Algoritmos y programas

En éste capítulo se presentará el concepto de *algoritmo*, visto como el proceso que conduce de un problema al programa que lo resuelve. Veremos algunas de las técnicas empleadas para representar algoritmos además del uso de un lenguaje de programación estructurado. Por último comentaremos algunas de las características deseables de un programa, así como una serie de pautas a seguir a la hora de escribir programas.

¿Qué es un algoritmo?

La palabra *algoritmo* procede de la deformación de la forma latina del nombre del matemático Persa ``*Abu Ja'far Mohamed ibn Musa* ***al Khowarizmi"*** *(825 D.C.)*. Si consultamos la definición actual de la palabra en un diccionario enciclopédico podemos encontrarnos con algo así: Conjunto ordenado y finito de operaciones que permite obtener la solución de un problema.

Ahora bien, aplicado a la informática y en concreto a la realización de programas, podemos verlo como la descripción precisa de una sucesión de instrucciones que permite a un computador llevar a cabo un trabajo para solucionar un determinado problema.

Cada uno de estos pasos debe cumplir lo siguiente:

- Ser finito y estar definido, es decir, debe estar perfectamente claro lo que hace, y además puede estar compuesto de uno o varios *sub-pasos* más sencillos.
- Ser efectivo, es decir, debería poder ser realizado por una persona con lápiz y papel en una cantidad de tiempo finita.

Es importante hacer notar que la realización o ejecución de un algoritmo producirá uno o varios resultados a partir de cero o más entradas al mismo, y lo que es más importante, lo hará en un tiempo finito.

Procesos

La ejecución de una o varias de las instrucciones que componen un algoritmo la denominaremos **proceso** o **cómputo**. En todo proceso nos encontraremos con una serie de magnitudes que son relevantes en el cálculo a realizar y las utilizaremos para almacenarlas unos objetos a los que llamaremos **variables**.

Variables

Podemos considerar una variable como una caja donde podemos guardar un dato para, posteriormente, recuperar su contenido. Puesto que podemos guardar un dato tantas veces como queramos en una variable, diremos que el **valor** de una variable es el del último dato introducido.

Para poder emplear una variable será imprescindible proporcionar cierta información de la misma antes de utilizarla. Esta información consiste en:

- El nombre que le damos, normalmente formado por una combinación de letras y dígitos, donde la primera letra del nombre nunca es un dígito.
- El tipo de los datos que puede contener, en un principio trataremos con *números*, *caracteres alfanuméricos* y *valores lógicos*.

Por ejemplo, la declaración de variables en C tendrá el siguiente aspecto:

```
int num, edad, i, j;
char cod, letra;
```

Áreas de interés en el trabajo con algoritmos

Actualmente podemos considerar cinco áreas distintas enfocadas al trabajo con algoritmos:

1. Cómo diseñar algoritmos. En ella se estudian varias técnicas que permiten la escritura de algoritmos eficientes, eliminando - siempre que es posible- la parte de *arte* que pone cada programador al crear un nuevo algoritmo.

2. Cómo expresar los algoritmos. Se pueden englobar aquí aspectos como la *programación estructurada*.

3. Cómo *validar* algoritmos. Denominamos *validar* un algoritmo al proceso por el cual comprobamos que proporciona una respuesta correcta *para todas* las posibles entradas que pueda tener.

4. Cómo analizar algoritmos. Se trataría de determinar a priori, es decir, antes de ejecutarse, y de manera correcta la cantidad de tiempo y memoria que necesitará un algoritmo para ejecutarse.

5. Cómo probar un algoritmo. En realidad lo que se hace en esta fase es:

   - Comprobar que el algoritmo no tiene fallos.
   - Una vez que sabemos que el algoritmo es correcto, medimos -a posteriori- el tiempo y la cantidad de memoria que necesita para realizar los cálculos. Este análisis a posteriori puede ser útil para comparar con los resultados obtenidos en el hecho a priori.

Como se puede comprobar, las disciplinas que abarca el concepto de *algoritmo* son diversas y diferentes entre sí, de manera que uno puede centrarse en aquella que más le interese.

Lenguaje algorítmico o seudocódigo.

Llamaremos **pseudocódigo** o **pseudolenguaje** a la combinación de las construcciones de un lenguaje formal de programación con proposiciones informales expresadas en el lenguaje empleado por el creador del algoritmo - *en nuestro caso el español*-.

