Programación 2

Presentación

Este libro contiene los apuntes de la asignatura Programación 2 de la Universidad de Alicante.

Autores: Antonio Pertusa, David Tomás, Carlos Pérez, Jaume Aragonés, Juan Antonio Pérez y Francisco Moreno.

Tema 1: Introducción

En este primer tema vamos a ver un repaso de los contenidos ya vistos en Programación 1, añadiendo algunos conceptos nuevos sobre diseño de algoritmos y programas, metodología de programación y sintaxis de C++. Te servirá para refrescar conocimientos y coger el ritmo para empezar la asignatura con buen pie antes de meternos con temas más serios. Si tuviste problemas con Programación 1, éste es el momento de ponerse al día.

Diseño de algoritmos y programas

Para hacer un programa es necesario crear uno o varios ficheros de código fuente escritos en un lenguaje de programación, que en el caso de Programación 2 es C++. Una vez tenemos el código, mediante un compilador podemos transformarlo en un programa ejecutable que es capaz de interpretar el ordenador.

Para la realización de un programa, antes de sentarte delante del ordenador a escribir código debes analizar detenidamente los requerimientos (¿qué tiene que hacer el programa exactamente?) y pensar en el diseño del algoritmo. Más concretamente, el desarrollo de un programa implica llevar a cabo las siguientes fases:

1. Estudio de los requerimientos del problema
2. Diseño del algoritmo (se puede hacer en papel)
3. Escritura del código fuente en el ordenador
4. Compilación del programa y corrección de errores
5. Ejecución del programa
6. Prueba de todos los casos posibles (o casi)

Es muy posible que en alguna de estas fases nos demos cuenta que arrastramos errores de alguno de los pasos anteriores y nos toque retroceder en el proceso. Por ejemplo, que al compilar el programa (fase 4) se produzca algún error y debamos volver escribir código (fase 3) para corregirlo, o que al ejecutar (fase 5) nos demos cuenta de que el programa no hace lo que debería y debamos incluso plantearnos un rediseño del

mismo (fase 2). Por esta razón es muy importante dedicar el tiempo necesario a las primeras fases, ya que

El proceso de escribir, compilar, ejecutar y probar debería ser iterativo, haciendo pruebas de funciones o módulos por separado del programa: no esperes a tener todo el programa escrito para compilar y probarlo.

Vamos a ver a continuación cada una de estas fases en más detalle.

1. Estudio de los requerimientos

Para empezar a crear un programa hay que tener claros los requerimientos, es decir, "qué" debe hacer. En algunos casos es sencillo; por ejemplo, si queremos hacer un programa para mostrar una cuenta atrás por pantalla. Sin embargo, en otros casos es mucho más complicado; por ejemplo, si queremos hacer un programa de contabilidad para una empresa. En estas situaciones hay que entrevistarse con los clientes para tener claras sus necesidades (trabajo que realiza habitualmente el *analista de sistemas*).

Como acabamos de indicar, antes de comenzar a diseñar el programa es necesario tener un listado de requerimientos con todas las opciones de nuestro programa. Por ejemplo, para hacer un juego como *Angry Birds* podríamos tener los siguientes requerimientos:

- Tendremos varios niveles ordenados por dificultad
- En cada nivel tendremos cerdos, pájaros y otros objetos con los que interactúan
- El usuario podrá lanzar un pájaro con un tirachinas para acabar con los cerdos
- Un cerdo se destruye si un pájaro o un objeto impacta contra él
- Podemos tener varios tipos de pájaros
- El usuario puede tocar un pájaro en vuelo para hacer acciones especiales
- ...

Como ves, la lista de requerimientos de un programa puede ser muy larga. Los enunciados de prácticas de Programación 2 resumen los requerimientos que tiene que cumplir la práctica, indicando de forma clara (o eso intentamos) todo lo que debe hacer el programa y cómo debe interactuar el usuario con él.

2. Diseño del algoritmo

Una vez tenemos claros los requerimientos, tenemos que pensar "cómo" vamos a implementar nuestro programa. Esto se hace en la fase de diseño, incluyendo tareas como detectar los tipos datos necesarios, decidir las funciones para trabajar con ellos y establecer el flujo del programa.

La fase de diseño es muy importante y se puede (te lo recomendamos) llevar a cabo en papel. Para esto, se debe pensar primero en qué datos tenemos (en el ejemplo anterior: pájaros, cerdos, objetos, niveles, paisajes de fondo, ...) y qué vamos a hacer con cada uno de ellos (lanzar un pájaro, impactar con un cerdo, tocar un pájaro, ...) para decidir las funciones que tendrá nuestro programa.

En teoría, a partir de un buen diseño ya puedes escribir todo el código del programa sin necesidad de volver a esta fase en ningún momento. En la realidad, para programas complejos es probable que, a pesar de que el diseño sea bueno, tengas que rehacer algunas partes. Por ejemplo, puedes hacer una aplicación para el teléfono móvil y darte cuenta de que un botón no queda del todo bien en un determinado sitio y que eso te obligue a cambiar el diseño (en este caso de la interfaz).

En cualquier caso, como ya se dijo más arriba, cuanto mejor sea el diseño del programa más sencillo será avanzar en las siguientes fases, por lo que es (muy) recomendable invertir el tiempo que sea necesario para conseguir un buen diseño de partida.

3. Escritura del código fuente

Una vez tenemos claro cómo vamos a hacer el programa, procederemos a escribir el código fuente. Para ello puedes usar un editor de textos cualquiera (vale hasta el *Bloc de notas*, aunque no te lo recomendamos) o alternativamente un entorno de desarrollo integrado (*Integrated Development Environment* - IDE), como puede ser Eclipse o Visual Studio Code.

Es recomendable no escribir mucho código de golpe (por ejemplo, más de 50 líneas). Mejor escribir unas pocas líneas de código (o una función) y compilar, para evitar que se produzcan muchos errores al mismo tiempo que haga difícil localizarlos y solucionarlos. Tras arreglar los posibles errores de compilación y probar que el código hace lo esperado, podemos seguir escribiendo.

4. Compilación y corrección de errores

El compilador interpreta o convierte nuestro código fuente en un programa que el ordenador puede ejecutar. Como ya se ha comentado, lo normal tras escribir código es que tengamos algunos (o muchos) errores de compilación. Estos errores hay que corregirlos, reescribiendo el código y compilando hasta que no se produzcan más errores. Mientras haya errores de compilación no se generará el programa ejecutable.

Hay que distinguir dos tipos de errores cuando se realiza la compilación: (i) los errores de compilación propiamente dichos, que impiden que se genere el programa ejecutable; (ii) las advertencias (*warnings*), que sí permiten obtener el fichero ejecutable pero que nos avisan de que puede que haya algo mal. Es conveniente arreglar todos los *warnings*, ya que indican un problema que puede acabar produciendo errores durante la siguiente fase de ejecución del programa. Un ejemplo de *warning* se produciría si hemos declarado una variable sin inicializar y decidimos llevar a cabo alguna operación con ella, como usarla de índice en un vector.

5. Ejecución del programa

Una vez tenemos nuestro código fuente compilado, debemos ejecutarlo para ver que el programa generado hace lo esperado. A veces no es así, en cuyo caso deberemos volver a la fase de diseño, reescribir código, compilar y volver a ejecutar.

6. Prueba

Tras ejecutar el programa ves que todo funciona correctamente pero, ¿es así en todos los casos? ¿Qué ocurre si, por ejemplo, el usuario introduce por teclado un valor incorrecto? ¿Sigue funcionando todo bien?

Es importante que pruebes tu programa con todos los casos posibles que se puedan dar durante su ejecución. Esta no es una tarea fácil, pero hay algunos trucos que pueden ayudarte.

Por ejemplo, supongamos que tenemos la siguiente función que calcula la división entre dos números enteros:

```
1 float division(int a,int b){
2   float resultado=a/b;
```

```
  return resultado;
}
```

Si compilamos y ejecutamos este código, nos daremos cuenta de que hay dos errores. ¿Puedes verlos? El primero de ellos se identifica rápidamente cuando hacemos la llamada a la función:

```
cout << division(3,4) << endl;
```

El resultado que muestra por pantalla este código es `0`, ya que `division` está realizando una división entera, cuyo resultado es un número entero, eliminando por tanto la parte decimal (no es lo que queríamos). Podemos arreglarlo fácilmente forzando a que el primer operando sea de tipo `float`:

```
float resultado=(float)a/b;
```

De esta manera, en cuanto uno de los dos operandos es de tipo `float` el resultado será un número real.

El segundo error es más difícil de localizar. Para cada función debemos considerar todos los posibles valores de sus parámetros de entrada y ver si la salida es correcta con ellos. En este caso, los parámetros `a` y `b` podrían ser positivos, negativos o cero. Si alguno de estos dos parámetros fuera negativo, ¿funcionaría el código? Sí, no habría problema. ¿Y si alguno de los parámetros fuera cero? En este caso tendríamos un problema, ya que no se pueden hacer divisiones por cero y si `b` podría tener ese valor.

Para que la función fuera correcta tendríamos que hacer algunos cambios:

```
float division(int a,int b){
  float resultado=0;
  if(b!=0){
    resultado=(float)a/b;
  }
  else{
    cout << "Error: ¡no se permiten divisiones por cero!" << endl;
  }
  return resultado;
}
```

Elementos básicos de C++

Tras ver cómo es el ciclo de desarrollo de un programa, vamos a centrarnos ahora en el lenguaje C++, que es el que usaremos en Programación 2. Todo lo que necesitas saber de C++ en esta asignatura te lo contarán tus profesores en las clases de teoría y prácticas. No obstante, si quieres ampliar tus conocimientos más allá de lo que verás en la asignatura, existen infinidad de libros y páginas web sobre C++ a los que puedes acudir. Una muy buena referencia del lenguaje la puedes encontrar en la web cplusplus.

En un código fuente podemos encontrarnos distintos elementos básicos:

- *Identificadores*: Nombres de variables, funciones, constantes, etc.
- *Constantes*: `123`, `12.3`, `'a'`, ...
- *Palabras reservadas*: `if`, `while`, ...
- *Símbolos*: `{}`, `()`, `[]`, `;`, ...
- *Operadores*: `++`, `--`, `+`, `*`, `/`, ...
- *Tipos de datos*: `int`, `char`, `float`, `double`, `bool`, `void`, ...

Por ejemplo, en este código:

```
1 int main(){
2   for(int i=0;i<10;i++){
3     cout << "Hola mundo" << endl;
4   }
5 }
```

encontramos los siguientes elementos básicos:

- Identificadores: `i`
- Constantes: `0`, `10`, `"Hola mundo"`
- Palabras reservadas: `for`, `main`, `std::cout`, `std::endl`
- Símbolos: `(`, `)`, `;`
- Operadores: `=`, `<`, `++`, `<<`
- Tipos de datos: `int`

En las siguientes secciones vamos a ver en más detalle cada uno de estos elementos.

Vamos a ver en detalle cada uno de estos elementos.

Identificadores

Los identificadores son los nombres que le damos a nuestras variables, constantes o funciones. Los eliges tú como programador y deben seguir una serie de recomendaciones.

En primer lugar, los identificadores deben ser **significativos**, es decir, su nombre debe indicar para qué se utiliza. Estos son ejemplos de una variable y una función con nombres significativos:

```
1 int numeroAlumnos=0;
2 void visualizarAlumnos(...)
```

Por convenio, en C++ se suele seguir la notación *lowerCamelCase*, es decir, el nombre de las variables y funciones debe empezar por una letra minúscula y, si el nombre se compone de más de una palabra, la primera letra debe ser mayúscula como en el ejemplo anterior.

Estos son ejemplos de nombres no significativos y que deberías evitar usar:

```
const int OCHO=8; // No se debe llamar a una constante con su valor
int p,q,r,a,b; // Una sóla letra no es significativa
int contador1,contador2; // mejor int i,j;
```

También por convenio, en C++ los contadores empiezan por la letra *i*. Por tanto, si tenemos varios contadores deberíamos llamarlos `i`, `j`, `k`, ...

Además, existen palabras reservadas por el lenguaje que no se pueden utilizar como nombres definidos por el usuario. Por ejemplo, en C++ no podemos llamar a una variable con el nombre `int`, `long`, `friend` o `for`, porque se produciría un error de compilación al ser palabras reservadas por el lenguaje:

```
int friend=10; // Dará un error de compilación
```

El nombre de constantes se suele poner en mayúsculas, para distinguirlas de las variables.

```
const int MAXALUMNOS=100;
```

Variables

Las variables pueden usarse para almacenar valores que cambien durante la ejecución del programa. Permiten almacenar diferentes tipos de datos, cuyo tipo se debe indicar cuando se declara.

Estos son los principales **tipos básicos** (o **primitivos**) de datos en C++:

Tipo	Descripción	Tamaño (en bits)
`int`	Número entero	32
`char`	Carácter	8
`float`	Número real	32
`double`	Número real de doble precisión	64
`bool`	Booleano	8
`void`	No es un tipo	-

> Realmente, el número de bytes necesarios cada tipo de dato depende de la plataforma (por ejemplo, si es sistema operativo es de 32 bits o 64 bits).

Además, para los números enteros tenemos su versión sin signo (`unsigned int`). Un `int` puede almacenar valores comprendidos entre -2.147.483.648 y 2.147.483.647. Si declaramos la variable como `unsigned int` no podremos almacenar números negativos, pero el rango de números positivos será el doble, pudiendo almacenar valores entre 0 y 4.294.967.295.

Es muy recomendable que siempre que se declare una variable de tipo simple (por ejemplo `int`, `char`, `float`, `double` y `bool`) se le asigne un valor de inicialización, bien en la misma línea o bien en la siguiente. Por ejemplo:

```
int numeroProfesores=0; // Inicialización en la misma línea

int numeroAlumnos;
numeroAlumnos=10; // Inicialización en la siguiente línea
```

Si no se inicializa una variable, su valor será el que haya en memoria en ese momento, es decir, cualquiera (no podemos controlarlo). Este código sería problemático:

```
int i;
cout << i << endl; // No sabemos qué valor mostrará
```

Ámbito

Todas las variables (y constantes) que declaramos en nuestro código tienen un ámbito. El ámbito de una variable o constante comienza cuando se declara y termina cuando acaba el bloque de llaves que la contiene.

Sólo podemos usar las variables o constantes en su ámbito:

```
if(i<10){
  int j=20;
}

j=10; // No podemos usar la variable j aquí
      // Su ámbito ha terminado y se ha destruído al finalizar el if
```

Ejemplo para un bucle:

```
for(int i=0;i<10;i++){
  cout << "Hola mundo" << endl;
}

i=2; // No podemos usar la variable i porque ya se ha destruído
```

Podemos declarar (aunque no es conveniente) dos variables que se llamen igual dentro de la misma función, siempre que tengan un ámbito distinto. Por ejemplo, este programa compilaría sin problema:

```
int numCajas=0; // Ya se puede usar numCajas

if(i<10){

   // numCajas se puede usar aquí
  int numCajas=100; // Mismo nombre pero distinto ámbito
  cout << numCajas << endl; // Muestra "100"
```

```
}
cout << numCajas << endl; // Muestra "0", ya que se usa la primera variable
```

Variables globales

Las variables normalmente se declaran dentro de una función (y por tanto su ámbito se reduce a esa función), pero cuando se declaran fuera se llaman variables globales y son accesibles desde cualquier función del programa. En Programación 2 está prohibido utilizar variables globales, ya que pueden producir efectos colaterales no deseados si no se usan de manera adecuada.

Vamos a ver esta problemática con un ejemplo de código:.

```
#include <iostream>

using namespace std;

int contador=10; // Variable global

void cuentaAtras(){
  while(contador>0){
    cout << contador << " ";
    contador--;
  }
  cout << endl;
}

int main() {
  cuentaAtras();   // Muestra "10", "9", "8", ... "0"
  cuentaAtras();   // Aqui no muestra nada
}
```

En este ejemplo, tenemos una variable global llamada `contador` que inicialmente vale 10. Podemos ver que si llamamos a la función `cuentaAtras`, la primera vez hará una cosa y la segunda otra distinta. Este código es complicado de controlar, ya que tenemos que una misma función con los mismos parámetros se comporta de forma distinta dependiendo de en qué punto del código hagamos la llamada.

Constantes

En C++ podemos tener constantes de diferentes tipos:

Tipo	Ejemplos
`int`	`123`, `007`, `-4`
`float`	`123.0`, `-0.4`, `.3`, `1.23e-12`
`char`	`'a'`, `'1'`, `';'`, `'\''`
`char[]`	`"hola"`,`""`,`"doble: \""`

`bool`	`true`, `false`

Cuando las constantes aparecen directamente en el código con su valor, como en el siguiente ejemplo, se dice que son **implícitas**:

```
if(i<255){ // 255 es una constantes implícita
  cout << "Valor correcto" << endl;
}
```

Sin embargo, si las declaramos con un nombre que las identifique, se dice que son **explícitas**:

```
const int MAXVALUE=255; // MAXVALUE es una constante explícita
const char MESSAGE[]="Valor correcto"; // MESSAGE es también explícita

if(i<MAXVALUE){
  cout << MESSAGE << endl;
}
```

Podemos declarar constantes explícitas de cualquier tipo:

```
const int MAXALUMNOS=600; // Tipo int
const double PI=3.141592; // Tipo double
const char DESPEDIDA[]="ADIOS"; // Tipo vector de caracteres
```

¿Cuándo debemos declarar constantes explícitas? La respuesta es "siempre que podamos", ya que deberíamos declarar como constantes aquellos valores que podríamos querer cambiar en futuras versiones del programa.

La principal ventaja de usar constantes es que normalmente en el código tenemos que usar un mismo valor muchas veces (por ejemplo, un mensaje de error). Si lo declaramos como una constante explícita y queremos cambiar su valor en un futuro, sólo tendríamos que hacer este cambio en la declaración. Si no lo declaramos como constante, tendríamos que buscar en el código todas las líneas en las que aparece este mensaje para cambiar su valor, lo cual es laborioso y propenso a errores.

Tipos de datos

En ocasiones podemos querer convertir una variable de un tipo a otro tipo. Las conversiones entre tipos pueden ser **implícitas** (el compilador lo hace por nosotros sin que tengamos que hacer nada especial) o **explícita** (tenemos que indicarle al compilar que lo haga).

A continuación podemos ver algunos ejemplos de conversiones **implícitas** hechas por el compilador:

Tipo origen	Tipo destino	Ejemplos
`char`	`int`	`int num = 'A' + 2;` // num == 67

`int`	`float`	`float pi = 1 +`
`float`	`double`	`double piDouble = pi;`
`bool`	`int`	`int c = true;` // c == 1
`int`	`bool`	`bool c = 77212;` // b == true

En estos ejemplos anteriores no hay problemas de conversión, por que la variable destino es más general que la variable original. Por ejemplo, en un `int` (4 bytes) siempre podremos almacenar un `char` (1 byte). Sin embargo, hay otros casos en los que haciendo la conversión podemos perder información. En estas situaciones el compilador mostrará una advertencia (*warning*) y tendremos que forzar la conversión en el código, haciendo lo que se conoce como *casting*, poniendo el tipo de dato destino entre paréntesis.

A continuación se muestran algunos ejemplos de conversiones **explícitas**:

Tipo origen	Tipo destino	Ejemplos
`int`	`char`	`char c = (char)('A' + 2);` // c == 'C'
`float`	`int`	`int enteroPi = (int)pi;` // enteroPi == 3
`double`	`float`	`float d = (double)pi;`

Si tratamos de convertir un número entero a un carácter podríamos tener problemas (si es mayor de 255 no nos cabría en un byte). Por tanto, el compilador nos da un *warning* para que lo tengamos en cuenta. Para indicar explícitamente que queremos convertir un tipo a otro incluso si puede haber problemas tenemos que hacer un *casting*, como hemos visto en los ejemplos anteriores.

Definición de nuevos tipos

En C++, como en la mayoría de lenguajes de programación, se pueden definir tipos de datos nuevos. Para empezar, podemos asignar un alias a un tipo de dato ya existentes usando `typedef`:

```
1 typedef int entero;
2 entero i,j; // i,j son de tipo int
3
4 typedef bool logico,booleano;  // logico y booleano son de tipo bool
5 logico b; // b es de tipo bool
```

También podemos declarar un array como un nuevo tipo de dato con `typedef`:

```
1 typedef char cadena[50]; // cadena es un array de char de 50 elementos
```

En Programación 2 no recomendamos redefinir tipos de datos ya existentes, básicamente porque un

programador de C++ no está acostumbrado a ver tipos de datos que se llamen *logico* o *cadena* en el código. Piensa qué pasaría si vieras un código lleno de tipos de datos que no conoces: tendrías que ir a la definición de esos tipos para saber qué significan en cada caso. Esto resulta muy poco cómodo a la hora de compartir y reutilizar código.

Además, en C++ los nombres que aparecen después de `struct`, `class` y `union` son también tipos.

Operadores de incremento y decremento

Los operadores `++` y `--` se usan para incrementar o decrementar el valor de una variable de tipo entero. Podemos usarlos antes o después del nombre de la variable:

```
1 ++i; // Preincremento
2 i++; // Postincremento
```

La diferencia entre ponerlos antes o después es importante, ya que el resultado no siempre es el mismo. Si van en una línea como en el ejemplo anterior, el resultado es idéntico. Sin embargo, cuando van acompañadas de otras instrucciones, el resultado cambia:

```
1 int i=3;
2 int k=++i; // k vale 4, i vale 4
3
4 i=3;
5 int j=i++; // j vale 3, i vale 4
```

Como puedes ver, con preincremento (o predecremento) la suma se hace **antes** de calcular el resto de la expresión. En cambio, con postincremento (o postdecremento) se hace **al final**. En el primer caso, primero se hace el incremento (`++i`) y luego se asigna este valor a `k`. Sin embargo, en el segundo caso se hace la asignación (`j = i`) y al finalizar se incrementa el valor de `i`.

