Tabla de contenido

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.

> Este libro está en construcción y se actualizará conforme vaya avanzando la asignatura durante el segundo cuatrimestre. Sólo estarán realmente terminados los capítulos que correspondan al tema tratado en las clases de teoría.

Tema 1 - Introducción

En este tema veremos un repaso de los contenidos de Programación 1, añadiendo conceptos sobre diseño de algoritmos y programas, metodología y sintaxis de C++.

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. Una vez tenemos el código, mediante un compilador podemos transformarlo en un programa ejecutable que es capaz de interpretar el ordenador.

Por tanto, es necesario escribir código fuente, compilarlo y como resultado ya podemos ejecutar nuestro programa. Sin embargo, antes de comenzar a picar código tenemos que analizar los **requerimientos** y pensar en el **diseño** de nuestro algoritmo.

Las fases de desarrollo de un programa son las siguientes:

1. Estudio de los requerimientos del problema
2. Diseño del algoritmo (a ser posible 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)

El proceso de escribir, compilar, ejecutar y probar debería ser iterativo, haciendo pruebas de funciones o módulos por separado del programa.

1. Requerimientos

Para empezar tenemos que tener claros los **requerimientos**, es decir, **qué** debe hacer nuestro programa. En algunos casos es sencillo ,por ejemplo, si queremos hacer un programa para mostrar los *n* primeros números primos. Sin embargo en otros casos es mucho más complicado, por ejemplo, si queremos hacer un programa de contabilidad para una empresa.

Antes de comenzar a diseñar el programa es necesario tener un listado de requerimientos con todas las opciones de nuestro programa. Este es un listado de ejemplo para hacer un juego:

- 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 que dependen de su tipo.
- etc...

Como ves, la lista de requerimientos puede ser muy larga. En las prácticas de Programación 2 (P2) daremos el listado de requerimientos de forma clara para que no haya dudas sobre qué debe hacer el programa y cómo debe interactuar el usuario.

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, e incluye detectar los tipos datos necesarios, decidir las funciones para trabajar con ellos, y elaborar un flujo de programa.

La fase de diseño es muy importante, y conviene hacerla en papel. Para esto, lo recomendable es 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 (lanzarPájaro, impactarConCerdo, tocarPájaro, etc.) para inferir las funciones.

En el caso de los programas con interfaz gráfico (por ejemplo, programas para móviles), podemos hacer el diseño en el mismo entorno de programación, colocando pantallas, botones, vistas, etc. En el caso de un programa sin gráficos (como será en esta asignatura), el diseño se limita a crear los tipos de datos necesarios y las funciones para trabajar con ellos.

En teoría, con un buen diseño ya podemos hacer el código, y no es necesario rediseñar el programa. En la realidad, para programas complicados es probable que, a pesar de que el diseño sea bueno, nos toque rehacer algunas partes. Por ejemplo, podemos hacer una app para móvil y darnos cuenta de que un botón no queda todo lo bien que pensábamos en un sitio determinado, y esto nos obligue a cambiar el diseño. Independientemente, lo que está claro es que cuanto mejor diseñemos un programa más sencillo será continuar con las siguientes fases, por lo que es recomendable dedicarle tiempo para pensarlo bien.

3. Escritura del código

Una vez tenemos claro cómo vamos a hacer el programa, procederemos a **escribir el código fuente**, usando un editor de textos, o alternativamente un entorno de desarrollo integrado (IDE, *Integrated Development Environment*).

Es muy importante no escribir mucho código (por ejemplo más de 50 líneas) de golpe. Lo que debe hacerse es escribir unas pocas líneas de código (o una función), y después compilar. Tras arreglar los errores de compilación y probar que el código hace lo esperado podemos seguir escribiendo.

Si hacemos mucho código sin comprobarlo, lo más normal cuando intentemos compilarlo es que nos salgan muchísimos errores de compilación y no seamos capaces de resolverlos adecuadamente.

4. Compilación y corrección de errores

Un compilador interpreta o convierte nuestro código fuente en un programa que el ordenador puede ejecutar. Lo normal tras escribir código es que tengamos algunos errores de compilación. Estos errores hay que corregirlos, reescribiendo el código y compilando hasta que no se produzcan más errores.

Hay dos tipos de errores: los errores de compilación, que impiden generar un ejecutable, y los *warnings* de compilación, que nos dejan generar el ejecutable pero nos avisan de que puede que haya algo mal. Es conveniente arreglar todos los *warnings*, ya que muchas veces nos avisan de algo que no está bien y acaba produciendo errores durante la siguiente fase, la ejecución del programa.

5. Ejecución

Una vez tenemos una parte de nuestro programa compilado, debemos ejecutarlo para ver que el código que hemos añadido hace lo esperado. A veces no es así, y nos toca volver a la fase 3, reescribiendo el código, compilándolo y volviendo a ejecutar.

6. Prueba

Tras ejecutar el programa hemos visto 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?

En principio es complicado controlar todos los posibles casos que pueden darse durante la ejecución del programa, pero hay algunos trucos para esto.

Por ejemplo, tenemos la siguiente función:

```
float division(int a, int b) {
    float resultado = a/b;
    return resultado;
}
```

Hay dos errores de ejecución en el código, ¿Puedes verlos?

Cuando la ejecutemos, veremos enseguida el primer error.

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

El resultado será 0, ya que hemos hecho una división entera. Podemos arreglarlo:

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

En cuanto uno de los dos operandos sea de tipo *float*, el resultado será *float*. Sin embargo, el segundo error es más complicado. ¿cómo lo encontramos?

La respuesta es que **para cada función debemos considerar todos los posibles valores de sus parámetros** y ver si la salida será correcta con ellos. Por ejemplo, en esta función recibimos dos enteros. Su valor puede ser positivo, negativo, o cero.

¿Si algún parámetro es negativo, funcionaría? Sí. ¿Si algún parámetro es cero, funcionaría? No, ya que no pueden hacerse divisiones por cero y b podría tener ese valor. Por tanto, para que la función fuera correcta tendríamos que hacer 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;
}
```

C++

Tras ver cómo hacer un programa en líneas generales, vamos a centrarnos ahora en el lenguaje C++, que es el que usaremos en Programación 2. Existen muchas referencias sobre C++, pero si quieres complementar este libro de apuntes recomendamos cplusplus y minidosis.

Elementos básicos

En un código fuente podemos encontrarnos distintos elementos básicos:

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

Por ejemplo, en este código:

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

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

Vamos a ver en detalle cada uno de estos elementos.

Identificadores

Los identificadores son los nombres que le damos a nuestras constantes, variables o funciones. Los elige el programador, por lo que debe seguir una serie de recomendaciones:

- Los identificadores deben ser **significativos**, es decir, su nombre debe indicar para qué se utiliza. Estos son ejemplos correctos:

```
int numeroAlumnos = 0;
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 hay más de una palabra la primera letra debe ser mayúscula, como en el ejemplo anterior.

Estos son ejemplos de nombres incorrectos:

```
const int kOCHO=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*, etc.

Evidentemente, existen palabras reservadas 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`, `for`, etc.

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

Constantes

Podemos tener constantes de varios 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) {
  cout << "Valor correcto" << endl;
}
```

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

```
const int kMAXVALUE = 255;
const char kMESSAGE[] = "Valor correcto";

if (i<kMAXVALUE) {
  cout << kMESSAGE << endl;
}
```

Podemos declrar constantes explícitas de cualquier tipo:

```
const int MAXALUMNOS=600;
const double PI=3.141592;
const char DESPEDIDA[] = "ADIOS";
```

La pregunta es, ¿cuándo debemos hacerlo?. Y la respuesta es: **casi siempre**, ya que deberíamos declarar como constantes aquellos valores que podríamos querer cambiar en futuras versiones del programa. Por ejemplo, es posible que queramos cambiar el texto de un mensaje, por lo que debemos declararlo como constante.

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 explícito, nos tocaría buscar en el código todas las líneas en las que se aparece este mensaje para cambiar su valor, lo cual es más incómodo y propenso a errores por despistes.

Variables

A diferencia de las constantes, que sólo pueden tener un valor fijo asignado, las variables pueden usarse para almacenar valores que cambien durante la ejecución del programa.

Es muy recomendable que siempre que se declare una variable de (tipo simple) 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 inicializa una variable de tipo simple, su valor será el que haya en memoria en ese momento, es decir, cualquiera (no podemos controlarlo).

> Los tipos simples más comunes en C/C++ son `int`, `char`, `float`, `double`, `unsigned` y `bool`.

Este código sería erróneo:

```
int i;
cout << i << endl; // El valor será aleatorio
```

Ámbitos

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. Por tanto, sólo podemos usar las variables o constantes durante su ámbito.

```
if (i<10) {
    int j = 20;
}
j = 10; // No podemos usar la variable j aquí porque se ha destruído cuando ha termina
do 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 nada conveniente) dos variables que se llamen igual dentro de la misma función, siempre que tengan un ámbito distinto. Por ejemplo, esto compilaría:

```
int ncajas=0;
// ya se puede usar ncajas
if (i<10) {
  // se puede usar
  int ncajas=100; // Dentro podemos declarar otra variable con el mismo nombre (no aco
nsejable)
  cout << ncajas << endl; // imprime 100
}
cout << ncajas << endl; // imprime 0, ya que se usa la primera variable
```

Variables globales

Las variables normalmente se declaran dentro de una función, pero cuando se declaran fuera se llaman variables globales. En general, se recomienda no utilizar variables globales (son peligrosas) y en Programación 2 están terminantemente prohibidas.

¿Por qué prohibimos estas variables? Veamos un ejemplo de código en la que se puede ver lo complicadas que son de manejar.