Para llegar a la fase en la que se realiza la escritura en seudocódigo, previamente, tenemos que haber realizado un modelo apropiado del problema. En muchos casos, este modelo, suele ser un modelo matemático del mismo.

Una vez tenemos escrita una primera versión del algoritmo -que denominaremos de Nivel 1- , y salvo que éste sea muy sencillo, debemos empezar a *refinarlo*, lo descompondremos en varios subproblemas más o menos independientes entre sí; en este instante nos encontraremos ante una descomposición de Nivel 2. Si fuera necesario, aplicaremos este proceso de manera repetida a cada uno de los subproblemas de Nivel 2 -obteniendo así un descomposición de Nivel 3-, y seguiríamos así hasta llegar a un punto en el que sólo tengamos que ir sustituyendo las proposiciones generales en español que hayamos empleado por instrucciones cada vez más pequeñas y definidas. A este proceso se le conoce por el nombre de *refinamiento por pasos*, y fue descrito por primera vez por Niklaus Wirth, el diseñador del lenguaje Pascal.

En algún instante de este proceso, el algoritmo expresado en pseudocódigo estará lo suficientemente detallado como para que podamos abordar la etapa de escritura del programa correspondiente de la manera más sencilla posible, de hecho, en muchos casos la escritura de este programa no consistirá más que en la traducción del pseudocódigo final obtenido al lenguaje de programación elegido; requiriendo esta tarea un esfuerzo relativamente pequño por parte del programador.

Especificación de un algoritmo.

En puntos anteriores del tema hemos visto que la 'ejecución' de un algoritmo supone la realización de una serie de procesos y el uso y -por tanto- modificación del contenido de una serie de variables.

Una vez en este punto podemos describir la ejecución de un algoritmo en base a los cambios que `produce' en su entorno. Para describir estos cambios partimos de un *estado inicial* del algoritmo -en realidad sería del entorno-, es decir, antes de que éste se ejecute, y vemos en qué estado se encuentra el entorno una vez finaliza la ejecución del algoritmo. A este último estado al que hemos llegado lo denominamos *estado final*. Podemos definir pues, el concepto de **especificación de un algoritmo** como la relación que existe entre los estados inicial y final del algoritmo una vez se ha ejecutado éste. Para representar la relación entre los estados *inicial* y *final* del algoritmo se hace uso de los *predicados*.

Por ejemplo, si queremos realizar un algoritmo que sume dos números reales, como estado inicial deberemos tener dos datos (variables) y como resultado seguiremos teniendo las dos variables y la suma de ellas.

```
algoritmo Suma reales
  var p,q,r : reales; fvar
  {p = P ^q = Q}
  suma reales
  {r = P + Q ^ p = P ^ q = Q}
falgoritmo.
```

Podemos hacer la siguiente interpretación: partimos de un estado en el que dos variables reales '*p*' y '*q*' contienen los valores *P* y *Q* respectivamente, después de realizar el proceso *suma reales* tenemos en la variable '*r*' el resultado de sumar los valores iniciales de '*p*' y '*q*', además se nos garantiza que tanto '*p*' como '*q*' siguen valiendo lo mismo que antes de ejecutar el algoritmo.

(E*n general el nombre de una variable en mayúsculas representa el valor inicial de esa variable antes de ejecutar cualquier instrucción)*.

Estructuras básicas en todo programa

Ya hemos visto qué es lo que hace un algoritmo partiendo de su especificación, ahora nos vamos a centrar en conocer las instrucciones que emplearemos para escribir programas.

Las distintas estructuras con las que vamos a trabajar a la hora de escribir programas son las siguientes:

- Declaración de variables y constantes.
- Asignación.
- Estructura secuencial.
- Estructura alternativa.
- Estructura iterativa.

Para empezar pasamos a estudiar la estructura general de un programa en C.