Aunque se pueden utilizar en cualquier punto de una expresión, lo recomendable es no mezclar estos operadores en la misma instrucción, ya que es muy complicado predecir el resultado. Por ejemplo, ¿cuál sería el valor de `j` en este caso?:

```
1 i = 3;
2 j = i++ + --i;  // valor de j?
```

Expresiones aritméticas

Las expresiones se usan cuando queremos hacer cálculos y evalúan una o varias operaciones devolviendo un resultado. En una expresión tenemos operadores y operandos, como puede verse en el siguiente ejemplo:

```
1 int i=10+12; // Expresión con operadores 10 y 12 y operando +
```

En C++ los principales operadores aritméticos son la suma (`+`), la resta (`-`), la multiplicación (`*`), la

división (`/`) y el el resto o módulo (`\%`).

Si en una expresión aritmética aparece un operando de tipo `char` o `bool`, se convierte implícitamente a `int`. Por ejemplo:

```
1 int i='a'+3; // i == 100
2 char c='A'+3; // c == 'D'
```

Cuando hacemos una división entre dos números enteros con el operador `/`, el resultado es un número entero. Por ejemplo:

```
1 float f1=7/2; // f1 == 3
```

Si queremos que el resultado sea un número real, uno de los dos operandos deberá ser real. En el ejemplo anterior, podemos hacer un *cast* de uno de los operandos para conseguirlo:

```
1 float f2=(float)7/2; // f2 == 3.5
```

En cuanto uno de los dos operandos es `float` (o `double`), el resultado de la división es un número real, como puede verse en el ejemplo anterior.

El operador `%` devuelve el resto de una división entre dos números enteros:

```
1 int resto=30%7; // resto == 2
```

En las expresiones aritméticas hay operadores que tienen preferencia sobre otros. En concreto, los operadores `*` y `/` se evalúan antes que los operadores `+` y `-`:

```
1 int num=2+3*2; // Se evalúa 2+6, ya que la multiplicación se hace antes
```

En caso de que haya varias operaciones en una expresión es recomendable usar paréntesis para evitar problemas:

```
1 int num=2+(3*(7/2.5));
```

Expresiones relacionales

Las expresiones relacionales evalúan una operación devolviendo únicamente `true` o `false`. Los principales operadores relacionales en C++ son `==`, `!=`, `>=`, `>`, `<=` y `<`.

Si los operandos son de distinto tipo, se convierten implícitamente al tipo más general:

```
1 bool resultado=2<3.4; // Internamente la operación es 2.0 < 3.4
```

En C++ los operandos se agrupan de dos en dos por la izquierda. Por tanto, la expresión `a < b < c` deberá realizarse de la siguiente manera:

```
if(a<b && b<c){
  // Se cumple la relación
}
```

Expresiones lógicas

Las expresiones lógicas evalúan una expresión de tipo lógico devolviendo `true` o `false`. Los principales operadores lógicos en C++ son `!`, `&&` y `||`. La negación (`!`) devuelve `true` si lo que viene a continuación es `false`, o `false` si lo que viene a continuación es `true`. Por ejemplo:

```
int i=2;
bool salir=false;

if(i==2 && !salir){
  // Si la condición es verdadera entra aquí
}
```

Hay una característica de C++ que es importante conocer: la **evaluación en cortocircuito**. Cuando tenemos un operador `||`, si el operando izquierdo es `true` el operando derecho no se llega a evaluar, ya que `true || x` siempre será `true`. Por ejemplo:

```
int i=2;
bool salir=false;

if(i==2 || !salir){
  // La variable salir no llega a comprobarse, ya que se cumple que i==2
}
```

Asimismo, cuando tenemos un operador `&&`, si el operando izquierdo es `false`, el operando derecho no se llega a evaluar, ya que `false && x` siempre será `false`. Por ejemplo:

```
char v[]="Hola mundo";
char letraBuscada='k';

for(int i=0;i<strlen(v) && v[i]!=letraBuscada;i++){
  cout << v[i]; // El bucle imprime "Hola mundo"
}
```

Este fragmento de código imprime una cadena hasta que se encuentra con la letra `'k'`. Fíjate en la condición `&&`. Cuando `i==strlen(v)` esta expresión será `false` y por tanto no llega a comprobarse `v[i]!=letraBuscada`. Si no existiera la evaluación en cortocircuito y esta segunda condición se comprobara, se produciría un fallo de segmentación (*segmentation fault*) al intentar mirar en una posición del array mayor de su tamaño. Si implementáramos la condición al revés (`v[i]!=letraBuscada &&`

`i<strlen(v)`) se produciría este error. Por tanto, **el orden de los operandos en una expresión lógica**

Entrada y salida

En C++ disponemos de los *streams* de entrada y salida `cin` y `cout`, respectivamente. Podemos mostrar una variable por pantalla de la siguiente forma:

```
1 int i=3;
2 cout << i << endl;
```

Como ves, las flechas (`<<`) van en el sentido de la operación (se envían los datos a `cout`). Por defecto, todo lo que se envía a la salida estándar (`cout`) se imprime por pantalla. Sin embargo, podemos cambiar este comportamiento.

Imagina que tenemos este código fuente y lo compilamos creando un fichero ejecutable llamado `prueba` :

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
6   cout << "Hola mundo" << endl;
7 }
```

Si lo ejecutamos en el terminal de la siguiente manera:

```
1 $ ./prueba
```

veremos que se muestra por pantalla el mensaje "Hola mundo".

Si lo ejecutamos ahora de la siguiente manera:

```
1 $ ./prueba > salida.txt
```

En lugar de mostrarse por pantalla, la salida se guardará en el fichero `salida.txt` , que puede abrirse con cualquier editor de texto. A esto se le llama **redirección** de la salida estándar.

Igual que podemos mostrar información por pantalla (o guardar en fichero) desde nuestro código, también podemos leer el valor de una variable desde teclado:

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main(){
6   int n;
7   cin >> n; // Leemos un valor entero por teclado
```

```
  cout << "He leido" << n << endl;
}
```

En este caso leemos de la **entrada estándar** con `cin`. Los datos a leer dependerán del tipo de la variable. En este ejemplo es un número entero, pero podría ser un carácter (sólo se leería una letra) o una cadena. El operador de entrada lee desde teclado ignorando blancos y tabuladores hasta leer el tipo de dato de la variable que se le indica, dejando el puntero de lectura justo después.

Al igual que con `cout`, podemos leer varias variables seguidas:

```
#include <iostream>

using namespace std;

int main(){
  int n;
  char c;
  cin >> n >> c; // Leemos un valor entero y un carácter por teclado
  cout << "He leido" << n << " y " << c << endl;
}
```

Cuando ejecutemos este programa, quedará a la espera de que introduzcamos un número entero y a continuación un carácter.

Podemos ejecutar el programa redireccionando la entrada estándar. Para ello, creamos un nuevo fichero llamado `entrada.txt` con cualquier editor de texto y escribimos lo siguiente:

```
124 k
```

A continuación ejecutamos:

```
$ ./prueba < entrada.txt
```

En este caso el programa ya no pide los datos de la entrada, sino que los lee desde el fichero `entrada.txt`, asignando a `n` el valor `124` y a `c` el valor `k`.

Podemos también usar la redirección de entrada y salida a la vez:

```
$ ./prueba < entrada.txt > salida.txt
```

El programa leerá la entrada estándar desde el fichero `entrada.txt` y escribirá el resultado en `salida.txt`.

Existe un tercer *stream*: la salida de error `cerr`. En el programa anterior, podemos reemplazar `cout` por `cerr`:

```
#include <iostream>
```

```
2
3 using namespace std;
4
5 int main(){
6   int n;
7   char c;
8   cin >> n >> c; // Leemos un valor entero y un carácter por teclado
9   cerr << "He leido" << n << " y " << c << endl;
10 }
```

La salida de error se puede redirigir también a fichero, usando en este caso el operador `2>`:

```
1 $ ./prueba < entrada.txt 2> error.txt
```

En `error.txt` se almacenará el mensaje "He leído 124 y k".

La salida de error se suele usar para mostrar mensajes de error de los programas. Por ejemplo, el compilador `g++` muestra los errores de compilación por la salida de error. Si ejecutamos el siguiente comando:

```
1 g++ -o prueba prueba.cc 2> errores.txt
```

los errores de compilación se guardarán en el fichero `errores.txt`.

Control de flujo

El control de flujo de código nos permite añadir condiciones al código, de forma que nuestro programa ejecute sólo ciertas instrucciones en función de una condición.

if

La instrucción `if` ejecuta lo que hay a continuación siempre y cuando la condición sea verdadera (`true`). Podemos también añadir un `else` para que se ejecuten instrucciones alternativas cuando no se cumple la condición principal:

```
1 if(condicion){
2   // Instrucciones
3 }
4 else{
5   // Otras instrucciones
6 }
```

while

La instrucción de control de flujo `while` crea un bucle que finaliza cuando se deja de cumplir una condición.

```
while(condicion){
  // Instrucciones
}
```

Es desaconsejable usar `||` en la condición de un `while`, ya que es muy complicado de controlar y poco intuitivo. Por ejemplo, piensa cuándo pararía el bucle en el siguiente código:

```
bool encontrado=false;
int i=0;
while(i<10 || !encontrado){
  cout << i << endl;
  i++;
}
```

Respuesta: Nunca, ya que se debe cumplir que `i<10` **y también** que `encontrado==true`.

> Nota: Para parar un programa que se ha quedado en un bucle infinito, pulsa `Ctrl + C`.

do-while

Su funcionamiento es similar a `while`, pero el bucle se ejecuta al menos una vez:

```
do{
  // Instrucciones
}while(condicion);
```

Ejemplo de uso:

```
int i=0;
do{ // Muestra el valor de i al menos una vez
  cout << "i vale: " << i << endl;
  i++;
}while(i<10);
```

Son útiles, por ejemplo, cuando queremos mostrar un menú por pantalla para que el usuario introduzca alguna opción. El menú se mostrará seguro la primera vez. Luego se volverá a mostrar o no dependiendo de las acciones que realice el usuario (por ejemplo, si el usuario decide salir del programa, no se volverá a mostrar).

for

Los `for` son bucles con una inicialización y una instrucción que se ejecuta tras cada iteración:

```
for(inicializacion;condicion;finalizacion){
  // Instrucciones
}
```

Los `for` son útiles cuando recorremos una serie de elementos, como es el caso de un vector:

```
bool salir=false;
int vec[]={1,3,5,2,5,6,1,2};

for(int i=0;i<8 && !salir;i++){
  cout << i << endl;
  if(array[i]==6){
    salir=true;
  }
}
```

En realidad un `for` es equivalente a una condición `while` pero con una sintaxis más elegante y compacta:

```
inicializacion;
while(condicion){
 // Instrucciones
 finalizacion;
}
```

switch

Las instrucciones `switch` se usan cuando queremos hacer una acción para una variable que puede tener varios valores distintos. Por ejemplo:

```
switch(variable){
  case valor1: // Instrucciones 1
               break;
  case valor2: // Instrucciones 2
               break;
  default: // Instrucciones 3
           break; // No es necesario si es la última instrucción
}
```

En C++ la variable del `switch` debe ser de tipo `int` o `char` (que se convierte implícitamente a entero). Por ejemplo:

```
char c;
cin >> c;
switch(c){
  case 'a': cout << "Seleccionado a" << endl;
            break;
  case 'b': // Como no hay 'break' pasará a la siguiente instrucción
  case 'c': cout << "Seleccionado b o c" << endl;
            break;
  default: cout << "Valor desconocido" << endl;
           break;
}
```

Si la variable es de otro tipo como `float` o `double`, el programa no compilará.

> Como ves, tenemos que usar `break` en los `switch` para que solo se ejecute la instrucción del `case` y salga del `switch`. También se puede usar `break` para salir de un bucle `while`, `do` o `for`.

Vectores y matrices

Un vector (o *array*) es un contenedor que permite almacenar una secuencia de variables o constantes. En C/C++, cuando declaramos un *array* lo hacemos con un **tamaño fijo** que no puede cambiar en tiempo de ejecución.

Podemos usar una constante para indicar su tamaño:

```
int arrayAlumnos[10];
char fila[MAXTABLERO];
```

También podemos crear un *array* inicializándolo con una serie de elementos. En este caso no hay que especificar el tamaño:

```
int numeros[]={1,3,5,2,5,6,1,2};
```

En C++ podemos crear un array con un tamaño asignado por una variable:

```
int n;
cin >> n; // No sabemos qué número introducirá el usuario
int v[n]; // El tamaño del array se conoce en tiempo de ejecución
```

El estándar del lenguaje C++ no recomienda hacerlo así. Si el tamaño del *array* no se conoce en tiempo de compilación, entonces es mejor usar vectores STL, como veremos a continuación.

Para acceder a los valores de un *array* podemos usar corchetes (`[]`):

```
int numeros[]={1,3,5,2,5,6,1,2};
cout << numeros[2] << endl; // Muestra "5"
```

El índice de un *array* comienza en la posición 0. Asimismo, nunca podemos sobrepasar el número de elementos de un *array*, ya que lo más probable es que se produzca un fallo de segmentación:

```
int numeros[]={1,3,5,2,5,6,1,2};

numeros[20]=12; // Error, el array tiene menos de 20 elementos
numeros[8]=3; // Error, las posiciones van de 0 a 7

for(int i=0;i<8;i++){
  numeros[i]=4; // Correcto
}
```

Es importante comprobar siempre los límites de un *array* para no acceder fuera de ellos.

Una matriz es un *array* de más de una dimensión:

```
int matrix[10][10];

matrix[0][2]=31; // Asignación de un valor a la fila 0 columna 2
```

Cadenas de caracteres

Las cadenas de caracteres son *arrays* que contienen caracteres y que terminan en el carácter nulo `'\0'`. Por ejemplo:

```
const char cadena[]="hola"; // El compilador introduce el \0 al final
```

Carácter	h	o	l	a	\0
Posición	0	1	2	3	4

Para representar una cadena de caracteres usamos comillas dobles (`"`), mientras que para representar un sólo carácter usamos comillas simples (`'`).

Es necesario que todas las cadenas acaben con el carácter nulo para que se ejecuten correctamente las funciones que trabajan sobre ellas, como `strlen` , `strcpy` , `strcmp` , etc.

Al igual que cualquier otro *array*, si lo declaramos sin inicializarlo debemos especificar su tamaño:

```
1 const int TAM=10;
2 char cadena[TAM]; // Correcto
3 char cadena2[]; // Dará un error de compilación
```

Funciones

Una **función** es un conjunto de líneas de código que realizan una tarea y, opcionalmente, puede devolver un valor. También puede recibir (opcionalmente) una serie de parámetros, por valor o por referencia.

```
1 // Recibe dos parámetros por valor (a y b) y uno por referencia (c)
2 // Devuelve un número entero
3 int funcion(int a,int b,int &c){
4   int ret=0; // Declaramos una variable del tipo de retorno
5 
6   // Instrucción 1
7   // Instrucción 2
8   // ...
9 
10  return ret; // Devolvemos el resultado
11 }
```

Como puedes ver en el ejemplo anterior, cuando una función devuelve un valor normalmente se declara la variable al principio y se hace un único `return` al final.

Una función no debería tener mucho código. Lo normal es que el código de la función "quepa" en la pantalla cuando se visualice con un editor (unas 30 líneas máximo). Si la función es más larga, lo recomendable es dividirla en varias funciones más cortas.

A veces no tenemos claro cuándo hay que crear una nueva función. Hay varios casos en los que hace falta, pero existe una regla sencilla: si tienes que copiar y pegara código en varios puntos de tu programa, entonces necesitas una función. Cuando copiamos un trozo de código para pegarlo en otro lugar es mejor crear una función para este trozo, de forma que el código quede más compacto y además sea más fácil de mantener (si queremos hacer un cambio en el código de la función, sólo lo tendremos que hacer en un sitio, en lugar de en dos si no la hemos creado).

> Importante: Es muy recomendable compilar y probar las funciones por separado, no esperar a tener todo el programa para empezar a compilar y probar. Cada vez que tengamos una nueva función, hay que comprobar que funcione correctamente con cualquier número de parámetros y cuando esté probada pasar a escribir la siguiente.

El compilador de C++ lee el código desde el principio al final, por lo que si hacemos una llamada a una función que está declarada más adelante obtendremos un error de compilación. Para evitarlo, se puede mover la función antes de su llamada o bien indicar su declaración (también llamada cabecera o **prototipo**) antes. Este es un ejemplo del segundo tipo (declaración mediante prototipo):

```
// Prototipo / cabecera / declaración de la función
int funcion(bool,char,double []);

char otraFuncion(){
  double vr[MAXNOTAS];
  a=funcion(true,'a',vr); // Llamada a la función
}

// Cuerpo / implementación de la funcion
int funcion(bool comer,char opcion,double vectorNotas[]){
  // Instrucciones
}
```

En Programación 2, tal como hace la mayoría de desarrolladores en C++, cuando tenemos todo el código en un único fichero recomendamos mover el código de la función en lugar de indicar su cabecera:

```
// Cuerpo / implementación de la funcion
int funcion(bool comer,char opcion,double vectorNotas[]) {
  // Instrucciones
}

char otraFuncion(){
  double vr[MAXNOTAS];
  a=funcion(true,'a',vr); // Llamada a la función
}
```

Esta segunda forma agiliza la escritura del código, ya que cuando vayamos a cambiar los parámetros de una función sólo tendremos que hacerlo en el cuerpo en lugar de en dos sitios (cuerpo y declaración).

En C++ las funciones aceptan parámetros pasados por valor o por referencia (con `&`). Internamente, cuando una función recibe un parámetro por valor, éste se **copia** en una variable local que se destruye cuando termina la función. De este modo, los cambios que se hagan sobre los valores de esta variable no tendrán efecto a la salida de la función. Cuando un parámetro se pasa por **referencia** no se hace una copia del mismo, por lo que los cambios realizados durante la función sí que afectarán a esta variable.

El problema surge cuando queremos pasar por valor una variable muy grande (por ejemplo, una cadena de 1 millón de caracteres). Si el compilador hace una copia local, el rendimiento del programa se verá seriamente afectado (además de que necesitaremos mucha más memoria). Para evitar esto, en C++ se puede pasar un parámetro por referencia con `const`, como en el siguiente ejemplo:

```
void funcion(const string &s){
  // El compilador no hace copia de s al pasarla por referencia
  // Si intentamos modificar esta variable nos da un error de compilación
}
```

Lo que le decimos al compilador con esta declaración es: "No hagas una copia de la variable, pero prometo no modificarla dentro de la función". De hecho, si la intentamos modificar nos dará un error de compilación.

> En Programación 2 no se permite el paso de parámetros por referencia cuando no es necesario hacerlo, excepto si se hace como en el ejemplo anterior, poniendo delante `const`.

En C++ hay un caso particular de paso por valor o referencia. Se trata de los *arrays* y matrices, ya que estos siempre se pasan por referencia. La explicación la veremos en el Tema 4 ("Memoria dinámica"), pero básicamente esto sucede porque internamente son punteros y en realidad lo que pasamos por valor o referencia es el puntero en sí.

```
1 // El tamaño de la primera dimensión de la matriz m no se indica
2 int sumaVM(int v[],int m[][MAXCOL]){
3   // Instrucciones
4 }
5 sumaVM(vector,matriz); // Pasamos el nombres de la variable sin corchetes
```

Como ves, no hay que indicar el tamaño de la primera dimensión, pero si hay más dimensiones C++ obliga a ponerlas.

Existe también una función especial en C++ llamada `main`. La función `main` es la primera que se invoca cuando comienza un programa. Puede recibir parámetros (lo vamos a ver más adelante) y devolver un valor (aunque se puede dejar sin indicar, normalmente devuelve 0). Por ejemplo:

```
1 int main(){
2   // Instrucciones
3
4   return 0; // Opcional
5 }
```

¿Cómo debe ser una función `main`?. Lo ideal es que, viendo la función, se sepa cuál va a ser el flujo del programa. Un ejemplo de una función `main` adecuada:

```
1 int main(){
2   int n;
3   leer(n);
4
5   if(n<0){
6     cout << "Error, no puede ser negativo";
7   }
8   else{
9     procesar(n);
10  }
11 }
```

Como puedes ver, hay poco código y se entiende intuitivamente qué hace el programa. A veces no es tan sencillo cuando el programa es muy largo, pero la idea es ésta. Lo que nunca debe hacerse es poner todo el código en el `main`, ni tampoco dejar un `main` con una sola llamada a una función que lo haga todo:

```
int main(){
  principal(); // Mal ejemplo de función principal
}
```

Vectores STL

Como hemos visto anteriormente, una vez hemos declarado un *array* no podemos cambiar su tamaño. Si lo que queremos es usar *arrays* de tamaño variable, entonces tenemos que usar los **vectores STL** de C++.

Un vector STL es un *array* con acceso eficiente a elementos y con la habilidad para cambiar automáticamente de tamaño cuando se añaden o se eliminan elementos. Estos vectores inicialmente pertenecían a la biblioteca *STL (Standard Template Library)*, que implementa una serie de contenedores dinámicos, algoritmos e iteradores.

