



@prog2ua

Tema 1: Introducción

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2024-2025



Índice

1. Diseño de algoritmos y programas
2. Compilación
3. Elementos básicos de C++
4. Depuración
5. Ejercicios

Diseño de algoritmos y programas

Cómo se hace un programa

1. Estudio del problema y de las posibles soluciones
2. Diseño del algoritmo **en papel**
3. Escritura del programa **en el ordenador**
4. Compilación del programa y corrección de errores
5. Ejecución del programa
6. ... y prueba de todos los casos posibles (o casi)

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

Metodología recomendada para programar

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

Compilación

El proceso de compilación

- El *compilador* permite convertir un código fuente en un código objeto
- En Programación 2 usamos el compilador GNU C++ para transformar el código fuente en C++ en un programa ejecutable
- El compilador de GNU se invoca con el programa `g++` y admite numerosos argumentos:^{*}
 - `-Wall`: muestra todos los *warnings*
 - `-g`: añade información para el depurador
 - `-o`: para indicar el nombre del ejecutable
 - `-std=c++11`: para usar el estándar de C++ de 2011
 - `--version`: muestra la versión actual del compilador
- Ejemplo de uso:

Terminal

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

*Puedes ver la lista completa de argumentos ejecutando `man g++` en el terminal de Linux

Elementos básicos de C++

Estructura básica de un programa en C++

```
#include <ficheros de cabecera estándar>
...
#include "ficheros de cabecera propios"
...
using namespace std; // Permite usar cout, string...
...
const ... // Constantes
...
typedef struct enum ... // Definición de nuevos tipos
...
// Variables globales: ¡¡PROHIBIDO en Programación 2!!
...
funciones ... // Declaración de funciones
...
int main() { // Función principal
...
}
```

Mantenemos algunas normas de Programación 1

- No se permite usar variables gloables
- No deben aparecer warnings al compilar los ficheros fuente de las prácticas y los exámenes
- No se permite el uso de break y continue en estructuras de repetición
- No se permiten múltiples return en una misma función

Identificadores

- Los *identificadores* son nombres de variables, constantes y funciones
- Han de comenzar por letra minúscula, mayúscula o guión bajo
- C++ distingue entre letras mayúsculas y minúsculas:

```
int grupo, Grupo; // Son dos variables diferentes
```

- El identificador debe indicar para qué se utiliza:

```
int numeroAlumnos=0;  
void visualizarAlumnos() {...}
```

- Malos ejemplos:

```
const int OCHO=8;  
int p,q,r,a,b;  
int contador1,contador2; // Más habitual: int i,j;
```

Palabras reservadas

- En C++ hay *palabras reservadas* que no se pueden utilizar como nombres definidos por el usuario:

```
if while for do int friend long auto public union ...
```

- Si las usamos como identificadores nos dará un error de compilación:

```
int friend=10;
```

Terminal

```
error: expected unqualified-id before '=' token
```

- Este tipo de mensajes de error no es fácil de interpretar

Variables > Definición y tipos

- Las *variables* permiten almacenar diferentes tipos de datos
- Se debe indicar el tipo de la variable cuando se declara
- *Tipos básicos* (o primitivos) de datos en C++:

Tipo	Tamaño (en bits)*
int	32
char	8
float	32
double	64
bool	8
void	No es un tipo

- Se puede usar `unsigned` con `int` para tener solo números positivos (sin signo):

```
int i=3; // Valores entre -2.147.483.648 y 2.147.483.647  
unsigned int j=3; // Valores entre 0 y 4.294.967.295
```

*En la arquitectura x86

Variables > Inicialización

- Siempre que se declara una variable se debe *inicializar*:

```
int numeroProfesores=11;
```

- No es necesario inicializarla si lo primero que se va a hacer después de declarar la variable es asignarle valor:

```
int i;  
for(i=0;i<25;i++){...}
```

Variables > Ámbito (1/3)

- El *ámbito* de un variable (o constante) es la parte del programa donde se puede acceder a esa variable
- Una variable se puede usar desde que se declara y dentro del bloque entre llaves que la contiene:

```
int numCajas=0;

if(i<10) {
    // numCajas se puede usar aquí
    int numCajas=100; // Mismo nombre pero otro ámbito
    cout << numCajas << endl; // Imprime 100
}

cout << numCajas << endl; // Imprime 0
```

Variables > Ámbito (2/3)

- *Variable local* a una función:
 - Aquella que se declara dentro de una función
 - Normalmente se declara al principio, aunque pueden introducirse en un punto intermedio:

```
void imprimir() {
    int i=3,j=5; // Al principio de la función
    cout << i << j << endl;
    ...
    int k=7; // En un punto intermedio
    cout << k << endl;
}
```

- *Variable global*:
 - Se declara fuera de las funciones
 - Se recomienda no utilizar variables globales (son peligrosas)
 - **En Programación 2 está prohibido usar variables globales**

Variables > Ámbito (3/3)

- Ejemplo de efecto colateral al usar una variable global:

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

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

int main(){
    cuentaAtras();
    cuentaAtras(); // Aquí no imprime nada
}
```

Constantes

- Las *constantes* tienen un valor fijo (no puede ser cambiado) durante toda la ejecución del programa
- Se declaran anteponiendo `const` al tipo de dato:

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

- Son útiles para definir valores que se usen en múltiples puntos de un programa y que no cambien de valor (como el tamaño de un vector o de un tablero de ajedrez)

Tipo	Ejemplos
int	123 017* 1010101
float/double	123.7 .123 1e1 1.231E-12
char	'a' '1' ';' '\n' '\0' '\\'
char[] (cadena)	"" "hola" "doble: \\""
bool	true false

*Un valor constante con un cero al principio se trata como un número octal

Tipos de datos > Conversión (1/2)

- *Conversión de tipo implícita*: la hace el compilador de manera automática

Tipos	Ejemplo
char → int	int a='A'+2; // a vale 67
int → float	float pi=1+2.141592;
float → double	double piMedios=pi/2.0;
bool → int	int b=true; // b vale 1
int → bool	bool c=77212; // c vale true

- *Conversión de tipo explícita*: la define el programador utilizando el operador *cast* (poniendo el tipo de dato entre paréntesis)

```
char laC=(char)('A'+2); // laC vale 'C'  
int pEnteraPi=(int)pi; // pEnteraPi vale 3
```

Tipos de datos > Conversión (2/2)

- A veces, si no se hace *cast*, el compilador da un aviso (*warning*) de que se están comparando tipos que no son iguales
- **Es importante no ignorar los *warnings***
- Cuando comparamos un entero (*int*) con un entero sin signo (*unsigned int*) se produce un *warning*:

```
int num=5;
char cad[]={ "Hola" };

if(num<strlen(cad)){ // strlen devuelve entero sin signo
    // Se puede evitar el warning con un cast:
    // if((unsigned)num<strlen(cad))
}
```

Terminal

warning: comparison between signed and unsigned integer...

Tipos de datos > Definición de nuevos tipos

- En C++ se pueden definir nuevos tipos mediante `typedef`:

```
typedef int entero;  
entero i,j;  
  
// logic y boolean son equivalentes al tipo bool  
typedef bool logic,boolean;
```

- Es posible declarar un vector como un tipo:

```
typedef char tCadena[50]; // tCadena es un vector de char
```

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

Tipos de datos > Comprobaciones

- En C++ se pueden comprobar si una variable es alfanumérica (un dígito o una letra) mediante la función `isalnum()`:

```
int isalnum(int c);
```

- Devuleve `true` si lo es y `false` en caso contrario
- Se debe incluir la librería `ctype` en el código para poder usarla:

```
#include <ctype.h>
...
if (isalnum(c)){ // Verificar si 'c' es alfanumérico
    cout << c << " es alfanumérico";
}
else{
    cout << c << " no es alfanumérico";
}
```

Operadores de incremento y decremento

- Los operadores ++ y -- se usan para incrementar o decrementar el valor de una variable entera en una unidad
- Preincremento/predecremento:* se incrementa/decrementa antes de tomar su valor

```
int i=3, j=3;  
int k=++i; // k vale 4, i vale 4  
int l=--j; // l vale 2, j vale 2
```

- Postincremento/postdecremento:* se incrementa/decrementa después de tomar su valor

```
int i=3, j=3;  
int k=i++; // k vale 3, i vale 4  
int l=j--; // l vale 3, j vale 2
```

- Es recomendable que aparezcan solos en la instrucción:

```
i++; // Equivalente a ++i  
j=(i++)+(--i); // ??
```

Expresiones aritméticas (1/2)

- Las *expresiones aritméticas* están formadas por operandos (int, float y double) y operadores aritméticos (+ - * /):

```
float i=4*5.7+3; // i vale 25.8
```

- Si aparece un operando de tipo char o bool se convierte a entero implícitamente:

```
int i=2+'a'; // i vale 99
```

- Si dividimos dos enteros el resultado es un entero:

```
cout << 7/2; // La salida es 3
```

- Si queremos que el resultado de la división entera sea un valor real hay que hacer un *cast* a float o double:

