Bloque 0

Introducción

Prácticas de
Fundamentos de Computadores y Redes
16 de enero de 2017

SESIÓN 1

Introducción a C/C++ para Fundamentos de Computadores y Redes

Objetivos

Esta práctica tiene como objetivo servir de introducción a los conceptos de C/C++ que serán necesarios en la asignatura de Fundamentos de Computadores y Redes. Se presenta una idea general de estos lenguajes, suponiendo que el alumno ya conoce Java. Los conceptos de paso de parámetros por valor y por referencia son los más importantes.

Los conocimientos relativos a C/C++ se utilizarán sobre todo en el tema 4, dedicado al lenguaje de la máquina, y son un prerrequisito para poder hacer el trabajo en grupo. Además, serán muy útiles en asignaturas posteriores de la carrera como Arquitectura de Computadores y Sistemas Operativos.

Conocimientos y materiales necesarios

Para aprovechar adecuadamente esta sesión de prácticas, el alumno necesita:

- Tener conocimientos básicos de programación: concepto de variable, estructura de control, función y paso de parámetros.
- Conocer las estructuras básicas de Java.

Desarrollo de la práctica

1. Introducción

En esta asignatura se estudia cómo hace el computador para ejecutar los programas desarrollados en lenguajes de alto nivel. Python o Java son lenguajes que se ejecutan habitualmente sobre una máquina virtual que utiliza un lenguaje intermedio y entender la relación entre el código fuente en esos lenguajes y lo que ejecuta el computador resulta más complejo que comprenderla cuando se utiliza C/C++, pues no utilizan una máquina virtual. Por otro lado, C/C++ tienen una sintaxis muy parecida a Java (de hecho, Java se inspiró en C/C++), por lo que al nivel que se va a ver en esta asignatura la mayoría de los aspectos sintácticos no van a suponer ningún problema para el alumno. Por último, de los lenguajes que no utilizan

una máquina virtual, el conjunto C/C++ es sin duda en la actualidad el más importante. Por todas estas razones, en esta asignatura usaremos C/C++.

El lenguaje C es un lenguaje imperativo creado por Dennis Ritchie entre 1969 y 1973. En los años 80 del siglo XX Bjarne Stroustrup desarrolló una extensión de C que facilitaba el desarrollo de programas orientados a objetos y la llamó C++. Es prácticamente un superconjunto de C (añade elementos a C), así que los programas de C suelen poder ser traducidos con compiladores de C++, pero no a la inversa.

2. Entorno de programacion Microsoft Visual Studio

En esta asignatura vamos a utilizar el compilador de C/C++ que viene incluido dentro de Visual Studio, un IDE (*Integrated Development Environment*) de Microsoft que también se puede utilizar para desarrollar en otros lenguajes. Existen varias versiones de Visual Studio. La Escuela tiene una licencia DreamSpark de Microsoft que permite que todos los alumnos obtengan legalmente una copia de la versión Professional para desarrollar sus prácticas[1]. Además, Microsoft ofrece una versión gratuita para todo el mundo denominada Visual C++ Express que se puede descargar directamente desde su página.

Visual Studio organiza el desarrollo en lo que denomina «Soluciones» que a su vez están compuestas de «Proyectos», que pueden ser de distintos tipos según el lenguaje, las bibliotecas y las plataformas a los que vayan dirigidos. Vamos a crear un nuevo proyecto:

- ❏ Arranca el Visual Studio. Si pregunta qué entorno de programación quieres utilizar, escoge C++.
- ❏ Selecciona en el menú *Archivo→Nuevo→Proyecto*. Como en esta asignatura se van a utilizar solamente características de C/C++, de entre las plantillas de Win32 de C++, escoge *Aplicación de consola Win32*.
- ❏ Proporciona el nombre `0-1Ejemplo` como nombre de proyecto, cambia la ubicación del proyecto para que esté en un sitio donde la puedas encontrar fácilmente (pero no uses tu lápiz USB ya que la compilación será muy lenta en este dispositivo; copia tu trabajo al lápiz al final de la práctica), deselecciona la opción *Crear directorio para la solución* (nuestra solución sólo va a tener un proyecto, asi que no es necesario) y pulsa *Aceptar*.
- ❏ Aparecerá el asistente para crear una nueva aplicacion de Win32. Pulsa *Siguiente* en la pantalla de introducción.
- ❏ En la pantalla de configuracion de la aplicacion, escoge *Proyecto vacío* y pulsa *Finalizar*.