Veamos un ejemplo de cómo usarlos:

```
vector<int> v; // Declara un vector de enteros
vector<int> v2(3); // Declara un vector de 3 enteros
v.resize(4); // Cambia dinámicamente su tamanyo

v.push_back(12); // Añade un valor al final del vector

// Acceso a elementos
for(unsigned int i=0;i<v.size();i++){
  v[i]=23; // Asignación
}
```

Como puedes ver en este fragmento de código, podemos declarar un vector sólo indicando su tipo y sin especificar su tamaño:

```
vector<int> v;
```

También podemos indicar un tamaño inicial:

```
vector<int> v2(3);
```

Para acceder a sus elementos, podemos usar la misma sintaxis que con los *arrays* convencionales mediante `[]`. Añadimos un elemento al final de un vector con `push_back()` y podemos saber el tamaño (número de elementos) que contiene un vector con `size()`. Con la función `clear` podemos borrar todos los elementos y con `erase` uno en concreto:

```
vec.erase(vec.begin()+3); // Elimina el cuarto elemento
vec.clear(); // Elimina todos los elementos del vector
```

La función `erase` recibe como parámetro un iterador (`iterator`), que es un tipo de dato especial para

recorrer vectores. Por ese motivo, debemos ayudarnos de `begin`, que devuelve un iterador al inicio del vector y al que le sumaremos un valor entero que indica la posición del elemento al que queremos acceder (teniendo en cuenta que el primero de ellos está en la posición 0).

Estas son sólo algunas de las funciones de la clase vector, cuya referencia puedes encontrar aquí.

Registros

Un registro en C++ es una agrupación de datos, los cuales no tienen por qué ser del mismo tipo. Se definen con la palabra reservada `struct`:

```
struct Alumno{ // Define un nuevo tipo de dato Alumno
  int dni;
  double nota;
  char nombre[50];
  int turno;
};
```

De esta manera hemos creado un nuevo tipo de dato llamado `Alumno`. A partir de aquí, podemos crear una variable de este tipo de la siguiente forma:

```
Alumno alu;
```

La principal ventaja de usar registros es que nos permiten agrupar datos de diferente tipo. Imaginemos que tenemos la siguiente función:

```
int gestionarAlumno(int dni,double nota,char nombre[],int turno);
```

Si usáramos un registro `Alumno` para agrupar todos estos datos, la llamada sería más sencilla:

```
int gestionarAlumno(Alumno alu);
```

Además, podríamos crear arrays:

```
Alumno alumnos[100];
```

Para acceder a los campos de un registro debemos usar un punto (`.`) tras su nombre. Por ejemplo:

```
Alumno a,b;

a.dni=123133; // Asignacion a un campo
b=a; // Asignacion de un registro entero
```

Tipos enumerados

Los tipos enumerados se utilizan cuando tenemos un rango determinado de posibles valores para una variable. A estos posibles valores se les llama **enumeradores** y las variables de los tipos enumerados pueden tomar cualquier valor de estos enumeradores, como puede verse en el siguiente ejemplo:

```
enum colors_e {black,blue,green,red}; // Definición del tipo enumerado

colors_e mycolor; // Declaración de una variable de tipo colors_e

mycolor=blue; // Asignación de un enumerador

if(mycolor==green){ // Comparación con un enumerador
  mycolor=red;
}
if(mycolor==0){ // Comparación con un entero (0 == black)
  cout << "Black" << endl;
}
```

Internamente, los enumeradores se convierten implícitamente a números enteros, como puede verse en el ejemplo anterior. Estos valores enteros se corresponden con el índice de la posición que ocupa el enumerador en el tipo enumerado. Para el ejemplo anterior de `colors_e`, `black` se representaría con el valor 0, `blue` con el 1, `green` con el 2 y `red` con el 3.

Argumentos del programa

Como comentamos anteriormente, la función `main` también puede recibir parámetros. Cuando lo hacemos estamos pasando argumentos a nuestro programa y esto sirve para la ejecución en lotes (de forma no interactiva). Un ejemplo de paso de argumentos es el que usa el programa `ls` cuando lo llamamos por línea de comando:

```
$ ls -l -a
```

En este caso, pasamos dos parámetros: `-l` y `-a`. Con esta información, el programa `ls` mostrará los datos de una forma determinada, añadiendo información extra sobre los ficheros (por la opción `-l`) y mostrando ficheros ocultos (por la opción `-a`). Imagínate que `ls` no admitiera parámetros. Cada vez que lo ejecutáramos, nos preguntaría si queremos mostrar los archivos ocultos, si queremos mostrar la información por líneas, etc, lo cual sería muy tedioso para el usuario.

Vamos a ver un ejemplo de un programa que admite parámetros a través de la función `main`:

```
int main(int argc,char *argv[]){
  // En este punto, argc contiene el numero de parametros, y argv su valor.
  for(unsigned i=0;i<argc;i++){
    cout << "Argv[" << i << "]=" << argv[i] << endl;
  }
}
```

Si ejecutamos el programa de esta forma:

```
$ ./programa uno dos tres
```

Se mostrará lo siguiente por pantalla:

```
Argv[0]=./programa
Argv[1]=uno
Argv[2]=dos
Argv[3]=tres
```

Por tanto, la variable `argc` contiene el número de parámetros que ha introducido el usuario y `argv` es un array de cadenas de caracteres, donde cada posición contiene uno de los valores de los parámetros introducidos por el usuario.

En principio parece sencillo, pero se complica cuando tenemos muchos parámetros y pueden ir en cualquier orden. Por ejemplo, `g++` admite muchos parámetros y el orden puede ser importante:

```
$ g++ -Wall -o programa programa.cc -g
```

Gestionar toda esta variabilidad es algo complicado. Por ejemplo, si queremos un programa que acepte tres parámetros (`uno`, `dos` y `tres`) podemos usar un bucle como éste:

```
int main(int argc,char *argv[]){
  if(argc>4) {
    cout << "Sintaxis: " << argv[0] << " [uno|dos|tres]" << endl;
    return(-1);
  }
  for(unsigned i=1;i<argc;i++){
    if(strcmp(argv[i],"uno")==0){
      // Hacer algo con argumento uno
    }
    else if(strcmp(argv[i],"dos")==0){
      // Hacer algo con argumento dos
    }
    else if(strcmp(argv[i],"tres")==0){
      // Hacer algo con argumento tres
    }
    else{
      cout << "Sintaxis: " << argv[0] << " [uno|dos|tres]" << endl;
      return(-1);
    }
  }
}
```

Como ves, es fácil que con unos pocos parámetros para gestionar haga falta escribir bastante código, por lo que es conveniente usar una función aparte para gestionar los parámetros.

Estructura de un programa

A modo de resumen, veamos la estructura típica de un programa en C++:

```
#include <ficheros de cabecera estandar>
...
#include "ficheros de cabecera propios"
...
using namespace std; // Permite usar cout, string, ...
...
const ... // Declaración de constantes
...
typedef struct enum ... // Declaración de tipos de datos propios
...
// Declaración de variables globales: ¡¡PROHIBIDO EN PROGRAMACIÓN 2!!
...
// Funciones
...
int main(){ // Función principal
...
}
```

Compilación y depuración

El compilador de C++ que usaremos en la asignatura es GNU GCC, que viene por defecto en Linux. Podemos llamar al compilador desde un terminal poniendo `g++`. El compilador de C++ admite muchos parámetros, vamos a ver los más importantes:

- `-Wall` : muestra todos los *warnings*, no sólo los más importantes (**recomendado en Programación 2**)
- `-g` : compila en un modo que facilita encontrar los errores mediante un depurador (**recomendado en Programación 2**)
- `-o` : sirve para indicar el nombre del ejecutable, que es el parámetro que viene a continuación de esta opción
- `--version` : muestra la versión actual del compilador
- `-std=c++11` : usa el estándar de C++ 201.

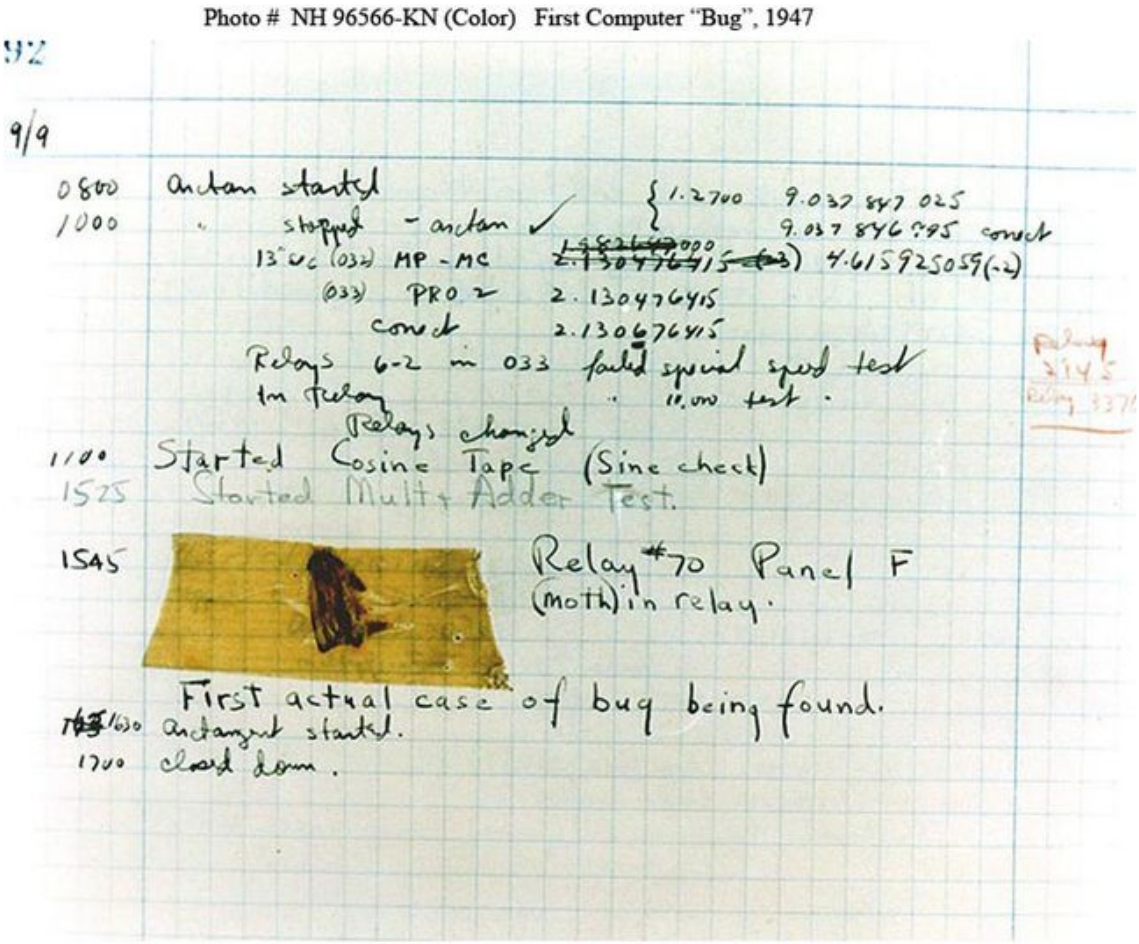
La forma recomendada en Programación 2 para compilar un programa es la siguiente:

```
g++ -Wall -g programa.cc -o programa
```

Es muy importante que justo a continuación de `-o` vaya el nombre del ejecutable, no el del código fuente. Si por error ponemos `-o programa.cc`, se nos borrará todo nuestro código fuente (estos despistes son habituales cuando estás aprendiendo a programar, así que ten cuidado).

Depuración

En algunas ocasiones el programa compila pero falla durante la ejecución. A veces es muy complicado encontrar qué línea del código ha provocado el error de ejecución. Afortunadamente, existen los **depuradores** (en inglés *debuggers*), unas herramientas que nos facilitan mucho esta tarea.
El nombre *debuggers* hace referencia a que sirven para eliminar *bugs* (bichos), que es como se conocen popularmente los errores de código. Este término viene porque en 1947, una polilla se introdujo accidentalmente en un circuito provocando errores en su programa. Cuando encontraron el problema, la pegaron con celo en el correspondiente informe de errores y escribieron: *First actual case of bug being found*:



A continuación se indican algunos depuradores muy usados:

- Valgrind. Es el depurador que usaremos en los correctores de las prácticas. Detecta errores de memoria: acceso a componentes fuera de un vector, variables usadas sin inicializar, punteros que no apuntan a una zona reservada de memoria, etc.
- GDB. Este depurador inicia nuestro programa, lo para cuando lo pedimos y mira el contenido de las variables. Si nuestro ejecutable da un fallo de segmentación, nos dice la línea de código dónde está el problema
- Más ejemplos en Linux: DDD, Nemiver, Electric Fence y DUMA

Casi todos estos depuradores se pueden instalar desde el instalador de paquetes de Linux (es más cómodo así que hacerlo desde su web).

Metodología recomendada para programar

A continuación y para terminar, un resumen de consejos finales sobre cómo afrontar la programación:

- Estudia detenidamente el problema y la posible solución
- Diseño del algoritmo en papel. El papel y el lápiz son dos de las principales herramientas de un/a ingeniero/a
- Diseñar el programa intentando hacer muchas funciones con poco código (unas 30 líneas por función)
- Evitar código repetido utilizando adecuadamente las funciones
- El `main` debería ser como el índice de un libro y permitir entender lo que hace el programa de un vistazo
- Compilar y probar las funciones por separado: no esperar a tener todo el programa para empezar a compilar y probar

Tema 2: La clase string

La mayoría de programas que desarrollamos tienen que trabajar en algún momento con datos textuales. Por ejemplo, mostrar un mensaje por pantalla a un usuario o leer el nombre de un cliente introducido por teclado implica manipular cadenas de texto.

El lenguaje C++ nos proporciona dos formas de representar las cadenas de texto:

- Cadenas de caracteres al estilo C
- La clase `string` de C++

En este tema veremos en primer lugar las cadenas en estilo C, para luego centrarnos en cómo trabajar con la clase `string` de C++.

La clase `string` facilita mucho el trabajo con cadenas, por lo que será nuestra opción preferida en la asignatura de Programación 2. La única circunstancia en la que tendremos que utilizar necesariamente cadenas al estilo C será cuando trabajemos con ficheros binarios. Veremos el por qué en el Tema 3.

Cadenas de caracteres en C

Declaración

Esta forma de manipular texto es originaria del lenguaje C pero puede utilizarse también en C++.

Las cadenas en C se representan con un array de caracteres (es decir, de tipo `char`) terminado en el carácter nulo (`'\0'`). Como todo array, tienen un tamaño fijo que se establece en el momento de declararlo y ya no puede variar a lo largo de la ejecución del programa.

Ejemplo:

```
1 char cad[10];
```

Este código define una cadena de caracteres, llamada `cad`, de 10 elementos. Como hay que reservar un espacio para introducir el carácter nulo de final de cadena, la variable `cad` del ejemplo anterior podría almacenar como máximo una cadena de 9 caracteres.

El lenguaje C/C++ nos permite inicializar las cadenas de caracteres con texto dentro de comillas dobles (`""`).

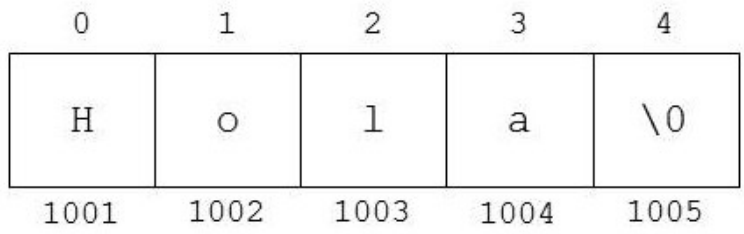
Ejemplo:

```
1 char cad1[5]="hola";
2 char cad2[]="hola";
```

Las dos declaraciones anteriores son equivalentes. En el primer caso, le asignamos a `cad1` el tamaño `5` para poder almacenar los cuatro caracteres de `"hola"` junto con el carácter nulo. En el segundo caso con `cad2`, se puede ver que no es necesario indicar el tamaño de la cadena cuando hacemos una declaración con inicialización: el compilador asignará al array el tamaño exacto que necesita para almacenar la cadena con la que se inicializa (en este caso `5`).

Otro detalle importante que muestra este ejemplo es que no hay que poner el carácter nulo al final de la cadena constante definida con comillas dobles. El compilador automáticamente coloca el `'\0'` al final de la cadena cuando se inicializa el array.

A continuación se muestra cómo sería la representación en memoria en C/C++ de cualquiera de las dos cadenas del ejemplo anterior:



Representación en memoria de la cadena "Hola"

La numeración superior (de 0 a 4) indica el índice que ocuparía cada uno de los caracteres en el array. La numeración inferior (de 1001 a 1005) representa la posición de memoria que ocuparía cada carácter. Se trata de direcciones de memoria ficticias simplificadas para este ejemplo (en realidad una dirección válida sería algo como esto: `0x7ffef832d670`). Lo importante es ver que cada carácter de una cadena se almacena en posiciones consecutivas de la memoria, cosa que sucede siempre con los elementos de un array, sean del tipo que sean.

Otra forma de inicializar una cadena es hacerlo carácter a carácter. Ejemplo:

```
1 char cad1[5]={'h','o','l','a','\0'};
2 char cad2[]={'h','o','l','a','\0'};
```

Esta inicialización sería equivalente a la del ejemplo anterior. En este caso sí que sería necesario introducir explícitamente el carácter `'\0'` al final de la cadena. Si declaráramos lo siguiente:

```
1 char cad[]={'h','o','l','a'};
```

estaríamos definiendo un array de cuatro caracteres con los elementos `'h'`, `'o'`, `'l'` y `'a'`, pero no se consideraría una cadena de caracteres "bien formada", ya que no acaba en el carácter nulo y por lo tanto no podríamos aplicar correctamente sobre ella las funciones que nos facilita el lenguaje C y C++ para el manejo de cadenas (y que veremos más adelante en este mismo tema).

Al igual que para arrays de otros tipos, no es necesario usar todo el espacio reservado para la cadena cuando se declara. Ejemplo:

```
1 char cad[100]="Hola";
```

Aquí estaríamos reservando `100` posiciones de memoria, aunque solo estaríamos ocupando 5 de momento (4 letras más el carácter nulo). Las otras 95 posiciones quedarían reservadas pero sin inicializar.

La cadena vacía se representa con las comillas dobles, una a continuación de la otra y sin espacio en medio:

```
1 char cadenaVacia[]="";
```

El array `cadenaVacia` tendría tamaño 1, ya que solo almacenaría el carácter `'\0'`.

Salida por pantalla

Para mostrar cadenas por pantalla se puede utilizar `cout`, como con cualquier otro tipo simple (`int`, `float`, etc.). Ejemplo:

```
1 char cad[]="Hola a todo el mundo";
2 cout << cad;
```

Este código mostraría por pantalla `Hola a todo el mundo`.

Entrada por teclado

Operador `>>`

La lectura de información por teclado se realiza con `cin` y el operador `>>`, de manera similar a como se hace con el resto de tipos simples, pero con alguna particularidad:

- Ignora los espacios en blanco que se introducen antes del primer carácter válido de la cadena. Es decir, si el usuario escribe `"   hola"`, con tres espacios en blanco delante, `cin` los ignorará y empezará a almacenar caracteres a partir de la `h`

- Después de haber leído un carácter válido, termina de leer cuando encuentra el primer *blanco* (espacio, tabulador o salto de línea). Ese blanco se deja en el buffer de teclado para la siguiente lectura

Ejemplo:

```
1 char cad[20];
2 cin >> cad;
```

Este programa quedaría a la espera de que el usuario introdujera la cadena por teclado.

`getline`

Leer de teclado usando `cin` y `>>` puede generar dos problemas:

- Problema 1: ¿Y si la cadena tiene espacios en blanco? Si por ejemplo el usuario escribiera `hola a todos`, el programa anterior leería `hola`, hasta encontrar el primer espacio en blanco, y dejaría de leer

```
1 char cad[20];
2 cin >> cad;
3 // El usuario escribe "buenas tardes"
4 // La variable cad almacena "buenas"
```

- Problema 2: El operador `>>` no limita el número de caracteres que se leen. ¿Y si la cadena escrita no cabe en el vector? Aquí podemos tener un problema serio, porque estaremos tratando de escribir en zonas de memoria fuera del vector. Si en el ejemplo anterior el usuario escribiera `supercalifragilisticoespialidoso`, se produciría un error en la memoria al haber sobrepasado el tamaño del vector

Una forma de solucionar estos dos problemas es utilizar la función `getline`, que pueden leer cadenas con blancos y controlar el número de caracteres que se almacenan. Ejemplo:

```
1 const int TAM=10;
2 char cad[TAM];
3 cin.getline(cad,TAM);
```

En este ejemplo, `getline` lee como máximo `TAM-1` caracteres o hasta que llegue al final de línea. El `'\n'` del final de línea se lee pero no se mete en la cadena `cad`. La función `getline` se encarga de añadir `'\0'` al final de lo que ha leído. Por esa razón solo lee `TAM-1` caracteres, dejando un espacio reservado para el carácter nulo.

Si el usuario introdujera `hola a todos`, el programa almacenaría en `cad` la cadena `hola a to` (los espacios en blanco cuentan como un carácter más).

Esta función tiene también un problema. ¿Qué sucede si el usuario escribe más caracteres de los que caben en la cadena? En este caso, los caracteres sobrantes se quedan en el buffer de teclado y provocan un fallo en la siguiente lectura.

