar al estándar de 2003 (también conocido como C++03). El nuevo lenguaje C++ no es una simple modificación de algunos aspectos de C, sino que además, se le ha dotado de nuevas posibilidades, destacando sobre todo la programación dirigida a objetos.

Un programa en este lenguaje, que corresponde al ejemplo anterior de suma de dos números es:

```
#include <iostream>
using namespace std;

int main()
{
  int n1;
  int n2;
  int res;

  cout <<"Introduzca un número: ";
  cin >> n1;
  cout <<"Introduzca un número: ";
  cin >> n2;
  res= n1+n2;
  cout << "El resultado de la suma es: " << res << endl;
}
```

donde no aparecen detalles de *bajo nivel* sobre la máquina donde se ejecuta. De hecho, este programa se puede usar para resolver nuestro problema en muchas máquinas distintas (en cualquiera que tenga un programa traductor de C++ conforme al estándar).

Finalmente, es interesante destacar que los lenguajes evolucionan, e incluso aparecen otros nuevos que se adaptan cada vez mejor a las necesidades actuales. Así, el lenguaje C se ha modificado hasta un nuevo estándar (C99). También han aparecido otros, entre los que podemos destacar *Java* y *C#*.

1.3.1. Desarrollo de software

En principio, la tarea a realizar consiste en saber cómo pasar del problema al programa en un lenguaje de alto nivel.

Pensemos en un problema sencillo, por ejemplo, supongamos que queremos que un programa lea la siguiente ecuación:

$$a \cdot x + b = 0$$

y escriba la solución. Para resolverlo, debemos indicar al ordenador los pasos que debe seguir para obtener la solución, es decir, el algoritmo que resuelve el problema.

Definición 1.7 (Algoritmo) *Un* algoritmo *es un conjunto finito y ordenado de pasos o instrucciones para obtener la solución a un problema.*

Lógicamente, cada uno de esos pasos debe ser preciso (no ser ambiguo), y el algoritmo debe acabar en un tiempo finito.

Nótese que si queremos obtener una solución a nuestro problema de programación, los pasos que se especifiquen en el algoritmo deben poder especificarse en el lenguaje de la solución. Así, podemos decir que el algoritmo consiste en:

1. Leer los valores a, b.
2. Calcular $x = -b/a$
3. Escribir como resultado el valor x.

que en C++ sería:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    double a;
7    double b;
8    double x;
9
10   cout << "Considere la ecuación ax+b=0." << endl;
11   cout << "Introduzca el valor de a: ";
12   cin >> a;
13   cout << "Introduzca el valor de b: ";
14   cin >> b;
15   x= -b/a;
16   cout << "La solución de la ecuación es: " << x << endl;
17 }
```

donde podemos ver que el primer paso del algoritmo se ha resuelto en las líneas 10-14, el segundo en la 15, y el tercero en la 16.

En otro ejemplo, podemos plantear un problema en el que un programa lee una imagen (obtenida con nuestra última cámara de fotos digital), recorta un trozo seleccionado, y la presenta, tras un zoom y un aumento del contraste, que nos revela esa persona oculta que no sabíamos distinguir. Ahora bien, esta tarea es muy compleja como para poder escribir, de forma directa, el programa que la resuelve.

El algoritmo necesario para resolver un problema mucho más complejo no es tan fácil de obtener. Para ello, será necesario desarrollar una metodología que simplifique el camino, desde el problema hasta la solución, en términos de un lenguaje.

Por otro lado, el lector puede entender que el objetivo es obtener cualquier programa que resuelva el problema del *usuario*[13]. Sin embargo, la ingeniería del software tiene como objetivo obtener programas de "calidad", es decir, programas fiables (que no fallen), que sean eficientes, y que puedan ser modificados (resolver fallos del programa o adaptar modificaciones del problema).

[13]La persona o sistema que luego usará el programa

En el apéndice A (página 345) se presenta la ingeniería del software de forma más detallada. Sin embargo, es importante insistir en este enfoque, ya que lo arrastraremos a lo largo de todo este libro como el motivo por el que dos programas, a pesar de obtener los mismos resultados, no serán igualmente válidos.

Lógicamente, los problemas que resolvamos irán creciendo en dificultad, de forma que no será tan sencillo como plantear un simple algoritmo y realizar la *implementación* (escritura de la solución en un lenguaje de alto nivel). El lector irá aprendiendo a enfrentarse a este tipo de problemas, conforme estudie los temas, a medida que vamos introduciendo nuevos conceptos y herramientas.

1.3.2. El proceso de traducción

El proceso de traducción desde un lenguaje como C++ a lenguaje máquina lo realiza un programa "externo", previamente disponible. Existen dos tipos de programas para realizar la traducción:

- *Intérpretes*. Realizan una traducción sentencia a sentencia, y las van ejecutando.
- *Compiladores*. Realizan una traducción completa desde el lenguaje de alto nivel a lenguaje máquina.

Los intérpretes traducen la primera sentencia a lenguaje máquina, detienen la traducción, ejecutan la sentencia, traducen la siguiente sentencia, la ejecutan, etc. Los compiladores traducen todo el programa. Así, no es necesario el compilador cuando necesitemos ejecutar el programa (ya traducido). En nuestro caso, utilizaremos un compilador de C++.

El proceso de traducción lo realiza un compilador, así que esta etapa parece muy sencilla, sin necesidad de que intervenga el programador. Sin embargo, cuando obtenemos un programa en C++, no hemos terminado en absoluto con el trabajo. No nos podemos olvidar que no somos perfectos, y el programa seguro que contendrá errores. Hablaremos de dos tipos de errores:

- *Errores de compilación y enlazado*. No hemos escrito correctamente el programa, y no corresponde al lenguaje. El compilador genera errores de traducción, que el programador debe interpretar para solucionar. Por ejemplo, si después de la línea de *main* no escribimos un carácter de llaves ({). el compilador genera un error de sintaxis en el momento de la compilación.
- *Errores de ejecución*. Se genera un error cuando el programa ya está traducido y lo estamos ejecutando. Generalmente se deben a errores de programación, es decir, que nos hemos equivocado en los algoritmos que deberían resolver el problema. Por ejemplo, si escribimos correctamente una operación de división de dos valores enteros, se podría generar un error en tiempo de ejecución en caso de que el divisor valga cero[14].

[14] La operación de división por cero puede provocar un error fatal, que detiene el programa

Algunos autores distinguen además los *errores lógicos*, cuando el programa termina correctamente, pero con una solución equivocada.

Los errores de compilación y enlazado son, con diferencia, los más sencillos de resolver. Ahora bien, para poder resolverlos debemos conocer bien el lenguaje, incluyendo la forma en que realiza la traducción.

Para obtener el fichero ejecutable se realizan dos pasos:

1. *Compilación*. Es la traducción propiamente dicha. Se traduce el fuente (ficheros en lenguaje C++) a ficheros "traducidos", es decir, a ficheros que contienen nuestro código en pero en lenguaje destino.

2. *Enlazado*. Se une el resultado de la compilación con los recursos necesarios para obtener el ejecutable.

Como ejemplo, volvamos al programa fuente de la sección anterior, en el que obtenemos la solución de una ecuación de primer grado. El programa fuente está en formato texto, es decir, está compuesto por una secuencia de caracteres ASCII que, supuestamente, sigue las reglas que especifica el lenguaje.