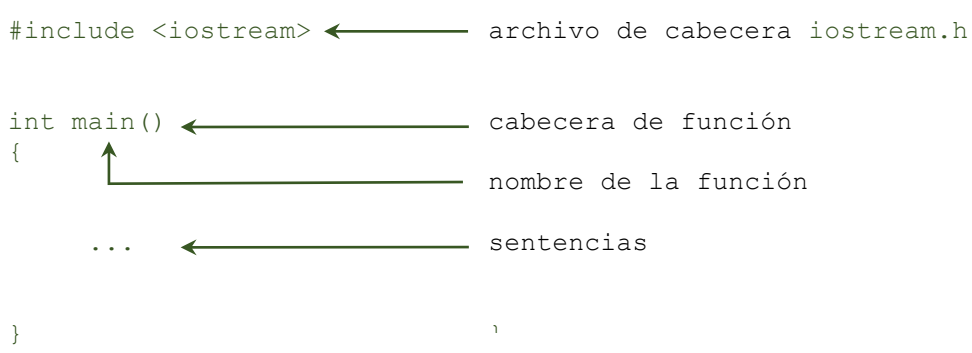
Estructura general de un programa en C

Un programa en C se compone de una o más funciones. Una de las funciones debe ser obligatoriamente **`main`**. Una función en C es un grupo de instrucciones que realizan una o más acciones (esto se verá más adelante en el capítulo dedicado a módulos). Así mismo, un programa contendrá una serie de directivas **`#include`** que permitirán incluir en el mismo archivos de cabecera que a su vez constarán de funciones y datos predefinidos en ellos.

De un modo más explícito, un programa C puede incluir:

- directivas del procesador;
- declaraciones globales;
- la función `main()`;
- funciones definidas por el usuario;
- comentarios del programa.

A continuación se muestra la estructura típica completa de un programa:



Un ejemplo de un programa sencillo en C podría ser:

```
1:  #include <iostream>
2:  // El programa imprime "Bienvenido a la programación en C"
3:  int main()
4:  {
5:      cout << "Bienvenido a la programación en C";
6:      return 0;
7:  }
```

La directiva `#include` de la primera línea es necesaria para que el programa tenga salida. Se refiere a un archivo externo denominado `iostream.h` en el que se proporciona la información relativa a la instrucción (objeto) `cout`. Obsérvese que los ángulos `<` y `>` no son parte del nombre del archivo; se utilizan para indicar que el archivo es un archivo de la biblioteca estándar de C/C++.

La segunda línea es un comentario, identificado por dobles barras inclinadas (//). Los comentarios se incluyen en programas que proporcionan explicaciones a los lectores de los mismos. Son ignorados por el compilador.

La tercera línea contiene la cabecera de la función `main()`, obligatoria en cada programa C. Indica el comienzo del programa y requiere los paréntesis `()` a continuación de `main()`.

La cuarta y séptima línea contienen sólo las llaves `{` y `}` que encierran el cuerpo de la función `main()` y son necesarias en todos los programas C/C++.

La quinta línea contiene la sentencia que indica al sistema que envía el mensaje `"Bienvenido a la programación en C"`, al objeto `cout`. Este objeto es el **flujo estándar de salida** y se verá más adelante.

La sexta línea contiene la sentencia `return 0`. Esta sentencia termina la ejecución del programa y devuelve el control al sistema operativo de la computadora. El número 0 se utiliza para señalar que el programa ha terminado correctamente (con éxito).

Los comentarios

En casi todos los algoritmos bien escritos aparece una construcción de gran importancia a la hora de tener que releerlos y entenderlos, es la que se conoce como **comentario**. Los *comentarios* no influyen de ninguna manera en la ejecución de un algoritmo, y sólo sirven de ayuda al *programador* para saber qué es lo que hace un determinado algoritmo, ya esté escrito por él o por otra persona.

Tan malo es hacer poco uso de los comentarios como el uso excesivo de los mismos, en general, el mal uso de los mismos. Un buen criterio para saber cuándo hacer uso de ellos es el siguiente: Un comentario debe indicar de forma clara y concisa qué es lo que hace una sección del código de un algoritmo, y no cómo lo hace, para eso ya está el propio código.

Todos los lenguajes de programación ofrecen la posibilidad de poner comentarios distribuidos a lo largo del programa. Algunos de estos lenguajes ofrecen distintas formas de poner comentarios, aunque lo normal es que solo tengamos disponible una de ellos. La diferencia entre uno y otro tipo suele estar en la posibilidad de que un comentario se pueda extender a lo largo de varias líneas del código, o que solo pueda estar en una línea. Por ejemplo, en lenguaje C++ podemos emplear el carácter ' // ' de modo que el comentario se extiende hasta el final de la línea en la que comienza, es decir, hasta que pulsamos la tecla *enter* o return. Sin embargo si queremos que el comentario abarque varias líneas escribiremos /* al inicio y */ al final del comentario.