Con estos pasos tendrás un proyecto vacío. En los siguientes apartados se va a ir añadiendo código y se va a explicar cómo compilar y ejecutar al mismo tiempo que se explican los aspectos más relevantes de C/C++.

[1] Ver https://www.dreamspark.com/

3. Estructura de un programa en C

Los programas en C/C++ se organizan en dos tipos de archivo:

- Archivos de encabezado (o cabecera). Tienen la extensión `.h` (a veces `.hpp` en C++) y se suelen utilizar para definir constantes y tipos de datos (por ejemplo, clases en C++) y declarar prototipos de funciones. El prototipo de una función dice qué devuelve la función, cuál es su nombre y qué argumentos recibe.
- Archivos de código fuente. Tienen la extensión `.c` para programas en C y `.cpp` para programas en C++.

En los archivos de encabezado se colocan elementos que se utilizan en varios ficheros de código fuente. Por ejemplo, una función `CuentaPalabras` que pueden utilizar muchos otros ficheros tendrá su prototipo en un archivo `.h` que incluirán todos los archivos de código fuente que quieran utilizarla. Su implementación estará en un solo archivo de código fuente con extensión `.c`.

No vamos a entrar aquí a explicar cómo organizar correctamente el código C/C++ en varios archivos[2]. Los programas que haremos en la asignatura tendrán todo el código en un solo archivo de código fuente, excepto los elementos que vamos a utilizar para imprimir y que forman parte de la biblioteca estándar de C++: un conjunto de funciones, clases y variables que permiten realizar operaciones comunes como escribir por pantalla, manejar cadenas o archivos.

Vamos a hacer el primer programa:

- ❏ Pulsa con el botón derecho sobre *Archivos de código fuente* en el explorador de soluciones de Visual Studio y escoge la opción *Agregar* y luego *Nuevo elemento*.
- ❏ Dentro de plantillas de *Visual C++*, escoge las de *Código* a la izquierda y selecciona a la derecha *Archivo C++ (.cpp)*.
- ❏ Dale el nombre `Ejemplo` y pulsa *Agregar*.

Verás que se ha creado un nuevo archivo de código fuente, denominado `Ejemplo.cpp`. Visual Studio abre automáticamente el editor para que empieces a programar. Vamos a hacer un programa que imprima en la consola “Hola, mundo". Copia el siguiente código:

```
1 #include <iostream>
2
3 int main()
4 {
5   std::cout << "Hola, mundo";
6
7   return 0;
8 }
```

Como puedes comprobar, el código tiene cierto aire a Java pero con elementos que te serán desconocidos:

[2] Más información en http://geosoft.no/development/cppstyle.html

- La primera línea incluye un fichero de cabecera, `iostream`[3], que contiene el prototipo de los elementos de la biblioteca estándar de C++ para realizar entrada/salida mediante flujos (*input/output stream*, de ahí el nombre). Los `include` cumplen una función similar a los `import` de Java o de Python.
- Todos los programas en C tienen que tener una función `main` y por ella empezará la ejecución del programa. Esto es similar al método público `main` que deben tener los programas de Java. En C, las funciones no tienen por qué estar dentro de una clase. En C++, tampoco, aunque en este lenguaje sí es lo habitual (excepto la función `main`, que no debe estar dentro de una clase).
- Para imprimir se utiliza un flujo estándar denominado `std::cout` (de *character out*). Para enviar elementos al flujo para que se impriman se utiliza el operador `<<`, que envía al flujo de su izquierda lo que tenga a la derecha.