Ejemplo:

```
char cad1[10];
char cad2[10];

cin.getline(cad1,10);
cout << "Cadena 1: " << cad1 << endl;
cin.getline(cad2,10);
cout << "Cadena 2: " << cad2 << endl;
```

En este programa, si el usuario introdujera `hola a todos` la salida mostrada por pantalla sería:

```
Cadena 1: hola a to
Cadena 2:
```

Problemas del uso combinado de `>>` y `getline`

Hemos visto dos formas de hacer lectura de teclado: mediante el operador `>>` y mediante `getline`. Cuando estos dos operadores se combinan, se pueden dar situaciones no deseadas. Ejemplo:

```
int num;
char cad[1000];

cout << "Escribe un numero: ";
cin >> num;
cout << "El numero leido es " << num << endl;

cout << "Escribe una cadena: " ;
cin.getline(cad,1000);
cout << "La cadena leida es: " << cad << endl;
```

En este ejemplo, si cuando el programa muestra por pantalla `Escribe un numero:` el usuario escribe `10`, por ejemplo, la salida completa por pantalla que se produce sería:

```
Escribe un numero: 10
El numero leido es 10
Escribe una cadena: La cadena leida es:
```

Vemos que no se le llega a preguntar al usuario por la cadena. ¿Por qué sucede esto?

En el código anterior, primero se lee un `int` mediante `>>`, para luego leer una cadena con `getline`. Cuando se lee `10` con el operador `>>`, éste deja de leer cuando encuentra el primer espacio en blanco (el salto de línea en este caso, cuando se pulsa la tecla *intro* después de escribir el `10`) y deja ese salto de línea en el buffer. Cuando ejecuta `getline`, lo primero que se encuentra en el buffer es el salto de línea `'\n'`, por lo que termina de leer y almacena en la variable `cad` una cadena vacía.

Una solución para evitar este problema es extraer el `'\n'` del buffer de teclado antes de hacer la

siguiente lectura. Esto lo podemos hacer con el método `get` de la siguiente manera:

```
...
cin >> num;
cin.get();
...
```

Aquí, `cin.get()` saca un carácter del buffer de teclado de manera que ya se puede usar `getline` a continuación sin problema.

La librería `string.h`

La librería estándar `string.h` de C/C++ ofrece una serie de funciones para trabajar con cadenas de caracteres. Para poder utilizarlas hay que importar la librería al comienzo de nuestro código utilizando la siguiente instrucción:

```
#include <string.h>
```

Entre las funciones más importantes que proporciona la librería estándar, tenemos `strlen`, `strcmp` y `strcpy`.

`strlen`

Esta función devuelve un `int` con el número de caracteres que contiene la cadena. Ejemplo:

```
char cad[20]="adios";
cout << strlen(cad);
```

Este ejemplo devolvería `5`, que es el número de caracteres que contiene `cad`, no `20` que sería el tamaño del array, ni `6` que sería el número de espacios de memoria ocupados si tenemos en cuenta que hay un `'\0'` al final.

`strcmp`

Esta función compara dos cadenas en orden lexicográfico, es decir, el orden que se da en un diccionario:

- La letra 'a' es más pequeña que la letra 'b'
- La cadena "adeu" es más pequeña que "adios", porque los dos primeros caracteres son iguales pero en la tercera posición 'i' es mayor que 'e' y ya no se miran los caracteres restantes
- Las letras minúsculas son mayores que sus correspondientes mayúsculas ('a' > 'A')
- Las letras son mayores que los números ('A' > '1')

De la lista anterior, los dos últimos puntos no son obvios. Este comportamiento viene dado por el código ASCII de cada carácter. El código ASCII es una representación numérica que tiene cada carácter en la memoria del ordenador.

A continuación se muestra la tabla de códigos ASCII de los 128 primeros caracteres:

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Código ASCII en decimal, hexadecimal, binario y octal

El código ASCII del carácter '1' es 49 (en decimal), mientras que el de la letra 'A' es 65 y el de la letra 'a' es 97. Por esta razón, 'a' > 'A' > '1'.

`strcmp` devuelve `-1` si la primera cadena es menor que la segunda, `0` si son iguales y `1` si la segunda cadena es mayor. Ejemplo:

```
1 char cad1[]="adios";
2 char cad2[]="adeu";
3
4 cout << strcmp(cad1,cad2) << endl;
5 cout << strcmp(cad2,cad1) << endl;
6 cout << strcmp(cad1,cad1) << endl;
```

Este código mostraría por pantalla:

```
1 1
2 -1
3 0
```

Por tanto, si queremos comprobar si dos cadenas introducidas por teclado son iguales, lo podemos hacer de la siguiente manera:

```
char cad1[1000];
char cad2[1000];

cout << "Introduce la cadena 1: ";
cin >> cad1;
cout << "Introduce la cadena 2: ";
cin >> cad2;

if(strcmp(cad1,cad2)==0){
  cout << "Las cadenas son iguales" << endl;
}
else{

  cout << "Las cadenas son diferentes" << endl;
}
```

Otra manera más compacta de expresar `strcmp(cad1,cad2)==0` sería `!strcmp(cad1,cad2)`.

`strcpy`

Esta función permite copiar una cadena en otra:

```
char cad[10];
strcpy(cad,"hola");
```

Las cadenas de caracteres en C son arrays y por tanto no se pueden asignar directamente. El siguiente código produciría un error de compilación:

```
char cad[10];
cad="hola";
```

El siguiente código también daría error de compilación:

```
char cad1[10]="hola";
char cad2[10];
cad2=cad1;
```

Al tratarse de arrays, debería de hacerse la copia elemento a elemento (carácter a carácter) de una cadena a otra. La función `strcpy` nos evita tener que hacer este tedioso proceso.

Un detalle importante es que la cadena receptora tiene que tener tamaño suficiente para almacenar la cadena que queremos guardar. Ejemplo:

```
char cad[10];
strcpy(cad,"Hoy es un dia fantastico para salir de paseo");
```

Este código produciría accesos a zonas de memoria no reservadas, y un más que probable fallo de segmentación, al tratar de copiar una cadena de mayor tamaño de lo que admite la variable `cad`. Hay que tener siempre en cuenta que tiene que haber también espacio suficiente en la cadena para el `'\0'`.

Otras funciones

Las funciones `strncmp` y `strncpy` son similares a las dos funciones anteriores, pero con la diferencia de que solo comparan o copian los `n` primeros caracteres. Por ejemplo:

```
char cad[10];
strncpy(cad,"Hola, mundo",4);
cad[4]='\0';
```

En este ejemplo, se copian los `4` primeros caracteres de `"Hola, mundo"` en `cad`. El carácter nulo `'\0'` se debe de añadir explícitamente al final de los caracteres copiados. Al copiar `4` caracteres, estos ocupan las posiciones 0, 1, 2 y 3 del array `cad`, por eso el carácter nulo se añade en la posición siguiente, en la `4`.

Conversión a `int` y `float`

Dos ejemplos más de funciones interesantes que trabajan con cadenas de caracteres, aunque estas no pertenecen a la librería `stdlib.h`, son `atoi` y `atof`. La primera sirve para convertir una cadena de texto que representa un valor entero a su equivalente valor de tipo `int`. La segunda función, exactamente igual pero para el caso de valores reales. Estas dos funciones están definidas en la librería estándar `cstdlib`, por lo que habrá que incluirla al comienzo de nuestro código si queremos poder utilizarlas:

```
#include <cstdlib>
```

Ejemplo:

```
char cad1[]="100";
int n=atoi(cad1);

char cad2[]="10.5";
float f=atof(cad2);
```

En este código, la variable `n` tomará el valor `100` y la variable `f` el valor `10.5`.

La clase `string` en C++

Definición

La librería C++ estándar proporciona la clase `string` que da soporte a todas las operaciones sobre cadenas de texto mencionadas anteriormente, además de muchas otras funcionalidades.

La gran ventaja con respecto a las cadenas de caracteres mediante arrays en C es que la clase `string` tiene un tamaño variable que puede cambiar a lo largo de la ejecución del programa en función de la cadena que queramos almacenar: puede aumentar si queremos almacenar una cadena más grande o puede disminuir para no desperdiciar memoria si queremos almacenar una cadena más pequeña.

La clase `string` usa internamente arrays de caracteres para almacenar los datos, pero el manejo de la memoria y la localización del carácter nulo lo hace de manera transparente la propia clase, que es lo que simplifica su uso.

El concepto de "clase" y su diferencia con un tipo simple lo veremos en el Tema 5, cuando nos adentremos en el paradigma de programación orientada a objetos. Para utilizar la terminología adecuada, en lugar de "variable de tipo `string`" deberíamos decir "objeto de la clase `string`", y en lugar de "funciones específicas" de `string` deberíamos de hablar de "métodos". No obstante, para no anticiparnos a los contenidos sobre programación orientada a objetos, en este tema seguiremos utilizando la terminología habitual y hablaremos de "tipos", "variables" y "funciones" en lugar de "clases", "objetos" y "métodos".

Las variables de tipo `string` se declaran como cualquier otro tipo de dato: ponemos el tipo, el nombre dado a la variable y opcionalmente un valor de inicialización. Ejemplo:

```
1 string cad1;
2 string cad2="hola";
3 const string cad3="hola";
```

Este código muestra una variable sin inicializar (`cad1`), una variable con el valor inicial `"hola"` (`cad2`) y una constante con ese mismo valor inicial (`cad3`). Como puede verse en este ejemplo, no se debe usar la sintaxis de corchetes ni indicar el tamaño de la cadena como se hacía con las cadenas en C.

El paso de parámetros a una función, ya sea por valor o referencia, es como con cualquier dato simple (`int`, `float`, etc.). Ejemplo:

```
1  #include <iostream>
2  #include <string>
3
4  using namespace std;
5
6  void analizarCadena(string cad1,string &cad2){
7    ...
8  }
9
10 int main(){
11   string cad1="hola";
12   string cad2="adios";
13
14   analizarCadena(cad1,cad2);
15 }
```

En este código se declaran dos variables de tipo `string` en la función principal, `cad1` y `cad2`. Ambas se pasan a la función `analizarCadena`, la primera por valor y la segunda por referencia.

Salida por pantalla

Al igual que con las cadenas de caracteres de C, se utiliza `cout` para mostrar el contenido de un `string` por pantalla. Ejemplo:

```
1 string cad="Hola a todo el mundo";
2 cout << cad;
```

Este código mostraría por pantalla `Hola a todo el mundo`.

Entrada por teclado

Operador `>>`

Para leer información de teclado, se puede utilizar `cin` y el operador `>>`, como ocurría con las cadenas en C, produciéndose el mismo resultado:

- Ignora los espacios en blanco que se introducen antes del primer carácter válido de la cadena
- Después de haber leído un carácter válido termina de leer cuando encuentra el primer blanco (espacio, tabulador o salto de línea). Ese blanco se deja en el buffer de teclado para la siguiente lectura

Ejemplo:

```
1 string s;
2 cin >> s;
3 // El usuario escribe "hola"
4 // La variable s almacena "hola"
5 ...
6 // El usuario escribe "buenas tardes"
7 // La variable s almacena "buenas"
```

`getline`

Si la cadena contiene espacios en blanco y queremos leerla entera, podemos utilizar también la función `getline` que utilizábamos para cadenas de C, aunque en este caso la sintaxis es diferente:

```
1 string cad;
2 getline(cin,cad);
3 // Si el usuario introduce "buenas tardes"
4 // en s se almacena "buenas tardes"
```

La ventaja que tiene usar `getline` con el tipo `string` es que no limita el número de caracteres leídos y, por tanto, no debe de indicarse este número como parámetro de la función.

Si queremos leer hasta la aparición de un determinado carácter (por defecto lee hasta que encuentra el salto de línea `'\n'`), podemos indicarlo como parámetro de `getline`. Ejemplo:

```
1 string cad;
2 getline(cin,cad,',');
```

En este ejemplo se leería de teclado hasta la primera aparición del carácter `','`.

Hay que tener en cuenta que, en caso de combinar lecturas de teclado con el operador `>>` y con `getline`, tendríamos el mismo problema que ya se mencionó con las cadenas en C.

Extraer palabras de un `string`

Dada una cadena de texto almacenada en un `string`, podemos extraer cada una de las palabras que contiene (asumiendo que están separadas por espacios en blanco) utilizando la clase `stringstream`. Para poder utilizar esta clase hay que importar la correspondiente librería al comienzo de nuestro código:

```
1 #include <sstream>
```

Ejemplo de uso:

```
1 stringstream ss("Hola mundo cruel 1 32 2.3");
2 string s;
3
4 // En cada iteración del bucle lee hasta encontrar un espacio en blanco
5 while(ss >> s){ // Extraemos las palabras una a una
6   cout << "Palabra: " << s << endl;
7 }
```

En este código, para cada iteración del bucle `while`, el operador `>>` lee de `ss` hasta que encuentra un espacio en blanco y lo almacena en `s`. `stringstream` se comporta como `cin` y se lee de la misma manera, ya que ambos representan flujos de caracteres.

Métodos de `string`

La librería estándar `string` proporciona una serie de funciones que facilitan el trabajo con cadenas. Al tratarse de una clase en lugar de un tipo simple, las funciones (en realidad se llama *métodos* a las funciones de una clase) de la clase `string` se invocan poniendo un `.` detrás del nombre de la variable. Como se comentó más arriba, veremos esto con detalle en el Tema 5 cuando introduzcamos la programación orientada a objetos.

Entre las funciones más importantes que proporciona la librería `string`, tenemos `length`, `find`, `replace`, `erase` y `substr`.

`length`

Esta función permite obtener el número de caracteres que contiene una cadena. Su prototipo es el

siguiente:

```
unsigned int length();
```

No recibe ningún parámetro y devuelve como resultado un valor de tipo `unsigned int`, es decir, un entero sin signo, ya que el número de caracteres de una cadena nunca puede ser un valor negativo. La palabra reservada `unsigned` del lenguaje C/C++ es un modificador de tipo que indica que el valor almacenado es siempre positivo (no puede tener signo negativo). Ejemplo:

```
string cad="hola";
unsigned int tam=cad.length();
```

Este ejemplo almacenaría el valor `4` en la variable `tam`.

`find`

Esta función permite buscar una subcadena dentro de otra. El prototipo de la función es el siguiente:

```
size_t find(const string str,unsigned int pos=0);
```

La función devuelve un valor de tipo `size_t`. Este tipo de dato se usa para representar el tamaño de un objeto. En muchas ocasiones se comporta de manera equivalente a `unsigned int`, pero no siempre es así. Si debemos almacenar el resultado devuelto por `find` en una variable, es conveniente definir ésta de tipo `size_t`.

El primer parámetro de `find` es la subcadena que se quiere buscar, mientras que el segundo parámetro indica a partir de qué posición de la cadena queremos comenzar la búsqueda de la subcadena (si no se indica nada ese valor será `0` y empezará a buscar por el principio de la cadena). La función devuelve un valor que indica la posición dentro de la cadena (`0` es la primera posición) donde ha encontrado la subcadena. Si no encuentra la subcadena buscada, devuelve la constante `string::npos`. Ejemplo:

```
string a="Hay una taza en esta cocina con tazas";
string b="taza";

// Longitud de a
unsigned int tam=a.length();

// Buscamos la primera aparición de la subcadena "taza" dentro de la cadena "Hay una taza.
size_t encontrado=a.find(b);
if(encontrado!=string::npos){
  cout << "Encontrada primera " << b << " en la posición " << encontrado << endl;

  // Buscamos la segunda aparición de la subcadena "taza"
  encontrado=a.find(b,encontrado+b.length());
  if(encontrado!=string::npos){
    cout << "Encontrada segunda " << b << " en la posición " << encontrado << endl;
  }
}
```

```
18 else{
19   cout << "Palabra " << b << " no encontrada";
20 }
```

La función `find` termina cuando encuentra la primera aparición de la subcadena. En la primera búsqueda (`a.find(b)`) se empieza desde el inicio de la cadena, por lo que para al encontrar la primera aparición de la subcadena. En la segunda búsqueda (`a.find(b,encontrado+b.length())`), se comienza justo a continuación de la aparición de la primera subcadena, por lo que devolverá la segunda subcadena que pueda encontrar en la cadena. Este proceso se podría repetir introduciéndolo dentro de un bucle si nos interesara recuperar todas las apariciones de una subcadena dentro de otra.

`replace`

Esta función permite reemplazar una subcadena dentro de una cadena. Su prototipo es:

```
1 string& replace(unsigned int pos,unsigned int tam,const string str);
```

El primer parámetro indica la posición dentro de la cadena donde comenzarán a reemplazarse caracteres. El segundo parámetro indica el número de caracteres que se van a reemplazar. Finalmente, el tercer parámetro indica la subcadena que se va a insertar como reemplazo. La función modifica directamente la cadena sobre la que se aplica, por lo que no es necesario asignar el valor que devuelve a otra variable. Ejemplo:

```
1 string a ="Hay una taza en esta cocina con tazas";
2
3 a.replace(8,4,"botella");
4 cout << a << endl;
```

La salida por pantalla de este ejemplo sería `Hay una botella en esta cocina con tazas`.

En este ejemplo, se han sustituido dentro de la cadena `a` un total de `4` caracteres comenzando en la posición `8` y se ha insertado en su lugar la subcadena `"botella"`.

`erase`

Esta función permite eliminar un conjunto de caracteres de una cadena o eliminarla por completo. Su prototipo es el siguiente:

```
1 string& erase(unsigned int pos=0,unsigned int tam=npos);
```

El primer parámetro indica a partir de qué posición se empieza la eliminación de caracteres. El segundo parámetro indica cuántos caracteres se van a eliminar. Si no se indica ningún parámetro, `erase` elimina todos los caracteres de la cadena. Ejemplo:

```
1 string cad="Hola a todo el mundo";
2 cad.erase(3,11);
3 cout << cad;
```

La salida por pantalla de este código sería `Hola mundo`.

`substr`

Esta función devuelve una subcadena de la cadena original. Su prototipo es el siguiente:

```
1 string substr(unsigned int pos=0,unsigned int tam=string::npos) const;
```

El primer parámetro es la posición a partir de la que empezará a extraer la subcadena. Si no se indica nada, comenzará desde la primera posición. El segundo parámetro es la cantidad de caracteres que queremos extraer. Si no se indica nada, se extraerán hasta el final de la cadena. Ejemplo:

```
1 string cad="hola mundo";
2 string subcad=cad.substr(2,5);
```

En `subcad` se almacenará `la mu`. Como puedes ver, el espacio en blanco cuenta como un carácter más.

Operadores

A las variables de tipo `string` se les pueden aplicar una serie de operadores aritméticos y de comparación.

Para asignar una cadena a otra, basta con usar el operador de igualdad `=`. Ejemplo:

```
1 string cad1="Hola";
2 string cad2;
3 cad2=cad1;
```

Recuerda que esto no se podía hacer directamente con cadenas de caracteres en C y tenía que utilizarse la función `strcpy`.

La concatenación de cadenas (es decir, añadir una cadena a continuación de otra) se lleva a cabo con el operador `+`. Ejemplo:

```
1 string cad1="Hola";
2 string cad2="mundo";
3 string cad3=cad1+", "+cad2+"!";
4 cout << cad3;
```

Este código mostraría por pantalla `Hola, mundo!`.

Para comparar cadenas, se pueden usar los mismos operadores que utilizamos para comparar números: `==`, `!=`, `>`, `<`, `>=` y `<=`. Ejemplo:

```
string cad1;
string cad2;

cin >> cad1;
cin >> cad2;

if(cad1>cad2){
  cout << "La primera cadena es mayor que la segunda" << endl;
}
else if(cad1<cad2){
  cout << "La primera cadena es menor que la segunda" << endl;
}
else if(cad1==cad2){
  cout << "Ambas cadenas son iguales" << endl;
}
```

Para acceder a los distintos caracteres de un `string` podemos utilizar la misma sintaxis que utilizamos para acceder a cualquier array, mediante `[]`. Ejemplo:

```
string cad="Hola";

for(int i=0;i<cad.length();i++){
  cout << cad[i] << endl;
}
```

Este código realiza un bucle que muestra por pantalla una cadena carácter a carácter:

```
H
o
l
a
```

También se puede cambiar el valor de un carácter concreto del `string` utilizando esta sintaxis. Ejemplo:

```
string cad="hola";
cad[0]='p';
cad[3]='o';
cout << cad;
```

Este código sustituye el primer y cuarto carácter de la cadena y muestra por pantalla `polo`. Es importante tener en cuenta aquí que solo podemos asignar caracteres a posiciones de la cadena que ya existan. Ejemplo:

```
string cad="hola";
cad[5]='!';
cout << cad;
```

En este ejemplo se está tratando de cambiar el valor del carácter en la posición `5` de la cadena, pero dicha posición no existe, ya que la variable `string` solo tiene reservado espacio para almacenar la cadena `"hola"`, que va de la posición 0 a la 3. La salida por pantalla sería `Hola`, ya que el carácter `!` no se puede almacenar en la posición especificada.