Veamos unos ejemplos:

Comentario sintácticamente correcto.

```
int main(){
   // Esto es un comentario correcto
   ...
}
```

Comentario sintácticamente incorrecto.

```
int main(){
   // Esto es un comentario
    incorrecto
   ...
}
```

Comentarios sintácticamente correctos.

```
int main(){
   // Esto es un comentario
   // correcto
   ...
}
```

```
int main(){
   /* Esto es un comentario
    correcto  */
   ...
}
```

Comentario sintácticamente correcto, pero formalmente innecesario.

```
int main(){
  int a=0;  // inicializamos la variable a con el valor 0

  ...
}
```

Declaración de constantes

Cualquier variable o constante que utilicemos en nuestros algoritmos ha de ser previamente declarada, es decir, debemos decir cuál es su nombre y qué tipo de datos puede contener.

Constantes en C

El lenguaje C permite definir constantes simbólicas que representan un valor determinado, que no cambia a lo largo del programa. Se definen mediante macros a través de la directiva **`#define`** usando la siguiente sintaxis

```
#define <nombre_constante> <valor>
```

donde `nombre_constante` representa el nombre simbólico que se da a la constante y valor su valor. La palabra reservada define indica que la constante tiene un valor que se fija durante todo el período de vida que dura la ejecución de un programa y que el procesador debe sustituir las ocurrencias de `nombre_constante` por su valor. Por ejemplo, `#define PI 3.141516` define una constante `PI` que representa al número `3.141516`.

No existe convención definida respecto a los nombres de las constantes, pero es aconsejable fijar un criterio para distinguirlas de las variables en el código del programa. Por ejemplo, se suelen escribir en mayúsculas como en el ejemplo anterior o con una k delante, como `#define kpi 3.141516`.

Hay que tener en cuenta que ya no es posible asignar valor a las constantes después de su declaración, porque ya lo tienen y no se puede modificar. Sin embargo, nunca se debe confundir una constante con una variable con valor inicial. Hay dos diferencias fundamentales:

1. Su declaración se diferencia porque las constantes tienen el calificador #define
2. El valor de la constante no se puede cambiar, por lo que se obtiene un error si se trata de asignar un valor a una constante.

Existen tres tipos de constantes en C:

- Carácter:
  ```
  #define LETRA 'a'
  ```
- Valores enteros:
  ```
  #define MAX 150
  ```
- Valores reales:
  ```
  #define PI 3.141516
  ```

Constantes en C++

En C++ se pueden definir constantes de dos formas, ambas válidas para nosotros. La primera es por medio del comando `#define nombre_constante valor` y la segunda es usando la palabra clave `const`, veamos ahora cada una de estas formas en detalle.

Vamos a compilar el siguiente fichero escrito en C++:

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

#define PI 3.141516; //Definimos una constante llamada PI

int main()
{
    cout << "Mostrando el valor de PI: " << PI;

    ...
}
```

Notemos que ha sido bastante fácil, sin embargo no todo es bueno. Lo que ocurre en realidad es que al usar la instrucción `#define` en C++ no estamos creando una constante, estamos creando una expresión y por tal motivo hay algunas cosas que se complican, veamos:

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

#define PI 3.141516; //Definimos una constante llamada PI

int main()
{
    cout << "Mostrando el valor de PI: " << PI << endl;

    return 0;
}
```

Al compilar este programa obtenemos el siguiente mensaje de error:

```
$ g++ -o prueba prueba.cc
prueba.cc:16:48: error: expected expression
     cout << "Mostrando el valor de PI: " << PI << endl;
    ^
```

El error se produce al haber usado el operador `<<` justo después de `PI`, esto sucede porque `PI` no es tratado exactamente como una variable cualquiera sino como una expresión, así que aunque podemos usar `#define` para declarar constantes no es la mejor opción.

La instrucción **`const`** nos permite declarar constantes de una manera más adecuada y acorde. Las constantes declaradas con **`const`** poseen un tipo de dato asociado (como debería ser siempre) y se declaran en el interior de nuestro código como un tipo cualquiera. Modificando el ejemplo anterior tendríamos:

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

const float PI = 3.141516; //Definimos una constante llamada PI

int main()
{
    cout << "Mostrando el valor de PI: " << PI << endl;

    return 0;
}
```

Cuando compilamos ya no se produce error y lo podemos ejecutar.