```
#include <iostream>
using namespace std;
int contador=10;

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

int main() {
  cuentaAtras();   // Imprime 10, 9, 8, ... 0
  cuentaAtras();   // Aqui no se imprime 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 muy complicado de controlar, ya que **una misma función con los mismos parámetros se comporta de forma distinta** dependiendo de en qué punto del código hagamos la llamada.

Tipos de datos

Estos son los principales **tipos de datos simples** en C++:

Tipo	Descripción
`short`	Entero corto
`int`	Entero, el doble de bytes que `short`
`char`	Caracter
`float`	Real
`double`	Real, el doble de bytes que `float`
`bool`	Booleano

Además, para los números enteros tenemos su versión sin signo (`unsigned short` y `unsigned int`). Si un entero corto (`short`) se representa con 2 bytes, su rango va desde -32768 a 32767. En cambio, si es `unsigned short`, podremos representar desde 0 a 65535, por lo que si sabemos que no podemos tener valores negativos podremos aprovechar para representar números más altos sin necesitar más bytes.

> 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).

Conversión

En nuestro código podemos convertir una variable de un tipo a otro. Esta conversión puede ser implícita (el compilador lo hace por nosotros directamente), o explícita (tenemos que indicarle que lo haga). A continuación podemos ver ejemplos de conversiones **implícitas**:

Conversión	Ejemplos
char -> int	int Ie = 'A' + 2;
int -> float	float pi = 1 + 2.141592;
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` (normalmente 2 bytes) siempre podremos almacenar un valor `char` (1 byte).

Sin embargo, hay otros casos en los que haceiendo la conversión podemos perder información. En estas situaciones el compilador mostrará un `warning`, y tendremos que forzar la conversión en el código, haciendo lo que conocemos como `casting`.

> Es importante no ignorar los *warnings*.

Vamos a ver algunos ejemplos de conversiones **explícitas**:

Conversión	Ejemplos
int -> char	char c = (char)('A' + 2); // c valdrá 'C'
float -> int	int epi = (int)pi; // epi valdrá 3
double -> float	double d = (double)pi;

En el primer caso, si convertimos un número entero a un caracter podríamos tener problemas (si es mayor de 255 no nos cabría en un byte). Por tanto, el compilador nos da un *warning* (a veces incluso un error) 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*, convirtiendo el tipo de dato mediante paréntesis antes del nombre de la variable.

Declaración de tipos

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

```
typedef int entero;
entero i,j; // i,j son int

typedef bool logico,booleano;  // logico y booleano son bool
logico b;  // b es bool
```

También podemos declarar un array como un tipo:

```
typedef char cadena[MAXCADENA];  // cadena es un array de char
```

En Programación 2 no recomendamos redefinir los tipos 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. Pensad que véis un código lleno de tipos de datos que no conocéis, habría que ir a la definición de los tipos para saber qué significan en cada caso, y esto no es cómodo a la hora de compartir y reutilizar código.

Además de redefinir tipos existentes, C++ nos permite crear nuevos tipos de datos. En Programación 2, los únicos tipos de datos nuevos que veremos serán los registros (en el último tema veremos cómo hacer clases, pero en realidad no son tipos de datos). Podemos declarar un registro de dos formas:

```
struct Alumno {      // Primera forma: 'Alumno' es un tipo en C++ (recomendado)
  int dni;
  double nota;
};
```

```
typedef struct {     // Segunda forma: 'Alumno' es un tipo en C
  int dni;
  double nota;
} Alumno;
```

En ambos casos declaramos un tipo de dato llamado `Alumno`. Después, podemos crear una variable de este tipo de la siguiente forma:

```
Alumno alu;
```

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

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

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 datos de un registro podemos usar un punto (`.`) tras su nombre. Por ejemplo:

```
Alumno a,b;

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

La asignación directa de registros (como aparece en la última línea del ejemplo anterior) es posible, pero no debemos hacerlo si dentro del registro tenemos un array o un puntero. El motivo es que hay que tener mucho cuidado al asignar directamente un registro o un puntero, por razones que veremos en el tema de memoria dinámica.

Expresiones

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:

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

Las expresiones pueden ser de varios tipos.

Expresiones aritméticas

En C++ los principales operadores aritméticos son la suma (`+`), la resta (`-`), la multiplicación (`*`), la división (`/`), y el el resto o módulo (`\%`).

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

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

Cuando hacemos una división entre dos números enteros (`int`), el resultado es un número entero. Por ejemplo:

```
float f1 = 7 / 2; // f1 == 3
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 resto (`%`) devuelve el resto de una división entre dos números enteros. Por ejemplo:

```
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 `-` .

```
int n = 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. Por ejemplo:

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

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 de la variable:

```
++i;  // Preincremento
i++;  // Postincremento
```

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

```
int i = 3;
int k = ++i;  // k == 4, i == 4

i=3;
int j = i++;  // j == 3, i == 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?:

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

> Piensa en la respuesta, y luego confirma el resultado ejecutando este código.

Expresiones relacionales

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

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

```
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, para saber si `a < b < c` , tenemos que indicar lo siguiente:

```
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 `!` , `&&` , `||` . La negación (`!`) devuelve `true` si lo que viene ab 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) {
    // La condición es verdadera, entra aquí
}
```

Hay una característica de C++ que es importante conocer : la **evaluación en cortocirtuito**. Cuando tenemos una condición `||`, 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 una condición `&&`, 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==strelen(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)`, nos saldría este error. Por tanto, **el orden de los operandos en una expresión lógica es importante**.

Entrada / 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:

```
int i = 3;
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. Escribe este programa, llámalo `prueba.cc` y compílalo:

```
#include <iostream>

using namespace std;

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

Y ejecútalo desde el terminal:

```
$ ./prueba
```

Verás que se muestra por pantalla el mensaje. Ahora vamos a ejecutarlo de otra forma:

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

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

Igual que podemos mostrar algo, también podemos leer el valor de una variable desde teclado:

```
#include <iostream>

using namespace std;

int main() {
    int n;
    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 caracter (sólo se leería una letra), o una cadena, por ejemplo. El operador de entrada lee desde teclado ignorando blancos y tabuladores hasta leer el tipo de dato de la variable que se le indica, y deja 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 caracter por teclado
    cout << "He leido" << n << " y " << c << endl;
}
```

Cuando ejecutemos el programa quedará a la espera de que introduzcamos un número entero y a continuación un caracter.

> Cada vez que leamos algo con `cin >>`, es conveniente poner a continuación `cin.get();`, `cin.ignore();` o llamar a una función propia `limpiarBuffer()` para que no se nos quede en el *buffer* el caracter de salto de línea, ya que puede dar problemas si se lee después una cadena como veremos en el tema de `strings`.

Vamos a ejecutar el programa anterior redireccionando la entrada estándar. Con cualquier editor de texto creamos un nuevo fichero llamado `entrada.txt`, y escribimos lo siguiente:

```
124 k
```

No te olvides de poner un salto de línea al final del fichero. A acontinuación, ejecutamos:

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

Verás como el programa ya no pide los datos de la entrada, sino que los lee desde el fichero. 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`). Cambia el programa anterior, reemplazando `cout` por `cerr` . Como verás, si ejecutamos el programa como hemos hecho la última vez, se mostrará la información por la pantalla y `entrada.txt` estará vacío. Esto ocurre porque lo que sale por la salida de error se redirecciona de forma distinta a lo que se muestra por la salida estándar. Para redireccionar el error, tenemos que poner:

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

La salida de error se suele usar para mostrar mensajes de error de nuestros programas. Por ejemplo, añade cualquier cosa al código anterior para que no compile. Ejecuta el compilador:

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

Verás que los errores de compilación se han guardado en `errores.txt` , ya que el programa `g++` muestra los mensajes de error mediante `cerr` .

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 `true` . Podemos también añadir un `else` para que se ejecuten instrucciones alternativas cuando no se cumple la condición:

```
if (condicion) {
    // Instrucciones
}
else {
    // Otras instrucciones
}
```

`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`.

`for`

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

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

Los `for` suelen ser necesariso cuando recorremos una serie de elementos, por ejemplo para recorrer un array:

```
bool salir = false;
int array[] = {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 usando menos código:

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

`do-while`

Las instrucciones de control `do-while` son similares a las instrucciones `while`, con la diferencia de que la primera vez **siempre** ejecutan su contenido.

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

Se usan, por ejemplo, cuando queremos mostrar un menú por pantalla.

`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;
}
```

En C++ la variable (o constante) del `switch` debe ser de tipo entero 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':
    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`. También se puede usar `break` para salir de un bucle `while`, `do` o `for`.

Arrays y matrices

Un *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[kMAXTABLERO];
```

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 crar un array con un tamaño asignado por una variable:

```
int n;
cin >> n;
int v[n]; // El tamaño del array se conoce en tiempo de ejecución. NO RECOMENDABLE!
```

Pero el estándar del lenguaje no recomienda hacerlo así. Si el tamaño del *array* no se conoce en tiempo de compilación, entonces es mejor usar vectores, 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; // Imprime 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, el array tiene 8 elementos cuyas 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
```

Cadenas de caracteres en C

Las cadenas de caracteres en C son simplemente *arrays* que contienen caracteres. Por ejemplo:

```
const char cadena[]="Hola";
```

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

La peculiaridad es que a estos *arrays* C les añade el carácter nulo al final: `'\0'` .