Conversiones de tipos

Conversión entre `string` y vector de caracteres en C

Para convertir una cadena en C a un `string`, podemos crear una variable de tipo `string` y utilizar directamente el operador `=` para asignarle su valor. Ejemplo:

```
char cad1[]="hola";
string cad2=cad1;
```

Para convertir un `string` a cadena de caracteres en C, podemos usar la función `c_str` de la clase `string`, que devuelve un vector de caracteres en C con el contenido del `string`, junto con la función `strcpy` para hacer la copia entre cadenas. Ejemplo:

```
string cad1="hola";
char cad2[10];
strcpy(cad2,cad1.c_str());
```

Conversión entre `string` y número

Para convertir un número entero o real a `string` podemos utilizar la clase `stringstream` que vimos más arriba . Ejemplo:

```
#include <sstream>
...
int num=100;
stringstream ss;
string s;

ss << num;
s=ss.str(); // Convierte el stringstream a string
```

Para convertir de cadena a número entero o real, podemos utilizar las mismas funciones que vimos para cadenas en C, `atoi` y `atof`, pero teniendo en cuenta que estas funciones esperan como entrada un array de caracteres y no un tipo `string`. Por ello, habrá que hacer una conversión previa utilizando la función `c_str` de la clase `string`, que permite convertir un `string` a cadena de caracteres en C. Este método, cuando se aplica a una variable de tipo `string`, devuelve una cadena de caracteres con su contenido. Ejemplo:

```
string cad1="100";
int n1=atoi(cad1.c_str());

string cad2="10.5";
float n2=atof(cad2.c_str());
```

Alternativamente, podemos usar las funciones `stoi` y `stof` definidas en el estándar 2011 de C++. Estas funciones reciben directamente como parámetro un `string`, por lo que no es necesario hacer la conversión a vector de caracteres mediante `c_str`. El ejemplo anterior quedaría de la siguiente manera:

```
string cad1="100";
int n1=stoi(cad1);

string cad2="10.5";
float n2=stof(cad2);
```

Comparativa

Resumimos en la siguiente tabla cómo se hacen las mismas cosas utilizando cadenas de caracteres en C y utilizando el tipo `string`:

vectores de caracteres	`string`
`char cad[TAM];`	`string s;`
`char cad[]="hola";`	`string s="hola";`
`strlen(cad);`	`s.length();`
`cin.getline(cad,TAM);`	`getline(cin,s);`
`if(!strcmp(c1,c2)){...}`	`if(s1==s2){...}`
`strcpy(c1,c2);`	`s1=s2;`
`strcat(c1,c2);`	`s1=s1+s2;`
`strcpy(cad,s.c_str());`	`s=cad;`
Terminan con `\0`	NO terminan con `\0`
Tamaño reservado fijo	El tamaño reservado puede crecer

Tamaño ocupado variable	Tamaño ocupado = tamaño reservado
Se usan en ficheros binarios	NO se pueden usar en ficheros binarios

Tema 3: Ficheros

En este tema veremos como trabajar con ficheros de texto y también con ficheros binarios. Aprenderemos a realizar las operaciones básicas: apertura, lectura, escritura y cierre. Además veremos los principales problemas que nos podemos encontrar al trabajar con ficheros y sus soluciones.

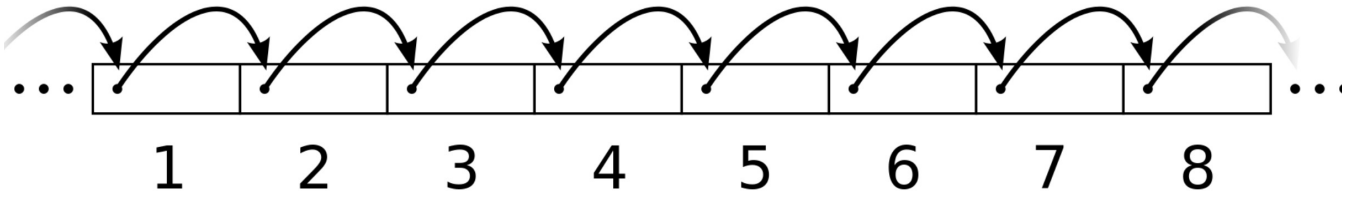
Introducción

Todos los datos con los que hemos trabajado hasta ahora se almacenan en la memoria principal del ordenador (*Random Access Memory* - RAM). Esta memoria permite un acceso rápido a los datos, pero su tamaño es limitado (unos cuantos *Gigabytes*). Además, los datos se borran cuando el programa termina o se desconecta el ordenador (*memoria volátil*).

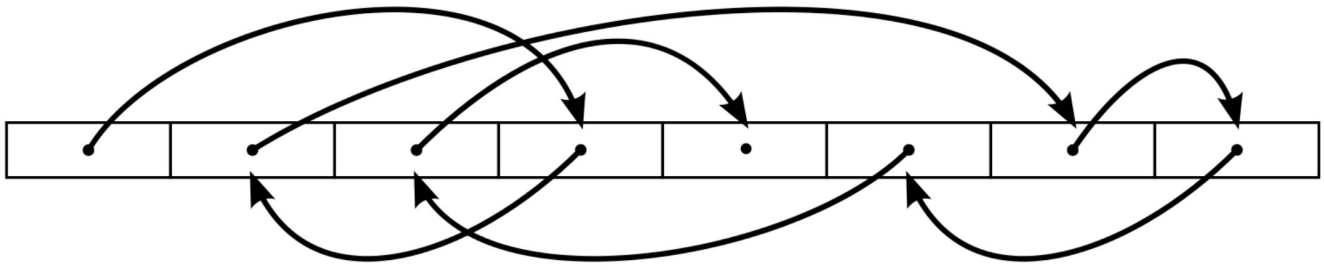
Los *ficheros* (o archivos) son la forma en la que C++ permite acceder a la información almacenada en disco (también conocido como *memoria secundaria*). Son estructuras dinámicas, ya que su tamaño puede variar durante la ejecución del programa según los datos que se necesiten almacenar.

Hay dos formas de acceder a un fichero:

- *Acceso secuencial*: se leen o escriben los elementos del fichero en orden, empezando por el principio y uno detrás de otro



- *Acceso directo* (o aleatorio): nos situamos en cualquier posición del fichero y lo leemos/escribimos directamente, sin pasar por los elementos anteriores



Existen dos tipos de ficheros en función de cómo se guarda dentro la información: *ficheros de texto* y *ficheros binarios*. En las siguientes secciones se describen cada uno de estos tipos.

Ficheros de texto

Definición

Los ficheros de texto (también llamados *ficheros con formato*) guardan la información en forma de secuencias de caracteres, tal como se mostrarían por pantalla. Por ejemplo, el valor entero `19` se guardará en fichero como los caracteres `1` y `9`. Este tipo de ficheros solo contiene caracteres imprimibles, es decir, aquellos cuyo código ASCII es mayor o igual a 32. Existen otros juegos de caracteres, como EBCDIC o Unicode (su codificación más comúnmente usada es UTF-8).

Algunos ejemplos de ficheros de texto son:

- El código fuente en C++ de tus prácticas
- Una página web, cuyo contenido está escrito en HTML
- Un fichero de configuración de un sistema operativo (.ini en windows, .conf en Unix)
- Un fichero *makefile* de dependencias de compilación de un proyecto en C++

El modo de lectura/escritura más habitual en ficheros de texto es el acceso secuencial.

Declaración de variables

Los ficheros son un tipo de datos más en C++. Para poder trabajar con ficheros de texto, debemos incluir la librería *fstream* en nuestro código:

```
1 #include <fstream>
```

Para declarar una variable de tipo fichero tenemos varias opciones, según el uso que le vayamos a dar.

- Si solo queremos acceder para leer el contenido del fichero:

  ```
  1 ifstream ficheroLec;
  ```

- Si solo vamos a usar el fichero para escribir en él:

  ```
  1 ofstream ficheroEsc;
  ```

- Si necesitamos acceder para ambas operaciones (lectura y escritura):

  ```
  1 fstream ficheroLecEsc;
  ```

No obstante, usar ficheros de texto en ambos modos simultáneamente es algo poco frecuente.

Apertura y cierre

Una variable de tipo fichero (*fichero lógico*) se ha de asociar a un fichero real en el sistema (fichero físico) para poder leer/escribir en él. Para establecer esta relación entre la variable y el fichero físico hay que hacer la apertura del fichero mediante la función `open`:

```
1 open(const char[] nombre,const int modo)
```

Ejemplo de uso:

```
1 ifstream fichero;
2 fichero.open("miFichero.txt");
```

El nombre del fichero se puede pasar como un array de caracteres o como un `string` (este último caso solo a partir de la versión 2011 de C++):

```
1 char nombre[]="mifichero.txt";
2 fichero.open(nombre);
```

A `open` se le puede pasar un segundo parámetro que indica el modo de apertura del fichero:

Modo	Observaciones	Constante en C++
Lectura	Sólo consultas, coloca el cursor al principio	`ios::in`
Escritura	Sólo escritura, colocando el cursor al principio, sobreescribiendo los contenidos	`ios::out`
Lectura y escritura	Permite leer y escribir contenidos en el fichero	`ios::in | ios::out`
Añadir al final	Sólo escritura y coloca el cursor al final del fichero para añadir contenidos en lugar de sobreescribirlos	`ios::out | ios::app`

Ejemplo de uso:

```
1 // Abrimos solo para leer
2 ifstream ficheroLec;
3 ficheroLec.open("miFichero.txt",ios::in);
4
5 // Abrimos para añadir información al final
```

```
ofstream ficheroEsc;
ficheroEsc.open("miFichero.txt",ios::out|ios::app);
```

Si abrimos un fichero que ya existe para escritura (`ios::out`) se borrará todo su contenido. Si abrimos con `ios::app` no borrará su contenido, sino que irá añadiendo la nueva información al final. Si el fichero no existe en algunos de estos dos casos, se creará uno nuevo con tamaño inicial `0`.

Por defecto, si no indicamos ningún modo de apertura, el tipo `ifstream` se abre para lectura y el `ofstream` para escritura. Además, se puede abrir el fichero en el momento de declararlo:

```
ifstream ficheroLec("miFichero.txt"); // Por defecto ios::in
ofstream ficheroEsc("miFichero.txt"); // Por defecto ios::out
```

La operación de abrir un fichero es delicada en tanto que puede producir errores en tiempo de ejecución. Es importante, tras abrir un fichero con `open`, comprobar si éste está correctamente abierto y listo para trabajar. Con la función `is_open()` podemos verificar si efectivamente nuestro fichero está abierto (`true`) o no (`false`). Además, al terminar de trabajar con el fichero, se debe cerrar para liberar el recurso con `close`:

```
ifstream ficheroLec("miFichero.txt");
if(ficheroLec.is_open()){
  // Ya se puede trabajar con el fichero
  ...
  ficheroLec.close(); // Cerramos el fichero
}
else{
  // Mostrar error de apertura
}
```

Lectura con el operador >>

La lectura de fichero permite recuperar información guardada en disco para ponerla en memoria y poder trabajar con ella. Los ficheros en C++ son objetos de la clase `stream`, los cuales permiten ser accedidos como *buffers* de entrada/salida, usando los operadores `>>` y `<<` de la misma forma que hacemos con los *buffers* `cin` y `cout` de entrada/salida estándar.

Ejemplo de lectura de fichero carácter a carácter:

```
ifstream fl("miFichero.txt")
if(fl.is_open()){
  char c; // Podríamos leer int, float, ...
  while(fl >> c){ // Lee mientras queden caracteres
    cout << c;
  }
  fl.close()
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

Al usar el operador `>>` descartamos los blancos (espacios en blanco, tabuladores y saltos de línea), al igual que sucedía con `cin`. Podemos usar la función `get` para leer todos los caracteres, sin descartar los blancos:

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
  char c;
  while(fl.get(c)){
    cout << c;
  }
  fl.close();
}
else{
  cout << "Error al abrir el fichero" << endl;
}
```

También se puede usar el operador `>>` para leer ficheros que contengan distintos tipos de datos. Por ejemplo, si tenemos un fichero que contiene en cada línea una cadena y dos enteros con una estructura como ésta:

```
hola 123 1024
mundo 43 23
```

El código para leer este fichero sería:

```
ifstream fichero("ejemplo.txt");
if(fichero.is_open()){
  string s;
  int i,j;
  while(fichero >> s){ // Leer string
    fichero >> i; // Leer primer numero
    fichero >> j; // Leer segundo numero
    cout << "Read: " << s << "," << i << "," << j << endl;
  }
}
```

En este otro ejemplo, leemos los datos de un fichero cuya estructura es: primero, un número entero que indica los nombres que hay a continuación y luego una sucesión de nombres separados por espacios o retornos de carro:

```
3 pedro antonio jordi
```

El código para leerlo sería este:

```
if(fichero.is_open()){
  string s;
```

```
  int cuantos;
  fichero >> cuantos;
  for(int i=0;i<cuantos;i++){
    fichero >> s;
    cout << "Nombre(" << i+1 << ")=" << s << endl;
  }
  fichero.close();
}
```

Lo primero que hacemos tras abrir el fichero es leer el número para saber cuantos nombres tenemos que leer. A continuación, mediante un bucle `for` leemos tanta `string` como nos indica la variable `cuantos` y los mostramos en pantalla.

Lectura por líneas

Podemos usar la función `getline` para leer una línea completa de fichero, al igual que hacíamos al leer de `cin`:

```
...
if(fichero.is_open()){
  string s="";
  // Hacemos una lectura anticipada, por si el fichero está vacío
  getline(fichero,s);
  while(!fichero.eof()){
    // Hacer algo con 's'
    getline(fichero,s);
  }
  fichero.close();
}
```

El bucle tiene un problema: si el fichero finaliza con un `\n` entonces produce una iteración de más. Eso se podría solucionar añadiendo una instrucción `if` que compruebe el contenido de la variable `s` antes de hacer algo con ella.

El código anterior se puede simplificar:

```
ifstream fichero("datos.txt");
if(fichero.is_open()){
  string s;
  while(getline(fichero,s)){
    // Hacer algo con 's'
  }
  fichero.close();
}
```

Esta otra forma de leer un fichero es más sencilla y **recomendable**. Aquí no usamos `eof()` porque la misma función `getline()`, cuando no puede leer falla y devuelve un valor `false` que provoca la salida del bucle.

Detección de final de fichero

Mientras leemos de un fichero tenemos que saber en qué momento hemos llegado al final del mismo y parar de leer. El método `eof()` nos indica si se ha alcanzado el final de fichero. Esta circunstancia se da cuando no quedan más datos por leer:

```
ifstream fichero;
...
while(!fichero.eof()){
  // Leemos utilizando alguno de los métodos vistos
}
```

Éste método devuelve un valor booleano (*true* o *false*) en función de si estamos o no al final de fichero. Cuando realizamos una operación de lectura de un dato (carácter, número, etc.) que no está en el fichero, devuelve *true*. **Cuidado**, después de haber leído **el último dato válido** todavía devolverá *false*. Por tanto, hace falta realizar una lectura adicional (cuyo resultado se debe descartar, pues son datos no válidos) para provocar la detección del final de fichero y, por tanto, que la función retorne *true*.

Escritura con el operador <<

Para escribir en un fichero de texto usaremos el operador de volcado `<<`, igual que hacíamos para escribir por pantalla con `cout`. Tal y como se indicó anteriormente, los ficheros son objetos *stream* y pueden ser usados como *buffers* de entrada/salida de datos.

```
ofstream fichero;
...
fichero.open("resultados.txt",ios::out);
if(fichero.is_open()){
  fichero << "El resultado es: " << numentero << endl;
  ...
  fichero.close();
}
```

En este ejemplo, abrimos un fichero en modo escritura y escribimos en él una frase y una variable llamada `numentero`. Otro ejemplo de escritura:

```
ofstream fichero;
...
fichero.open("resultados.txt",ios::out | ios::app);
if(fichero.is_open()){
  fichero << "Datos: ";
  for(int i=0;i<totales;i++){
    fichero << resultados[i] << " ";
  }
  fichero << endl; // Fin de linea al final del volcado de datos
  ...
  fichero.close();
}
```

En este otro ejemplo se escribe una línea de texto con el literal `Datos:` seguido de una sucesión de datos obtenidos durante el recorrido de un array llamado `resultados`. Cada dato estará separado por un espacio en blanco. Además, el fichero no será truncado en la apertura (al estar abierto con `ios::app`) y su contenido se mantendrá. Los datos de sucesivas ejecuciones serán añadidos al final del mismo.

Ficheros binarios

Definición

Un fichero binario es una secuencia de bits (habitualmente agrupados de ocho en ocho, es decir, en bytes). En estos ficheros la información se almacena tal y como están alojados en la memoria del ordenador, no se convierten en caracteres al guardarlos en el fichero. También son llamados *ficheros sin formato*.

Normalmente, para guardar/leer información en un fichero binario se usan registros (*structs*). De esta forma, es posible acceder directamente al *n-ésimo* elemento sin tener que leer los *n-1* anteriores. Es por esto que se dice que los ficheros binarios son ficheros de acceso directo (o aleatorio) y los ficheros de texto son de acceso secuencial.

Declaración de variables

Al igual que con los ficheros de texto, para poder trabajar con este tipo de ficheros, debemos incluir la biblioteca de funciones *fstream*:

```
1 #include <fstream>
```

Para declarar una variable de tipo fichero binario, según el uso que le vayamos a dar, tenemos:

- Solo para lecturas/consultas al fichero:

  ```
  1 ifstream fichLectura;
  ```

- Solo para escribir en él:

  ```
  1 ofstream fichEscritura;
  ```

- Si necesitamos acceder para ambas operaciones (lectura y escritura):

  ```
  1 fstream fichLecturaEscritura;
  ```

  Operaciones de apertura y cierre

Apertura y cierre

Para abrir un fichero binario, también usamos la función `open`, pero tenemos que añadir un modo adicional: `ios::binary`.

- Apertura para lectura:

  ```
  1 fichLectura.open("miFichero.dat",ios::in | ios::binary);
  ```

- Apertura para escritura:

```
fichEscritura.open("miFichero.dat",ios:out | ios::binary);
```

- Modo abreviado, declaración y apertura en una sola instrucción:

```
ifstream fbl("miFichero.dat",ios::binary); // Binaro de lectura
ofstream fbe("otroFichero.dat",ios::binary); // Binario de escritura
```

Combinado varios modos de apertura podemos obtener:

- Fichero binarios de lectura y escritura: `ios::in` | `ios::out` | `ios::binary`
- Ficheros binarios para escritura/añadir al final: `ios::out` | `ios::app` | `ios::binary`

Finalmente, para cerrar un fichero binario, igual que con los de texto, se usa la función `close()`:

```
fichBinario.close();
```

Lectura

Para leer ficheros binarios se utiliza la función `read()` a la que tenemos que pasar dos argumentos:

- El registro donde se almacenarán los datos tras la operación de lectura
- La cantidad de bytes que deseamos leer, la cual podemos obtener calculando el tamaño del registro con la función `sizeof()`

En el siguiente ejemplo leemos el contenido completo de un fichero binario formado por registros de tipo `TCiudad`:

```
struct TCiudad{
  ...
}
...
TCiudad ciudad;
ifstream fbl;

fbl.open("miFichero.dat",ios::in | ios::binary);

if(fbl.is_open()){
  while(fbl.read((char *)&ciudad,sizeof(ciudad))){
    // Procesar 'ciudad'
  }
  fbl.close();
}
```

Una vez comprobado que el fichero ha sido correctamente abierto, leemos su contenido mediante un bucle, invocando a cada iteración a la función *read*, pasándole como argumentos el registro `ciudad` y su

tamaño. En el cuerpo del bucle `while` trabajaremos con los datos de la ciudad recién leída. Finalmente,

Nota: para leer el contenido de un fichero binario, tenemos que conocer su estructura, es decir, el tipo de registros (`struct`) usado para almacenar datos en él.

También podemos acceder a un elemento directamente calculando su posición en el fichero en función del tamaño de los datos y la cantidad de elementos que hay antes. Para ello usamos la función `seekg()` , la cual recibe dos argumentos:

- El primero es la posición (en bytes) donde queremos desplazar el cursor o puntero de lectura. Puede tener un valor negativo, en cuyo caso realizará el desplazamiento hacia el inicio del fichero en lugar de desplazarse hacia el final
- El segundo parámetro es la referencia desde la que calcular la posición anterior

Este segundo parámetro, el de referencia, puede tener los siguientes valores:

Valor	Ejemplo	Descripción
`ios::beg`	`fi.seekg(pos,ios::beg)`	Desde el inicio del fichero
`ios::cur`	`fi.seekg(pos,ios::cur)`	Desde la posición actual del cursor de lectura
`ios::end`	`fi.seekg(pos,ios::end)`	Desde el final del fichero

Por ejemplo, si queremos acceder y leer el tercer elemento, haríamos:

```
1 ...
2 if(fbl.is_open()){
3   // Nos posicionamos justo ante el tercer elemento
4   fbl.seekg((3-1)*sizeof(ciudad),ios::beg); // Contamos el tercer registro de tamaño ciuda
5   fbl.read((char *)&ciudad,sizeof(ciudad));
6 }
7 ...
```

En este otro ejemplo, queremos acceder al último elemento del fichero:

```
1 ...
2 if(fbl.is_open()){
3   // Nos posicionamos al final del fichero con 'ios::end'
4   // Nos desplazamos un registro en direcció hacia el inicio (con -1)
5   fbl.seekg((-1)*sizeof(ciudad),ios::end);
6   fbl.read((char *)&ciudad,sizeof(ciudad));
7 }
8 ...
```

Escritura

Para escribir datos en un fichero binario se usa la función `write()` y, de forma parecida a la función de lectura, se le pasan dos parámetros:

- El registro que contiene los datos que van a enviarse al fichero
- La cantidad de bytes que vamos a escribir (podemos calcularla con la función `sizeof()`)

En el siguiente ejemplo de código escribimos en un fichero binario llamado `miFichero.dat` un registro de tipo `TCiudad`:

```
struct TCiudad{
  ...
};
...
TCiudad ciudad;
ofstream fbe("miFichero.dat",ios::binary);

if(fbe.is_open()){
  // introducimos datos en el registro 'ciudad'
  ...
  // escribimos en el fichero
  fbe.write((const char *)&ciudad,sizeof(ciudad));
  ...
}
fbe.close();
```

Si deseamos escribir en una posición concreta del fichero, podemos usar la función `seekp()`. Los argumentos son análogos a la función `seekg`:

- El primer parámetro es la ubicación (en bytes) donde queremos mover el cursor o puntero de escritura. Si la posición no existe en el fichero, éste se alargará para hacer posible la operación de escritura
- El segundo parámetro es la referencia o desplazamiento desde el que calcular la posición anterior. Los valores posibles son los mismo que para `seekg`

Por ejemplo, si deseamos escribir o modificar el quinto elemento de un fichero:

```
...
if(fbe.is_open()){
  // Nos posicionamos para escribir en el quinto elemento
  fbl.seekp((5-1)*sizeof(ciudad),ios::beg);
  fbl.write((const char *)&ciudad,sizeof(ciudad));
}
...
```

En este caso, si en el fichero hay 5 o más registros, sobreescribiremos el quinto con el contenido de la variable `ciudad`, pero si hubiera menos de cinco elementos entonces el fichero crecerá para permitir escribir el dato en la quinta posición.