Comprueba que el programa funciona pulsando Ctrl+F5, que compila, enlaza y ejecuta el programa. El resultado es similar a lo que lograrías en Java con un `System.out.println`, excepto en que no hay un salto de línea al final. Para añadirlo, hay que enviar un salto de línea al flujo de salida. Los saltos de línea se obtienen mediante el elemento `std::endl`. Por lo tanto, debes cambiar la línea de impresión a esta:

```
1   std::cout << "Hola, mundo" << std::endl;
```

El `std` que antecede a `cout` y `endl` es el espacio de nombres (*namespace*) de la biblioteca estándar y sirve para evitar conflictos si otra biblioteca decide denominar a un elemento `cout` o `endl`. Como es engorroso añadir el `std` continuamente cuando no hay conflictos, C++ ofrece la posibilidad de indicar que ciertos identificadores serán siempre de un `namespace` utilizando la directiva `using`. Aquí tienes el programa completo usando esta directiva y, a modo de ejemplo de cómo imprimir varios valores, con una línea que imprime números y cadenas:

```
 1 #include <iostream>
 2
 3 using std::cout; // siempre que pongamos cout será igual que std::cout
 4 using std::endl; // siempre que pongamos endl será igual que std::endl
 5
 6 int main()
 7 {
 8   cout << "Hola, mundo" << endl; // No hace falta poner los std
 9   cout << "Entero: " << 3 << " Real: " << 4.5 << endl;
10
11   return 0;
12 }
```

Para leer de consola se utiliza el flujo `std::cin` junto con el operador `>>`, como se muestra en este ejemplo:

```
1   int i;
2
3   cout << "Introduce un entero: ";
4   cin >> i; // Se supone que antes hay un using std::cin;
```

[3] Fíjate que, a pesar de ser un fichero de cabecera, no lleva `.h`. Es un caso especial por motivos históricos.

4. Similitudes y diferencias básicas con Java

Al haberse inspirado Java en la sintaxis de C/C++, hay muchos aspectos comunes, pero también hay muchos diferentes. No se van a tratar aquí todos estos aspectos, sino sólo los más relevantes para la asignatura.

Entre los aspectos comunes, la sintaxis general, incluyendo el uso de punto y coma para separar sentencias, las llaves para delimitar bloques de código y la forma de declarar variables, es la misma. Las estructuras de control básicas (`if`, `while`, `for` y `switch`) son similares (hay pequeñas diferencias que no afectan a lo que vamos a ver aquí).

Los tipos de datos básicos de C/C++ son muy parecidos a los de Java, pero con algunas variaciones. Existen en ambos lenguajes `int` para enteros, `float` para reales y `double` para reales más largos. Una de las diferencias básicas es que en C el tamaño de estos tipos no está definido exactamente por el estándar, asi que puede variar entre sistemas. En la actualidad, es bastante común que `int` sea un entero de 32 bits, como en Java. Existen los modificadores `short` y `long` que, añadidos a `int` (por ejemplo, `short int`), permite obtener enteros con menos o más bits en algunas arquitecturas. Los `float` suelen ser de 32 bits y los `double` de 64 bits, en ambos casos de manera similar a Java.

Otros modificadores que se pueden añadir a los enteros y que Java no posee son `signed` y `unsigned`. El primero permite definir números con signo y el segundo sin signo, de manera que si en una arquitectura los `int` son de 32 bits, `signed int` utilizará estos 32 bits para representar números positivos y negativos entre -2 147 483 648 y 2 147 483 647 (como en Java) y `unsigned int` utilizará los 32 bits para representar sólo números positivos, con lo que se podrán representar números mayores, en concreto, números entre 0 y 4 294 967 295 (el doble de números positivos que en Java). Estos rangos surgen del sistema de codificacion usado, que se estudiará con detalle en la asignatura. Lo habitual si no se indica el tipo de número es que se suponga que es un número con signo.