La etapa de compilación obtiene un programa traducido, que denominamos *programa o fichero objeto*. En la figura 1.8 se presenta un esquema de esta traducción. Como entrada, el programa fuente se encuentra almacenado en un fichero que denominamos, por ejemplo, *PrimerGrado.cpp*. El compilador traduce este fichero, generando un fichero objeto denominado *PrimerGrado.o*.

Figura 1.8
Del fuente al objeto.

El compilador realiza la traducción leyendo el programa fuente desde el primer carácter al último (en ese orden), es decir, que podemos decir que "lee" el programa desde la primera línea hasta la última. Durante este proceso, mantiene una serie de tablas de símbolos que contienen información sobre su situación en cada instante, las cosas que ha traducido, y lo que necesita para seguir (podríamos decir que son tablas donde escribe, "en sucio", todo lo que necesita). Las etapas que se siguen son:

1. *Análisis léxico*. Inicialmente tenemos una secuencia de caracteres. El primer paso consiste en convertirla a una secuencia de *tokens*. Por ejemplo, podemos recibir la secuencia de caracteres :

   cin>>a;res=-a/3.14;

   Y el analizador léxico la descompone en algo como:

"identificador cin", "operador >>", "identificador a", finalizador ;", "identificador res", "operador =", "operador -", "identificador a", "operador /", "real 3.14", "finalizador ;"

2. *Análisis sintáctico*. Comprueba que la secuencia de *tokens* sigue las reglas sintácticas que determina el lenguaje. Por ejemplo, si se está usando el operador suma en una expresión, la sintaxis indica que se escribe un operando, seguido del operador +, y seguido por otro operando. Si el programador escribe dos veces el operador +, la sintaxis no es correcta, y el compilador genera un error indicando que no se esperaba encontrar otro operador.

3. *Análisis semántico*. En esta etapa se comprueba que el programa, que sintácticamente es correcto (está bien escrito), tiene sentido. Por ejemplo, puedo escribir una operación correctamente como *Operando1 Operador Operando2*, sin embargo, no es correcto si lo que estamos es sumando un número entero, con una cadena de caracteres.

4. *Generación de código*. El programa es correcto y por tanto se puede generar el código objeto que le corresponde.

5. *Optimización*. El compilador revisa el código generado para optimizarlo, eliminando operaciones innecesarias, cambiando de orden las operaciones, etc. Por ejemplo, imagine que calcula la expresión $(a + b) * 5/(a + b)$. En el código generado, puede descubrir que la expresión $(a + b)$ se calcula dos veces, entonces modifica el código para que sólo se haga una vez. A lo largo de este libro iremos indicando algunos casos en los que el compilador podría aplicar cambios para la optimización.

Finalmente, el programa objeto no es directamente ejecutable y es necesario aplicar un paso más, de *enlazado* para obtener el resultado final. En esta etapa, se une el resultado de la compilación, con otros recursos externos que son necesarios para nuestro programa. Como es de esperar, también podemos obtener errores, denominados de *enlazado*, en este último paso.

En la figura 1.9 se presenta un esquema de esta etapa. Como entrada, el fichero objeto se encuentra almacenado, por ejemplo, en *PrimerGrado.o*. El *enlazador* une este fichero con otros ficheros y recursos, generando un fichero ejecutable denominado *PrimerGrado.exe*.

Más adelante, sobre ejemplos concretos, detallaremos esta última etapa. Así mismo, un estudio más amplio sobre este proceso, y que incluye otras posibilidades, se presenta en el tema 12 (página 317).

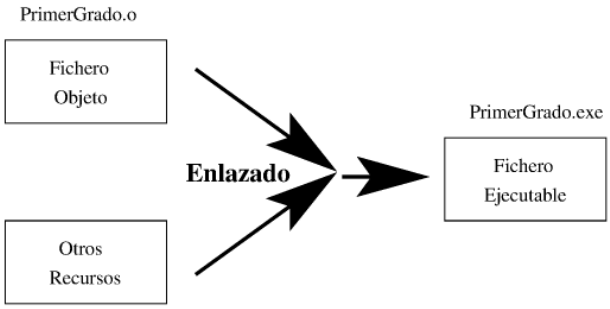

Figura 1.9
Del objeto al ejecutable.

1.4. Programación en C++

En esta sección, revisaremos los elementos básicos para realizar sencillos programas en C++, que servirán como introducción a la programación en este lenguaje, así como base para estudiar conceptos y estructuras más avanzadas en las siguientes lecciones.

1.4.1. El preprocesador

En C++, podemos decir que la compilación consiste realmente en dos etapas:

1. Preprocesamiento.
2. Compilación propiamente dicha.

La primera etapa consiste en un procesamiento previo del fichero fuente, para que un programa (el preprocesador) analice la existencia de instrucciones "especiales" que se deben llevar a cabo antes de que el código entre al compilador. Este tipo de instrucciones se denominan *directivas* del preprocesador, y se distinguen porque comienzan con una carácter # al principio de la línea.

Un ejemplo es la directiva `#include` que va seguida de un nombre de archivo que contiene, igualmente, código C++. Cuando se analiza el código fuente, al llegar a este punto, el preprocesador encuentra esta orden, localiza el fichero referenciado, y como efecto la elimina del programa, sustituyéndola con el código C++ que contiene el archivo indicado.

Como vemos, realmente no se realiza ningún tipo de compilación (recuerde, análisis léxico, sintáctico, etc.), sino que sólo se sustituye una línea por otras. De ahí, que se considere como una etapa previa a la compilación. A pesar de ello, cuando hablemos de forma general de compilación, nos referiremos a las dos etapas. El lector encontrará más detalles sobre esta directiva y otras en el tema 12 (página 317).

1.4.2. Un programa básico

En principio, consideraremos que un programa C++ implementa un conjunto ordenado de instrucciones indicadas en un módulo denominado *main*. Por lo tanto, la ejecución de

un programa consiste en localizar *main* y seguir todas sus instrucciones (delimitadas por dos caracteres '{','}') hasta que finalice.

Así, el programa más simple posible es el que no hace nada, y por tanto se escribiría como:

```
int main()
{ }
```

donde vemos que para especificar el módulo *main*, debemos escribir *int main()*. Y a continuación, las instrucciones del programa (en este caso, ninguna).

Un ejemplo algo más complejo es el siguiente:

```
1 #include <iostream>  // Para usar cout, endl
2 using namespace std; // Usamos el estándar
3
4 int main()
5 {
6   cout << "¡Bienvenido!" << endl;
7 }
```

Donde podemos observar:

- En la línea 1 y 2 se incluyen los caracteres "//". El compilador entiende estos caracteres como indicadores de comentarios. Para poder leer mejor nuestros códigos, podemos incluir cualquier comentario en el punto que deseemos del programa. Para ello, tenemos que indicar al compilador que no son parte del programa. Lo podemos realizar de dos formas:

  1. Escribir dos caracteres "//". Todo lo que sigue a estos caracteres hasta final de línea se ignora.
  2. Escribir los caracteres "/*" (como comienzo) y "*/" (como fin). Todo lo que se encuentra entre el comienzo y el fin se ignora.

- En la línea 1, se incluye una directiva. En este caso, el precompilador localiza el fichero con nombre *iostream* e incluye, el código C++ que contiene, al principio del programa. En principio, este código no nos interesa especialmente. Sólo, téngase en cuenta que con este nombre hacemos que el programa sea capaz de realizar entradas y salidas, por ejemplo, por el teclado y la pantalla.

- En la línea 2 aparece una línea indicando que se utilizará el espacio de nombres *std*. El estudio de los espacios de nombres se dejará hasta un tema posterior. En principio, podemos interpretarlo como la orden para que el compilador nos permita usar directamente las herramientas que nos ofrece el estándar.

- Línea 4-5. Comienzan las instrucciones a llevar a cabo.

- Línea 6. Indica que se escriba en la salida el mensaje, seguido de un final de línea (salto de línea y retorno de carro). En esta línea usamos *cout* y *endl*, que son dos

identificadores del estándar. Disponemos de ellos al haber incluido *iostream*, y podemos usarlos directamente al indicar que usamos el espacio de nombres *std*.

Por tanto, para comenzar el curso de programación en C++, consideraremos que un programa tiene la siguiente estructura:

```
[Inclusión de archivos cabecera]
using namespace std;

int main()
{
[Instrucciones]
}
```

donde la parte indicada por corchetes corresponde al código que debemos escribir para implementar nuestro programa concreto.

Nótese que en la inclusión de archivos hemos indicado que son de *cabecera*. Esto se debe a que estos archivos contienen información y se escriben especialmente para ser incluidos. De ahí que se denominen de cabecera[15].

Escritura del fichero fuente

Para poder usar el ejemplo anterior, tendremos que introducirlo en el ordenador, realizar la traducción y ejecutarlo. Para ello, en primer lugar tendremos que escribirlo en un archivo fuente.

Como indicamos, este fichero consiste en la secuencia de caracteres que componen el programa. Por tanto, necesitaremos usar un *editor* de texto que permita introducir estos fuentes almacenándolos en formato estándar ASCII. Damos un nombre a este fichero, y ya estamos preparados para la compilación.

El nombre del fichero puede ser cualquiera, aunque procuraremos que sea significativo del contenido del fichero. Como extensión usaremos [16] *cpp*, indicando que es de código C++. Por ejemplo, podemos nombrar nuestro ejemplo con el nombre *bienvenido.cpp*.

En este punto, podemos lanzar el proceso de traducción para obtener otro archivo, en este caso ejecutable, del programa fuente. El nombre de este archivo final puede estar predefinido en nuestro compilador o podemos indicar uno nosotros. Por ejemplo, podemos indicar que queremos traducir al archivo *bienvenido.exe*.

Una vez que se traduce, podemos pedir al sistema que se ejecute este archivo, obteniendo como resultado el mensaje "¡Bienvenido!".

Respecto al formato de escritura del fichero fuente, el lenguaje C++ es un lenguaje denominado de *formato libre*, es decir, no es necesario que "formateemos" el fuente

[15] Veremos como en sus nombres se puede incluir la extensión ".h" que proviene de *header*.

[16] Existen otras extensiones estándar, como son *cxx* o *C*.

de ninguna forma especial. Sólo es necesario tener en cuenta que el analizador léxico deberá obtener los *tokens* correctamente. Para ello, se considera que cualquier carácter espacio, salto de línea, tabulador, etc. (cualquier "espacio blanco") determina una separación entre *tokens*. Por ello, para el analizador léxico todos ellos son equivalentes, e incluso varios de ellos consecutivos lo son.

Como ejemplo de este funcionamiento, podemos reescribir el mismo programa, con los mismos resultados como sigue:

```
1  #include <iostream>  // Para usar cout, endl
2                       using
3  namespace
4                   std
5
6  ; // Usamos el estándar
7
8  int main
9  (
10 )
11 {
12                          cout
13   <<
14   "¡Bienvenido!"  /* no sirve */  << endl
15                        ;
16 }
```

donde vemos que los espaciados respectan el mismo conjunto de *tokens* e incluso hemos añadido un comentario en medio de una línea (que lógicamente se ignorará). Como el lector intuirá, intentaremos presentar un formato como el primero, que nos permite leer más fácilmente el código.

Por ejemplo, no hubiera sido válido escribir (el compilador detiene la traducción indicando un error) lo siguiente:

```
1 #include <iostream>  // Para usar cout, endl
2 using namespace std; // Usamos el estándar
3
4 int main()
5 {
6   cout < < "¡Bienvenido!" << endl;
7 }
```

donde la línea 6 produce dos tokens "<" (después de *cout*) en lugar de uno sólo "<<" (que veremos no tienen el mismo significado).

1.4.3. Datos y expresiones

En nuestros programas, necesitamos procesar información de entrada para obtener la salida deseada. Así, el sistema debe ser capaz de usar datos (números, mensajes, nombres, valores reales, etc.), y operar con ellos para obtener esos resultados.

El lenguaje C++ nos ofrece distintos tipos de datos *predefinidos*, junto con una serie de operadores sobre ellos, que nos permiten construir expresiones. Algunos tipos básicos son:

- *int*: Tipo entero.
- *double*: Tipo en coma flotante (número reales).
- *char*: Tipo carácter (por ejemplo la letra 'a', el espacio ' ', el salto de línea, etc.).
- *bool*: Tipo booleano. Sólo puede tomar dos valores, *false* o *true*.

El que un tipo de dato sea *predefinido* indica que se encuentra ya en el lenguaje, es decir, que el compilador conoce como se almacena (sus valores), como se escribe (la sintaxis), y qué significa cada una de sus operaciones.

Así, si tenemos que hacer un programa que gestione las calificaciones obtenidas por los alumnos de primer curso de una carrera, podemos usar un valor de tipo *int* para indicar el número de alumnos que hay en un grupo, un valor *double* para indicar la calificación obtenida en un examen, un valor *char* para codificar el grupo al que pertenece un alumno o un valor *bool* para indicar si un alumno ha superado la parte práctica de la asignatura.

Variables

Los datos que usa un programa se almacenan en memoria[17]. En C++ no nos tenemos que preocupar de la zona de memoria donde se almacena cada dato, ya que el compilador la asigna de forma automática.

Definición 1.8 (Variable) *Una variable es un nombre para una zona de memoria que almacena un valor de algún tipo.*

Por ejemplo, imagine que deseamos calcular la media aritmética de dos valores reales. Podemos considerar los siguientes pasos:

1. Leer el valor del primer número real.
2. Leer el valor del segundo número real.
3. Calcular un nuevo número, resultado de la media aritmética.
4. Escribir el resultado.

[17] Recuerde las secciones anteriores sobre representación de datos.

Para escribirlo en C++ utilizamos variables. Por ejemplo, podemos pensar que el proceso consiste en leemos un número y lo *"guardamos"*, leemos el segundo y lo *"guardamos"*, para calcular la suma *"recuperamos"* el primero y el segundo, para obtener el resultado, y *"guardarlo"*. El concepto de "*guardar*" y "*recuperar*" se corresponde con el de escribir en memoria o leer de memoria.

En la práctica, lo único que tenemos que hacer es seleccionar el tipo de dato que deseamos y darle un nombre. Por tanto, para poder usar una variable, lo primero que tenemos que realizar es una declaración, en la que informamos del tipo de dato que almacena y del nombre que referencia a ese dato. Es en la declaración, cuando el compilador "detecta" la necesidad del dato, y por tanto cuando "prepara" la zona de memoria donde se localizará (todo ello, de forma automática).

La forma de una declaración de variable es:

```
<tipo de la variable> <nombre de variable> ;
```

donde el tipo de la variable corresponden a uno de los tipos conocidos por el compilador (por ejemplo, *int* o *bool*), el nombre de la variable es un identificador que nosotros seleccionamos (secuencia de caracteres de letras[18], dígitos y carácter _ que no empieza con un dígito), y termina con el carácter ';' que indica al compilador que es el fin de la declaración.

Note que la declaración es fundamental para el compilador, no sólo porque conoce el nombre de la variable, sino porque tiene que preparar una zona de memoria para manejarla. Así, el tipo indica el tamaño de la zona de memoria a reservar.

Ejemplos de declaración son:

```
int valor;
double resultado;
char un_caracter;
bool apto;
```

Ejemplos de declaraciones incorrectas son:

```
int dato         // Falta ;
double 1val;     // Comienza por 1
bool -logico;    // Comienza por -
char una letra; // Tiene un espacio
```

Finalmente, los identificadores de variables no pueden ser ninguna de las palabras reservadas del lenguaje (ver tabla C.2, página 420). Por ejemplo, la palabra *int* está reservada para indicar un tipo entero, y por tanto, el usuario no puede usarla para identificar una variable.

Ejercicio 1.3 *Considere la siguiente lista de identificadores:*

- hola, primero, 1dato, datonúmero2, año, double, resultado, el_valor_5.

Indique los identificadores que son válidos en C++.

[18] Se incluyen las letras del alfabeto inglés, es decir, no son válidas caracteres como vocales acentuadas o la letra 'ñ'

Literales

Un literal es un valor concreto de un tipo. Por ejemplo,

- Un literal de tipo *int* se escribe como un conjunto de dígitos (0-9) consecutivos (precedido con un signo '-' para los negativos). Ejemplos: *123, -37, 5*
- Un literal de tipo *double* como un conjunto de dígitos (tal vez, precedido con el signo '-') que incluye el carácter punto (por ejemplo, "-1.23", o "5."). Además, es posible usar la notación científica añadiendo una letra 'e' y un exponente (por ejemplo, "5.2e-7").
- Un literal de tipo *char(carácter)*. Escribimos el carácter entre comillas simples. Por ejemplo, 'A' corresponde al carácter ASCII 65.
- Un literal de tipo *booleano (bool)*. Podemos especificar sus dos valores con "true" o "false" respectivamente.
- Un literal *cadena de caracteres*. Escribimos una secuencia de caracteres entre comillas dobles. Por ejemplo, "Mensaje". Note que no es lo mismo un carácter entre comillas simples que entre comillas dobles (entre dobles es una cadena de longitud uno).

Nótese que para el caso de los caracteres o las cadenas de caracteres es necesario escribir el carácter entre comillas simples o dobles respectivamente. Sin embargo, esto no es posible siempre. Por ejemplo, algunos de ellos son:

- No podemos escribir una comilla simple entre comillas simples (se entendería como el "cierra comillas").
- De forma similar, no podemos escribir unas comillas dobles dentro de una cadena de caracteres.
- ¿Cómo se escribe un tabulador?
- ¿y un salto de línea?

Para ello, es posible utilizar la barra invertida (\) para indicar al compilador que se desea indicar un carácter especial. En la tabla 1.2 se presentan algunos de estos códigos.

Así algunos ejemplos de su uso son:

- Caracteres: *'\n', '\t'*
- Cadenas de caracteres: "*\tComienza con tabulador*"

Ejercicio 1.4 *Identifique el tipo de los siguientes literales:*

- "X","Hola","","'Y',1345,30.0,true,15,"false",12.0,'\"'

Tabla 1.2
Ejemplos de códigos de barra invertida.

Código	Significado
\'	Comilla simple
\"	Comilla doble
\n	Salto de línea
\t	Tabulador
\r	Retorno de carro
\\	Barra invertida

Operaciones

Los tipos de datos de C++ no serían útiles sin un conjunto de operaciones asociadas. Así, cuando disponemos de un nuevo tipo, no sólo somos capaces de representar la información, sino que podemos operar con ella.