Si tenemos un registro que contiene un campo de tipo cadena de caracteres y queremos almacenarlo en un fichero binario, tenemos que usar un vector de caracteres en lugar de un `string`. Al hacer la conversión

puede ser que tengamos que recortar el `string` para adecuarlo al tamaño del vector:

```
struct TCiudad{
  int codigo;
  char nombre[MAXLONG];
};

string s="Alicante";
...
TCiudad ciudad;
ciudad.codigo=3;
// Copiamos el string en un vector de caracteres
strncpy(ciudad.nombre,s.c_str(),MAXLONG-1);
ciudad.nombre[MAXLONG-1]='\0'; // strncpy no pone el \0 si no esta en la cadena original
...
fichero.write((const char *)&ciudad,sizeof(ciudad)); // Dscribimos el registro
...
```

Posición actual

Existen dos funciones que nos permiten obtener la posición actual del puntero (el de lectura y de escritura):

- Para el puntero de lectura se usa `tellg()`.
- Para el de escritura usamos `tellp()`.

Devuelven la posición en bytes.Por ejemplo, si tenemos un fichero abierto y queremos obtener la cantidad de registros que contiene:

```
// Colocamos el puntero de lectura al final:
fichero.seekg(0,ios::end);

// Obtenemos el número de registros del fichero
cout << fichero.tellg()/sizeof(elRegistro) << endl;
```

Tema 4: Memoria dinámica

En este tema veremos cómo se organiza la memoria asignada a las variables de un programa. Aprenderemos las diferencias entre memoria estática y memoria dinámica, y cómo usar la memoria dinámica para almacenar datos en memoria cuyo tamaño es variable y/o se desconoce en el momento de escribir el programa.

Organización de la memoria

Cada variable que usamos en nuestros programas se usa para contener información que se almacena en la memoria asignada al programa en tiempo de ejecución. La cantidad de memoria que ocupa cada variable viene determinada por el tipo de datos al que pertenece. Además, usamos una variable simple cuando queremos almacenar un único valor (por ejemplo, un número entero o un carácter) o un *array* cuando queremos almacenar varios datos del mismo tipo bajo el mismo nombre de variable, especificando la cantidad máxima de valores que podemos guardar en el array en su declaración.

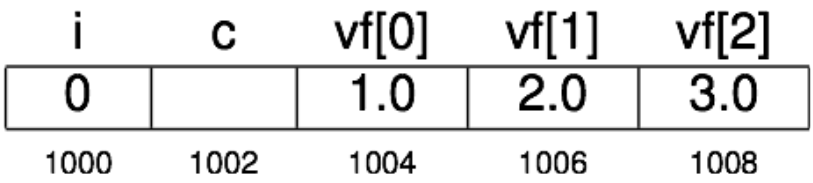
Sin embargo, no siempre podemos conocer la cantidad de información que necesitamos almacenar en memoria cuando escribimos el código. En ocasiones la cantidad de memoria necesaria sólo se conoce una vez que el programa se está ejecutando. Por ejemplo, si preguntamos al usuario el tamaño de una matriz y a continuación le pedimos que introduzca los valores a guardar dentro de la matriz.

Memoria estática

Se denomina memoria estática al conjunto de todas las variables cuyo tamaño es fijo y se conoce al implementar el programa. Se incluyen en esta categoría todas las variables y parámetros locales de las funciones (simples o *arrays*), así como las variables globales, declaradas tal como hemos estado haciendo hasta ahora. Por ejemplo:

```
1 int i=0;
2 char c;
3 float vf[3]={1.0,2.0,3.0};
```

En el siguiente gráfico podemos ver una representación de una posible asignación de direcciones de memoria a cada una de las variables del ejemplo anterior. Las celdas representan el valor que contiene la variable y los números debajo de cada celda la posición que ocupa en memoria (en realidad las direcciones de memoria suelen ser muy distintas, pero lo hemos representado así por simplicidad).



Memoria dinámica

Además de la memoria estática, un programa puede hacer uso de otra zona de memoria para almacenar información. Normalmente se utiliza para almacenar grandes volúmenes de datos, cuya cantidad exacta se desconoce al implementar el programa. La cantidad de datos a almacenar se calcula durante la ejecución del programa y además puede cambiar mientras el programa esté en ejecución.

Para hacer uso de esta **memoria dinámica** en C++ se usan unas variables especiales llamadas **punteros**.

Zonas de la memoria

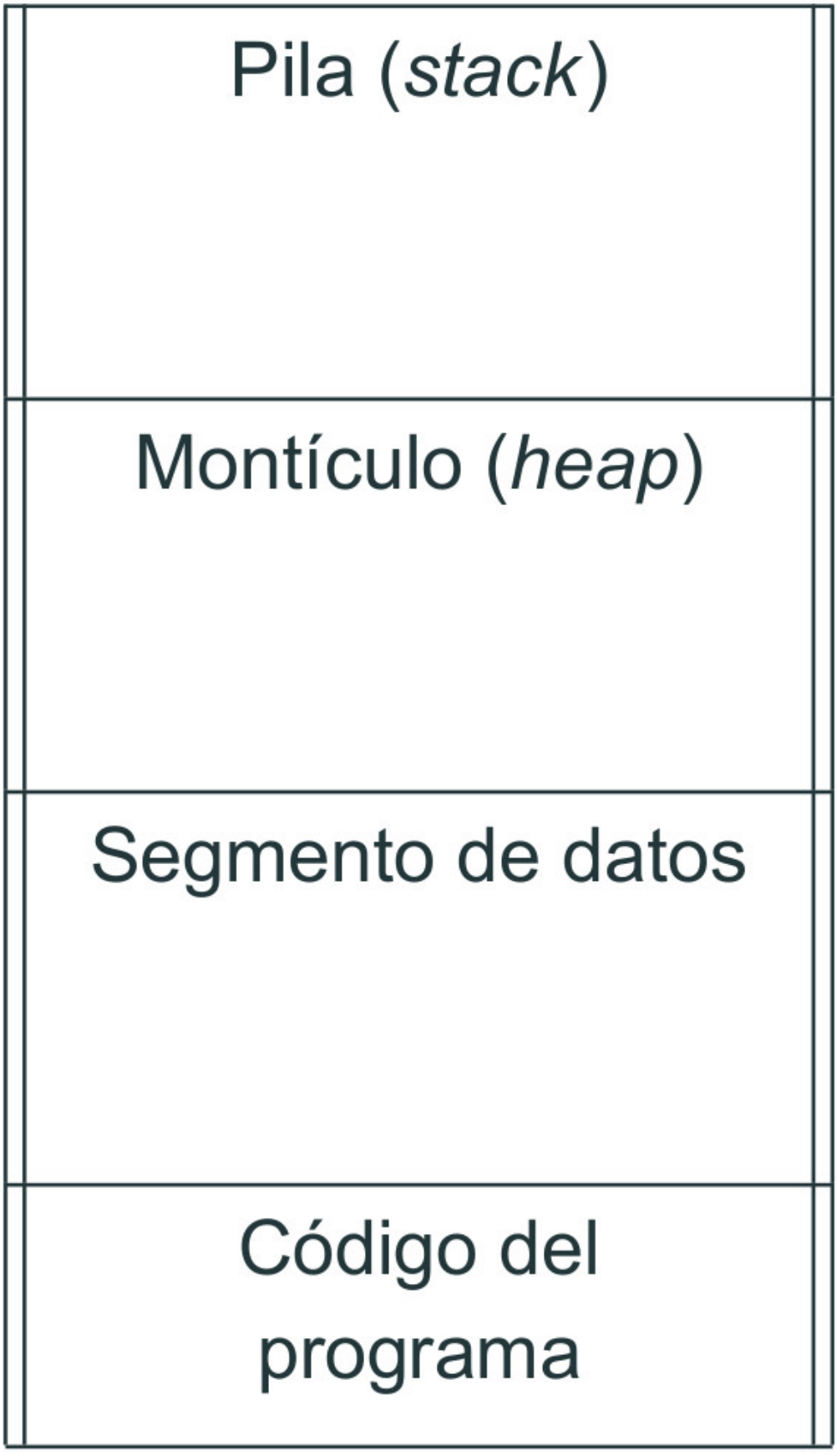
Durante la ejecución de un programa se utilizan zonas diferenciadas de la memoria:

- La *pila* almacena los datos locales de una función: parámetros por valor y variables locales
- El *montículo* almacena los datos dinámicos que se van reservando durante la ejecución del programa
- En el *segmento de datos* se almacenan los datos de estos tipos, cuyo tamaño se conoce en tiempo de

compilación

- El propio *código* también se almacena en la memoria, como los datos

La siguiente imagen muestra la estructura de las zonas de memoria:



Punteros

Definición y declaración

Un puntero es un **número** (entero largo) que se corresponde con una dirección de memoria. Para usar un puntero es necesario declarar una variable especificando que se trata de un puntero, además del tipo de dato que contendrá esa dirección de memoria, de manera que cuando accedamos a ella se pueda interpretar correctamente su valor.

Los punteros se declaran usando el carácter `*` antes del nombre de la variable. Por ejemplo:

```
1 int *punteroEntero;
2 char *punteroChar;
3 int *vectorPunterosEntero[20];
4 double **doblePunteroReal;
```

Como puedes ver en el último ejemplo, un puntero puede apuntar también a una dirección de memoria donde hay almacenado otro puntero (indicado en el ejemplo como `**`).

Operadores de punteros

En C++ hay dos operadores que podemos usar para trabajar con punteros:

- `*x`: contenido de la dirección apuntada por `x`
- `&x`: dirección de memoria de la variable `x`

Usaremos el operador `*` para acceder al valor almacenado en la dirección de memoria a la que apunta un puntero, tanto para obtener su valor como para modificarlo. Sin embargo, antes de poder acceder a un puntero tenemos que asegurarnos de que apunta a una dirección de memoria válida.

La forma más sencilla de asignar una dirección de memoria a un puntero es asignarle la dirección de una variable estática usando el operador `&`. De esta forma podemos usar el puntero para leer o modificar su valor como si se tratara de la misma variable. Por ejemplo:

```
1 int i=3;
2 int *pi; // El puntero todavía no está inicializado
3 pi=&i; // Inicializamos el puntero pi a la direccion de memoria de i
4 *pi=11; // Contenido de pi = 11. Por lo tanto, i = 11
```

En el siguiente gráfico puedes ver cómo quedaría el valor de las variables después de ejecutar el código anterior:

1000 1002 1004 1006 1008

Como puedes observar, al ejecutar `pi=&i` el valor de `pi` (el puntero) es `1000`, es decir, la dirección donde está almacenada la variable `i`. Al ejecutar la instrucción `*pi=11` se accede a esta posición de memoria y se almacena el valor `11`, por lo que en realidad es como si hubiésemos modificado directamente la variable `i`.

Declaración con inicialización

Igual que sucede con el resto de variables, es recomendable inicializar siempre un puntero en el momento de su declaración. Por ejemplo:

```
1 int *pi=&i; // pi contiene la direccion de i
```

Cuando declaramos un puntero pero todavía no sabemos a qué dirección tiene que apuntar es muy conveniente usar el valor especial `NULL`. El puntero `NULL` es aquel que no apunta a ninguna variable:

```
1 int *pi=NULL;
```

Siempre que un puntero no tenga memoria asignada debe valer `NULL`. Ésto nos permitirá comprobar antes de acceder a un puntero si apunta a una dirección de memoria válida. Por ejemplo:

```
1 if(pi!=NULL){
2   *pi=11;
3 }
```

Uso de punteros

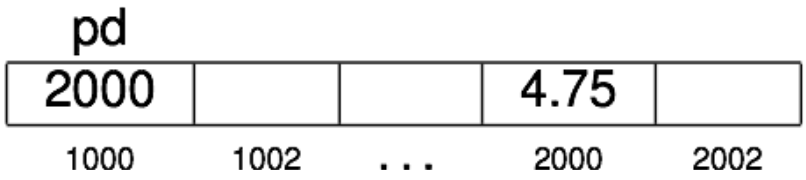
Hasta ahora hemos visto cómo usar punteros para acceder a variables estáticas. Sin embargo, la principal utilidad de los punteros es poder reservar dinámicamente memoria cada vez que necesitamos almacenar nueva información, sin tener que recurrir a variables estáticas.

Reserva y liberación de memoria

Para asignar una nueva posición de memoria dinámica a un puntero tenemos que **reservar** memoria con el operador `new`. En el momento en que esta memoria ya no es necesaria debemos **liberarla** usando el operador `delete`. Por ejemplo:

```
1 double *pd;
2 pd=new double; // Reserva memoria
3 *pd=4.75;
4 cout << *pd << endl; // Muestra el valor apuntado por pd (4.75)
5 delete pd; // Libera la memoria
```

Como puedes ver en la segunda línea, al reservar memoria hay que especificar de nuevo el tipo de dato que contendrá dicha memoria. Esto es necesario para que el programa reserve la cantidad de memoria necesaria (en bytes) para almacenar la nueva información. En el siguiente gráfico puedes ver la representación de las variables del ejemplo anterior:



Las variables locales y las reservadas con `new` van a zonas de memoria distintas. Cuando se usa `new`, el programa reserva memoria y devuelve la dirección de inicio de ese espacio reservado; esta dirección se almacena en el puntero. Si no hubiera suficiente memoria disponible no se podrá completar la reserva; en este caso el operador `new` devuelve `NULL`.

Siempre que se reserva memoria con `new` hay que liberarla con `delete` cuando ya no es necesario seguir usándola, de manera que ese espacio queda disponible para futuras reservas. Si un programa no libera correctamente la memoria dinámica reservada puede llegar a agotar toda la memoria disponible, si realiza muchas reservas o se ejecuta ininterrumpidamente durante mucho tiempo.

Tras hacer `delete`, el puntero no vale `NULL` por defecto**.** Si necesitas seguir usando ese puntero, es una buena práctica asignarle el valor `NULL` manualmente para evitar acceder a una zona de memoria que ya no está disponible para el programa. Recuerda que en programas más complejos deberías comprobar siempre si el puntero es distinto de `NULL` antes de acceder a su valor, especialmente cuando los punteros se usan en un lugar del código alejado de su declaración. También puedes volver a asignarle otra dirección de memoria usando `new`.

Punteros y arrays

En el ejemplo anterior hemos usado un puntero para reservar memoria que almacenará una única variable. Sin embargo, los punteros pueden usarse también para crear *arrays* o matrices. Para reservar memoria para un *array* hay que especificar su tamaño entre corchetes `[]`. También será necesario usar los corchetes para liberar la memoria al usar `delete`. Por ejemplo:

```
1 int *pv;
2 int n=10;
3 pv=new int[n]; // Reserva memoria para n enteros
4 pv[0]=585; // Usamos el puntero como si fuera un vector
5 delete [] pv; // Libera la memoria reservada, no hay que indicar el tamaño entre corchetes
```

Como los punteros pueden almacenar la dirección de memoria de cualquier variable, también se pueden usar para acceder directamente a una posición de un vector. Por ejemplo:

```
1 int v[TAM];
2 int *pv;
3 pv=&(v[7]); // pv contiene la posición de memoria del octavo elemento del vector v
4 *pv=117; // v[7]=117
```

Punteros definidos con typedef

Como ya vimos en el Tema 1, podemos usar el operador `typedef` para definir nuevos tipos de datos:

```
typedef int entero; // Entero es un tipo, como int
entero a,b; // Equivale a int a,b;
```

Para facilitar la claridad en el código, pueden definirse los punteros con `typedef`:

```
typedef int *punteroAEntero;
int i=0;
punteroAEntero pi=&i; // Equivale a int *pi;
```

Punteros a registros

Los punteros se pueden usar también para contener registros, igual que si se tratara de un tipo de datos simple. Una vez se le ha asignado memoria, podemos acceder a los campos del registro usando el **operador punto** (`.`) como si se tratara de un registro normal, aunque para esto es necesario acceder primero a la dirección del puntero con el operador `*`. Por ejemplo:

```
struct TRegistro{
  char c;
  int i;
};

TRegistro *pr;
pr=new TRegistro;
(*pr).c='a'; // Asigna un valor al campo c del registro
```

En el ejemplo anterior los paréntesis alrededor de la expresión `*pr` son necesarios, ya que si no interpretaría como un puntero el valor de `c` (como si hubiésemos escrito `*(pr.c)`).

Para evitar confusiones, para acceder a los campos de un registro referenciado por un puntero podemos usar el **operador** `->`. Por ejemplo:

```
struct TRegistro{
  char c;
  int i;
};

TRegistro *pr;
pr=new TRegistro;
pr->c='a'; // (*pr).c = 'a';
```

Punteros como parámetros de funciones

Las variables de tipo puntero se pueden pasar también como parámetro a una función. Esto permite pasar variables reservadas dinámicamente a otra función. Por ejemplo:

```
void f(int *p){ // El puntero se pasa por valor
  *p=2;
}

int main(){
  int *q=new int;
  f(q); // La función modifica el valor referenciado por el puntero
  cout << *q << endl; // Imprimirá el valor 2
}
```

Aunque un puntero se pase por valor, se puede modificar el contenido de la dirección de memoria a la que apunta. Ten en cuenta que si pasamos un puntero por referencia, estamos permitiendo que se modifique la dirección de memoria que contiene, pudiendo hacer que apunte a otra zona de memoria distinta. Por ejemplo:

```
void f2 (int *&p){ // El puntero se pasa por referencia
  int num=10;
  p=&num; // Ahora el puntero apunta a la dirección de la variable num
}

int main(){
  int i=0;
  int *p=&i;
  f2(p); // La función modifica la dirección de memoria contenida en p
  cout << *p << endl; // ¡ERROR! num ya no existe, no podemos acceder a su dirección
}
```

En el ejemplo anterior hemos hecho que el puntero apunte a una variable local de la función. Al terminar la ejecución de la función desaparecen de la memoria todas sus variables locales, por lo que la dirección de memoria contenida en el puntero ha dejado de ser válida. Si intentamos acceder a su contenido podemos tener un error de **violación de segmento** y el programa terminaría de forma abrupta.

Finalmente, si usamos `typedef` para definir tipos de datos con punteros, podemos usar estos nuevos tipos para pasar punteros como parámetros a funciones. Por ejemplo:

```
typedef int* PInt;
void f(PInt p){ // Por valor
  *p=2;
}

void f2(PInt &p){ // Por referencia
  int num=10;
  p=&num;
}

int main(){
  int i=0;
```

```
13   PInt p=&i;
14   f(p);
15   f2(p);
16 }
```

Errores comunes

A continuación se detallan los errores más comunes cuando se usan punteros en un programa:

- **Utilizar un puntero sin haberle asignado memoria.** Igual que sucede con las variables de tipos de datos simples sin inicializar, un puntero puede contener una dirección de memoria arbitraria si se crea y no se inicializa. Tampoco se debe acceder a punteros que contienen el valor `NULL`.

```
1 int *pEntero;
2 *pEntero=7; // Error: no apunta a ninguna zona de memoria válida
```

- **Usar un puntero tras haberlo liberado.** Igual que en el caso anterior, una vez liberada la memoria de un puntero ya no es seguro acceder a su contenido.

```
1 punteroREGISTRO p,q;
2 p=new REGISTRO;
3 ...
4 q=p;
5 delete p; // Se libera la zona de memoria a la que apunta el puntero
6 q->num=7; // Error: el puntero no apunta a una zona válida
```

- **Liberar memoria no reservada.** Si se intenta liberar la memoria de un puntero que guarda una referencia a una variable estática el programa terminará con un error de ejecución.

```
1 int *p=&i;
2 delete p;
```

Tema 5: Introducción a la programación orientada a objetos

Introducción

Definición

La programación que hemos estudiado hasta ahora con lenguajes como C sigue los principios del llamado *paradigma procedimental*, bajo el que un programa es una colección de funciones (a veces llamadas también *procedimientos*) que se invocan sucesivamente durante la ejecución. Los datos, además, suelen estar disociados de las funciones que los manipulan. El paradigma de la *programación orientada a objetos*

propone encapsular los datos y las funciones que los manejan bajo el concepto de *clase* (por ejemplo, una clase de tipo vector). Estas clases se *instancian* una o más veces durante la ejecución del programa (por ejemplo, usando distintos vectores) para crear *objetos*. Así, un programa se compone ahora de una colección de objetos que interactúan y se comunican entre ellos. Estos objetos, como hemos dicho, son instancias de una o más clases definidas para el problema en cuestión. Como veremos más adelante, las clases se organizan de forma jerárquica mediante un mecanismo conocido como *herencia*.

Ya que los distintos objetos se comunican mediante llamadas a funciones (en realidad, mediante un concepto más amplio denominado *paso de mensajes*), los lenguajes orientados a objetos permiten hasta cierto punto su uso en programas que siguen el paradigma procedimental. C++ es un claro ejemplo de ello. Aunque no todos los programas escritos en C son válidos en C++ (ni, evidentemente, a la inversa), muchos de ellos pueden ser transformados de forma relativamente sencilla en programas de C++. Por tanto, es posible en principio escribir programas en C++ que están más cerca del paradigma procedimental que del orientado a objetos, pero a la hora de la verdad los programadores suelen usar C++ siguiendo el paradigma orientado a objetos.