Para comprobar los problemas a los que puede llevar esto, modifica el código de la función `main` para que sea el siguiente:

```
1  int main()
2  {
3    unsigned int a = 23;
4    int b = -5;
5
6    cout << "a: " << a << " b: " << b << endl;
7
8    if (a < b)
9    {
10     cout  << "a menor que b" << endl;
11   }
12
13   return 0;
14 }
```

Compila y ejecuta el programa. Como verás, se imprime que `a` es menor que `b`, lo que no tiene sentido ya que antes se ha imprimido que `a` vale 23 y `b` vale -5. Las razones detalladas de este comportamiento las podrás comprender tras estudiar la asignatura. En cualquier caso, el compilador te puede ayudar a evitar este tipo de errores: en la ventana *Resultados* situada en la zona inferior, pulsa sobre el icono con la flecha verde a la izquierda (sirve para ir al mensaje anterior) y te llevará a un mensaje de aviso (*warning*) que te informa de que estás comparando variables con tipos que no coinciden. En general, debes considerar los

warnings como errores a no ser que comprendas exactamente por qué se producen y estés seguro de que la situación de la que te avisa el compilador no puede generar un error.

Otra diferencia con respecto a los tipos básicos entre Java y C/C++ es que utilizan sistemas de codificación distintos para los caracteres. En Java, un `char` son 16 bits codificados en Unicode, en concreto en UTF-16 (verás más adelante en la asignatura qué son estos sistemas de codificación). En C/C++ un `char` puede variar en distintas implementaciones pero lo mas habitual es que sea de 8 bits y se utilice un sistema de codificación derivado de ASCII como ISO Latin-1.

En esta asignatura no vamos a utilizar tipos complejos, como por ejemplo clases, pero debes saber que hay bastantes diferencias entre la forma de implementar los tipos complejos en Java y C/C++.

5. Paso de parámetros a funciones

La definición de funciones en C/C++ es similar a Java. Sin embargo, hay diferencias en el mecanismo de paso de parámetros. Como ya deberías saber, en Java los parámetros se pasan siempre por valor. En C también es así. En C++, en cambio, se pueden pasar por valor o por referencia. Cuando se pasan por valor, los cambios dentro de la función no afectan al valor fuera de la función; cuando se pasan por referencia, la función sí puede cambiar el valor que tiene la variable pasada fuera de la función.

Veamos primero un ejemplo de paso por valor:

```
1  #include <iostream>
2
3  using std::cout;
4  using std::endl;
5
6  void pon23(int numero)
7  {
8    numero = 23;
9  }
10
11 int main()
12 {
13   int i = 10;
14
15   pon23(i);
16   cout << "i ahora vale: " << i << endl;
17
18   return 0;
19 }
```

¿Qué valor para `i` crees que imprimirá el código anterior?[1] Introdúcelo en el Visual Studio y comprueba si tu respuesta es correcta. Como verás, una función así no tiene mucho sentido.

[1]

El paso anterior era por valor, al igual que en Java. En C/C++, para pasar por referencia un parámetro se debe utilizar en su definición el símbolo `&` (en español se llama *et*, pero en programación se usa habitualmente con su nombre inglés, *ampersand*). Modifica el programa anterior para que la definición de la función sea la siguiente:

```
1 void pon23(int& numero) {
```

Fíjate que en la declaración de la función el parámetro ahora es `int&`, es decir, una referencia a un entero. Compila el programa y ejecútalo. ¿Qué se imprime para `i`?[2] Como habrás comprobado, ahora la función modifica el parámetro que se le pasa.

2

Para entender la diferencia entre el paso por valor y por referencia, debes de tener claro que cuando se pasa por valor lo que se está haciendo es hacer una copia del valor original. Es decir, en el paso por valor del primer programa de esta sección, al hacer `pon23(i)` se está copiando el valor de `i` en otra zona de memoria, en concreto, en la variable `numero`. Sin embargo, cuando se hace un paso por referencia, no se realiza una copia del valor. En el segundo programa lo que se hace es que la variable `numero` haga referencia a la misma dirección de memoria que `i`. En el tema 4 se explicará esto con más detalle.

6. Vectores

A un nivel básico, los vectores (*arrays*) son similares en Java y C/C++. En Java, los vectores son estructuras de datos que guardan varios elementos del mismo tipo y pueden ser accedidos por un índice entero. En C, además, se asegura que los elementos están contiguos en memoria.