H	**o**	**l**	**a**	**\0**
0	1	2	3	4

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

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

```
const int tCADENA = 10;
char cadena[tCADENA];
```

Vectores

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** de C++.

Un vector 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);
```

Añadimos un elemento al final de un vector con `push_back()`, y podemos saber el tamaño efectivo de un vector con `size()`.

Estas son sólo algunas de las funciones de la clase vector, cuya referencia puedes encontrar aquí. Además de estas, los vectores tienen funciones para ordenar sus elementos siguiendo el criterio que elijamos, para borrar un elemento redimensionando su tamaño, etc.

En Programación 2 trabajaremos bastante con vectores.

Tipos enumerados

Los tipos enumerados se utilizan cuando tenemos un rango determinado de posibles valores para una variable (o constante). 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

mycolor = blue; // Asignación de un enumerador

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

Internamente, los enumeradores se convierten implícitamente a números enteros y viceversa, como puede verse en el ejemplo anterior.

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.

```
// Función que recibe dos parámetros por valor y uno por referencia, y devuelve un núm
ero entero

int funcion(int a, int b, int &c) {
    int ret = 0; // Declaramos una variable del tipo de retorno

    // Instrucción 1
    // Instrucción 2
    // ...

    return ret; // Devolvemos el resultado
}
```

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 50 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 hacer copy-paste, entonces necesitas una función**. Cuando copiamos un trozo de código para pegarlo en otro lugar, es porque lo estamos usando dos veces. Cuando esto ocurre, 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/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 de cabecera):

```
// 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, ésta 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, 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, pero si intentamos modificar esta variable no
s da un error
}
```

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 saldrá un error de compilación.

> Por motivos pedagógicos en Programacón 2 no se deben pasar parámetros por referencia cuando no es necesario hacerlo, excepto si es con `const`.

En C/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 de 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.

```
int sumaVM(int v[],int m[][MAXCOL]) {  // el tamano de la primera dimension no se indi
ca
  // Instrucciones
}

sumaVM(vector,matriz);  // llamada, sin corchetes
```

Como ves, tampoco 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/C++, llamada `main`. La función `main` es la primera que se invoca cuando comienza un programa. Puede recibir parámetros (veremos cómo hacerlo en el tema de paso de parámetros), y devuelve un valor (aunque se puede dejar sin indicar, normalmente devuelve 0). Por ejemplo:

```
int main() {
  // Instrucciones

  return 0; // Opcional
}
```

¿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:

```
int main() {
    int n;
    leer(n);

    if (n<0)
        cout << "Error, no puede ser negativo";
    else procesar(n);
}
```

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

```
int main() {
    principal();  // Incorrecto
}
```

Argumentos del programa

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`:

```
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. 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 desde 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 imprimirá lo siguiente:

```
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 y en cada posición contiene un valor introducido por el usuario.

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

```
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++) {
            string arg = argv[i];

            if (arg=="uno")  {
                // Hacer algo con argumento uno
            }
            else if (arg=="dos") {
                // Hacer algo con argumento dos
            }
            else if (arg=="tres") {
                // Hacer algo con argumento tres
            }
            else {
                cout << "Sintaxis: " << argv[0] << " [uno | dos | tres]" << endl;
                return(-1);
            }
        }
}
```

A veces se puede complicar tanto que conviene usar una función aparte para gestionar los parámetros.

Ejercicio (argumentos)

Vamos a hacer un programa que imprima por pantalla los *n* primeros números primos. Por defecto (si no se indican parámetros), el programa debe imprimir los primeros 10 números primos separados por espacios. Ejemplo:

```
./primos
2 3 5 7 11 13 17 19 23 29
```

El usuario debe poder también indicar las opciones `-L` y/o `-N n`.

La opción `-L` es para mostrar cada número en un linea distinta, y la opción `-N` seguida de un número es para mostrar los primeros `n` primos. Por ejemplo:

```
./primos -N 3
2 3 5
./primos -L -N 2
2
3
```

Si los parámetros indicados por el usuario no son correctos debe mostrarse un mensaje de error de sintaxis y finalizar el programa. Lo complicado es esta parte, es decir, comprobar que los parámetros son adecuados.

Estructura tipica de un programa en C++

Para repasar, veamos la estructura típica de un programa en C++:

```
#include <ficheros de cabecera estandar>
...
#include "ficheros de cabecera propios"
...
using namespace std; // permite usar bool (y string)
...
const ... // Declaración de constantes
...
typedef ... // Declaración de tipos de datos propios
...
// declaración de variables globales ¡¡¡PROHIBIDO!!!
...
// funciones
...
int main() {
...
}
```

Compilación

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 P2**).
- `-g` : Compila en un modo que facilita encontrar los errores mediante un depurador (**recomendado en P2**).
- `-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++0x`: Usa el nuevo estándar de C++ que permite programación no tipada con `auto`, funciones `lambda`, bucles `for_each`, y funciones de concurrencia, entre otras. No lo usaremos en P2.

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

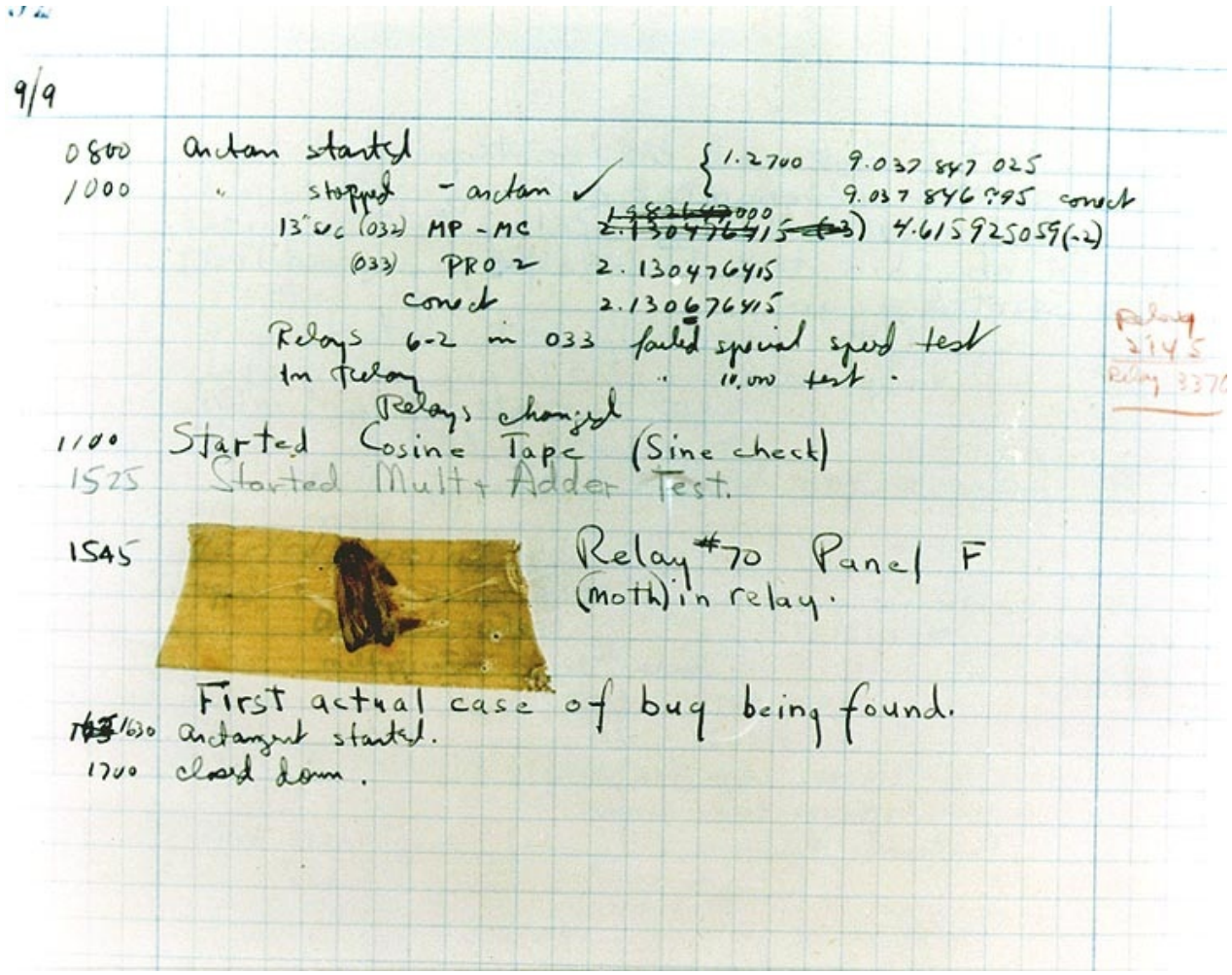
```
g++ -Wall -g programa.cc -o programa
```

Es muy importante que justo a continuación de `-o` vaya el nombre del ejecutable, **no** el del código fuente. Si ponemos `-o programa.cc`, se nos borrará todo nuestro código fuente (es habitual perderlo en los primeros cursos, 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.

> Se llaman *debuggers* porque encuentran *bugs*, que es como se conocen popularmente los errores de código. Este término viene porque en 1946, 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, Nevimer, Electric Fence, DUMA, etc.

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

Recordatorio

Por motivos pedagógicos, en Programacón 2 está **terminantemente prohibido usar variables globales**.

Referencias

Para aprender más sobre C++ recomendamos consultar las siguientes referencias:

http://www.cplusplus.com

http://www.minidosis.org

Tema 2 - Cadenas de caracteres

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 e inicialización

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:

```
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 9 caracteres.