```
cout << (float)7/2; // La salida es 3.5
cout << (float)(7/2); // ¡Ojo! La salida es 3
```

Expresiones aritméticas (2/2)

- El operador `%` devuelve el resto de la división entera:

```
cout << 30%7; // La salida es 2
```

- Precedencia de operadores:^{*}

<code>++ (incremento) -- (decremento) ! (negación) - (menos unario)</code>
<code>* (multiplicación) / (división) % (módulo)</code>
<code>+ (suma) - (resta)</code>

- En caso de duda usar paréntesis:

```
cout << 2+3*4; // La salida es 14  
// * tiene más precedencia que +  
cout << 2+(3*4); // La salida es 14  
cout << (2+3)*4; // La salida es 20
```

^{*}De mayor a menor precedencia. Los operadores de una fila tienen la misma precedencia

Expresiones relacionales (1/3)

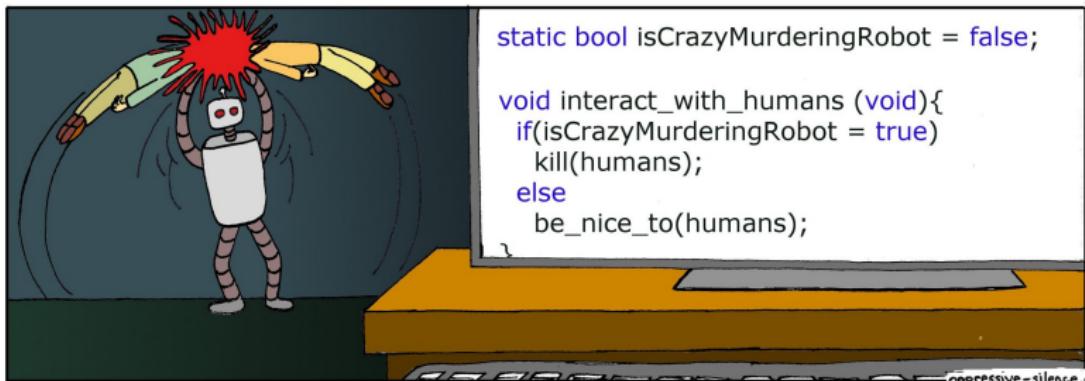
- Las *expresiones relacionales* permiten realizar comparaciones entre valores
- Operadores: == (igual), != (distinto), >= (mayor o igual), > (mayor estricto), <= (menor o igual) y < (menor estricto)
- Si los tipos de los operandos no son iguales se convierten (implícitamente) al tipo más general:

```
if(2<3.4){...} // Se transforma en: if(2.0<3.4)
```

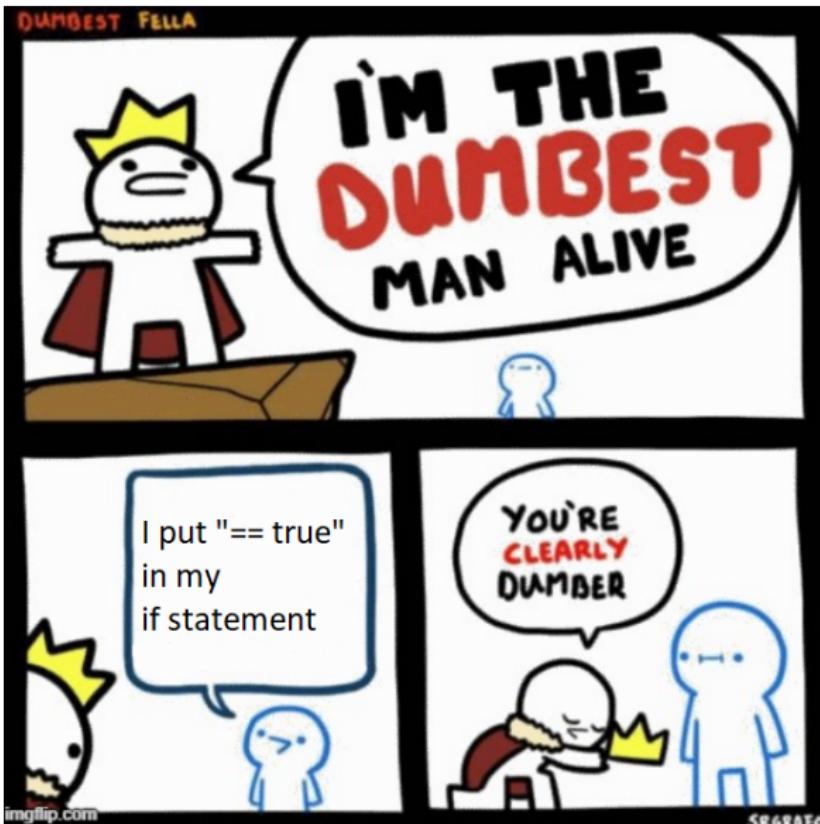
- Los operandos se agrupan de dos en dos por la izquierda. Para hacer $a < b < c$ hay que poner $a < b \&\& b < c$
- El resultado es 0 si la comparación es falsa y distinto de 0 si es cierta*

*En el compilador GCC es 1, pero el estándar de C++ no obliga a ello

Expresiones relacionales (2/3)



Expresiones relacionales (3/3)



Expresiones lógicas

- Las *expresiones lógicas* permiten relacionar valores booleanos y obtener un nuevo valor booleano
- Operadores: ! (negación), && (y lógico) y || (o lógico)
- Precedencia: ! > && > ||

```
if(a || b && c){...} // Equivale a: if(a || (b && c))
```

- *Evaluación en cortocircuito*:
 - Si el operando izquierdo de && es falso, el operando derecho no se evalúa (false && loquesea es siempre false)
 - Si el operando izquierdo de || es cierto, el operando derecho no se evalúa (true || loquesea es siempre true)

Entrada y salida

- Salida por pantalla con `cout`:

```
int i=7;  
cout << i << endl; // Muestra 7 y salto de línea (endl)
```

- Salida de error (por pantalla) con `cerr`:

```
int i=7;  
cerr << i << endl; // Muestra 7 y salto de línea (endl)
```

- Entrada por teclado con `cin`:*

```
int i;  
cin >> i; // Guarda en i un número escrito por teclado
```

*Más detalles en el Tema 2

Control de flujo > if

- Las *estructuras de control de flujo* evalúan una expresión condicional (true o false) y seleccionan la siguiente instrucción a ejecutar dependiendo del resultado
- if evalúa una condición y toma un camino u otro:

```
int num=0;  
cin >> num; // Leemos un número por teclado  
  
if(num<5){ // Si num es menor que 5 ejecuta esta parte  
    cout << "El número es menor que cinco";  
}  
else{ // Si no, ejecuta esta otra  
    cout << "El número es mayor o igual que cinco";  
}
```

Control de flujo > while

- while ejecuta instrucciones mientras se cumpla la condición:

```
int i=10;
while(i>=0){
    cout << i << endl; // Hará una cuenta atrás del 10 a 0
    i--; // Si no decrementamos entra en un bucle infinito
}
```

- Cuidado al utilizar || dentro de la condición, porque las dos partes han de ser falsas para que acabe el bucle:

```
while(i<tamano || !encontrado){
    // Las dos condiciones han de ser falsas para terminar
}
```

- Normalmente necesitaremos && en lugar de ||:

```
while(i<tamano && !encontrado){
    // Termina cuando alguna de las condiciones es falsa
}
```

Control de flujo > do-while

- do-while ejecuta el cuerpo del bloque al menos una vez:

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



Control de flujo > for

- for equivale a un while:

```
for(inicialización;condición;finalización){  
    // Instrucciones  
}
```

```
inicialización;  
while(condición){  
    // Instrucciones  
    finalización;  
}
```

- Tiene una sintaxis más elegante y compacta que while:

```
for(int i=10;i>=0;i--){  
    cout << i << endl; // Hará una cuenta atrás del 10 al 0  
}
```

Control de flujo > switch

- switch permite seleccionar entre varias opciones:

```
char opcion;
cin >> opcion; // Leemos un carácter de teclado

switch(opcion) {
    case 'a': cout << "Opción A" << endl;
                break; // Sale del switch
    case 'b': cout << "Opción B" << endl;
                break;
    case 'c': cout << "Opción C" << endl;
                break;
    default: cout << "Otra opción" << endl;
}
```

- La expresión en el switch (opcion en el ejemplo anterior) debe ser int o char (dará error de compilación en caso contrario)

Vectores y matrices (1/3)

- Los *vectores* (o *arrays*) almacenan múltiples valores en una única variable en posiciones de memoria contiguas
- Estos valores pueden ser de cualquier tipo que deseemos, incluso tipos de datos propios
- Al declarar un vector hay que especificar su tamaño (cuántos elementos almacena) mediante constantes o variables:

```
// Tamaño definido mediante constantes
const int MAXALUMNOS=100;
int alumnos[MAXALUMNOS]; // Puede almacenar 100 enteros
bool gruposLlenos[5]; // Puede almacenar 5 booleanos