Se pueden definir vectores en C/C++ de varias formas. En la asignatura vamos a utilizar sólo una que reserva memoria para el vector a la vez que lo declara. Para ello, simplemente se debe poner el tipo de los elementos del vector, el nombre del vector y, entre corchetes, cuántos elementos va a tener. Por ejemplo, el siguiente fragmento de código define y reserva memoria para un vector de enteros con cuatro elementos:

```
1 int v[4];
```

Como este tipo de vectores ya tiene memoria asignada, no es necesario, al contrario que en Java, llamar a `new` para reservar memoria. A continuación se muestra un ejemplo de definición y uso:

```
1 int v[4];
2
3 v[0] = 2131; // primer elemento
4 v[3] = 112;  // último elemento
```

También se pueden inicializar los elementos de un vector durante su definición. En este caso, no es necesario indicar entre los corchetes el número de elementos del array: se tomará del número de elementos que se inicialicen. Por ejemplo:

```
1 int v[] = {300, 123, 12}; // array de tres elementos
```

Para pasar un vector a una función, se pone el tipo de los elementos, el nombre del parámetro y los corchetes vacíos, como en este ejemplo:

```
1 void f(int v[]); // la función f recibe un vector de enteros
```

El tipo de vectores de C/C++ explicado aquí no tienen la propiedad `length` que tienen los vectores en Java. Por lo tanto, cuando se pasa un vector a una función, es habitual pasarle en otra variable el tamaño, como en este ejemplo:

```
1 void f(int v[], int longV);
```

Este tipo de vectores se pasan siempre por referencia.

7. Cadenas

En C las cadenas son vectores de caracteres que acaban con un carácter especial denominado `null`, que se suele representar como `\0` y tiene el código todo ceros. Es necesario que el vector de caracteres donde se almacena la cadena tenga una posición para almacenar este carácter.

Cuando se escribe una cadena entre comillas en el código, el compilador añade automáticamente este carácter terminador. Por ejemplo:

```
1 char cad[] = "abc"; // Vector de cuatro posiciones con estos valores:
2                     // cad[0]='a' cad[1]='b' cad[2]='c' cad[3]=\0
```

Para imprimir cadenas por consola, se hace como se ha visto mediante `cout` y el operador <<. Para leer cadenas de la consola, sin embargo, no se puede utilizar directamente `cin` con el operador >>, ya que este operador no funciona sobre el tipo de vectores que estamos utilizando[4]. Una posibilidad sería ir leyendo carácter a carácter en un bucle. Afortunadamente, el objeto `cin` ofrece una función para leer cadenas: `getline`, que recibe la cadena en la que debe dejar la secuencia de caracteres y el máximo número de caracteres que puede tener la cadena (incluyendo el terminador). Por ejemplo:

```
1 const unsigned int maxCad = 100; // const sirve para definir constantes
2 char cad[maxCad];
3 cin.getline(cad, maxCad); // lee hasta 99 caracteres de la consola cuando
4                           // se pulse return. Deja en la cadena los caracteres
5                           // leídos y añade un terminador
```

8. Ejemplo completo

Como ejemplo de todo lo visto hasta ahora, se va a hacer un programa que define e inicializa dos vectores de enteros y luego llama a una función que compara los elementos con el mismo índice de cada vector y, si son distintos, modifica el elemento del segundo vector para que tenga la suma de los dos elementos originales. Además, la función indica

[4] C++ define una clase String que sí permite utilizar el operador «. Sin embargo, no la utilizamos en la asignatura porque comprender cómo funciona a bajo nivel es muy complejo y, además, es fundamental entender cómo funcionan las cadenas de C ya que las utilizan el conjunto de funciones para desarrollar aplicaciones (API, de *Application Programming Interface*) que ofrecen los sistemas operativos.

cuántos elementos ha modificado. El programa principal utiliza una función para imprimir el nombre del vector y sus elementos.