Así, para el tipo *int*, podemos escribir expresiones con los operadores de suma, resta, producto, division, etc. Por ejemplo, podemos escribir la expresión:

$$\frac{a+b}{c}$$

Lógicamente, para escribirlas en un programa, tenemos que usar una notación que permita especificarla como una secuencia de caracteres. Así, usando los caracteres +-*/, podemos escribir la expresión anterior como:

$$(a+b)/c$$

Note que hemos añadido unos paréntesis, ya que el producto y la división tienen más prioridad que la suma y la resta. Conocer la prioridad de los operadores es fundamental, ya que afecta al resultado de la expresión. Al escribir las expresiones como una secuencia de caracteres, es necesario conocer el orden de evaluación. Éste viene predeterminado en el lenguaje (ver apéndice C, página 417). Para modificarlo podemos, como hemos visto, usar paréntesis. Por ejemplo, si no hubiéramos puesto los paréntesis en la expresión anterior, se hubiera obtenido:

$$a+\frac{b}{c}$$

Otros operadores con enteros son el menos unario, y el operador % (módulo). La precedencia es:

1. Paréntesis[19].

[19] Aunque existe el operador () en C++, aquí sólo aparece como una forma de cambiar el orden de evaluación en una expresión, y no como un operador propiamente.

2. Menos unario.

3. Producto, división, módulo.

4. Suma, resta.

Además, es necesario conocer el orden de evaluación o *asociatividad*, ya que si aparecen varios operadores de la misma precedencia, el resultado puede ser distinto si evaluamos de derecha a izquierda o al revés. Por ejemplo,

$$a/b * c$$

tiene dos operadores de igual precedencia, pero que darían lugar a distintos resultados dependiendo del orden de evaluación. Como norma general, los operadores binarios se evalúan de izquierda a derecha[20]. Así, en el ejemplo anterior, primero se realiza la división y, el resultado se multiplica por c. Si quisiéramos que a se dividiera por el resultado del producto, tendríamos que añadir paréntesis, para darle más prioridad.

Para el caso de los números reales (por ejemplo, *double*), también existen operadores similares, de forma que podemos escribir expresiones con datos de tipo real. Al igual que antes, tenemos los mismos operadores (excepto el de módulo) con la misma precedencia.

Además, note que el operador de división es distinto para enteros que para reales. Cuando dividimos un entero entre otro, obtenemos como resultado otro entero. Por tanto, el operador corresponde a la división entera. Por ejemplo *7/2* devuelve como resultado *3*, mientras que *7.0/2.0* devuelve *3.5*.

Ejercicio 1.5 *Supongamos las variables* a,b,c,d *de tipo* double. *Escribir en C++ las siguientes expresiones:*

$$\frac{a + \frac{b}{cd}}{a + b}$$

$$a + \frac{b + c}{d} + \frac{b}{c} - b$$

Conversiones aritméticas

Es posible que el lector se pregunte sobre la "mezcla" de distintos tipos. Efectivamente, podríamos escribir *7.0/2* que corresponde a la división de un número *double* por un número *int*. En este caso, cuando se realiza una operación entre dos valores de distinto tipo, se realiza una conversión, para que los dos tengan el mismo. En nuestro ejemplo, se convierte el entero a real, de forma que se lleva a cabo una división de valores *double*.

En este sentido, el lector no debería preocuparse por la mezcla de tipos que almacenan valores numéricos (enteros y reales), al menos para los ejemplos de esta primera parte. Podemos confiar en que el compilador realiza las conversiones necesarias.

[20]Más adelante, veremos que los operadores de asignación se evalúan de derecha a izquierda

Operador de asignación

Una operación especialmente importante, y que se define para todos los tipos básicos, es la de *asignación*. El operador es el signo =, a su izquierda se especifica una referencia a un objeto[21] en memoria (por ejemplo, una variable), y a su derecha un nuevo valor para ese objeto. Por ejemplo, podemos escribir:

```
x = 3 ;
```

para hacer que el valor de la variable `x` sea 3. O con una expresión:

```
x = (a+b)/2;
```

para hacer que `x` valga el resultado de evaluar la expresión $(a + b)/2$.

Además, en la parte derecha puede haber cualquier expresión, incluso conteniendo la variable de la izquierda. Por ejemplo, podemos hacer:

```
x = 3;
x = x*2;
```

en la primera línea asignamos el valor 3 a `x`, y en la segunda, asignamos el resultado de multiplicar `x` por 2. Aunque matemáticamente pueda resultar una barbaridad, el proceso de asignación consiste [22] en, primero evaluar la parte derecha (se usa el antiguo valor de `x`), y luego, realizar la asignación (actualiza el valor de `x`).

> **Ejercicio 1.6** *Suponga dos variables* x,y *que contienen dos valores desconocidos. Escriba las sentencias necesarias para hacer que los valores de las variables se intercambien.*

1.4.4. Entrada/Salida básica

El computador se relaciona con el exterior a través de los dispositivos de entrada/salida (E/S). Un programa también debe implementar este intercambio de información.

Definición 1.9 (Interfaz) *Una* Interfaz *se refiere a la conexión física y funcional entre dos sistemas independientes. Así, denominaremos* interfaz de usuario *a la parte del sistema que se encarga de la comunicación entre el programa y el usuario.*

Por ejemplo, podemos realizarlo *de forma gráfica*, mostrando *ventanas*, con botones, menús, casillas de texto, para introducir datos, mientras muestra los resultados en otras ventanas. En este caso, hablamos de *interfaz gráfica de usuario*[23].

El lector probablemente conoce varios medios para realizar la E/S de datos en un programa. Por ejemplo, gráficamente como hemos visto, E/S de datos de archivos, o

[21] Consideraremos objeto como "algo en memoria", es decir, una región de almacenamiento contiguo en memoria. El lector también puede encontrar en la literatura el concepto de *l-value* (valor-i) como una expresión que referencia a un objeto (constante o no).

[22] Realmente el proceso corresponde a la evaluación de una expresión que incluye el operador de asignación (de menor prioridad), sin embargo, para estos temas, no entraremos en detalles.

[23] En inglés, *Graphical User Interface (GUI)*.

incluso intercambio con otros programas en *Internet*. La programación de estos sistemas de comunicación es muy compleja y no forma parte del lenguaje de programación ya que depende en gran medida del sistema (por ejemplo, el sistema gráfico usado en *UNIX* difiere en gran medida por el usado en *WINDOWS*), y no es fácil llegar a un estándar[24].

El lenguaje C++ ofrece una forma estándar de comunicación que, siendo simple, es muy potente. En esta sección veremos únicamente los elementos más básicos para desarrollar los capítulos siguientes.

Cuando se lanza un programa C++, se realiza una asignación de los *flujos* de E/S estándar *cin, cout, cerr*[25]. Podemos pensar en el programa como una entidad que tiene un conector de entrada y dos de salida, como muestra la figura 1.10. Cuando ejecutamos el programa, el sistema "conecta" un flujo de datos en la entrada estándar, que el programa conoce con el identificador *cin*, un flujo de datos en la salida estándar, que el programa conoce con el identificador *cout*, y otro de salida estándar de error, conocido con *cerr*.

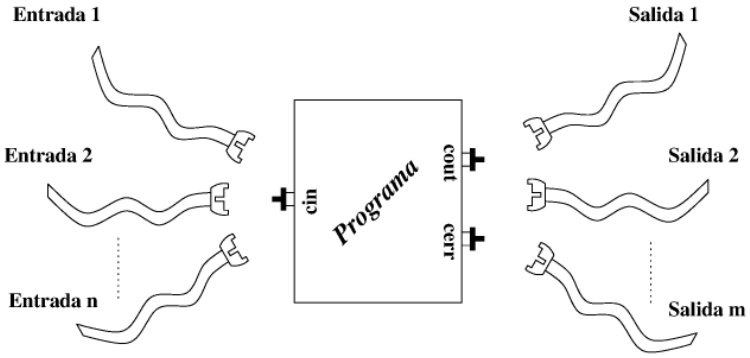

Figura 1.10
Flujos de E/S estándar.

Tenga en cuenta que:

- Estos identificadores están disponibles simplemente incluyendo el fichero cabecera *iostream*, es decir, escribiendo una línea al principio del programa, con la directiva *include* del preprocesador.

```
#include <iostream>
```

- Un *flujo de datos* es una secuencia de caracteres. Por ejemplo, si entra el número flotante *125.34*, lo que entra por *cin* es la secuencia '1', '2', '5', '.', '3', '4'.

Así, para el programa, no es importante lo que se haya "conectado" a la entrada o la salida, sino que podemos solicitar datos de entrada desde *cin* o sacar datos de salida

[24]En este sentido, es interesante nombrar otros lenguajes, como Java, que utiliza una máquina virtual para su ejecución. De esta forma, es más fácil implementar bibliotecas estándar que resuelvan estos y otros problemas.

[25]Podríamos hablar de un cuarto, *clog* (aunque se asigna al mismo sitio que *cerr* con la diferencia de que éste es un flujo sin *búfer*) o sus correspondiente extendidos. El lector interesado puede consultar por ejemplo Stroustrup[22], aunque este tema es muy avanzado para este capítulo.