// Tamaño definido mediante variables (no recomendable)
int numElementos;
cin >> numElementos; // No sabemos qué número introducirá
float listaNotas[numElementos];
```

Vectores y matrices (2/3)

- Cuando se inicializa un vector al declararlo no hace falta indicar su tamaño:

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

- Asignación y acceso a valores mediante el operador []:

```
const int TAM=10;
int vec[TAM];
vec[0]=7;
vec[TAM-1]=vec[TAM-2]+1; // vec[9]=vec[8]+1;
```

- Si un vector tiene tamaño TAM, el primer elemento se halla en la posición 0 y el último en la posición TAM-1
- Podemos tener un fallo en tiempo de ejecución si intentamos leer o escribir en un elemento fuera del vector:

```
int vec[5];
vec[5]=7; // Puede haber fallo en tiempo de ejecución
          // El último elemento válido está en vec[4]
```

Vectores y matrices (3/3)

- Una *matriz* es un vector cuyas posiciones son, cada una de ellas, otro vector
- Hay que dar tamaño a sus dos dimensiones (filas y columnas):

```
const int TAM=10;  
char tablero[TAM][TAM]; // Matriz de 10 x 10 elementos  
int tabla[5][8]; // Matriz de 5 x 8 elementos
```

- Como los vectores, comienzan en 0 y acaban en TAM-1
- Asignación y acceso a valores mediante el operador []:

```
int matriz[8][10];  
matriz[2][3]=7; // Hay que indicar fila y columna
```

- Es posible utilizar filas de matrices como si fueran vectores:

```
leeArray(matriz[4]); // Pasamos la fila 4 como un vector
```

Cadenas de caracteres

- Las *cadenas de caracteres* son vectores que contienen una secuencia de caracteres terminada en el carácter nulo '\0':*

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

"hola" →

h	o	l	a	\0
---	---	---	---	----

- Si no la inicializamos hay que especificar su tamaño:

```
const int TAM=10;
char cadena[TAM]; // Ok
char cadena2[]; // Error de compilación
```

- Recuerda: "a" es una cadena y 'a' es un carácter

```
char cadena[]={a}; // Ok
char cadena2[]='a'; // Error de compilación
```

*Más detalles sobre cadenas de caracteres en el Tema 2

Funciones > Definición (1/2)

- Una función es un bloque de código que realizan una tarea
- Permite agrupar operaciones comunes en un bloque reutilizable
- Puede opcionalmente tener parámetros de entrada y devolver un valor como salida:

```
tipoRetorno nombreFuncion(parametro1,parametro2,...){  
    tipoRetorno ret;  
  
    instrucion1;  
    instrucion2;  
    ...  
  
    return ret;  
}
```

- Una función no debería tener mucho código
- Si tengo que hacer *copy-paste* en el código es porque necesito una función

Funciones > Definición (2/2)

- Siempre se puede encontrar la forma de utilizar un único `return` en el cuerpo de una función:

```
// No permitido en Programación 2
bool buscar(int vec[], int n){
    for(int i=0;i<TAM;i++){
        if(vec[i]==n)
            return true; // Primer return
    }
    return false; // Segundo return
}
```

```
// Versión alternativa con un return
bool buscar(int vec[],int n){
    bool encontrado=false;
    for(int i=0;i<TAM && !encontrado;i++){
        if(vec[i]==n)
            encontrado=true;
    }
    return encontrado; // Un único return
}
```

Funciones > Parámetros (1/2)

- Se permite paso de parámetros por *valor* o por *referencia* (con &)

```
// a y b se pasan por valor, c por referencia
void funcion(int a,int b,bool &c){
    c=a<b; // c mantiene este valor al acabar la función
}
```

- Cuando se pasa un parámetro por valor, el compilador hace una copia local del mismo para usarlo dentro de la función
- Si es un tipo de dato muy grande, es conveniente pasarlo por referencia con `const` por eficiencia:

```
void funcion(const string &s){
    // El compilador no hace copia de s, pero si
    // intentamos modificarlo nos da un error
}
```

- En Programación 2 no se permite pasar parámetros por referencia si no van a ser modificados, excepto si es con `const`, como se ha explicado

Funciones > Parámetros (2/2)

- Los vectores y matrices se pasan implícitamente por referencia (no hay que poner & delante)
- El nombre de un vector o matriz, sin corchetes, contiene la dirección de memoria donde está almacenado*
- Al pasar una matriz como parámetro no hay que poner el tamaño de la primera dimensión en la declaración de la función:

```
void sumar(int v[],int m[] [TAM]) {  
    // En m no se pone el tamaño de la primera dimensión  
    ...  
}  
...  
// No se ponen corchetes en la llamada a la función  
sumar(v,m);
```

*Más información en el Tema 4

Funciones > Prototipos

- A veces es necesario utilizar una función antes de que aparezca su código (o una función cuyo código esté en otro módulo)*
- En esos casos se debe poner el *prototipo* de la función:

```
void miFuncion(bool, char, double[]); // Prototipo

char otraFuncion(){
    double vr[20];
    // Todavía no se ha declarado miFuncion
    // pero podemos usarla gracias al prototipo
    miFuncion(true, 'a', vr);
}

// Declaración de la función
void miFuncion(bool exist, char opt, double vec[]){
    ...
}
```

*Más información sobre la creación de módulos en el Tema 5

Registros

- Un *registro* es una agrupación de datos, los cuales no tienen por qué ser del mismo tipo
- Se definen con la palabra **struct**:

```
struct Alumno{ // Define un nuevo tipo de dato Alumno
    int dni;
    float nota;
};
```

- Para acceder a sus campos se debe indicar el nombre de la variable y del campo, separados por un punto:

```
Alumno a,b;
a.dni=123133; // Asignación de datos a un campo
b=a; // Asignación de un registro completo bit a bit
```

Tipos enumerados

- Los *tipos enumerados* pueden declararse con un conjunto de posibles valores (*enumeradores*):

```
// Creamos un nuevo tipo de dato color
enum color{black,blue,green,red}; // Cuatro enumeradores
```

- Las variables de este tipo pueden tomar cualquier valor de entre estos enumeradores:

```
color myColor=blue;
if(myColor==green) {
    cout << "Green!" << endl;
}
```

- Los valores de los tipos enumerados se convierten internamente en `int` y viceversa:

```
enum animal{cat,dog,monkey,fish};
cout << monkey << endl; // Mostrará 2 por pantalla
// Es la posición que ocupa monkey en los enumeradores
```

Vectores STL (1/2)

- La *Standard Template Library* (STL) es una librería de funciones para C++
- Proporciona diferentes estructuras de datos y algoritmos
- Incluye la clase `vector`, que permite almacenar elementos de cualquier tipo, como un vector normal, pero sin tener que preocuparnos del tamaño:

```
#include <vector> // Siempre que vayamos a usar vector
vector<int> vec; // Declara un vector de enteros
                  // No es necesario indicar su tamaño
```

- El tamaño inicial de un vector STL es 0 y crece de manera dinámica en función de las necesidades
- Para añadir elementos al final del vector usamos `push_back`:*

```
vec.push_back(12); // Añade 12 al final del vector
vec.push_back(8); // Añade 8 detrás del 12
```

*Al ser una clase, sus métodos se invocan poniendo un punto tras el nombre de la variable

Vectores STL (2/2)

- Acceso a elementos mediante el operador []:

```
vec[10]=23; // Igual que un vector convencional  
cout << vec[8] << endl;
```

- Con size obtenemos el número de elementos del vector:

```
// Recorremos todos los elementos del vector  
for(unsigned int i=0;i<vec.size();i++) {  
    vec[i]=10;  
}
```

- Con clear podemos borrar todos los elementos y con erase uno en concreto:

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

- Existen muchas otras funciones para trabajar con vectores STL*

*Más información en <http://www.cplusplus.com/reference/vector/vector/>

Argumentos del programa (1/4)

- Los *argumentos* de un programa se usan para proporcionarle información (normalmente opciones) desde línea de comandos
- Su uso es muy habitual y permite modificar el comportamiento del programa:

Terminal

```
$ ls           // Muestra los ficheros de un directorio  
$ ls -a       // Muestra también los ficheros ocultos (opción "-a")  
$ ls -a -l   // Añade información extra de cada fichero (opción "-l")
```

Argumentos del programa (2/4)

- El `main` es una función y como tal puede recibir dos parámetros: `argc` y `argv`
- Estos parámetros permiten gestionar el paso de argumentos por línea de comandos a nuestro programa:

```
// Siempre en este orden
int main(int argc,char *argv[]){
    ...
    return 0;
}
```

- `int argc`: número de argumentos pasados al programa (contando también el nombre del programa)
- `char *argv[]`: vector de cadenas de caracteres con los argumentos pasados al programa

Argumentos del programa (3/4)

- Ejemplo de uso:

```
int main(int argc, char *argv[]){
    for(int i=0;i<argc;i++){
        cout << "Arg. " << i << " : " << argv[i] << endl;
    }
}
```

Terminal