Clases y objetos

En la asignatura ya hemos usado clases y objetos. No lo hemos hecho cuando declaramos una variable como:

```
1 int i; // Declaramos una variable i de tipo int
```

ya que en C++ los tipos como `int` o `float` se consideran *tipos primitivos* y no clases, ya que hay cosas que podemos hacer con clases y objetos que no podemos hacer con ellos (por ejemplo, sobrecargar sus operadores). Sin embargo, sí hemos declarado objetos del tipo (o clase) `string`:

```
1 string s; // Declaramos un objeto s de clase string
```

Las clases o tipos compuestos son similares a los tipos simples (o primitivos) pero permiten muchas más funcionalidades.

Podemos considerar una clase como un modelo para crear (*instanciar*) objetos de ese tipo que define las características que les son comunes a todos ellos. Un objeto de una determinada clase se denomina una instancia de la clase (`s` es una instancia de `string`). En una primera aproximación, una clase es similar a un registro como los estudiados hasta el momento, pero añadiendo funciones. De esta manera conseguimos *encapsular* los datos y las funciones (llamadas también funciones miembro o *métodos* miembro) que los manipulan.

El programador de una clase puede decidir que sólo algunas funciones se puedan invocar desde el código externo (código cliente) que declara y usa objetos de la clase (lo que constituiría la parte *pública* de la clase) y que el resto de funciones e incluso los datos de la clase sean *privados* y solo se pueda acceder a ellos desde el código de la propia clase (y, a veces, desde el código de otras clases con privilegios especiales). A la parte pública se le conoce también como *interfaz* de la clase. A la idea de esconder ciertos detalles al código cliente se le denomina *ocultación de información* y permite, entre otras cosas, que el programador de una clase modifique la representación usada para los datos de la clase sin que haya que modificar el código cliente que usa objetos de dicha clase.

El equivalente aproximado al siguiente registro en C++:

```
struct Fecha{
  int dia;
  int mes;
  int anyo;
};
```

sería una clase como la siguiente:

```
class Fecha{
  public:
    int dia;
    int mes;
    int anyo;
}; // Ojo: el punto y coma del final es necesario
```

Si no se indica lo contrario (con la palabra reservada `public`), todos los miembros de la clase son privados. El acceso a los elementos públicos de un objeto se realiza de forma similar a como se hace con registros:

```
Fecha f;
f.dia=12;
```

Sin embargo, como se ha comentado anteriormente, lo habitual es *esconder* del código cliente los datos de la clase de forma que en nuestro ejemplo la modificación desde fuera de la clase del atributo `dia` se tenga que realizar invocando un método público de la clase:

```
class Fecha{
  private: // Solo accesible desde métodos de la clase
    int dia;
    int mes;
    int anyo;
  public:
    bool setFecha(int d,int m,int a){...};
};
```

Conceptos básicos

A continuación introduciremos brevemente algunos principios en los que se basa el paradigma orientado a objetos: abstracción, encapsulación, modularidad, herencia y polimorfismo.

Abstracción

La *abstracción* es el mecanismo mediante el que determinamos las características esenciales de un objeto y su comportamiento en el contexto del programa que queremos escribir, a la vez que descartamos todo aquello que no es relevante en dicho contexto. La abstracción implica renunciar a una parte de la realidad (por ejemplo, la estatura de un contribuyente no suele ser relevante para una aplicación de gestión tributaria, pero sí lo será probablemente en un programa de supervisión dietética) y definir un modelo de esta adecuado para un propósito concreto. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevas clases, y tiene lugar en la fase de diseño que precede a la de implementación.

Encapsulación

La *encapsulación* significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad al mismo nivel de abstracción. En programación orientada a objetos, cada objeto puede realizar tareas, informar y cambiar su estado, y comunicarse con otros objetos en el sistema sin revelar cómo se implementan estas características. El *código cliente* no tiene acceso a los detalles de implementación e interactúa con cada objeto a través de la interfaz de su clase.

La interfaz es la parte del objeto que es visible para el resto de los objetos (la parte pública): es el conjunto de métodos (y a veces datos) del cual disponemos para comunicarnos con un objeto. Cada objeto, por tanto, oculta su implementación y expone al resto de la aplicación una interfaz. Tanto la implementación como la interfaz son compartidas por todos los objetos de la misma clase.

La encapsulación protege a las propiedades de un objeto contra su modificación: solamente los propios métodos internos del objeto pueden acceder a su estado. Además, el programador de una clase puede realizar cambios en la implementación y, mientras la interfaz no cambie, no será necesario modificar el código cliente.

Modularidad

Se denomina *modularidad* a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos) tan independientes como sea posible. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos.

Generalmente, cada clase se implementa en un módulo independiente, aunque clases con funcionalidades similares pueden pertenecer al mismo módulo. Lo ideal es que los componentes de un módulo estén bien cohesionados y que el acoplamiento entre módulos sea bajo. De esta manera, los cambios en una funcionalidad concreta de una aplicación implicarán normalmente cambiar las clases de un único módulo y no modificarlas de otros módulos.

Herencia

Las clases se pueden relacionar entre sí formando una jerarquía de clasificación en un mecanismo conocido como *herencia*. Por ejemplo, un coche (*subclase*) o una moto (otra subclase) son vehículos (*superclase*). La superclase define las propiedades y comportamiento común y las diferentes subclases los especializan. Los objetos de las subclases heredan las propiedades y el comportamiento de todas las clases a las que pertenecen (un objeto de la clase *Coche* es también un objeto de la clase *Vehículo* y puede ser tratado a conveniencia como uno u otro).

La herencia facilita la organización de la información en diferentes niveles de abstracción. Así, los objetos derivados pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Cuando

un objeto hereda de más de una clase se dice que hay *herencia múltiple*. C++ permite definir relaciones de herencia múltiple, pero no todos los lenguajes lo hacen.

Polimorfismo

El *polimorfismo* es la asignación de la misma interfaz a entidades de tipos diferentes. Por ejemplo, el método `desplazar()` puede referirse a acciones distintas si se trata de una moto o de un coche. En este caso la idea es que comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre. Otra de las formas más usadas de polimorfismo es el conocido como *polimorfismo de subtipos*, en el que una misma variable puede referenciar a instancias de diferentes clases que comparten una misma superclase, como en el siguiente ejemplo de variable polimórfica:

```
Animal *a=new Perro;
...
a=new Gato;
...
a=new Gaviota;
```

El polimorfismo implica una ambigüedad que ha de ser resuelta en algún momento. Cuando esto ocurre en *tiempo de ejecución*, esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en *tiempo de compilación*) de polimorfismo, tales como las plantillas (*templates*) y la sobrecarga de operadores de C++.

Programación orientada a objetos en C++

Sintaxis

El siguiente fichero de cabecera define la clase `SpaceShip` con una serie de atributos privados (que no forman parte de la interfaz pública de los objetos de la clase) y una serie de métodos que los manipulan (también conocidos como *funciones miembro*):

```
// SpaceShip.h (declaracion de la clase)
class SpaceShip{
  private:
    int maxSpeed;
    string name;
  public:
    SpaceShip(int maxSpeed,string name); // Constructor
    ~SpaceShip(); // Destructor
    int trip(int distance);
    string getName() const;
};
```

La implementación de los métodos de la clase `SpaceShip` se realiza habitualmente en otro fichero:

```
// SpaceShip.cc (implementacion de los metodos)
```

```
#include "SpaceShip.h"

// Constructor
SpaceShip::SpaceShip(int maxSpeed,string name){
  this->maxSpeed=maxSpeed;
  this->name=name;
}

// Destructor
SpaceShip::~SpaceShip(){}

int SpaceShip::trip(int distance){
  return distance/maxSpeed;
}

string SpaceShip::getName() const{
  return name;
}
```

Muchas de las características del código anterior se irán analizando en las próximas secciones.

Diseño modular

En C++ el programa principal `main` usa y comunica las clases. Una clase `Clase` se implementa con dos ficheros fuente: `Clase.h`, que contiene constantes que se usen en este fichero, la declaración de la clase y la de sus métodos; y `Clase.cc`, que contiene constantes que se usen en este fichero y la implementación de los métodos (y puede que la de tipos internos que use la clase).

La tarea de traducir un programa fuente en ejecutable se realiza en dos fases:

- Compilación: en C++ el compilador traduce un programa fuente en un programa en código objeto (no ejecutable)
- Enlace: el enlazador o *linker* de C++ junta el programa en código objeto con las librerías del lenguaje (C/C++) y genera el ejecutable

En C++, se realizan las dos fases a la vez con la siguiente instrucción:

```
g++ programa.cc -o programa
```

Con la opción `-c`, sin embargo, solo se realiza la compilación a código objeto (.o) pero sin hacer el enlace:

```
g++ programa.cc -c
```

Cuando un programa se compone de varias clases, para obtener el ejecutable se debe compilar cada clase por separado, obteniendo varios ficheros en código objeto y, a continuación, enlazar los ficheros en código objeto (las clases compiladas) con las librerías del sistema y generar un ejecutable. Para compilar cada módulo y el programa principal por separado se ejecutaría lo siguiente:

```
g++ -Wall -g -c C1.cc
g++ -Wall -g -c C2.cc
g++ -Wall -g -c prog.cc
```

Para enlazar las clases compiladas y el programa, y obtener el ejecutable, haríamos:

```
g++ -Wall -g C1.o C2.o prog.o -o prog
```

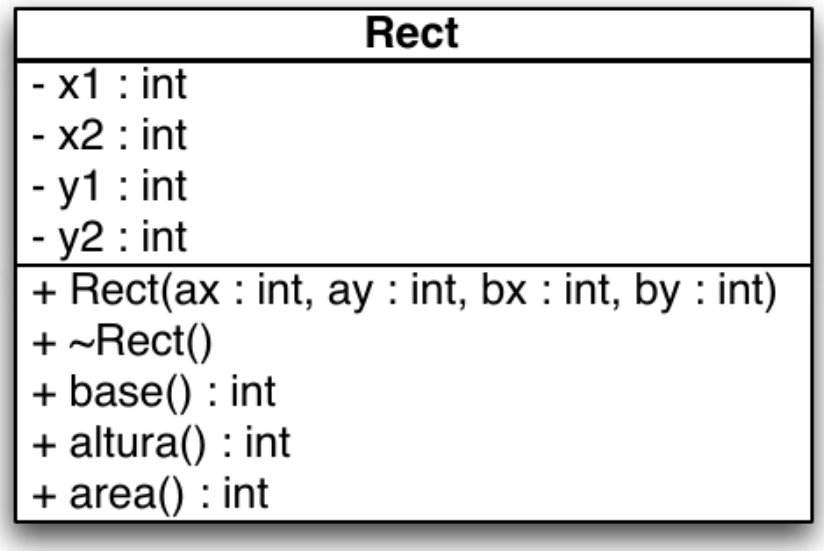
En el caso de programas pequeños, puede hacerse todo de una vez con:

```
g++ -Wall -g C1.cc C2.cc prog.cc -o prog
```

Cuando los programas son más grandes, no debemos recompilar cada vez todos los ficheros tras haber hecho un cambio mínimo en uno de ellos, ya que el tiempo necesario puede ser muy grande y entorpecer la labor del desarrollador. Más adelante veremos cómo el programa `make` se emplea para gestionar proyectos con decenas, cientos o incluso miles de clases.

Otro ejemplo de clase en C++

Las líneas siguientes muestran el código generado para la clase `Rect` representada en el siguiente diagrama UML de clases:




```
// Rect.h (declaracion de la clase)
class Rect{
  private:
    int x1,y1,x2,y2;
  public:
    Rect(int ax,int ay,int bx,int by); // Constructor
    ~Rect(); // Destructor
    int base();
```

```
    int altura();
    int area();
};
```

```
// Rect.cc (implementacion de métodos)
Rect::Rect(int ax,int ay,int bx,int by){
  x1=ax;
  y1=ay;
  x2=bx;
  y2=by;
}

Rect::~Rect(){}

int Rect::base(){ return (x2-x1); }
int Rect::altura(){ return (y2-y1); }
int Rect::area(){
  return base()*altura();
}
```

```
// main.cc
int main(){
  Rect r(10,20,40,50);
  cout << r.area() << endl;
}
```

Funciones inline

En escenarios donde el rendimiento es importante, los métodos con poco código también se pueden implementar directamente en la declaración de la clase. El código generado para las funciones `inline` se inserta en cada punto donde se invoca a la función, en lugar de hacerlo una sola vez en otro lugar y hacer una llamada. Para que una función sea considerada como `inline` basta con definirla dentro de la declaración de la clase:

```
// Rect.h (declaracion de la clase)
class Rect{
  private:
    int x1, y1, x2, y2;
  public:
    Rect(int ax, int ay, int bx,int by);
    ~Rect() {}; // Inline
    int base(){ return (x2-x1); }; // Inline
    int altura(){ return (y2-y1); }; // Inline
    int area();
};
```

Las funciones *inline* también se pueden implementar fuera de la declaración de clase (en el `.cc`) usando

la palabra reservada `inline`:

```
inline int Rect::base(){
  return(x2-x1);
}
```

Hay que reseñar que el compilador puede decidir *motu proprio* no implementar como *inline* una función que en principio ha sido declarada como tal.

Métodos accesores

Por el principio de encapsulación ya comentado, no es conveniente permitir al código cliente acceder directamente a los datos miembro de una clase. Lo normal es definirlos como `private` e implementar funciones *set/get/is* (llamadas `accesores`) que permitan acceder a ellos desde el exterior de la clase.

Los accesores `set` nos permiten controlar que los valores de los atributos sean correctos y solo modificarlos después de comprobar la validez de los nuevos valores.

Forma canónica

Todas las clases deben implementar al menos cuatro métodos importantes:

- Constructor
- Destructor
- Constructor de copia
- Operador de asignación (no lo estudiaremos en este curso)

Estas operaciones conforman lo que se conoce como *forma canónica* de una clase en C++ y son definidas *de oficio* por el compilador si el programador no las aporta.

Constructores

Las clases suelen tener al menos un método *constructor* y otro *destructor*. El constructor se invoca automáticamente cuando se crea un objeto de la clase, y el destructor cuando se termina de usar. El constructor se suele encargar de inicializar los atributos del objeto y de reservar recursos adicionales como la memoria dinámica necesaria para ellos; el destructor habitualmente libera estos recursos. Si no definimos un constructor, el compilador creará uno por defecto sin parámetros y que no hará nada. Los datos miembros de los objetos declarados así contendrán basura. En una clase puede haber varios constructores con parámetros distintos; diremos en ese caso que el constructor está *sobrecargado* (la sobrecarga es un tipo de polimorfismo), como en el siguiente ejemplo:

```
Fecha::Fecha(){
  dia=1;
  mes=1;
  anyo=1900;
}

```

```
Fecha::Fecha(int d,int m,int a){
  dia=d;
  mes=m;
  anyo=a;
}
```

Los siguientes son tres ejemplos de llamadas a los constructores anteriores:

```
Fecha f;
Fecha f(10,2,2010);
Fecha *f1=new Fecha(11,11,2011);
```

Aunque en otros lenguajes lo siguiente es una forma válida de ejecutar el constructor sin parámetros, en C++ no lo es:

```
Fecha f(); // Error de compilación
```

Los constructores pueden tener parámetros con valores por defecto que solo deben ponerse en el fichero `.h`:

```
Fecha(int d=1,int m=1,int a=1900);
```

Con este constructor podemos entonces crear un objeto de varias formas:

```
Fecha f;
Fecha f(10,2,2010);
Fecha f(10); // dia=10
```

Excepciones

Las excepciones son el mecanismo que permite gestionar de forma eficiente los errores que, por diversas causas, se producen en un programa durante su ejecución. Un uso habitual de las excepciones se produce cuando un constructor no puede crear el objeto correspondiente porque los parámetros suministrados por el código cliente son incorrectos. En casos como este, el constructor detecta el error pero no sabe qué hacer para solucionarlo (puede ser cambiar el valor del parámetro incorrecto y volverlo a intentar, mostrar un mensaje y pedir por consola un nuevo valor, informar al usuario a través de una ventana gráfica, emitir un sonido de alerta, terminar inmediatamente la ejecución del programa, etc.); es el código cliente que ha invocado el constructor el que probablemente sabe lo que hay que hacer para tratar el error. En otras ocasiones, puede que el tratamiento del error no se pueda hacer tampoco en la función que ha invocado al constructor, sino en la función que invocó a esta función. La idea aquí es que una vez lanzada una excepción en el punto donde se detecta el error, la excepción circulará *hacia atrás* por el programa, hasta un lugar donde se *capture* y se trate. Las excepciones no son necesarias en aquellos casos en los que la misma función que detecta un error es capaz de gestionar la situación y solucionar el problema.

Las excepciones en C++ se lanzan con la instrucción `throw` y se capturan en un bloque `try`/`catch` en el que la parte del `try` contiene el código que puede potencialmente lanzar una excepción y la parte

`catch` contiene el código que gestiona el error. Si se produce una excepción y no se captura ni siquiera desde la función `main`, el programa terminará. La función `root` del siguiente ejemplo lanza una excepción si su parámetro es negativo; la función `main` captura la excepción y muestra un mensaje de error.

```
int root(int n){
  if(n<0){
    throw exception(); // La funcion finaliza con una excepción
  }
  return sqrt(n);
}

int main(){
  try{ // Intentamos ejecutar las siguientes instrucciones
    int result=root(-1); // Provoca una excepcion
    cout << result << endl; // Esta linea no se ejecuta
  }
  catch(...){ // Si hay una excepcion la capturamos aquí
    cerr << "Negative number" <<  endl;
  }
}
```

Volviendo al tema de los constructores, este es un ejemplo de constructor con excepción:

```
Coordenada::Coordenada(int cx,int cy){
  if(cx>=0 && cy>=0){
    x=cx;
    y=cy;
   }
   else{
     throw exception();
   }
}
```

```
int main(){
  try{
    Coordenada c(-2,4); // Este objeto no se crea
  }
  catch(...){
    cout << "Coordenada incorrecta" << endl;
  }
}
```

Destructores

Todas las clases de C++ necesitan un destructor (si no se especifica, el compilador crea uno por defecto). Un destructor debe liberar los recursos (normalmente, memoria dinámica) que el objeto haya estado usando. Un destructor es una función miembro sin parámetros, que no devuelve ningún valor y con el mismo nombre que la clase precedido por el carácter ~. Una clase solo puede tener una función

destructora. El compilador genera código que llama automáticamente a un destructor del objeto cuando su ámbito acaba. También se invoca al destructor al hacer `delete`:

```
// Declaración
~Fecha();

// Implementación
Fecha::~Fecha(){
  // Liberar la memoria reservada (nada en este caso)
}
```

El destructor generado por defecto por el compilador invoca a los destructores de todos los atributos de la clase cuando estos son a su vez objetos.

Constructores de copia

De modo similar a la asignación, un constructor de copia crea un objeto a partir de otro objeto existente. Estos constructores sólo tienen un argumento, que es una referencia a un objeto de su misma clase:

```
// Declaración
Fecha(const Fecha &f);

// Implementación
Fecha::Fecha(const Fecha &f){
  dia=f.dia:
  mes=f.mes;
  anyo=f.anyo;
}
```

El constructor de copia se invoca automáticamente cuando:

- Una función devuelve un objeto (pero no un puntero o una referencia a un objeto)
- Se declara un objeto usando la asignación:

```
Fecha f2(f1);
Fecha f2=f1; // Esta línea es equivalente a la anterior
f1=f2; // Aquí NO se invoca al constructor, sino al operador =
```

- Un objeto se pasa por valor a una función (pero no cuando se pasan punteros o referencias):

```
void funcion(Fecha f1);
funcion(f1);
```

Si no se especifica ningún constructor de copia, el compilador crea uno por defecto con el mismo comportamiento que el operador `=.`

Operador de asignación

Podemos hacer una asignación directa de dos objetos (sin usar constructores de copia) usando el operador de asignación.

```
Fecha f1(10,2,2011);
Fecha f2;
f2=f1; // Copia directa de valores de los datos miembro
```

Por defecto el compilador crea un operador de asignación `=` que copia bit a bit cada atributo. Podemos redefinirlo para nuestras clases si lo consideramos necesario, pero no lo haremos en este curso.

Métodos constantes

Igual que ocurre con las variables de tipos primitivos, en ocasiones resulta de utilidad poder definir objetos constantes cuyo estado inicial (es decir, el valor de sus atributos establecido por el constructor) no pueda cambiar durante la ejecución del programa. C++ permite declarar métodos, llamados *métodos constantes*, que no modifican el valor de los atributos:

```
int Fecha::getDia() const{ // Método constante
  return dia;
}
```

En un objeto constante no se puede invocar a métodos no constantes. Por ejemplo, este código no compilaría:

```
int Fecha::getDia(){
  return dia;
}

int main(){
  const Fecha f(10,10,2011);
  cout << f.getDia() << endl;
}
```

De igual modo, el compilador emitirá un error si intentamos modificar los atributos del objeto desde un método constante. Obviamente, los métodos `get` deben ser constantes.

Funciones amigas

La parte privada de una clase (tanto si son métodos como atributos) sólo es accesible en principio desde los métodos de la propia clase, pero en C++ es conveniente poder saltarse esta regla puntualmente para permitir a funciones no definidas en la clase acceder a los elementos privados de esta. Para que una función pueda hacer lo anterior se ha de declarar como *amiga* de la clase en cuestión de la siguiente forma:

```
class MiClase{
  friend void unaFuncionAmiga(int,MiClase&);
```

```
  public:.
  private:
    int datoPrivado;
};
```

```
void unaFuncionAmiga(int x,MiClase& c){
  c.datoPrivado=x; // Ok
}
```

Las funciones amigas resultan especialmente útiles en C++ a la hora de sobrecargar los operadores de entrada/salida como veremos en el siguiente apartado.