Descarga del Campus Virtual el fichero `0-1EjemploVector.zip`. Descomprímelo y abre el fichero de solución en Visual Studio. Lee el código, incluyendo los comentarios, y complétalo para que funcione. Compílalo y ejecútalo para comprobar que está todo bien.

9. Para saber más: punteros

Una de las grandes ventajas de C/C++ es que incorpora un sistema de punteros que permite acceder directamente a posiciones de memoria, lo que es fundamental para trabajar a bajo nivel y también para desarrollar estructuras con un alto rendimiento. Como contrapartida, es fácil utilizar incorrectamente esta potencia y generar errores con consecuencias graves.

Un puntero es una variable que contiene una dirección de memoria. Se dice que el puntero apunta a lo que hay en la zona de memoria de la dirección que contiene. En C/C++, cuando se declara un puntero hay que indicar de qué tipo es lo que hay en la dirección a la que se apunta. Por ejemplo, de esta forma se declararían un puntero para apuntar a un `int` y otro para apuntar a un `float`:

```
1 int* pI; // pI es un puntero a un int
2 float* pF; // pF es un puntero a un float
```

El asterisco después del tipo en la declaración es lo que indica que es un puntero. En general, se desaconseja definir varios punteros en la misma línea porque la sintaxis no es intuitiva:

```
1 int* a, b; // a es un puntero a un int, b es un int (no es un puntero)
2 int* c, * d; // tanto c como d son punteros a int. Desaconsejado
```

Una de las formas de uso de los punteros más habitual es utilizarlos para guardar la dirección de otra variable. Para obtener la dirección de una variable se utiliza el simbolo `&`. Por ejemplo:

```
1 int* p; // p es un puntero a entero
2 int numero = 35; // numero es un entero
3 p = &numero; // copiar a p la dirección de numero. p ahora apunta a numero
```

La expresión `&i` se lee «dirección de i». Otra forma alternativa de decir que un puntero apunta a algo es decir que referencia ese algo. Los punteros se pueden desreferenciar, es decir, obtener lo que hay en la memoria a la que apuntan. Por ejemplo:

```
1 int* p; // p es un puntero a entero
2 int numero = 35; // numero es un entero
3
4 cout << "numero: " << numero << endl;
5
6 p = &numero; // copiar a p la dirección de numero. p ahora apunta a numero
7 *p = 555; // poner 555 en la dirección a la que apunta p, es decir en numero
8
```

```
9  cout << "numero: " << numero << endl;
10 cout << "*p: " << *p << endl;
```

Ejecuta el código anterior, y comprueba que accediendo a la zona de memoria de la variable `numero` con su nombre o con `*p` se muestra lo mismo. Si ejecutases esta instrucción a continuación:

```
1 numero = 101;
```

¿Cuánto valdría `*p`?[3] Comprueba tu respuesta añadiendo la instrucción anterior e imprimiendo después el valor de `*p`.

[3]

Fíjate que el operador asterisco se utiliza en dos contextos relacionados con punteros de manera muy distinta: en la definición de variables (por ejemplo, `int* p`) indica que se está declarando un puntero y en la desreferenciación (por ejemplo, `*p =555`) indica que se quiere acceder al valor que hay donde apunta el puntero.

Uno de los problemas de los punteros es que pueden no estar inicializados. Por ejemplo, este programa es incorrecto:

```
1 int* p; // p es un puntero a entero
2 *p = -3400; // ERROR: no se sabe a dónde apunta p
```

Como `p` no se ha inicializado, puede contener cualquier dirección. Al escribir en `*p` ¡estamos escribiendo en cualquier dirección! Eso puede tener consecuencias desastrosas: imagínate que en un programa de un banco justo en esa dirección está lo que tiene un cliente en la cuenta; con esa línea, pasaría a tener -3400 euros, tuviese lo que tuviese antes. Probablemente, al cliente no le haría gracia... a no ser que tuviese una cantidad inferior, en cuyo caso al que no le haría gracia sería al banco.