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

Ejemplo:

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

Las dos declaraciones anteriores son equivalentes. En el primer caso, a `cad1` le asignamos 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 hace falta poner explícitamente 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:

![Falta imagen](./images/fig_1.jpg "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 caracter 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:

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

```
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:

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

Aquí estaríamos reservando `100` posiciones de memoria, aunque solo estaríamos ocupando 5 en este 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:

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

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

Entrada y salida

Salida por pantalla con `cout`

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

Ejemplo:

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

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

Lectura de teclado con `cin` y `>>`

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:

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

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

Lectura de teclado con `cin` y `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
- 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:

```
const int TAM = 10;
char cad[TAM];
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 almacenariá 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 probocan 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 solucionar este problema es extraer el `'\n'` del buffer de teclado antes de hacer la siguiente lectura. Esto lo podemos hacer con el método `ignore` de la siguiente manera:

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

Aquí, `cin.ignore()` saca un carácter del buffer de teclado y lo descarta.

Funciones de la librería `string.h`

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

El código ASCII del carácter '1' es 49, 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:

```
char cad1[] = "adios";
char cad2[] = "adeu";

cout << strcmp(cad1,cad2) << endl;
cout << strcmp(cad2,cad1) << endl;
cout << strcmp(cad1,cad1) << endl;
```

Este código mostraría por pantalla:

```
1
-1
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 ese 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. 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. En este ejemplo, al copiar `4` caracteres, estos ocuparn las posiciones 0, 1, 2 y 3 del array `cad`, por eso el carácter nulo se añade en la posición siguiente, en la `4`.

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 primerar 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++

Declaración e inicialización

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 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 más pequeña.

La clase `string` usa intérnamente 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".

Para hacer uso de algunas de las funciones que vamos a mencionar en este apartado es necesario incluir la librería del mismo nombre:

```
#include <string>
```

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:

```
string cad1;
string cad2 = "hola";
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:

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

using namespace std;

void analizarCadena(string cad1, string &cad2)
{
    ...
}

int main()
{
    string cad1 = "hola";
    string cad2 = "adios";

    analizarCadena(cad1,cad2);
}
```

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.

Entrada y salida

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

Ejemplo:

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

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

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:

```
string cad;
cin >> cad;
```

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:

```
string cad;
getline(cin,cad);
```

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:

```
string cad;
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.

Funciones de la librería `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 llaman *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` y `erase`.

`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:

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

El primer parámetro 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 entero indicando 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 t
aza..."
unsigned int encontrado = a.find(b);

if(encontrado != string::npos)
    cout << "Encontrada primera << b << en la posición " << encontrado << endl;
else
    cout << "Palabra << b << no encontrada";

// 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;
else
    cout << "Palabra << b << no encontrada";
```

`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:

```
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 devuleve a otra variable.

Ejemplo:

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

a.replace(8,4,"botella");
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:

```
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:

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

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

Operaciones con `string`

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:

```
string cad1 = "Hola";
string cad2;
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:

```
string cad1 = "Hola";
string cad2 = "mundo";
string cad3 = cad1 + ", " + cad2 + "!";
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.

Conversión entre `string` y números

Para convertir un número entero o real a `string` podemos utilizar la función `to_string`.

Ejemplo:

```
int n = 100;
string numero = to_string(n);
```

Esta función pertenece a la versión C++11 del lenguaje, por lo que para poder utilizarla en nuestro código habría que compilar añadiendo a `g++` el parámetro `std=c++11`. Por ejemplo, si nuestro código se llama `prog.cc` deberíamos de compilarlo con el comando `g++ -std=c++11 prog.cc`.

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());
```

Sacar las 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:

```
#include <sstream>
```

Ejemplo:

```
stringstream ss("Hola mundo cruel 1 32 2.3");
string s;

while(ss >> s)
{
    cout << "Palabra: " << s << endl;
}
```

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.

Conversión entre cadenas en C y `string`

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 mencionamos más arriba, junto con la función `strcpy` para hacer la copia entre cadenas.

Ejemplo:

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

Comparativa entre cadenas de caracteres en C y `string`

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

vectores de caracters	**`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.

Ficheros de texto

Definición

Los ficheros de texto (también llamados **ficheros con formato**) son aquellos cuyo contenido solo son caracteres imprimibles (en código ASCII son los caracteres a partir del código 32, el cual corresponde al espacio en blanco: ' '). Éstos pueden estar creados con diversos juegos de caracteres: ASCII (el código por defecto de los ficheros de texto), EBCDIC, Unicode (en su codificación más comúnmente usada, UTF-8). Un juego de caracteres o código de caracteres consiste en asignar un número a cada símbolo escrito: letras, números, símbolos, etc. Su contenido no tiene porque seguir ningún formato o disposición especial (no se puede 'decorar' el texto). En algunos casos tienen caracteres especiales para indicar los finales de linea (EOLN por *End of line*) y para señalar el final de fichero (EOF por *End of File*).

Ejemplos de ficheros de texto:

- Un programa en C++, concretamente el fichero que contiene su código fuente sin compilar
- 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++

Declaración

Para poder trabajar con ficheros de texto, debemos incluir la biblioteca de funciones *fstream*:

```
#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:

```
ifstream fich_lectura;
```

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

```
ofstream fich_escritura;
```

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

```
fstream fich_txt_lect_escr;
```

  Aunque usar ficheros de texto en ambos modos simultáneamente es algo poco frecuente.

Operaciones de apertura y cierre

Para trabajar con un fichero, es necesario abrirlo, y para ello se usa la función

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

Por ejemplo:

```
ifstream fichero; // declaramos la variable del fichero
const char nombre[]="mifichero.txt"; // declaramos una variable con el nombre del fich
ero
fichero.open(nombre,ios::in); // abrimos el fichero en modo lectura
```

Si tenemos el nombre del fichero almacenado en una variable de tipo *string*, entonces deberemos usar la función de conversión `c_str()`, por ejemplo:

```
ifstream fichero; // variable del fichero
string nombre="mifichero.txt"; // variable tipo string con el nombre del fichero
fichero.open(nombre.c_str(),ios::in); // abrimos el fichero en modo lectura
```

En función de como vayamos a trabajar con el fichero (sólo lectura, sólo escritura, lectura y escritura, añadir al final, etc.), tendremos que utilizar un modo de apertura diferente. En C++ tenemos los siguientes modos de apertura:

Modo	Observaciones	Constante en C++
Lectura	Sólo consultas, coloca el cursor al principio	`ios::in`
Escritura	Sólo escritura, colocando el cursor al principio, puede sobreescribir 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	`ios::out | ios::app`
Ir al final	Este modo permite moverse al final del fichero tras abrirlo	`ios::ate`
Fichero binario	Este modo permite abrir ficheros binarios, lo veremos en el siguiente apartado	`ios::binary`
Truncar/eliminar contenido	Este modo permite borrar TODO el contenido del fichero tras abrirlo en modo escritura	`ios::trunc`

Existe una forma de realizar una declaración y apertura de un fichero con una sola instrucción.

```
ifstream fichero("datos.txt");
```

Con esta instrucción abrimos un fichero llamado *datos.txt* en modo lectura (`ios::in`) y lo asignamos a la variable `fichero` .

```
ofstream ficheroSalida("resultado.txt");
```

Esta instrucción nos permite abrir un fichero llamado *resultado.txt* en modo escritura (`ios::out`) y queda asignado a la variable `ficheroSalida` .

La operación de abrir un fichero es delicada en tanto que puede producir errores en tiempo de ejecución. Por lo que es importante, tras intentar abrir un fichero, comprobar si éste está correctamente abierto y listo para trabajar. Con la función `is_open()` podemos verificar si efectivamente nuestro fichero está abierto o no.

```
if (fichero.is_open()) {
   // ya se puede trabajar con el ...
} else {
   // error de apertura
}
```

Finalmente, tras trabajar con un fichero, tenemos que cerrarlo, para liberar recursos y permitir que otros procesos/programas puedan trabajar con el. Para ello, usamos la función `close()`:

```
fichero.close();
```

Lectura de ficheros de texto

Detección de final de fichero

Durante el trabajo (consulta de contenidos) con un fichero tenemos que saber en qué momento hemos llegado al final del mismo y parar de leer (para no incurrir en un error). Para saber si estamos en el final de un fichero tras una o varias operaciones de lectura, se usa el método `eof()`. Por tanto la forma típica de recorrer el contenido de un fichero es mediante un bucle que tras cada operación de lectura, comprueba si estamos al final del fichero:

```
ifstream fichero;
...
while(!fichero.eof()) { ... }
```

Éste método devuelve un valor booleano (*true* o *false*) en función de si estamos o no al final. Cuando realizamos una operación de lectura de un dato (carácter, número, etc.) que no está en el fichero, devuelve *true*. Pero, **cuidado**, despues de haber leido **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*.

Ejemplos de lectura de un fichero

Vamos a ver varias formas de leer el contenido de un fichero de texto:

Primer ejemplo

```
...
if (fichero.is_open()) {
   string s="";
   getline(fichero,s);
   // otra opción: fichero.getline(cad,tCAD);
   // siendo 'cad' de tipo 'char []' y tCAD el tamaño de la cadena 'cad'.
   while (!fichero.eof()) {
      // hacer algo con 's'
      getline(fichero,s);
   }
   fichero.close();
}
```

Éste código se puede simplificar más. Además el bucle tiene un problema al final, 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, antes de utilizarla.

Segundo ejemplo, más compacto

```
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.