Sobrecarga de los operadores de entrada/salida

En C++ podemos sobrecargar los operadores de entrada/salida de cualquier clase (en general, pueden sobrecargarse la mayoría de operadores del lenguaje, pero se recomienda no asignarles cometidos que estén muy alejados del original atribuido por el lenguaje; por ejemplo, el operador `+` debería sobrecargarse para reflejar operaciones similares a sumas o concatenaciones, pero nunca para algo como, por ejemplo, buscar un elemento en una lista) para permitir con una sintaxis sencilla la modificación del estado de un objeto a partir de los valores suministrados por un `ifstream` (por ejemplo, `cin`) o el volcado del estado actual sobre un objeto de clase `ofstream` (por ejemplo, `cout`):

```
MiClase obj;
cin >> obj;
cout << obj;
```

El problema que surge al intentar sobrecargar operadores que puedan usarse como en el código anterior es que no pueden ser funciones miembro de `MiClase` porque el primer operando no es un objeto de esa clase (es un `stream`); por otro lado, no podemos añadir las funciones que sobrecargan los operadores a las clases `ifstream` u `ofstream` porque son clases de la librería de C++ que no podemos modificar. La solución es declarar las funciones de sobrecarga de los operadores fuera de cualquier clase, pero haciéndolas *amigas* de la clase `MiClase` para poder acceder a sus elementos privados:

```
friend ostream& operator<<(ostream &o,const MiClase& obj);
friend istream& operator>>(istream &o,MiClase& obj);
```

Por ejemplo, la declaración de estos operadores para una clase `Fecha` quedaría:

```
class Fecha{
  friend ostream& operator<<(ostream &os,const Fecha& obj);
  friend istream& operator>>(istream &is,Fecha& obj);
  public:
    Fecha(int dia=1,int mes=1,int anyo=1900);
     ...
  private:
    int dia, mes, anyo;
```

```
};
```

Y una posible implementación sería la siguiente:

```
ostream& operator<<(ostream &os,const Fecha& obj){
  os << obj.dia << "/" << obj.mes << "/" << obj.anyo;
  return os;
}
```

```
istream& operator>>(istream &is,Fecha& obj){
  char dummy;
  is >> obj.dia >> dummy >> obj.mes >> dummy >> obj.anyo;
  return is;
}
```

Atributos y métodos de clase

Los *atributos de clase* y los *métodos de clase* también se llaman *estáticos*. Se representan *subrayados* en los diagramas UML. Un atributo de clase es una variable que existe en una única posición de memoria y que es compartida por todos los objetos de la clase. Los métodos de clase, por otra parte, sólo pueden acceder a atributos de clase. En C++, un atributo o método estático se declara utilizando la palabra reservada `static`:

```
class Fecha{
  public:
    static const int semanasPorAnyo=52;
    static const int diasPorSemana=7;
    static const int diasPorAnyo=365;
    static string getFormato();
    static boolean setFormato(string);
    void setDia(int d);
    void getDia();
  private:
    static string cadenaFormato;
    int dia;
    int mes;
    int año;
};
```

Cuando el atributo estático no es un tipo simple o no es constante, debe declararse en la clase pero tomar su valor fuera de ella:

```
// Fecha.h (dentro de la declaracion de la clase)
static const string findelmundo;

// Fecha.cc
const string Fecha::findelmundo="2020";
```

Este código ejemplifica cómo acceder a atributos o métodos *static* desde fuera de la clase:

```
cout << Fecha::diasPorAnyo << endl; // Atributo static
cout << Data::getFormat() << endl; // Método static
```

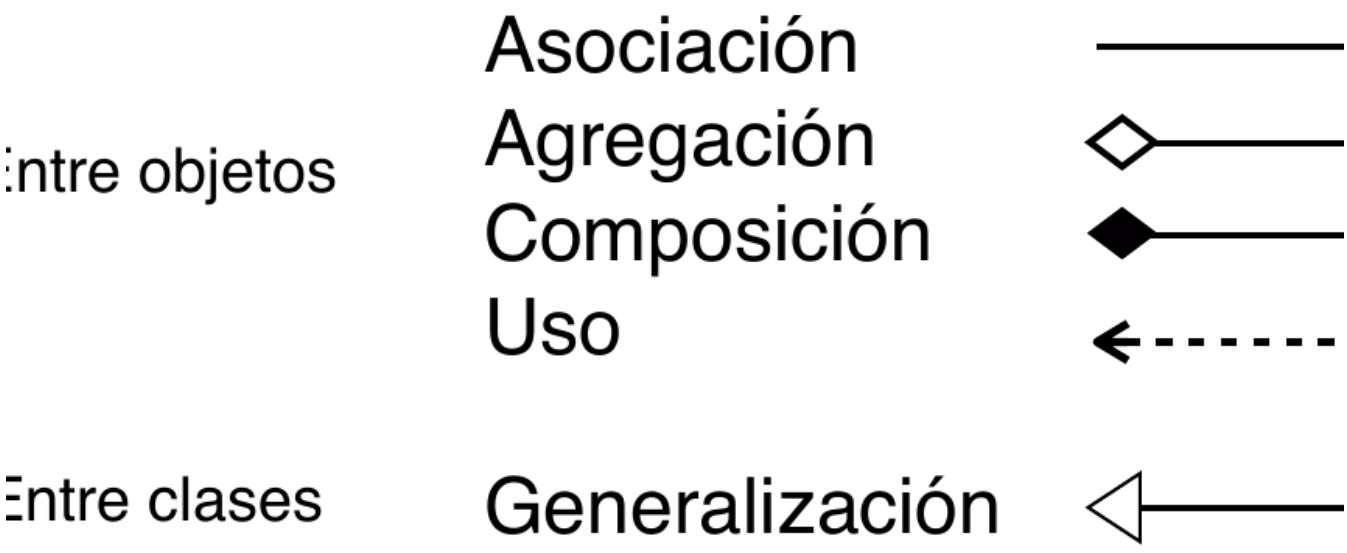
El puntero *this*

El puntero `this` es una pseudovariable que no se declara explícitamente en ningún punto del programa ni se puede modificar. Es un argumento implícito que reciben todas las funciones miembro (excluyendo funciones `static`) y que apunta al objeto receptor del mensaje actual. Suele omitirse para acceder a los atributos mediante funciones miembro. Es necesario usar `this`, sin embargo, cuando queremos referirnos a un atributo del objeto y existe una variable homónima declarada en un ámbito más cercano, o cuando queremos pasar como argumento el objeto a una función anidada.

```
void Fecha::setDia(int dia){
  // dia=dia; // Asigna el valor del parámetro al propio parámetro
  this->dia=dia; // Asigna el valor del parámetro al atributo homónimo del objeto actual
  cout << this->dia << endl;
}
```

Relaciones

Existen distintos tipos de relaciones que pueden establecerse entre objetos y clases. La figura siguiente muestra las que vamos a estudiar en esta sección, junto con la notación gráfica que se utiliza en UML para representarlas:



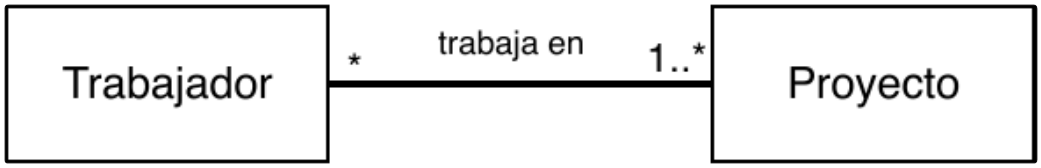
Las relaciones poseen una *cardinalidad*, que define el número de clases u objetos que pueden estar implicados en ellas. Esta cardinalidad puede ser:

- Uno o más: representada con *1..** (o *1..n* si hay un máximo definido)
- Cero o más: representada con *

- Número fijo: *m*

Asociación

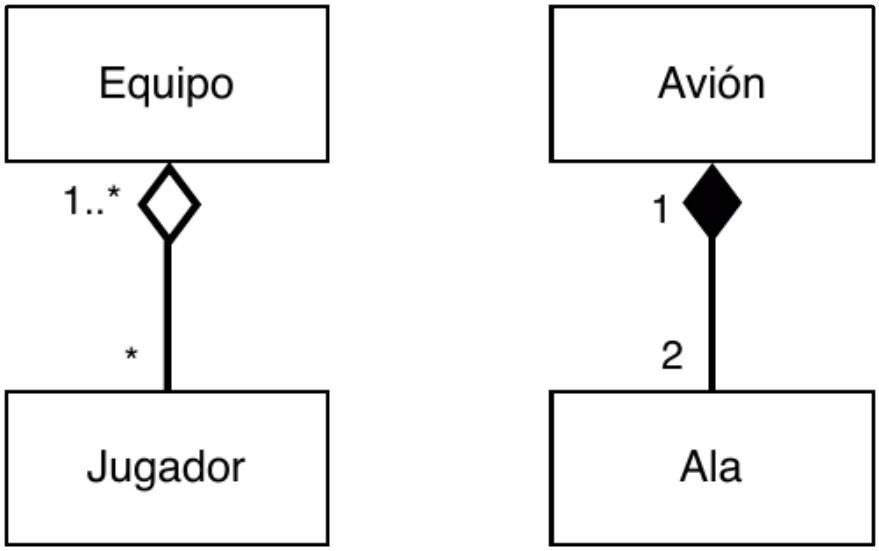
La asociación expresa una relación (unidireccional o bidireccional) general entre los objetos instanciados a partir de las clases conectadas. Por ejemplo, la relación de la figura siguiente expresa que un trabajador está vinculado a uno o más proyectos y que en un proyecto pueden trabajar cero o más trabajadores:



El sentido en que se recorre la asociación se denomina *navegabilidad* de la asociación.

Agregación y composición

La *agregación* y la *composición* son relaciones *todo-parte*, en la que un objeto (o varios) forma parte de la naturaleza de otro. A diferencia de la asociación, son relaciones asimétricas. Las diferencias entre agregación y composición son la fuerza de la relación. La agregación es una relación más débil que la composición. Considera, por ejemplo, el siguiente diagrama de clases:



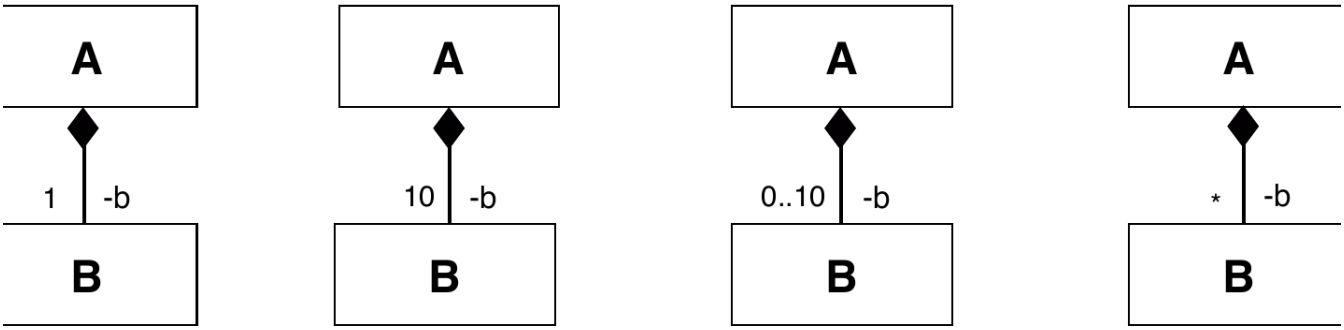
En el caso de la relación *fuerte* de composición (representada en UML mediante un rombo relleno en el lado de la clase *todo*), cuando se destruye el objeto contenedor también se destruyen los objetos que contiene; así el ala forma parte del avión y podemos considerar que en nuestra aplicación no tiene sentido fuera del mismo. Si vendemos un avión, lo hacemos incluyendo sus alas; si llevamos un avión al desguace para su eliminación, no salvamos sus alas.

En el caso de la agregación (representada en UML mediante un rombo vacío en el lado de la clase *todo*), no ocurre así: el objeto parte puede existir sin el todo. La relación de agregación de la figura refleja que podemos vender un equipo, pero los jugadores pueden irse a otro club.

Algunas relaciones pueden ser consideradas como agregaciones o composiciones, en función del contexto y de la aplicación para la que se establezcan (por ejemplo, la relación entre bicicleta y rueda).

Algunos autores consideran que la única diferencia entre ambos conceptos radica en su implementación; así una composición sería una *agregación por valor*. La composición es la relación que más usaremos en las prácticas de esta asignatura.

La siguiente figura representa esquemáticamente cómo se implementan varias composiciones con diferentes cardinalidades:



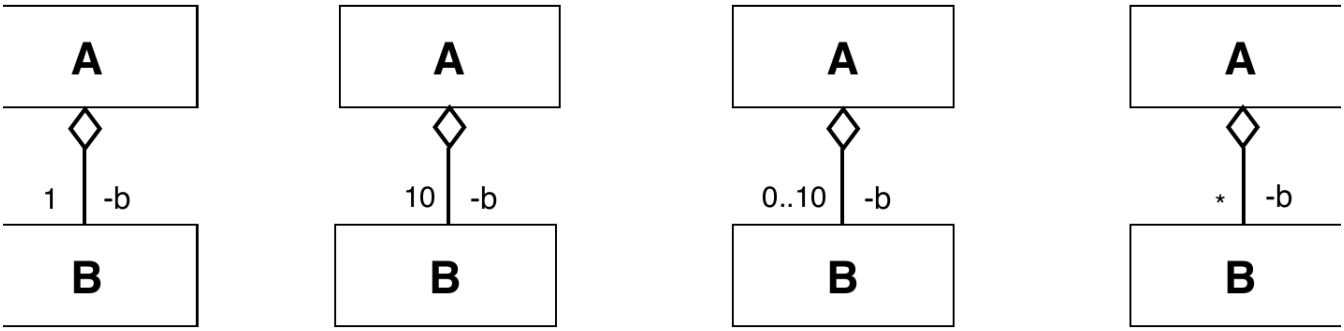

```
class A {
   private:
      B b;
   ...
};
```

```
class A {
   private:
      B b[10];
   ...
};
```

```
class A {
   private:
      vector<B> b;
      static const int N=10;
   ...
};
```

```
class A {
   private:
      vector<B>
   ...
};
```

En estas composiciones, la clase `A` (la clase que representa el *todo*) tiene su propia copia de los objetos de clase `B`, por lo que el borrado de un objeto de la clase `A` implica el borrado de sus objetos de clase `B`. En la agregación, sin embargo, la clase `A` incluye punteros a objetos de clase `B` creados fuera de `A`, objetos cuyo ciclo de vida no es gestionado por la clase A:




```
class A {
   private:
      B *b;
   ...
};
```

```
class A {
   private:
      B *b[10];
   ...
};
```

```
class A {
   private:
      vector<B*> b;
      static const int N=10;
   ...
};
```

```
class A {
   private:
      vector<B*>
   ...
};
```

El siguiente código muestra un ejemplo de relación de agregación y la instanciación de los objetos correspondientes:

```
class A{
  private:
    B *b;
  public:
    A(B *b){
```

```
      this->b=b;
    }
    ...
};
```

```
int main(){
  // Dos formas:
  // 1- Mediante un puntero
  B *b=new B;
  A a(b);

  // 2- Mediante un objeto
  B b;
  A a(&b);
}
```

Lo comentado en los párrafos anteriores constituye una aproximación básica a la implementación de las relaciones *todo-parte*, pero esta implementación puede ser diferente en determinados contextos. Por ejemplo, si la cardinalidad de una composición es *0..1* puede interesar implementarla como un puntero que sea nulo o no nulo dependiendo de si existe el objeto *parte*.

Uso

A diferencia de las anteriores, el *uso* es una relación no persistente (tras la misma, se termina todo contacto entre los objetos). Diremos que una clase `A` usa una clase `B` cuando:

- Invoca algún método de la clase `B`.
- Tiene alguna instancia de la clase `B` como parámetro de alguno de sus métodos.
- Accede a sus variables privadas (esto sólo se puede hacer si son clases *amigas*).

El siguiente diagrama UML y el código en C++ que le sigue representan una relación de uso de la clase `Gasolinera` por la clase `Coche` que se ajusta a los dos primeros casos de la lista anterior:

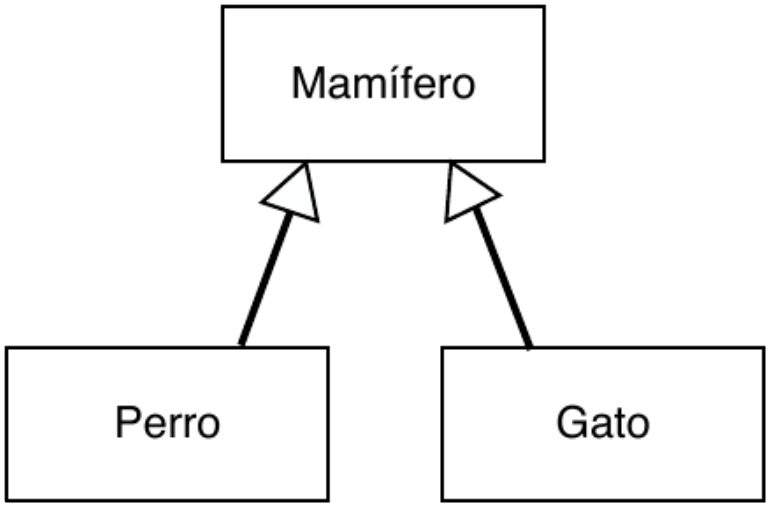

```
float Coche::Repostar(Gasolinera &g,float litros){
  float importe=g.dispensarGaso(litros,tipoC);
  lgaso=lgaso+litros;
  return importe;
}
```

Generalización

La *herencia* es el mecanismo de los lenguajes orientados a objetos que permite definir una nueva clase

(clase *derivada* o *subclase*) como una especialización de otra (clase *base* o *superclase*); se aplica cuando hay suficientes similitudes y las características de la clase base son adecuadas para la clase derivada.

El siguiente diagrama de clases UML muestra una jerarquía de herencia en la que las subclases `Perro` y `Gato` heredan los métodos y atributos especificados por la superclase `Mamífero`:



La herencia nos permite adoptar características ya implementadas por otras clases y refinarlas o sustituirlas en las clases derivadas. El siguiente código muestra una clase `Rectangle` que deriva de una clase `Shape`. Los atributos declarados como `protected` son visibles en las clases derivadas (a diferencia de los privados), pero no son públicos:

```
class Shape{ // Clase base
  public:
    void setWidth(int w){
      width=w;
    }
    void setHeight(int h){
      height=h;
    }
  protected:
    int width;
    int height;
};
```

```
class Rectangle: public Shape{ // Clase derivada
  public:
    int getArea(){
      return (width*height);
    }
};
```

```
int main(){
  Rectangle rect;

  // Podemos llamar a los metodos de la clase base
```

```
  rect.setWidth(5);
  rect.setHeight(7);

  // ...y a los de la clase derivada
  cout << "Total area: " << rect.getArea() << endl;
}
```

El programa Make

Supongamos el caso de un fichero de cabecera `Clase.h` que se usa en varios ficheros `.cc`. Si cambiamos algo en el fichero de cabecera, no tiene sentido recompilar todo el código de la aplicación, sino solo aquellos ficheros que usan `Clase.h`. El programa `make` ayuda a compilar programas grandes compuestos por muchos ficheros; `make` nos permite definir las *dependencias* entre los diferentes ficheros de código fuente de manera que un determinado fichero solo se recompile cuando cambie su contenido o lo haga el de alguno de los ficheros de los que depende.

Un fichero de texto llamado `makefile` especifica las dependencias entre los ficheros y qué hacer cuando cambian; si ejecutamos `make` dentro de un determinado directorio, se buscará por defecto un fichero `makefile` que contenga uno o más *objetivos* a cumplir por `make`. El fichero `makefile` tiene un objetivo principal (normalmente el programa ejecutable) seguido de otros objetivos secundarios. El formato de cada objetivo es:

```
<objetivo> : <dependencias>
[tabulador]<instrucción>
```

El algoritmo del programa `make` es sencillo y puede definirse informalmente como: "Si la fecha de alguna dependencia es más reciente que la del objetivo, ejecutar la instrucción". Un ejemplo de fichero `makefile` para un proyecto con dos clases `C1` y `C2` y un fichero `prog.cc` cuyo programa principal usa ambas clases quedaría como sigue:

```
prog : C1.o C2.o prog.o
        g++ -Wall -g C1.o C2.o prog.o -o prog
C1.o : C1.cc C1.h
        g++ -Wall -g -c C1.cc
C2.o : C2.cc C2.h C1.h
        g++ -Wall -g -c C2.cc
prog.o : prog.cc C1.h C2.h
        g++ -Wall -g -c prog.cc
```

Si se cambia `C2.cc` y ejecutamos `make`, se ejecutarán las siguientes instrucciones:

```
g++ -Wall -g -c C2.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

Si se cambia `C2.h` y ejecutamos `make`, se ejecutarán estas otras:

```
g++ -Wall -g -c C2.cc
g++ -Wall -g -c prog.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

Podemos definir variables que son sustituidas donde corresponda. Una versión mejorada del `makefile` anterior que usa algunas variables sería:

```
CC = g++
CFLAGS = -Wall -g
MODULOS = C1.o C2.o prog.o

prog : $(MODULOS)
	$(CC) $(CFLAGS) $(MODULOS) -o prog
C1.o : C1.cc C1.h
	$(CC) $(CFLAGS) -c C1.cc
C2.o : C2.cc C2.h C1.h
	$(CC) $(CFLAGS) -c C2.cc
prog.o : prog.cc C1.h C2.h
	$(CC) $(CFLAGS) -c prog.cc
clean:
	rm -rf $(MODULOS)
```