Pero también puede que el *valor basura* que contiene `p` antes de ser inicializado coincida con una dirección del sistema operativo y, por lo tanto, escribas un -3400 una zona del sistema operativo. Como verás en la asignatura de Arquitectura de Computadores en 2º curso, existen mecanismos para proteger al sistema operativo que hacen que falle el programa al intentar acceder a estas direcciones en lugar de hacer fallar a todo el sistema.

A veces interesa definir un puntero a una zona de memoria sin saber qué se va a almacenar en esa zona de memoria, es decir, interesa poder tener un «puntero a cualquier cosa». En C/C++ eso se logra utilizando el tipo `void*`. Los punteros de este tipo no se pueden desreferenciar sin hacerles antes un *casting*, como muestra este ejemplo:

```
1 void* p; // p es un puntero a cualquier cosa
2 int i = 23;
3 float f = 55.2;
4 p = &i;
5 int i2 = *((int*) p);
6 p = &f;
7 float f2 = *((float*) p);
```

Como ves, la sintaxis empieza a ser engorrosa. Para entender el `*((int*) p)` debes pensar que se parte de `p`, que es un `void*` (puntero a cualquier cosa) y se le hace un casting a `int*`, es decir, `((int*) p)` es un puntero a entero; el asterisco que se añade a la izquierda en `*((int*) p)` quiere decir desreferenciar `((int*) p)`, es decir, ir a la dirección que contiene

`p`, suponer que ahí hay un entero y obtener el valor de ese entero. Con `*((float*) p)` se hace lo mismo pero suponiendo que `p` está apuntando a un número en coma flotante.

Estos conceptos de punteros son complejos. Precisamente el estudio de cómo funciona el computador a bajo nivel que se hace en esta asignatura te permitirá entenderlos mucho más fácilmente.

10. Para saber más: printf

Como se vio anteriormente, en C++ se puede imprimir por pantalla mediante `cout`. Esto no funciona en C, donde la forma estándar de hacerlo es utilizando la función `printf` que está incluida en el fichero de cabecera `stdio.h`, que contiene el prototipo de las funciones de entrada/salida de la biblioteca estándar de C (*standard input/output*, de ahí el nombre). Por ejemplo, para imprimir `Hola, mundo`, se haría así:

```
1 printf("Hola, mundo\n");
```

El `\n` final produce un salto de línea, ya que `printf` no produce un salto de línea por defecto. No se puede utilizar `endl` para este cometido porque también es una característica de C++.

Debido a la forma de manejar las cadenas en C, imprimir varios valores es más complejo que en Java y en C++. No se puede utilizar el operador `+` para concatenar cadenas ni el operador `<<` para encadenar salidas por pantalla. La forma de imprimir varios valores es utilizando varios parámetros en la función `printf`. El primero es una cadena, denominada cadena de formato, que puede incluir unos indicadores de elementos que se sustituirán por los valores de los siguientes parámetros. La mejor forma de entenderlo es con un ejemplo:

```
1 printf("Entero1: %d. Entero2: %d. Flotante: %f\n", 23, 100, 45.3);
```

Si añades esto a un programa, lo compilas y lo ejecutas (recuerda que también debes incluir `stdio.h`), comprobarás que el primer indicador `%d` se ha sustituido por el segundo parámetro, el segundo indicador `%d` se ha sustituido por el tercer parámetro y el indicador `%f`, por el cuarto. Existen indicadores para distintos tipos de datos. En Internet puedes encontrar más información sobre `printf`. Es interesante conocerla porque la sintaxis de cadenas de formato ha pasado a muchas funciones de otros lenguajes. En Java, por ejemplo, está `String.format` y en Python se utiliza con el operador de cadenas `%`.

11. Ejercicios adicionales

- ✏ Crea un programa que defina dos vectores de 6 enteros. Inicialízalos con valores aleatorios, la mitad positivos y la mitad negativos. El programa debe contener una función que reciba un vector de enteros y su longitud y ponga a cero los elementos negativos. El programa principal debe llamar a esta función para los dos vectores. Deben imprimirse los vectores antes y después de modificarlos.
- ✏ Crea un programa que defina dos cadenas e imprima el número de vocales y el número de consonantes que tienen. Para ello debe utilizar una función que realice el cálculo.