Lectura carácter a carácter

```
ifstream fichero("datos.txt");

if (fichero.is_open()) {
   char c; // Tambien puede ser de otros tipos: int, float, etc.
   while (fichero >> c) {
      // hacer algo con 'c'
   }
   fichero.close();
}
```

Este ejemplo muestra como trabajar con un fichero leyendolo carácter a carácter. 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.

Otro ejemplo, similar al anterior, leemos el fichero carácter a carácter, pero cuyo contenido tiene espacios en blanco:

```
if (fichero.is_open())   {
   char c;
   while (fichero.get(c)) {
      // hacer algo con 'c'
   }
 fichero.close();
}
```

Lectura de ficheros de texto con contenido 'conocido'

En este ejemplo leeremos el contenido de un fichero que tiene una o más lineas cuyo contenido es un *string* y a continuación dos números enteros. Por ejemplo, un fichero con esta estructura seria:

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

El código:

```
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;
   }
}
```

Otro ejemplo más: En este 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, por ejemplo:

```
3 pedro antonio jordi
```

Y el código

```
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 tantas *strings* como nos indica *cuantos* y los mostramos en pantalla.

Escritura de ficheros de texto

Para escribir en un fichero de texto usaremos el operador de volcado `<<`, porque tal y como dijimos anteriormente, los ficheros son objetos *stream* y pueden ser usados como buffers de entrada/salida de datos.

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

En este ejemplo, simplemente abrimos un fichero en modo escritura y volvamos o escribimos en él una frase y una variable llamada *numentero*. Se debe tener en cuenta que tal y como está escrito este ejemplo, el contenido del fichero será eliminado en cada ejecución y sustituido por el nuevo resultado.

Otro Ejemplo:

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

En este otro ejemplo:

- volcamos 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 y su contenido se mantendrá. Los datos de sucesivas ejecuciones serán añadidos al final del mismo.

Ejercicios

Ejercicio 3.1:

Implementa un programa que lea un fichero llamado 'fichero.txt' e imprima por pantalla las lineas del fichero que contienen la cadena 'Hola'.

Ejercicio 3.2:

Escribe un programa que lea un fichero llamado 'fichero.txt' y escriba en un segundo fichero, llamado 'FICHERO.TXT' el contenido del primero con todas sus letras (carácteres alfanuméricos) pasados a mayúsculas. Ejemplo:

fichero.txt	FICHERO.TXT
Hola, mundo.	HOLA, MUNDO.
Como estamos?	COMO ESTAMOS?
Adios y 1000 veces adios	ADIOS Y 1000 VECES ADIOS

Ejercicio 3.3:

Haz un programa que lea dos ficheros de texto llamados 'f1.txt' y 'f2.txt' y escriba por pantalla las lineas que sean distintas en cada fichero (número de línea, y '< ' delante de la linea del primer fichero y '> ' precediendo la linea del segundo. Ejemplo:

f1.txt	f2.txt
Hola, mundo.	Hola, mundo.
Me llamo andreu	Me llamo Andreu
Adios, adios	Hasta la vista

El resultado debe ser:

```
Línea 2:
< Me llamo andreu
> Me llamo Andreu
Línea 3:
< Adios, adios
> Hasta la vista
```

Ejercicio 3.4:

Diseña una función "finfichero" que reciba dos parámetros: el primero debe ser un número entero positivo *n*, y el segundo el nombre de un fichero de texto. La función deberá mostrar en pantalla las últimas *n* lineas del fichero. Ejemplo:

```
finfichero(3, "texto.txt");
```

Tenemos dos opciones para implementar esta función:

- Solución *bestia*: leer el fichero para contar las líneas que tiene, y volver a leer el fichero para escribir las n líneas finales. Problema: si el fichero tiene **muchas lineas**??
- Otra solución: Utilizar un vector de string de tamaño *n* que almacene en todo momento las n últimas líneas leídas (aunque al principio tendrá menos de n líneas)

Ejercicio 3.5:

Dados dos ficheros de texto "f1.txt" y "f2.txt", en los que cada línea es una serie de números separados por ":", y suponiendo que las líneas están ordenadas por el primer número de menor a mayor en los dos ficheros, haz un programa que lea los dos ficheros línea por línea y escriba en el fichero “f3.txt” las líneas comunes a ambos ficheros, como en el siguiente ejemplo:

f1.txt	f2.txt	f3.txt
10:4543:23	10:334:110	10:4543:23:334:110
15:1:234:67	12:222:222	15:1:234:67:881:44
17:188:22	15:881:44	20:111:22:454:313
20:111:22	20:454:313	

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 estan alojados en la memoria del ordenador, no se convierten en caracteres al gardarlos 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

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

```
#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:

  ```
  ifstream fich_lectura;
  ```

- Solo para escribir en él:

  ```
  ofstream fich_escritura;
  ```

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

```
fstream fich_lect_escr;
```

Operaciones de 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:

```
fich_lectura.open("mifichero.dat", ios::in | ios::binary);
```

- Apertura para escritura:

```
fich_escritura.open("mifichero.dat", ios:out | ios::binary);
```

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

```
ifstream fbl("mifichero.dat", ios::binary); // fichero binaro de lectura
ofstream fbe("otrofichero.dat", ios::binary); // fichero binario de escritura
```

Combinando varios modos de apertura podemos obtener:

- Ficheros binarios de lectura y escritura: `ios::in | ios::out | ios::binary`
- Ficheros binarios para ascritura/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()`.

```
fich_binario.close();
```

Lectura de ficheros binarios

Para leer ficheros finarios se utiliza la función `read(registro, tamaño)` a la que le 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 se obtiene 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 *TIPOCIUDAD*.

```
typedef struct { ... } TIPOCIUDAD;
...
TIPOCIUDAD 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 recien leida. Finalmente, cuando lleguemos al final de fichero, el *while* finalizará y cerraremos el fichero.

`NOTA: Para leer el contenido de un fichero binario, tenemos que saber su estructura (el tipo de resgistro o *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(posición, desde)`, donde:

- posición: es la posición (medida en bytes) donde queremos desplazar el cursor o puntero de lectura. Puede tener **un valor negativo**.
- desde: es la referencia o desplazamiento desde la que calcular la posición anterior.

Esta puede tener los siguientes valores:

Valor	ejemplo	relativo a
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 deseamos acceder y leer el tercer elemento, hariamos:

```
...
if (fbl.is_open()) {
    // nos posicionamos justo ante el tercer elemento:
    fbl.seekg( (3-1)*sizeof(ciudad), ios::beg); // contamos el tercer registro de tama
ño ciudad desde el principio
    fbl.read( (char *)&ciudad, sizeof(ciudad) );
}
...
```

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

```
...
if (fbl.is_open()) {
    // nos posicionamos justo ante el tercer elemento:
    fbl.seekg( (-1)*sizeof(ciudad), ios::end);
    fbl.read( (char *)&ciudad, sizeof(ciudad) );
}
...
```

A la función *seekg* le pasamos una posición negativa, relativa al final del fichero.

Escritura de ficheros binarios

Para guardar datos en un fichero binario se usa la función `write(registro, tamaño)`, 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. Con la función `sizeof()` calcularemos la cantidad de bytes a escribir.

En el siguiente ejemplo de código escribimos en un fichero binario llamado "mifichero.dat" un registro de tipo *TIPOCIUDAD*.

```
typedef struct { ... } TIPOCIUDAD;
...
TIPOCIUDAD ciudad;
ofstream fbe("mifichero.dat", ios::binary);

if (fbe.is_open())
{
    // introducimos datos en el registro 'ciudad'
    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(posición, desde)`, los argumentos son análogos a la función *seekg*:

- posición: 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.
- desde: es la referencia o desplazamiento desde la que calcular la posición anterior. Puede tener los mismos valores que en la función *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 carácteres y queremos almacenarlo en un fichero binario, tenemos que usar un vector de carácteres en lugar de un *string*. Al hacer la conversión puede ser que tengamos que recortar el *string* para adecuarlo al tamaño del vector. Por ejemplo:

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

string s="Alicante";
...
TIPOCIUDAD ciudad;
ciudad.codigo=3;
strncpy(ciudad.nombre, s.c_str(), MAXLONG-1); // convertimos el string a vector de car
ácteres
ciudad.nombre[MAXLONG-1]='\0'; // strncpy no pone el \0 si no esta en la cadena origin
al.
...
fichero.write((const char *)&ciudad, sizeof(ciudad)); // escribimos el registro
...
```

Funciones tellg() y tellp()

Existen dos funciones que nos permiten obtener la posición actual del puntero (el de lectura y de escritura). Devuelven la posición en **bytes**.

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

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;
```

Gestión de errores

Al trabajar con ficheros (tanto de texto como binarios) hay dos operaciones que son especialmente subceptibles de producir un error en tiempo de ejecución, que en ocasiones puede ser fatal:

- en la apertura de un fichero: fichero inexistente, falta de permisos, etc.
- en las operaciones de lectura/escritura (sobretodo las de escritura): permisos, fichero bloqueado, etc.