```
$ ./miPrograma -a -h X // Ejemplo de llamada con tres parámetros
Arg. 0 : ./miPrograma
Arg. 1 : -a
Arg. 2 : -h
Arg. 3 : X
```

- Los argumentos no tienen por qué empezar con un guión (-) pero es una práctica bastante habitual

Argumentos del programa (4/4)

- Parece fácil gestionar los argumentos del programa, pero a veces puede ser complicado
- El usuario no siempre usa el mismo orden a la hora de introducir los argumentos:

Terminal

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

- Puede haber errores en la introducción y hay que mostrar mensajes de ayuda al usuario
- Es recomendable usar una función aparte para gestionar los argumentos

Depuración

Depuración de código en C++ (1/3)

- Cuando hay un error en tiempo de ejecución en nuestro código es difícil a veces localizar en qué punto está el fallo
- Un *depurador (debugger)* es un programa que nos ayuda a encontrar y corregir errores de ejecución en el código (*bugs*)

9/9

0800 Auton startd
1000 - stopped - auton ✓ { 1.2700 9.032 847 025
13°uc (03) HP - MC 1.30476415 9.037 846 995 connect
033 PRO 2 2. 13.047 6415
connect 2.13.047 6415
Relays 6-2 in 033 failed special speed test
in relays " 11.000 test -
1100 Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

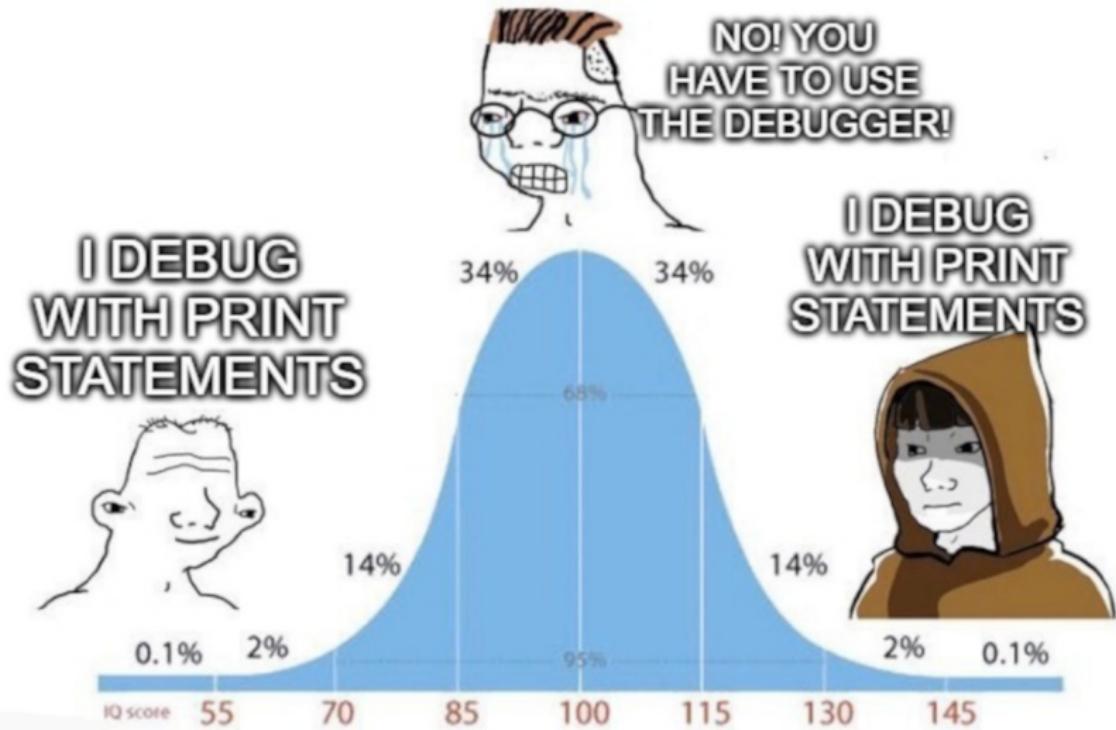
1545  Relay #70 Panel F
(Moth) in relay.
First actual case of bug being found.
1650 Autostart startd.
1700 clear down.

Relay 2145
Relay 3371

Depuración de código en C++ (2/3)

- Un depurador permite, por ejemplo, ejecutar el código línea a línea o ver qué valores tienen las variables en un determinado punto de ejecución
- Existen numerosos programas que facilitan la tarea de localizar errores en el código:
 - *GDB*: 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
 - *Valgrind*: 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.)
 - Otros ejemplos en Linux: *DDD*, *Nemiver*, *Electric Fence* y *DUMA*

Depuración de código en C++ (3/3)



Ejercicios

Ejercicios

Ejercicio 1

Implementa un programa que contenga una función con el siguiente prototipo: `int primeNumber(int n)`. Esta función devolverá el enésimo número primo. El programa debe imprimir números primos por pantalla con las siguientes opciones:

- `-L` imprimir cada número en una línea distinta (por defecto se imprimen todos en la misma línea)
- `-N n` imprimir los `n` primeros números primos (por defecto 10)

Ejemplos de ejecución:

Terminal

```
$ primes -N 5
1 2 3 5 7
$ primes -N -L 5
Error: primes [-L] [-N n]
```



@prog2ua

Tema 2: La clase string

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2024-2025



Índice

1. Cadenas de caracteres en C
2. La clase `string` en C++
3. Conversiones de tipos
4. Comparativa
5. Ejercicios

Cadenas de caracteres en C

Declaración (1/3)

- Las *cadenas de caracteres* son vectores que contienen una secuencia de tipo `char` terminada en el carácter nulo ('`\0`'):

```
// El compilador mete el '\0' al final automáticamente
char cad[]="holá";
// Otra forma de inicializar, carácter a carácter
char cad[]={ 'h', 'o', 'l', 'a', '\0' };
// Falta el '\0': no es una cadena de caracteres válida
char cad[]={ 'h', 'o', 'l', 'a' };
```

- Muchas de las funciones* que trabajan con cadenas buscan el '`\0`' para saber dónde termina la cadena
- Si no tenemos el '`\0`' puede que el resultado de estas funciones no sea el esperado

*Como aquellas que pertenecen a la librería `cstring` y que veremos más adelante

Declaración (2/3)

- Las cadenas de caracteres en C tienen tamaño fijo y una vez declaradas no pueden cambiar de tamaño:

```
char cad[10]; // Almacena como máximo 10 elementos
```

- Hay que tener en cuenta que se debe reservar siempre un espacio para almacenar el carácter nulo ('\0'):

```
char cad[10]; // Almacena como máximo 9 letras y el '\0'
```

- Se pueden inicializar al declararlas, en cuyo caso no hace falta poner el tamaño:

```
char cad[]="hola"; // Tamaño 5 (4 letras + '\0')
char cad2[10]="hola"; // Tamaño 10, aunque solo ocupa 5
```

- Las cadenas de caracteres en C se pueden usar también en C++

Declaración (3/3)

- Errores comunes al declarar cadenas de caracteres:

```
// El vector es demasiado pequeño para guardar la cadena
char cad[5]="paralelepipedo"; // Error de compilación

// Se usan comillas simples ('') en lugar de dobles ("")
char cad[]='h'; // Error de compilación
char cad[]='hola'; // Error de compilación

// No se pone el tamaño y no se inicializa
char cad[]; // Error de compilación

// Se intenta asignar valor con '=' después de declarar
char cad[10];
cad="hola"; // Error de compilación
```

Salida por pantalla

- Salida por pantalla con `cout` y `cerr` como el resto de tipos simples (`int`, `float`, etc.)
- Podemos combinar en la salida variables, constantes y datos de distinto tipo:

```
char cad[]="Nota";
int num=10;

cout << cad << " -> " << num; // Muestra "Nota -> 10"
```

Entrada por teclado > Operador >> (1/2)

- Podemos leer una cadena de caracteres desde teclado como con otros tipos simples, utilizando `cin` y el operador `>>`
- Existen algunas diferencias a la hora de leer desde teclado con respecto a otros tipos de datos
- Ignora los blancos* antes de la cadena:

```
char cad[32];
cin >> cad;
// El usuario escribe "    hola"
// La variable cad almacena "hola"
```

*Entendemos por "blanco" un espacio, tabulador o salto de línea ('`\n`')

Entrada por teclado > Operador >> (2/2)

- Termina de leer en cuanto encuentra el primer blanco en la cadena. **No nos permite leer entera una cadena que contenga blancos:**

```
char cad[32];
cin >> cad;
// El usuario escribe "buenas tardes"
// La variable cad almacena "buenas"
```

- No limita el número de caracteres que se leen. **El usuario puede escribir una cadena más grande de lo que admite el vector:**

```
char cad[5];
cin >> cad;
// El usuario escribe "esternocleidomastoideo"
// Puede invadir zonas de memoria que no debería y
// producirse un fallo de segmentación
```

Entrada por teclado > getline (1/4)

- También podemos leer una cadena de caracteres de teclado mediante `cin` y la función `getline`
- Esta función permite leer cadenas con blancos y limitar el número de caracteres leídos:

```
const int TAM=100;
char cad[TAM];
// cad: variable donde almacenamos la cadena
// TAM: número de caracteres a leer
cin.getline(cad,TAM);
// Si el usuario introduce "buenas tardes"
// en cad se almacena "buenas tardes"
```

- 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 guarda en la cadena
- La función añade '`\0`' al final de lo que ha leído (por eso sólo lee `TAM-1` caracteres)

Entrada por teclado > getline (2/4)

- Si el usuario introduce más caracteres de los que caben, estos se quedan en el *buffer* de teclado y la siguiente lectura falla:

```
char cad[10];
cout << "Cadena 1: ";
cin.getline(cad,10);
cout << "Leido 1: " << cad << endl;
cout << "Cadena 2: ";
cin.getline(cad,10);
cout << "Leido 2: " << cad << endl;
```

Terminal

```
$ miPrograma
Cadena 1: hola a todo el mundo
Leido 1: hola a to
Cadena 2: Leido 2:
```

Entrada por teclado > getline (3/4)

- Pueden haber problemas cuando leemos de `cin` combinando el operador `>>` y la función `getline`:

```
int num;
char cad[100];

cout << "Num: ";
cin >> num;
cout << "Escribe una cadena: " ;
cin.getline(cad,100);
cout << "Lo que he leido es: " << cad << endl;
```

Terminal

```
$ miPrograma
Num: 10
Escribe una cadena: Lo que he leido es:
```

Entrada por teclado > getline (4/4)

- ¿Por qué sucede esto?
 - Con el operador `>>` se lee 10, pero se deja de leer cuando se encuentra el primer carácter no numérico ('`\n`' en este caso)
 - Lo primero que encuentra en el *buffer* la función `getline` es un '`\n`', por lo que termina de leer y no guarda nada en `cad`
- Solución:

```
...
cin >> num;
cin.ignore(); // Añadimos esta línea
              // Saca el '\n' del buffer
// Ya se puede usar getline sin problema
...
```

La librería `cstring` (1/3)

- La librería `cstring` contiene una serie de funciones que facilitan el trabajo con cadenas de caracteres
- Para poder utilizarla hay que incluir la librería en el código:

```
#include <cstring>
```

- `strlen` devuelve la longitud (número de caracteres) de una cadena:

```
char cad[10] = "adios";
cout << strlen(cad); // Imprime 5
```

- `strcpy` copia una cadena en otra. Hay que llevar cuidado de no superar el tamaño del vector de destino:

```
char cad[5];
strcpy(cad, "hola"); // Cabe: 4 + '\0' = 5 caracteres
strcpy(cad, "adios") // No cabe!! Violación de segmento
```

La librería `cstring` (2/3)

- `strcmp` compara dos cadenas en orden lexicográfico*, devolviendo 1 si `cad1>cad2`, 0 si `cad1==cad2` y -1 si `cad1<cad2`:

```
char cad1[]="adios";
char cad2[]="adeu";
cout << strcmp(cad1,cad2) << endl; // Imprime 1
cout << strcmp(cad2,cad1) << endl; // Imprime -1
cout << strcmp(cad1,cad1) << endl; // Imprime 0
```

- `strcat` añade el contenido de una cadena al final de otra. **Debe haber suficiente espacio en la cadena destino:**

```
char cad[10]="hola";
strcat(cad, ", mu"); // En total 9 caracteres (cabe)
strcat(cad, "ndo"); // Añade 3 más (¡ya no cabe!)
```

*Orden que siguen las palabras en un diccionario

La librería `cstring` (3/3)

- Las funciones `strncpy`, `strncpy` y `strncat` comparan, copian o concatenan sólo los `n` primeros caracteres:

```
char cad[8];
strncpy(cad,"hola, mundo",4); // Solo copia "hola"
cad[4]='\0'; // No añade el '\0' de manera automática
               // Lo hemos de añadir nosotros al final
```

```
char cad1[8]="adios";
char cad2[8]="adeu";
// Solo compara los dos primeros caracteres
cout << strncmp(cad1,cad2,2) << endl; // Imprime 0
```

```
char cad1[50]="Hola, ";
char cad2[]="mundo maravilloso";
strncat(cad1,cad2,5); // cad1 valdrá "Hola, mundo"
```

Conversión a int y float

- Para pasar una cadena de caracteres a `int` o `float` se pueden usar las funciones `atoi` o `atof`
- Estas funciones pertenecen a la librería `cstdlib`:

```
#include <cstdlib> // Siempre que se vayan a usar

char cad[]="100";
int num=atoi(cad); // num vale 100

char cad2[]="10.5";
float num2=atof(cad2); // num2 vale 10.5
```

La clase string en C++

Definición (1/2)

- En C++ se pueden usar las cadenas de caracteres en C, pero además cuenta con la clase* `string` que permite trabajar de manera más cómoda y flexible con cadenas de caracteres:

```
// Declaración de una variable de tipo string
string s; // No hay que indicar el tamaño de la cadena
// Declaración con inicialización
string s2="Alicante";
// Declaración de una constante
const string SALUDO="hola";
```

*Más información sobre lo que es una “clase” en el Tema 5

Definición (2/2)

- Un `string` tiene tamaño variable y puede crecer en función de las necesidades de almacenamiento del programa:

```
string s="hola"; // Almacena 4 caracteres
s="hola a todo el mundo"; // Almacena 20 caracteres*
s="ok"; // Almacena 2 caracteres
```

- No es necesario preocuparse del '`\0`'
- El paso de parámetros (valor y referencia) se hace como con cualquier tipo simple:

```
void miFuncion(string s1,string &s2) {
    // s1 se pasa por valor
    // s2 se pasa por referencia
}
```

*Un espacio en blanco cuenta como un carácter más

Salida por pantalla

- Salida por pantalla con `cout` y `cerr` igual que con los vectores de caracteres en C:

```
string s="Nota";
int num=10;

cout << s << " -> " << num; // Muestra "Nota -> 10"
```

Entrada por teclado > Operador >>

- Se puede leer de teclado con `cin` y el operador `>>` de la misma forma que con vectores de caracteres en C
- Ignora los blancos antes de la cadena y termina de leer cuando encuentra el primer blanco:

```
string s;
cin >> s;
// El usuario escribe "      hola"
// La variable s almacena "hola"
...
// El usuario escribe "buenas tardes"
// La variable s almacena "buenas"
```

Entrada por teclado > getline (1/2)

- Al igual que con los vectores de caracteres en C, podemos usar la función `getline` para leer cadenas
- Permite leer cadenas que contengan blancos:

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

- No limita el número de caracteres que se leen, ya que con un `string` no es necesario
- ¡Ojo! Cambia la sintaxis con respecto a los vectores de caracteres en C

Entrada por teclado > getline (2/2)

- Si combinamos lecturas con el operador `>>` y `getline` tenemos el mismo problema que con los vectores de caracteres en C*
- Por defecto, `getline` lee hasta que encuentra el carácter salto de línea (`'\n'`)
- Podemos pasárle un parámetro adicional para indicar que lea hasta un determinado carácter:

```
string s;
// Lee hasta que encuentra la primera coma
getline(cin,s,',');
// Lee hasta que encuentra el primer corchete
getline(cin,s,['']);
```

*La solución es la misma que en la transparencia 11

Extraer palabras de un string

- Se pueden extraer palabras fácilmente de un string usando la clase `stringstream`:

```
#include <sstream> // Necesario si se usa stringstream  
...  
stringstream ss("Hola mundo cruel 666");  
string s;  
  
// En cada iteración del bucle lee hasta encontrar blanco  
while(ss>>s){ // Extraemos las palabras una a una  
    cout << "Palabra: " << s << endl;  
}
```

Métodos de string (1/3)

- Al ser una clase, los métodos se invocan poniendo un punto tras el nombre de la variable
- `length` devuelve el número de caracteres de la cadena:

```
// unsigned int length()
string s="hola, mundo";
cout << s.length(); // Imprime 11
```

- `find` devuelve la posición en la aparece una subcadena dentro de una cadena:

```
// size_t find(const string &s,unsigned int pos=0)
cout << s.find("mundo"); // Imprime 6
// Si no encuentra la subcadena devuelve string::npos
```

Métodos de string (2/3)

- replace sustituye una cadena (o parte de ella) por otra:

```
// string& replace(unsigned int pos,unsigned int tam,
                  const string &s)
string s="hola mundo";
s.replace(0,4,"hello"); // s vale "hello mundo"
```

- erase permite eliminar parte de una cadena:

```
// string& erase(unsigned int pos=0,unsigned int tam=
                  string::npos);
string cad="hola mundo";
cad.erase(4,3); // cad vale "holando"
```

- substr devuelve una subcadena de la cadena original:

```
// string substr(unsigned int pos=0,unsigned int tam=
                  string::npos) const;
string cad="hola mundo";
string subcad=cad.substr(2,5); // subcad vale "la mu"
```

Métodos de string (3/3)

- Ejemplo de uso:

```
string a="Hay una taza en esta cocina con tazas";
string b="taza";
unsigned int tam=a.length(); // Longitud de a
// Buscamos la primera palabra "taza"
size_t encontrado=a.find(b);
if(encontrado!=string::npos) {
    cout << "Primera en: " << encontrado << endl;
    // Buscamos la segunda palabra "taza"
    encontrado=a.find(b,encontrado+b.length());
    if(encontrado!=string::npos)
        cout << "Segunda en: " << encontrado << endl;
}
else{
    cout << "Palabra '" << b << "'" no encontrada";
}
// Sustituimos la primera "taza" por "botella"
a.replace(a.find(b),b.length(),"botella");
cout << a << endl;
```

Operadores (1/2)

- Comparaciones: == (igual), != (distinto), > (mayor estricto), >= (mayor o igual), < (menor estricto) y <= (menor o igual)

```
string s1,s2;
cin >> s1; cin >> s2;
if(s1==s2) // La comparación es en orden lexicográfico
    cout << "Son iguales" << endl;
```

- Asignación de una cadena a otra con el operador =, como cualquier tipo simple:

```
string s1="hola";
string s2;
s2=s1;
```

- Concatenación de cadenas con el operador +:

```
string s1="hola";
string s2="mundo";
string s3=s1+" "+s2; // s3 vale "hola, mundo"
```

Operadores (2/2)

- Acceso a componentes como si fuera un vector de caracteres en C, con el operador []:

```
string s="hola";
char c=s[3]; // s[3] vale 'a'
s[0] = 'H';
cout << s << ":" << c << endl ; // Imprime "Hola:a"
```

- No se pueden asignar caracteres a posiciones que no pertenecen al string:

```
string s;
s[0]='h'; s[1]='o'; s[2]='l'; s[3]='a';
// No almacena nada, porque s es una cadena vacía y esas
    posiciones no las tiene reservadas
```

- Ejemplo de recorrido de un string carácter a carácter:

```
string s="hola, mundo";
for(unsigned int i=0;i<s.length();i++)
    s[i]='f'; // Sustituye cada carácter por 'f'
```

Conversiones de tipos

Conversión entre string y vector de caracteres en C

- Para asignar un vector de caracteres en C a `string` se utiliza el operador de asignación (=):

```
char cad[]="holo";
string s;
s=cad;
```

- Para asignar un `string` a un vector de caracteres en C hay que usar `strcpy` y `c_str`:*

```
char cad[10];
string s="mundo";
// Debe haber espacio suficiente en cad
strcpy(cad,s.c_str());
```

*El método `c_str` devuelve un vector de caracteres en C con el contenido del `string`

Conversión entre string y número

- Convertir un número entero o real a string:

```
#include <string> // ¡Ojo! No es lo mismo que <cstring>
...
int num=100;
string s=to_string(num);
```

- Convertir un string a número entero:^{*}

```
string s="100";
int num=stoi(s);
```

- Convertir un string a número real:

```
string s="10.5";
float num=stof(s);
```

^{*}to_string, stoi y stof están disponibles a partir de la versión 2011 de C++

Comparativa

Vector de caracteres en C vs. string

Vector de caracteres en C	string
char cad[TAM]; char cad[]="hola";	string s; string s="hola";
strlen(cad) cin.getline(cad,TAM); if(!strcmp(cad1,cad2)){...} strcpy(cad1,cad2); strcat(cad1,cad2);	s.length() getline(cin,s); if(s1==s2){...} s1=s2; s1=s1+s2;
strcpy(cad,s.c_str());	s=cad;
Terminan con '\0' Tamaño reservado fijo Tamaño ocupado variable	No terminan con '\0' El tamaño reservado puede variar Tamaño ocupado == tamaño reservado
Se usan con ficheros binarios	No se pueden usar con ficheros binarios

Ejercicios

Ejercicios (1/4)

Ejercicio 1

Diseña una función `subCadena` que devuelva la subcadena de longitud `n` que empieza en la posición `p` de otra cadena. Tanto el argumento como el valor de retorno deben ser de tipo `string`.

```
subCadena ("hooooola", 2, 5) // Devuelve "la"
```

Ejercicio 2

Diseña una función `borraCaracterCadena` que, dados un `string` y un carácter, borre todas las apariciones del carácter en el `string` y lo devuelva.

```
borrarCaracterCadena ("cocobongo", 'o') // Devuelve "ccbng"
```

Ejercicios (2/4)

Ejercicio 3

Diseña una función `buscarSubcadena` que busque la primera aparición de una subcadena `a` dentro de una cadena `b` y devuelva su posición, o `-1` si no está. Tanto `a` como `b` deben ser de tipo `string`.

```
buscarSubcadena("ooool", "hooooola") // Devuelve 2
```

Ampliaciones:

1. Añadir otro parámetro a la función que indique el número de aparición (si vale 1 sería como la función original)
2. Crear otra función que devuelva el número de apariciones de la subcadena en la cadena

Ejercicio 4

Diseña una función `codifica` que codifique una cadena sumando una cantidad `n` al código ASCII de cada carácter, pero teniendo en cuenta que el resultado debe ser una letra.

Por ejemplo, si `n=3`, la `a` se codifica como `d`, la `b` como `e`,..., la `x` como `a`, la `y` como `b`, y la `z` como `c`.

La función debe admitir letras mayúsculas y minúsculas. Los caracteres que no sean letras no se deben codificar. El argumento debe ser de tipo `string`.

```
codifica("hola, mundo", 3) // Devuelve "krod, pxqgr"
```

Ejercicios (4/4)

Ejercicio 5

Diseña una función `esPalindromo` que devuelva `true` si el string que se le pasa como parámetro es palíndromo.

```
esPalindromo("larutanatural") // Devuelve true  
esPalindromo("hola, aloh") // Devuelve false
```

Ejercicio 6

Diseña una función `crearPalindromo` que añada a un string el mismo string invertido, de forma que el resultado sea un palíndromo.

```
crearPalindromo("hola") // Devuelve "holaaloh"
```



@prog2ua

Tema 3: Ficheros

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2024-2025



Índice

1. Introducción
2. Ficheros de texto
3. Ficheros binarios

Introducción

Qué es un fichero (1/3)

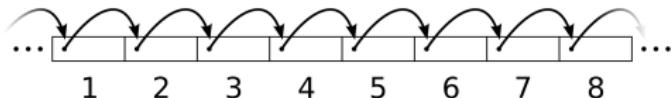
- Todos los datos con los que hemos trabajado hasta ahora se almacenan en la memoria principal del ordenador (*RAM*)
- El tamaño de la memoria principal es bastante limitado (unos pocos Gigabytes)
- Todos los datos se borran cuando el programa termina (*memoria volátil*)

Qué es un fichero (2/3)

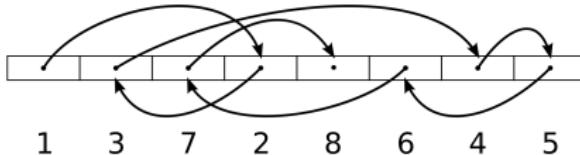
- Los *ficheros* (o *archivos*) son la forma en la que C++ permite acceder a la información almacenada en disco (memoria secundaria)
- Los ficheros son estructuras dinámicas: su tamaño puede variar durante la ejecución del programa según los datos que almacena
- Existen dos tipos de ficheros, en función de cómo se guarda dentro la información: *ficheros de texto* y *ficheros binarios*

Qué es un fichero (3/3)

- Hay dos formas de acceder a un fichero:
 - *Acceso secuencial*: leemos/escribimos los elementos del fichero en orden, empezando por el principio y uno detrás de otro



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



Ficheros de texto

Definición (1/2)

- Los ficheros de texto también se denominan *ficheros con formato*
- Guardan la información en forma de secuencias de caracteres, tal como se mostrarían por pantalla
- Por ejemplo, el valor entero 19 se guardará en fichero como los caracteres 1 y 9
- Ejemplos de ficheros de texto: un código fuente en C++, una página web (HTML) o un fichero creado con el bloc de notas
- El modo de lectura/escritura más habitual en ficheros de texto es el acceso secuencial

Definición (2/2)

- Son ficheros que contienen solamente caracteres imprimibles: aquellos cuyo código ASCII es mayor o igual que 32
- El código ASCII es un código que asigna a cada carácter un número para su almacenamiento en memoria:

Dec	Hex	Car									
0	00	NUL	32	20	SPC	64	40	�	96	60	'
1	01	SOH	33	21	!	65	41	A	97	61	a
2	02	STX	34	22	"	66	42	B	98	62	b
3	03	ETX	35	23	#	67	43	C	99	63	c
4	04	ECT	36	24	\$	68	44	D	100	64	d
5	05	ENQ	37	25	%	69	45	E	101	65	e
6	06	ACK	38	26	&	70	46	F	102	66	f
7	07	BEL	39	27	,	71	47	G	103	67	g
8	08	BS	40	28	(72	48	H	104	68	h
9	09	HT	41	29)	73	49	I	105	69	i
10	0A	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	SI	47	2F	/	79	4F	O	111	6F	o
16	10	DLE	48	30	0	80	50	P	112	70	p
17	11	DC1	49	31	1	81	51	Q	113	71	q
18	12	DC2	50	32	2	82	52	R	114	72	r
19	13	DC3	51	33	3	83	53	S	115	73	s
20	14	DC4	52	34	4	84	54	T	116	74	t
21	15	NAK	53	35	5	85	55	U	117	75	u
22	16	SYN	54	36	6	86	56	V	118	76	v
23	17	ETB	55	37	7	87	57	W	119	77	w
24	18	CAN	56	38	8	88	58	X	120	78	x
25	19	EN	57	39	9	89	59	Y	121	79	y
26	1A	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	ESC	59	3B	;	91	5B	[123	7B	{
28	1C	FS	60	3C	<	92	5C	\	124	7C	
29	1D	GS	61	3D	=	93	5D]	125	7D	}
30	1E	RS	62	3E	>	94	5E	^	126	7E	-
31	1F	US	63	3F	?	95	5F	_	127	7F	DEL

Declaración de variables

- Los ficheros son un tipo de dato más en C++
- Hay que incluir la librería `fstream` en nuestro código para poder trabajar con ellos:

```
#include <fstream>
```

- Existen tres tipos de datos básicos para trabajar con ficheros, dependiendo de lo que queramos hacer con ellos:^{*}

```
ifstream ficheroLec; // Leer de fichero  
ofstream ficheroEsc; // Escribir en fichero  
fstream ficheroLecEsc; // Leer y escribir en fichero
```

*Es poco habitual usar el tipo `fstream` con ficheros de texto

Apertura y cierre (1/4)

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

```
ifstream fichero; // Vamos a leer del fichero  
fichero.open("miFichero.txt");  
// Ahora ya podemos leer de "miFichero.txt"
```

- El nombre del fichero se puede pasar como un array de caracteres o como un `string`:*

```
char nombreFichero[]="miFichero.txt";  
fichero.open(nombreFichero);
```

*Solo se puede usar `string` a partir de la versión 2011 de C++

Apertura y cierre (2/4)

- A `open` se le puede pasar un segundo parámetro que indica el *modo de apertura* del fichero:
 - Lectura: `ios::in`
 - Escritura: `ios::out`
 - Lectura/escritura: `ios::in | ios::out`
 - Añadir al final: `ios::out | ios::app`

```
ifstream ficheroLec;
ofstream ficheroEsc;
// Abrimos solo para leer
ficheroLec.open("miFichero.txt",ios::in);
// Abrimos para añadir información al final
ficheroEsc.open("miFichero.txt",ios::out|ios::app);
```

Apertura y cierre (3/4)

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

Apertura y cierre (4/4)

- Por defecto, el tipo `ifstream` se abre para lectura y el `ofstream` para escritura
- Se puede abrir el fichero en el momento de declararlo:

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

- Antes de leer/escribir, se debe comprobar con `is_open` si el fichero se ha abierto correctamente (`true`) o no (`false`)
- Al terminar de usar el fichero, se debe liberar con `close`:

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

Lectura con el operador >> (1/3)

- La lectura de fichero permite recuperar información guardada en disco para ponerla en memoria y poder trabajar con ella
- Podemos utilizar el operador >> para leer de fichero igual que hacíamos con `cin` para leer de teclado
- Bucle para leer un fichero carácter a carácter:

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

Lectura con el operador >> (2/3)

- Al usar el operador `>>` descartamos los blancos, al igual que sucedía al leer de `cin`
- Podemos usar la función `get` para leer carácter a carácter sin descartar blancos:

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

Lectura con el operador >> (3/3)

- También se puede usar el operador >> para leer ficheros que contengan distintos tipos de datos
- Por ejemplo, si tenemos un fichero que contiene en cada línea una cadena y dos enteros (ej. Hola 1032 124):

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
    string s;
    int num1,num2;
    while(fl >> s){ // Lee el string
        fl >> num1; // Lee el primer número
        fl >> num2; // Lee el segundo número
        cout << s << "," << num1 << "," << num2 << endl;
    }
    fl.close()
}
...
```

Lectura por líneas

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

```
ifstream fl("miFichero.txt");
if(fl.is_open()){
    string s;
    while(getline(fi,s)){
        cout << s << endl;
    }
    fl.close();
}
else{
    cout << "Error al abrir el fichero" << endl;
}
```

Detección de final de fichero

- El método `eof` nos indica si se ha alcanzado el final de fichero
- Esta circunstancia se da cuando no quedan más datos por leer:

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

- Cuando se intenta leer datos que quedan fuera del fichero, el método devuelve `true`
- Despues de haber leido el ultimo dato valido el metodo sigue devolviendo `false`
- Es necesario hacer una lectura mas para provocar que `eof` devuelva `true`

Escritura con el operador <<

- Podemos utilizar el operador << para escribir en fichero igual que hacíamos con cout para escribir por pantalla:

```
ofstream fe("miFichero.txt");
if(fe.is_open()){
    int num=10;
    string s="Hola, mundo";
    fe << "Un numero entero: " << num << endl;
    fe << "Un string: " << s << endl;
    fe.close();
}
else{
    cout << "Error al abrir el fichero" << endl;
}
```

Ejercicios (1/6)

Ejercicio 1

Implementa un programa que lea un fichero `fichero.txt` e imprima por pantalla las líneas del fichero que contienen la cadena `Hola`.

Ejercicios (2/6)

Ejercicio 2

Haz un programa que lea un fichero `fichero.txt` y escriba en otro fichero `FICHERO.TXT` el contenido del fichero de entrada con todas las letras en mayúsculas.

Ejemplo:

<code>fichero.txt</code>	<code>FICHERO.TXT</code>
Hola, mundo.	HOLA, MUNDO.
Como estamos?	COMO ESTAMOS?
Adios, adios...	ADIOS, ADIOS...

Ejercicios (3/6)

Ejercicio 3

Haz un programa que lea dos ficheros de texto, f1.txt y f2.txt, y escriba por pantalla las líneas que sean distintas en cada fichero, con < delante si la línea corresponde a f1.txt y con > si corresponde a f2.txt.

Ejemplo:

f1.txt	f2.txt
hola, mundo. como estamos? adios, adios...	hola, mundo. como vamos? adios, adios...

La salida debe ser:

```
< como estamos?  
> como vamos?
```

Ejercicios (4/6)

Ejercicio 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 debe mostrar por pantalla las `n` últimas líneas del fichero.

Ejemplo:

```
finFichero(3, "cadenas.txt")
```

```
with several words  
unapalabra  
muuuuchas palabras, muchas, muchas...
```

Ejercicio 4 (continuación)

Hay dos soluciones:

1. A lo bestia: leer el fichero para contar las líneas que tiene y volver a leer el fichero para escribir las n líneas finales.
Problema: ¿y si el fichero tiene 100000000000000 de líneas?
2. Utilizar un array de `string` de tamaño n que almacene en todo momento las n últimas líneas leídas (aunque al principio tendrá menos de n líneas)

Ejercicios (6/6)

Ejercicio 5

Tenemos dos ficheros de texto, f1.txt y f2.txt, en los que cada línea es una serie de números separados por :. Cada línea está ordenada 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 um fichero f3.txt las líneas comunes a ambos ficheros.

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 (1/2)

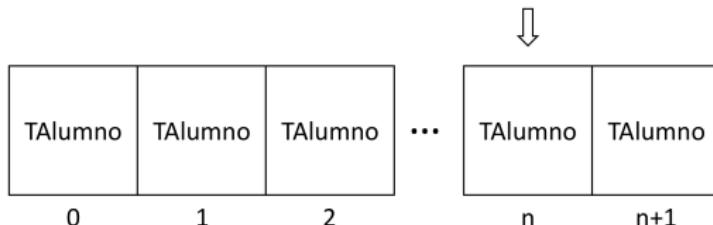
- También se le denominan *ficheros sin formato*
- Guardan la información tal y como se almacena en la memoria principal del ordenador
- Por ejemplo, el valor entero 19 se guardará en fichero como la secuencia 00010011
- Para leer y escribir se utilizan funciones diferentes a las de los ficheros de texto
- Con los ficheros binarios se suele emplear tanto acceso secuencial como directo
- Las lecturas y escrituras son más rápidas que con los ficheros de texto (no hay que convertir a carácter)
- Generalmente ocupan menos espacio en disco que sus equivalentes ficheros de texto

Definición (2/2)

- Es muy habitual que cada elemento del que se quiere guardar información se almacene en un *registro* (*struct*):

```
struct TAlumno{  
    char nombre[100];  
    int grupo;  
    float notaMedia;  
};
```

- Mediante acceso directo, se puede acceder directamente al elemento n del fichero sin tener que leer los $n-1$ anteriores



Declaración de variables

- Se declaran igual que en los ficheros de texto:

```
#include <fstream> // Siempre que se trabaja con ficheros  
  
ifstream ficheroLec; // Leer de fichero  
ofstream ficheroEsc; // Escribir en fichero  
fstream ficheroLecEsc; // Leer y escribir en fichero
```