```
$ g++ -o prueba prueba.cc
$ ./prueba
Mostrando el valor de PI: 3.141516
```

Declaración de variables

Una variable es una representación alfanumérica, es decir, un nombre para identificar una posición de memoria. Como tal, se caracteriza por tres propiedades: posición de memoria que almacena el valor, tipo de datos almacenado y nombre que se refiere a esa posición de memoria. Una variable debe estar obligatoriamente ligada a un tipo de datos.

Las variables pueden contener valores diferentes durante la ejecución de un programa, eso sí, uno cada vez, **no** varios a la vez.

La sintaxis de declaración de una variable es:

```
<tipo de datos> <nombre1> [<nombre2> .. <nombren>]
```

Las variables son identificadores y, como todo identificador en C, deben escribirse siguiendo las reglas vistas para construir identificadores válidos. Todas las variables en C deben definirse antes de su uso.

```
int temperatura;     // variable de tipo int
float precio;    // variable de tipo float
char tipo;           // variable de tipo char
```

Además, en una misma línea se pueden declarar varias variables del mismo tipo. Por ejemplo:

```
int temperatura, dia, mes, anyo;   // variables de tipo int
float precio, distancia;           // variables de tipo float
```

En C es posible definir variables sin valor inicial y variables con valor inicial. El formato de definición con valor inicial seguiría el siguiente formato:

```
<tipo de datos> <nombre1> = <valor>
```

Por ejemplo:

```
float interes = 2.3;
```

La asignación

Podemos considerarla como la instrucción fundamental, ya que es la que nos permite asociar un valor a una variable.

La sintaxis que tiene la *asignación* en C es la siguiente:

```
<nombre_variable> = <valor>
```

O de modo más general:

```
<nombre_variable> = <expresión>
```

Que quiere decir: La variable llamada <nombre_variable> contiene una copia del valor de la expresión <expresión>, previa evaluación de ésta última. O sea, se evalúa la expresión que aparece a la derecha del igual y se asigna a la variable que aparece a la izquierda del igual.

Es obvio que la expresión debe ser una expresión correcta y además debe proporcionar un resultado del mismo tipo que la variable a la cual se va a asignar.

Veamos algunos ejemplos de asignación:

```
// Asignación de un valor real a una variable real (float)
```

```
int main() {
  float x;

  x = 3.2;
   ...
}
```

```
// Asignación de un valor de tipo carácter a una variable de tipo char
```

```
int main() {
  char letra;

  letra = 'c';
  ...
}
```

```
// Asignación de una variable real a otra variable real
```

```
int main() {
  float a, b;

  a = 3.2;
  b = a;
  ...
}
```

```
// Asignación de una expresión real a una variable real
```

```
int main() {
  float a, b;

  a = 3.2;
  b = a / 5.3;
  ...
}
```

```
// Incremento en una unidad de una variable de tipo entero (int)
```

```
int main() {
  int num;

  num = 3;
  num = num + 1;   // en C esto es equivalente a num++;
  ...
}
```

Este último ejemplo, no significa que *num* sea igual a *num + 1*, sino que a la variable *num* le asignamos una nueva cantidad que es igual al valor que ya contenía incrementado en una unidad.

A medida vayamos construyendo algoritmos más complejos es lógico que necesitemos declarar más variables que las que hasta ahora mismo estamos utilizando, con el fin de clarificar nuestro programa deberemos realizar la declaración de variables al inicio del módulo principal (main) o del módulo correspondiente que las utilice.

Para terminar con la sentencia de asignación aclarar que por el hecho de declarar una variable, está creada pero no se le ha asignado ningún valor todavía. De este tipo de variables se dice que son variables **no inicializadas**. Las variables no inicializadas son una fuente de problemas muy difíciles de detectar tanto para un programador experto como para un novato.

La entrada/salida

El nombre genérico de *entrada/salida* lo empleamos para referirnos a las posibilidades que nos ofrece un lenguaje de programación para:

1. Mostrar el contenido de las variables y constantes empleadas en nuestros programas en la pantalla del computador, en una impresora o incluso para 'guardarlas' en dispositivos de almacenamiento externos como discos, usb, etc...

2. Obtener información que pueda ser guardada en variables de nuestro programa. Esta información podrá ser leída desde el teclado o recuperada desde cualquier otro dispositivo donde se encuentre almacenada.

Para evitar complicar demasiado nuestro programa, las operaciones de entrada/salida se limitarán a mostrar la información en pantalla y leerla desde teclado.

- En C:
  - Para mostrar la información en pantalla disponemos de