Para comprobar si la operación de apertura ha sido exitosa podemos usar la función `is_open()` que nos dice si el fichero esta correctamente abierto, tal y como hicimos al principio del capítulo.

Tras una operación de lectura o escritura, es recomendable comprobar si ha habido algún error, para ello tenemos el método `fail()` (entre otras funciones). Un ejemplo de uso:

```
fbl.read( (char *)&registro, sizeof(registro) );
if (fichero.fail() && !fichero.eof() ) {
   ... // error de lectura
}
```

En este otro ejemplo más elaborado leemos un fichero hasta el final y en cada operación de lectura comprobamos si se ha producido un error, en ese caso, salimos del bucle, emitimos un mensaje de error y cerramos el fichero.

```
...//abrimos el fichero
if (fi.is_open()) {
   bool error=false; // no hay error por ahora
   string s;

   while (getline(fi,s) && !error) { // acabamos al llegar al final o bien si hay error

      // leemos y comprobamos si hay error
      if (fi.fail() && !fi.eof()) {
         error=true;
      } else {
         // Hacer algo con s
      }
   }

   if (error) {
      cout << "Error de lectura" << endl;
   }
   fi.close();
}
```

Ejercicios

Ejercicio 3.6:

Dado un fichero binario "alumnos.dat" que tiene registros de alumnos, con la siguiente información por cada alumno:

dni	vector de 10 caracteres
apellidos	vector de 40 caracteres
nombre	vector de 20 caracteres
turno	entero

Apartado 1: Haz un programa que imprima por pantalla el DNI de todos los alumnos del turno 7. **Apartado 2:** Haz un programa que intercambie los alumnos de los turnos 4 y 8 (los turnos van del 1 al 10).

Ejercicio 3.7:

Dado el fichero “"lumnos.dat" del ejercicio anterior, haz un programa que pase a mayúsculas el nombre y los apellidos del quinto alumno del fichero, y lo vuelva a escribir en el fichero.

Ejercicio 3.8:

Diseña un programa que construya el fichero "alumnos.dat" a partir de un fichero de texto "alu.txt" en el que cada dato (dni, nombre, etc) está en una línea distinta. Ten en cuenta que en el fichero de texto el dni, nombre y apellidos pueden ser más largos que los tamaños especificados para el fichero binario, en cuyo caso se deben recortar.

Ejercicio 3.9:

Escribe un programa que se encarge de la asignación automática de alumnos en 10 turnos de prácticas. A cada alumno se le asignará el turno correspondiente al último número de su DNI (a los alumnos con DNI acabado en 0 se les asignará el turno 10). Los datos de los alumnos están en un fichero "alumnos.dat" con la misma estructura que en los ejercicios anteriores. La asignación de turnos debe hacerse leyendo el fichero una sola vez, y sin almacenarlo en memoria. En cada paso se leerá la información correspondiente a un alumno, se calculará el turno que le corresponde, y se guardará el registro en la misma posición.

Tema 4 - Memoria dinámica (en construcción)

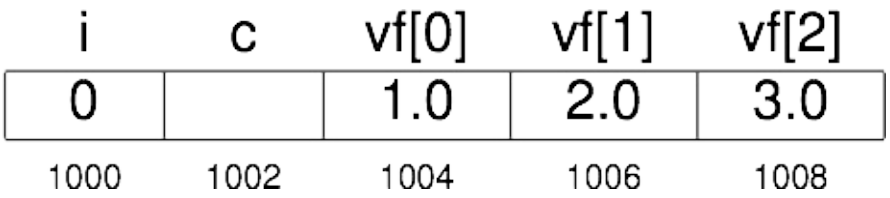
Organización de la memoria

Memoria estática

El tamaño es fijo y se conoce al implementar el programa.

Declaración de variables:

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



Memoria dinámica

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 puede cambiar.

En C++ se puede hacer uso de la memoria dinámica usando punteros.

Punteros

Definición y declaración

Un puntero es un **número** (entero largo) que se corresponde con una dirección de memoria.

En la declaración de un puntero, se debe especificar el tipo de dato que contiene la dirección de memoria.

Se declaran usando el carácter *

Ejemplos:

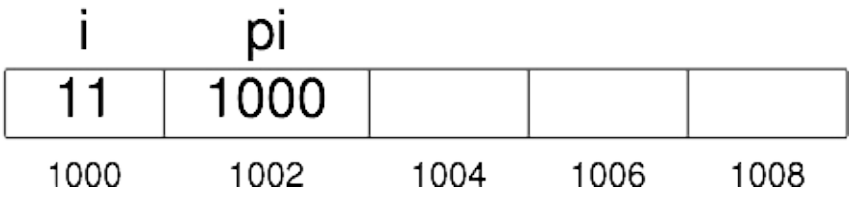
```
int *punteroAEntero;
char *punteroAChar;
int *VectorPunterosAEntero[tVECTOR];
double **punteroAPunteroAReal;
```

Dirección y contenido

`*x`	**Contenido de la dirección apuntada por** `x`
`&x`	Dirección de memoria de la variable `x`

Ejemplo:

```
int i=3;
int *pi;
pi=&i; // pi=direccion de memoria de i
*pi = 11; // contenido de pi=11. Por lo tanto, i = 11
```



Declaracion con inicialización

Declaración con inicialización:

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

El puntero NULL es aquel que no apunta a ninguna variable:

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

Precaución: siempre que un puntero no tenga memoria asignada debe valer NULL.

Ejercicios

Ejercicio 1

Indica cuál sería la salida de los siguientes fragmentos de código:

```
int e1;
int *p1, *p2;
e1 = 7;
p1 = &e1;
p2 = p1;
e1++;
(*p2) += e1;
cout << *p1;
```

```
int a=7;
int *p=&a;
int **pp=&p;
cout << **pp;
```

Uso de punteros

Reserva y liberación de memoria

Los punteros pueden usarse para reservar (con new) o liberar (con delete) memoria dinámica.

Ejemplo:

```
double *pd;
pd = new double; // reserva memria
pd = 4.75; *
cout << *pd << endl; // muestra 4.75
delete pd; // libera memoria
```

Ojo: Las variables locales y las reservadas con `new` van a zonas de memoria distintas.

Cuando se usa `new`, el compilador reserva memoria y devuelve la dirección de inicio de ese espacio reservado.

Si no hay suficiente memoria para la reserva, new devuelve `NULL`.

Siempre que se reserva memoria con new hay que liberarla con delete.

Ojo: Tras hacer delete, el puntero no vale NULL por defecto.

Un puntero se puede reutilizar; tras liberar su contenido se puede reservar memoria otra vez con `new`.

Punteros y vectores

Los punteros también pueden usarse para crear vectores o matrices.

Para reservar memoria hay que usar corchetes y especificar el tamaño.

Para liberar toda la memoria reservada es necesario usar corchetes.

Ejemplo:

```
int *pv;
int n = 10;
pv = new int[n]; // reserva memoria para n enteros
pv[0] = 585; // usamos el puntero como si fuera un vector
delete [] pv; // liberar la memoria reservada
```

Los punteros se pueden usar como accesos directos a componentes de vectores o matrices.

Ejemplo:

```
int v[tVECTOR];
int *pv;
pv = &(v[7]);
*pv = 117; // v[7] = 117;
```

Punteros definidos con typedef

Se pueden definir nuevos tipos de datos con typedef:

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

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

```
typedef int *punteroAEntero;
typedef struct {
    char c;
    int i;
} Registro, *PunteroRegistro;
```

Punteros y registros

Cuando un puntero referencia a un registro, se puede usar el operador `->` para acceder a sus campos.

```
typedef struct {
    char c;
    int i;
} Registro, *PunteroRegistro;

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

Parámetros de funciones

Paso de parámetros a funciones:

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

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

int main() {
    int i = 0;
    int *p = &i;
    f(p); // Llamada a funciones
    f2(p);
}
```

Paso de parámetros a funciones usando typedef:

```
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;
    PInt p = &i;
    f(p); // Llamada a funciones
    f2(p);
}
```

Errores comunes

Utilizar un puntero sin haberle asignado memoria o apuntando a nada.

```
int *pEntero;
*pEntero = 7; // Error !!!
```

Usar un puntero tras haberlo liberado.

```
punteroREGISTRO p,q;
p = new REGISTRO;
...
q = p;
delete p;
q->num =7; // Error !!!
```

Liberar memoria no reservada.

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

Ejercicios

Ejercicio 2

Dado el siguiente registro:

```
typedef struct {
    char nombre[32];
    int edad;
} Cliente;
```

Realizar un programa que lea un cliente (sólo uno) de un fichero binario, lo almacene en memoria dinámica usando un puntero, imprima su contenido y finalmente libere la memoria reservada.

Tema 5: Introducción a la programación orientada a objetos (en construcción)

Introducción a la programación orientada a objetos

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 más bien 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 no suelen usar C++ para lo primero. En este curso no ahondaremos excesivamente en los elementos de C++ relacionados con la programación orientada a objetos, lo que se hará en cursos posteriores.

Clases y objetos

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

```
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`:

```
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. Además, el programador de una clase puede decidir que solo 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 o 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 metodos 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, a saber: 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 implemenmtación.

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 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 modificar las 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 `desplazate()` 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;
}

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

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++ e 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 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
```

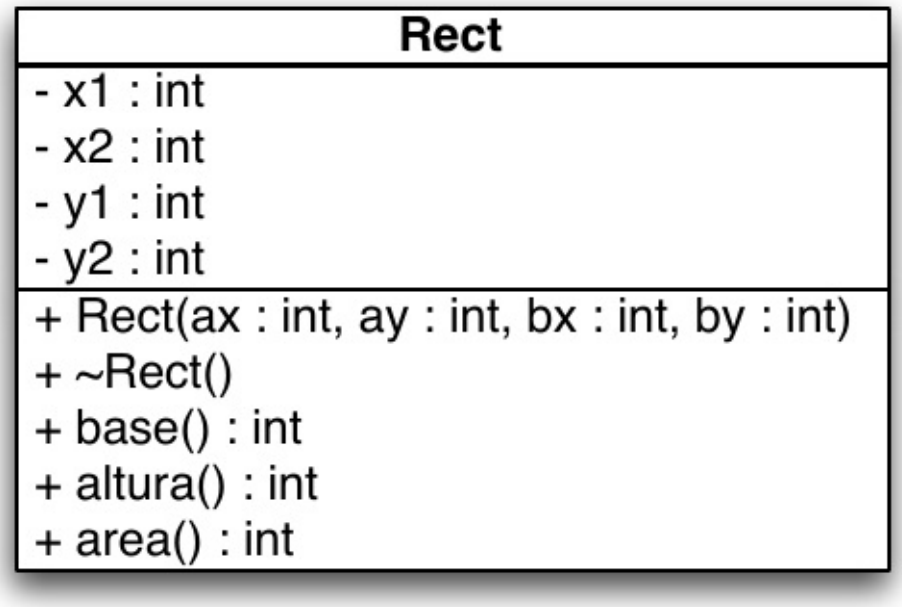
En el caso de programas pequeños, puede hacerse todo de una vez con:

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

Cuando los programas son más grandes, no podemos 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 metodos)
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 correspondiente.

```
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);
```

Nótese que 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 se ponen en el `.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 excepcion

  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 aqui
  {
     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`. Se puede invocar explícitamente de la forma `f.~Fecha()`, aunque son muy pocas las situaciones en las que esta invocación explícita es necesaria.

```
    // Declaracion
    ~Fecha();

    // Implementacion
    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.

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

// Implementacion
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 { // Metodo 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, en principio, 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.

```
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="2112";
```

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;  // Metodo static
```

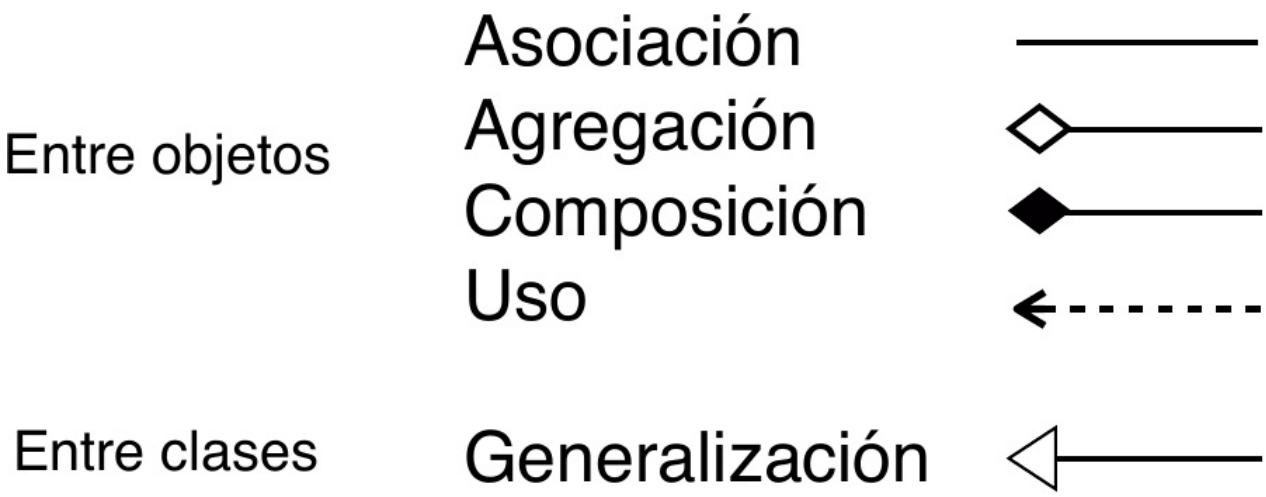
El puntero *this*

El puntero `this` es una pseudovariable que no se declara explícitamente en ningún punto del programa ni se puede modificar explícitamente. 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 a
ctual
    cout << this->dia << endl;
}
```

Relaciones

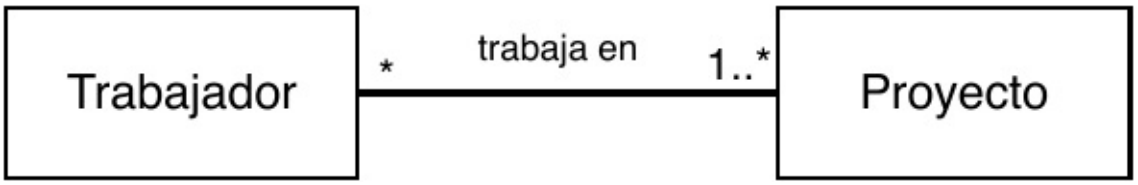
Existen distintos tipos de relaciones que pueden establecerse entre objetos y clases. La figura siguiente muestra las que vamos a estudiar en esta sección, junto con la notación gráfica que se utiliza en UML para representarlas.



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 bien 1..*n*, si hay un máximo definido)
- Cero o más: representada con *
- Número fijo: *m*

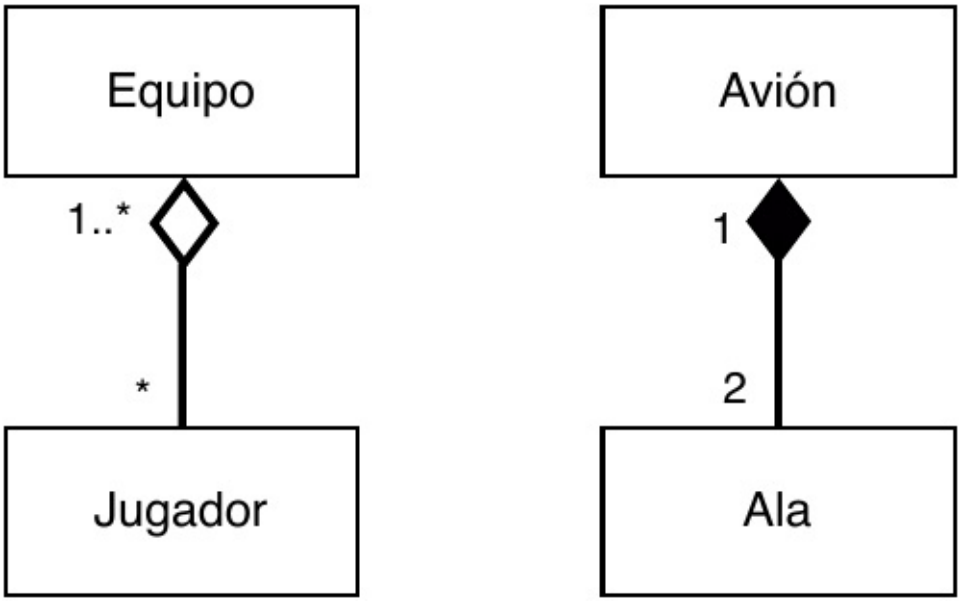
Asociación



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.

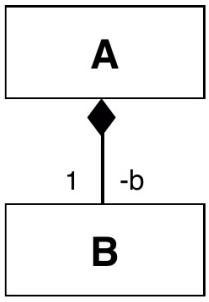
En el caso de la agregación (representada en UML mediante un rombo vacío en el lado de la clase *todo*), no ocurre así: el objeto parte puede existir sin el todo. La relación de agregación de la figura refleja que podemos vender un equipo, pero los jugadores pueden

irse a otro club.

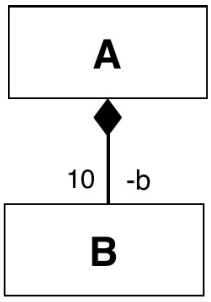
Algunas relaciones pueden ser consideradas como agregaciones o composiciones, en función del contexto y de la aplicación para la que se establezcan (por ejemplo, la relación entre bicicleta y rueda).

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.