```
printf(formato, arg1, arg2, arg3, ... argn);
```

    donde `formato` incluye la cadena de caracteres que se van a imprimir y una serie de especificadores de formato. Un especificador de formato incluye, entre otros, un especificador de conversión, que indica la forma en la que se imprimirán los datos que se encuentran en `arg1, ... argn`. Por ejemplo, el especificador de conversión que se utiliza para imprimir un número entero es `%d`. Por ejemplo:

```
printf("Número %d" , num);
```

    siendo `num` una variable entera cuyo valor en ese momento es `67`.

    imprimirá por pantalla:

```
Número 67
```

  - Para leer la información desde teclado utilizaremos

```
scanf(formato, arg1, arg2, arg3, ... argn);
```

    donde la cadena `formato` es similar a la utilizada en la función `printf`. Los especificadores de conversión utilizados también son los mismos que los empleados en la función `printf`. Cada uno de los argumentos de la función es un puntero a la variable en la que se quiere almacenar el dato que se va a leer. Un puntero representa una dirección de memoria. Por ejemplo:

```
scanf("%f", &temperatura);
```

    previamente habremos declarado la variable `temperatura` de tipo `float`.

En C para poder ejecutar operaciones de entrada/salida estándar deberemos incluir el archivo de cabecera **stdio.h**, que entre todas las funciones que contiene algunas son muy utilizadas: `getchar`, `putchar`, `scanf`, `printf`, `gets` y `puts`. Estas seis funciones permiten la transferencia de información entre la computadora y los dispositivos de entrada/salida estándar tales como teclado y monitor.

- En C++, la entrada/salida funciona a través de flujos de bytes de un dispositivo a otro.
  - Para mostrar la información en pantalla combinaremos los operadores de desplazamiento a la izquierda **<<** (inserción de flujo) con el objeto de flujo **cout** (flujo de salida estándar).

```
cout << formato << variable << ... ;
```

Siguiendo el ejemplo anterior:

```
cout << "Número " << num;
```

siendo `num` una variable entera cuyo valor en ese momento es `67`.

imprimirá por pantalla:

```
Número 67
```

  - Para recoger la información desde teclado combinaremos los operadores de desplazamiento a la derecha **>>** (extracción de flujo) con el objeto de flujo **cin** (flujo de entrada estándar).

```
cin >> variable >> ... ;
// podríamos recoger más de una variable siempre separadas
// por el operador de desplazamiento a la derecha
```

Siguiendo el ejemplo anterior:

```
cin << temperatura;
```

recogeríamos por pantalla un valor que asignaríamos a la variable `temperatura`.

La biblioteca `iostream` es la biblioteca de entrada/salida estándar en C++. El acceso a esta biblioteca se realiza mediante el archivo cabecera **iostream.h**. Un programa escrito en C++ toma la entrada o salida como un flujo de datos. En la entrada un programa extrae byes de un flujo de entrada y en la salida un programa inserta bytes en un flujo de salida. Por tanto, un flujo se puede considerar como una secuencia lineal de bytes con un significado. Los bytes pueden formar una representación binaria de datos carácter o numéricos. Los bytes de entrada pueden venir del teclado (entrada estándar), pero también pueden proceder de otro dispositivo de almacenamiento como un disco duro u otro programa. Y de forma análoga los bytes del flujo de salida pueden ir destinados a la pantalla (salida estándar), a una impresora, a un fichero, o a otro programa. Puesto que son varias las opciones de entrada/salida deberemos indicar cuál de ellas vamos a utilizar a lo largo de nuestro programa, de ese modo evitamos escribir en cada momento `std::cin` o bien `std::cout` . El modo de evitar esto es escribiendo:

```
using namespace std;
```

Al escribir esto le estamos indicando al compilador que usaremos el espacio de nombres `std` (estándar) por lo que no tendremos que incluirlo cada vez que usemos elementos de este espacio de nombres como son `cout` y `cin`.

Ejemplo de entrada/salida en C:

```
#include <stdio.h>

int main(){
```

```
   int anyo, edad;

   printf("¿En qué año naciste? ");
   scanf("%d", &anyo);
   edad = 2016 - anyo;
   printf( "Durante este año cumplirás %d años \n", edad);
   return 0;
}
```

Ejemplo de entrada/salida en C++:

```
#include <iostream>

using namespace std;

int main(){
   int anyo, edad;

   cout << "¿En qué año naciste? ";
   cin >> anyo;
   edad = 2016 - anyo;
   cout << "Durante este año cumplirás " << edad << " años " << endl;
   return 0;
}
```

Definición de precisión y tamaño de la salida

C++ proporciona varios manipuladores de flujo que ejecutan tareas de formato, cabe destacar dos que actúan sobre el flujo de salida: precisión y ancho (*precision*, *width*).

Uso de *precision*: en el caso de números reales se puede definir la precisión, esto es, el número de dígitos a la derecha del punto decimal que queremos visualizar. Ejemplo:

```
cout.precision(int)
```

donde `int` se refiere al tipo del argumento de esta función, en este caso un entero que define el número de decimales que vamos a visualizar en las posteriores salida que se realicen, hasta que se vuelva a efectuar otra llamada a esta función para modificar la precisión. Para que esta opción funcione correctamente se debe utilizar previamente una llamada a `cout.setf(ios::fixed)` que hace que un número de punto flotante se muestre con un número determinado de dígitos. Por ejemplo:

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

int main(){
   float num;
   num = 118.0 / 3;

   cout.setf(ios::fixed);

   // muestra la precisión definida por defecto
   cout << "La precisión es " << cout.precision() << endl;
   cout << num << endl << endl;
   // definimos la precisión a 3
   cout.precision(3);
   cout << "La precisión es " << cout.precision() << endl;
```

```
    cout << num << endl << endl;
    // definimos la precisión a 4
    cout.precision(4);
    cout << "La precisión es " << cout.precision() << endl;
    cout << num << endl << endl;
    // definimos la precisión a 0
    cout.precision(0);
    cout << "La precisión es " << cout.precision() << endl;
    cout << num << endl << endl;
    return 0;
}
```

Cuya ejecución sería:

```
La precisión es 6
39.333332

La precisión es 3
39.333

La precisión es 4
39.3333

La precisión es 0
39
```

Uso de `width`: permite definir el ancho del campo del valor a visualizar, es decir, decir la cantidad total de caracteres con la que vamos a escribir el valor, mediante la función:

```
cout.width(int)
```

donde `int` se refiere al tamaño. Si el ancho definido es mayor que lo que se va a visualizar se introducen espacios de relleno a la izquierda del valor, por el contrario, si lo que deseamos visualizar tiene un tamaño mayor que el ancho del campo, el valor NO será truncado, visualizándose en su totalidad, o sea, no hará nada. Por ejemplo:

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

int main(){
   int num1 = 12345;
   int num2 = 678;

   // cout << "Cadena texto " <<  endl;
   // delimitando el ancho de la cadena a 15
   cout.width(15);
   cout << "Probando texto " <<  endl;
   // delimitando el ancho de la cadena a 15
   cout.width(15);
   cout << "texto " <<  endl;
   // delimitando el ancho de la cadena a 8,
   // como la cadena de texto es mayor la visualiza completa
   cout.width(8);
   cout << "Probando texto " <<  endl;
   // delimitando el ancho de la cadena, en este caso un número a 7
   cout.width(7);
   cout << num1 << endl;
```

```
   // delimitando el ancho de la cadena, en este caso un número a 7
   cout.width(7);
   cout << num2 << endl;
   // delimitando el ancho de la cadena, en este caso un número a 3,
   // como el total de dígitos es mayor visualiza el número completo
   cout.width(3);
   cout << num1 << endl << endl;
   return 0;
}
```

Cuya ejecución sería:

```
Probando texto
        texto
Probando texto
  12345
    678
12345
```

La estructura secuencial

La *estructura secuencial* está formada por una secuencia de instrucciones que se ejecutan en orden una a continuación de la otra. Cada una de las instrucciones están separadas por el carácter punto y coma (`;`). No obstante, en algunos casos nos interesará agrupar en un bloque una serie de instrucciones, como veremos al explicar las estructuras de selección y de iteración. El bloque de sentencias se define por el carácter llave de apertura (`{`) para marcar el inicio del mismo, y el carácter llave de cierre (`}`) para marcar el final.

Ejemplo:

```
{
  instrucción 1;
  instrucción 2;
  instrucción 3;
  .....
  instrucción N;
}
```

Sin embargo, en caso de que el bloque de sentencias este constituido por una única sentencia no es obligatorio el uso de las llaves de apertura y cierre (`{ }`).

Se debe tener siempre presente que según el orden en el que se escriben las instrucciones se produce un resultado u otro. Las mismas instrucciones pero escritas en distinto orden producen resultados distintos. Veamos un ejemplo:

Si intercambiamos las instrucciones en el ejemplo anterior el resultado nunca será el mismo.

```
#include <stdio.h>

int main(){
   int anyo, edad;

   edad = 2016 - anyo;
   printf("¿En qué año naciste? ");
   scanf("%d", &anyo);
   printf( "Durante este año cumplirás %d años \n", edad);
   return 0;
}
```

Acabamos de estudiar que la composición secuencial de instrucciones nos permite crear programas más complejos que aquellos que solo constan de una única instrucción, sin embargo esta estructura tiene una limitación importante: las instrucciones ejecutadas son siempre las mismas y siempre en el mismo orden, independientemente de los valores que puedan ir tomando las variables de nuestro algoritmo. Esta limitación nos impide escribir programas que, por ejemplo, traten de resolver problemas que necesiten de la toma de decisiones para ser resueltos o problemas que necesitan que un conjunto de instrucciones se repitan mientras se den unas determinadas condiciones. Más adelante se analizarán las estructuras alternativa e iterativa que nos permitirán afrontar estas situaciones.

Características deseables de un programa.

Todo estudiante de un curso de *fundamentos de programación*, y en general toda persona que se dedica a la programación debería tener en mente una serie de pautas a seguir a la hora de construir un algoritmo con el fin de conseguir una serie de objetivos básicos. Los objetivos que vamos a considerar son los que se explican a continuación.

Corrección

Un algoritmo siempre debe cumplir su especificación de manera rigurosa. Algunas de las causas que a menudo interfieren en esta labor son:

- La complejidad del propio algoritmo.
- Mala comprensión del problema por parte del programador.
- Descuidos a la hora de escribir el código por parte del programador.
- Una mala especificación del problema original.

Un buen criterio a seguir a la hora de escribir un algoritmo para resolver un problema consiste en empezar con una versión lo más sencilla posible del mismo, pero que sabemos a ciencia cierta que cumple la especificación. A partir de este momento nos podemos dedicar a intentar optimizar dicho algoritmo para intentar que sea lo más eficiente posible, siempre cuidando de que se cumpla su especificación.

Claridad

El texto del algoritmo debe estar escrito de una manera clara y sencilla ya que un algoritmo escrito de manera clara es de gran ayuda, tanto para el propio autor del mismo, como para otras personas que lo puedan leer posteriormente para corregir errores o para modificarlo.

Para conseguirlo podemos seguir estas normas:

- Dividir el texto del algoritmo en partes dotadas de unidad lógica.
- Aprovechar el lenguaje de programación elegido con el fin de expresar lo que queremos decir de la forma más precisa posible, aunque teniendo siempre presente que el código escrito debe ser fácilmente legible.
- Elegir nombres adecuados para las variables, subprogramas que construyamos, etc.
- Hacer un uso adecuado de los comentarios.
- Hacer un uso adecuado de los distintos elementos de composición del texto tales como:
    - Líneas en blanco.
    - Espacios en blanco.
    - Indentado o `*sangrado*' de las líneas.

Todo ello con el fin de hacer patente la relación entre las distintas partes del mismo.

Eficiencia

Posteriormente hablaremos del coste que implica la ejecución de un algoritmo pero sólo en términos del tiempo que emplea en resolver el problema para el que se diseñó. Hay que ser consciente de que cuando un algoritmo se ejecuta tiene unos ciertos requerimientos de memoria. Estos requerimientos se deben a la cantidad de datos que hay que almacenar en memoria principal, así como al propio código del algoritmo, que también se encuentra en la memoria.

Hay algoritmos que son eficientes en cuanto a tiempo de ejecución se refiere a costa de ser tremendamente ineficientes en cuanto a memoria requerida, y al revés, algoritmos que hacen un uso mínimo de la memoria a costa de ser más lentos en su ejecución. Siempre que sea posible se debe llegar a una solución de compromiso entre estos dos tipos de eficiencia, y si no, elegir entre uno u otro dependiendo de:

- el problema concreto que queramos resolver.
- el entorno en el que se va a ejecutar el algoritmo -*memoria y tiempo de CPU disponibles, capacidad de cálculo del computador utilizado*, etc...-.