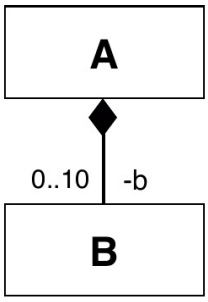
La siguiente figura representa esquemáticamente cómo se implementan varias composiciones con diferentes cardinalidades:



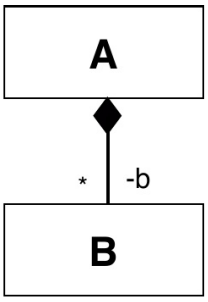

```
class A {
   private:
      B b;
   ...
};
```




```
class A {
   private:
      B b[10];
   ...
};
```

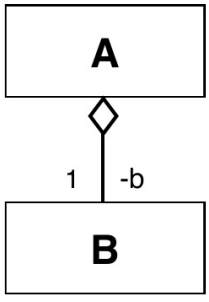



```
class A {
   private:
      vector<B> b;
      static const int N=10;
   ...
};
```

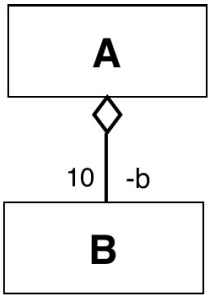



```
class A {
   private:
      vector<B> b;
   ...
};
```

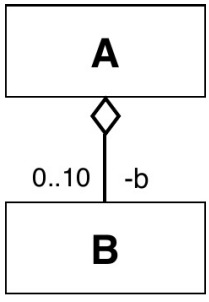
En estas composiciones, la clase `A` (la clase que representa el *todo*) tiene su propia copia de los objetos de clase `B`, por lo que el borrado de un objeto de la clase `A` implica el borrado de sus objetos de clase `B`. En la agregación, sin embargo, la clase `A` incluye punteros a objetos de clase `B` creados fuera de `A`, objetos cuyo ciclo de vida no es gestionado por la clase A:



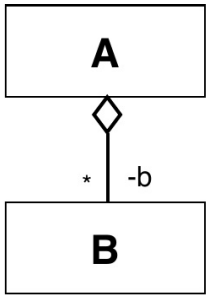

```
class A {
   private:
      B *b;
   ...
};
```




```
class A {
   private:
      B *b[10];
   ...
};
```




```
class A {
   private:
      vector<B*> b;
      static const int N=10;
   ...
};
```




```
class A {
   private:
      vector<B*> 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*; en cualquier caso, para mantener la naturaleza de la relación, la clase *todo* tendrá que crear probablemente una copia profunda del objeto *parte* que le permita controlar su ciclo de vida.

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*).

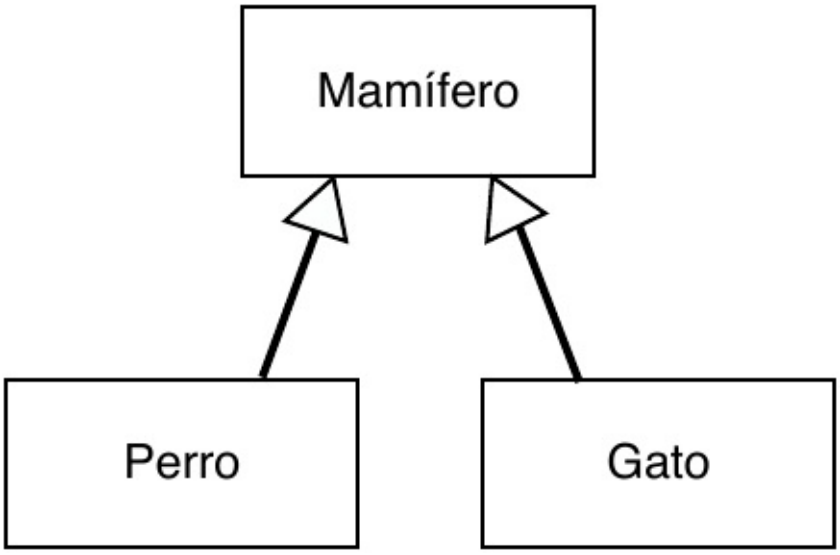
El siguiente diagrama UML y el código en C++ que le sigue representan una relación de uso de la clase `Gasolinera` por la clase `Coche` que se ajusta a los dos primeros casos de la lista anterior.


```
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
```

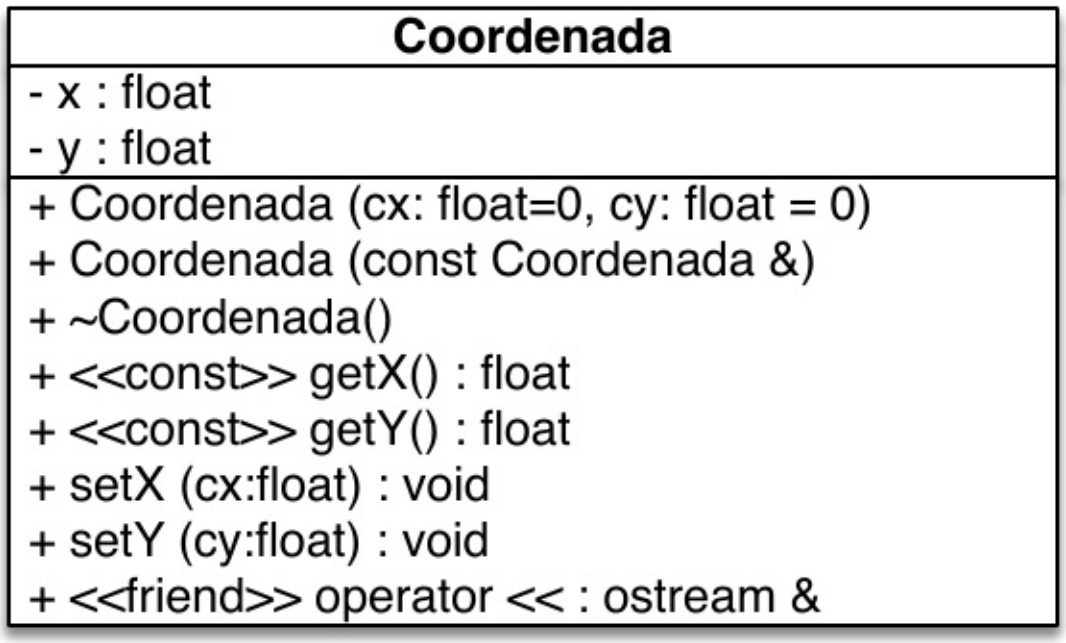
Podemos definir variables que son sustituidas donde corresponda. Una versión mejorada del `makefile` anterior que usa algunas variables sería:

```
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)
```

Ejercicios

Ejercicio 1

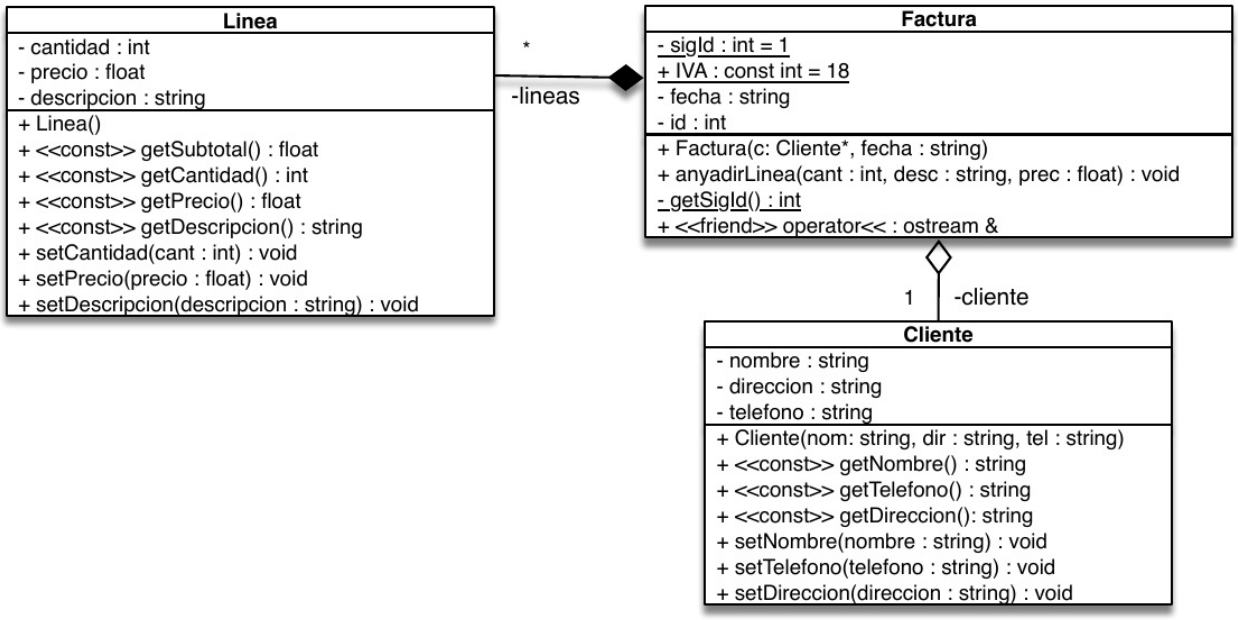
Implementa la clase del siguiente diagrama:



Debes crear los ficheros `Coordenada.cc` y `Coordenada.h`, y un `makefile` para compilarlos con un programa `principal.cc`. En el `main` se debe pedir al usuario dos números y crear con ellos una coordenada para imprimirla con el operador salida en el formato `x,y`.
Escribe el código necesario para que cada método sea utilizado al menos una vez.

Ejercicio 2

Implementa el código correspondiente al siguiente diagrama de clases:



Se debe hacer un programa que cree una nueva factura, añada un producto y lo imprima. Desde el constructor de `Factura` debe llamarse al método `getSigid`, que debe devolver el valor de `sigid` e incrementarlo. Este es un ejemplo de salida al imprimir una factura:

```
Factura nº: 12345
Fecha: 18/4/2011

Datos del cliente
----------------------
Nombre: Agapito Piedralisa
Dirección: c/ Río Seco, 2
Teléfono: 123456789

Detalle de la factura
---------------------
Línea;Producto;Cantidad;Precio ud.;Precio total
--
1;Ratón USB;1;8,43;8,43
2;Memoria RAM 2GB;2;21,15;42,3
3;Altavoces;1;12,66;12,66![](/assets/agregacion-composicion-crop.png)

Subtotal: 63,39 €
IVA (18%): 11,41 €
TOTAL: 74.8002 €
```