Apertura y cierre

- Al abrir el fichero se debe indicar que es binario mediante el modo de apertura `ios::binary`:
 - Lectura: `ios::in | ios::binary`
 - Escritura: `ios::out | ios::binary`
 - Lectura/escritura: `ios::in | ios::out | ios::binary`
 - Añadir al final: `ios::out | ios::app | ios::binary`

```
ifstream ficheroLec;
ofstream ficheroEsc;
// Abrimos solo para leer en modo binario
ficheroLec.open("miFichero.dat",ios::in | ios::binary);
// Abrimos solo para escribir en modo binario
ficheroEsc.open("miFichero.dat",ios::out | ios::binary);
// Forma abreviada
fstream ficheroLecEsc("miFichero.dat",ios::binary)
```

- Igual que con los ficheros de texto, se puede comprobar si está abierto con `is_open` y se cierra con `close`

Lectura (1/3)

- Para leer de fichero binario utilizamos la función `read`
- Esta función recibe dos parámetros: el primero indica dónde se guardará la información leída de fichero y el segundo la cantidad de información (número de bytes) que se va a leer:^{*}

```
TAlumno alumno;
ifstream fichero;

fichero.open("miFichero.dat",ios::in | ios::binary);
if(fichero.is_open()){
    // En cada iteración leemos un registro TAlumno
    while (fichero.read((char *)&alumno, sizeof(TAlumno))){
        // Mostramos el nombre y la nota de cada alumno
        cout << alumno.nombre << ":" << alumno.nota << endl;
    }
    fichero.close();
}
```

*Para saber el número de bytes que ocupa una variable se puede usar la función `sizeof`

Lectura (2/3)

- Se puede leer directamente un elemento n del fichero sin tener que leer los $n-1$ anteriores (acceso directo)
- La función `seekg` permite situarse en un punto específico del fichero
- Recibe dos parámetros: el primero indica cuántos bytes nos queremos saltar, mientras que el segundo indica el punto de referencia para hacer ese salto

```
// Tenemos un fichero con registros de tipo TAlumno
ifstream fichero("miFichero.dat",ios::binary);
TAlumno alumno;
...
// Podemos leer directamente el tercer registro
// Nos saltamos los dos primeros registros
fichero.seekg(2*sizeof(TAlumno),ios::beg);
// Ya podemos leer el tercer registro
fichero.read((char *)&alumno,sizeof(alumno));
...
```

Lectura (3/3)

- Puntos de referencia posibles:
 - ios::beg: contando desde el principio del fichero
 - ios::cur: contando desde la posición actual
 - ios::end: contando desde el final del fichero
- Si el primer parámetro de seekg es un número negativo, la ventana de lectura se mueve hacia el principio del fichero:

```
ifstream fichero("miFichero.dat",ios::binary);
TAlumno alumno;
...
fichero.seekg(-1*sizeof(TAlumno),ios::end);
// Leemos el último registro del fichero
fichero.read((char *)&alumno,sizeof(TAlumno));
...
```

Escritura (1/3)

- Para escribir en fichero binario utilizamos la función `write`
- Esta función recibe dos parámetros: el primero indica dónde está guardada la información que queremos escribir y el segundo la cantidad de información (número de bytes) que se van a escribir
- La sintaxis es muy parecida a la de `read`:

```
ofstream fichero("miFichero.dat", ios::binary);
TAlumno alumno;

if(fichero.is_open())
{
    strcpy(alumno.nombre, "Pepe Pi");
    alumno.notaMedia=7.8;
    alumno.grupo=5;

    fichero.write((const char *)&alumno, sizeof(TAlumno));
    fichero.close();
}
```

Escritura (2/3)

- Al igual que para la lectura, se puede escribir directamente en el registro n sin tener que escribir los $n-1$ anteriores
- La función `seekp` permite posicionarse para escritura (`seekg` es para lectura)
- Los parámetros son los mismos que para `seekg`:

```
ofstream fichero("miFichero.dat",ios::binary);
TAlumno alumno;
...
// Nos posicionamos para escribir el tercer registro
fichero.seekp(2*sizeof(TAlumno),ios::beg);
fichero.write((const char *)&alumno,sizeof(TAlumno));
...
```

- Si la posición a la que se va con `seekp` no existe en el fichero, éste se "alarga" para que se pueda escribir en él

Escritura (3/3)

- Para almacenar cadenas de caracteres en un fichero binario se deben usar arrays de caracteres, nunca `string`
- El problema de `string` es que es un dato de tamaño variable, por lo que no podemos tener registros que tengan todos el mismo tamaño
- Puede ser necesario recortar la cadena para que quepa en el registro antes de guardarla en fichero:

```
const int TAM=20;
char cad[TAM];
string s;
...
strncpy(cad,s.c_str(),TAM-1); // Máximo 19 caracteres
cad[TAM-1]='\0';
```

Posición actual

- La posición actual (en bytes) de la ventana de lectura se puede obtener mediante la función `tellg` y la de escritura mediante `tellp`
- Se puede usar, por ejemplo, para calcular el número de registros de un fichero:

```
ifstream fichero("miFichero.dat",ios::binary);
// Colocamos la ventana de lectura al final del fichero
fichero.seekg(0,ios::end);
// Calculamos el numero de registros TAlumno del fichero
cout << fichero.tellg()/sizeof(TAlumno) << endl;
```

Ejercicios (1/3)

Ejercicio 6

Tenemos un fichero binario `alumnos.dat` que tiene registros de alumnos con la siguiente información:

- `dni`: array de 10 caracteres
- `apellidos`: array de 40 caracteres
- `nombre`: array de 20 caracteres
- `grupo`: entero

Haz un programa que imprima por pantalla el DNI de todos los alumnos del grupo 7.

Ampliación: haz un programa que intercambie los alumnos de los grupos 4 y 8 (los grupos van del 1 al 10).

Ejercicios (2/3)

Ejercicio 7

Dado el fichero `alumnos.dat` del ejercicio anterior, haz un programa que pase a mayúsculas el nombre y los apellidos del quinto alumno del fichero, volviéndolo a escribir en él.

Ejercicio 8

Diseña un programa que cree el fichero `alumnos.dat` a partir de un fichero de texto `alu.txt` en el que cada dato (dni, apellidos, 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 deberán recortar.

Ejercicios (3/3)

Ejercicio 9

Escribe un programa que se encargue de la asignación automática de alumnos en 10 grupos de prácticas. A cada alumno se le asignará el grupo correspondiente al último número de su DNI (a los alumnos con DNI acabado en 0 se les asignará el grupo 10). Los datos de los alumnos están en un fichero `alumnos.dat` con la misma estructura que en los ejercicios anteriores.

La asignación de grupos 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 grupo que le corresponde y se guardará el registro en la misma posición.



@prog2ua

Tema 4: Memoria dinámica

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2024-2025



Índice

1. Organización de la memoria
2. Punteros
3. Uso de punteros
4. Referencias
5. Implementación de una pila

Organización de la memoria

Memoria estática

- Los *datos estáticos* son aquellos cuyo tamaño es fijo y se conoce al escribir el programa
- Las variables que hemos usado hasta ahora son estáticas:

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

i	c	vf[0]	vf[1]	vf[2]
0		1.0	2.0	3.0

1000 1002 1004 1006 1008

Memoria dinámica

- Permite almacenar grandes volúmenes de datos, cuya cantidad exacta se desconoce al implementar el programa
- Durante la ejecución del programa se ajusta el uso de la memoria a lo que necesita en cada momento
- En C++ se puede hacer uso de la memoria dinámica usando punteros

Zonas de la memoria

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

Pila (<i>stack</i>)	<ul style="list-style-type: none">La <i>pila</i> almacena los datos locales de una función: parámetros por valor y variables locales
Montículo (<i>heap</i>)	<ul style="list-style-type: none">El <i>montículo</i> almacena los datos dinámicos que se van reservando durante la ejecución del programa
Segmento de datos	<ul style="list-style-type: none">El <i>segmento de datos</i> se almacenan los datos de estos tipos, cuyo tamaño se conoce en tiempo de compilación
Código del programa	<ul style="list-style-type: none">El propio código también se almacena en la memoria, como los datos

Punteros

Definición y declaración

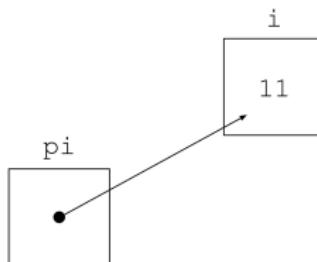
- Un *puntero* almacena la dirección de memoria donde se encuentra otro dato
- Se dice que el puntero “apunta” a ese dato
- Los punteros se declaran usando el carácter *
- El dato al que apunta el puntero será de un tipo concreto que deberá indicarse al declarar el puntero:

```
int *punteroEntero; // Puntero a entero
char *punteroChar; // Puntero a carácter
int *vecPunterosEntero[20]; // Array de punteros a entero
double **doblePunteroReal; // Puntero a puntero a real
```

Operadores de punteros (1/2)

- El operador `*` permite acceder al contenido de la variable a la que apunta el puntero
- El operador `&` permite obtener la dirección de memoria en la que está almacenada una variable:

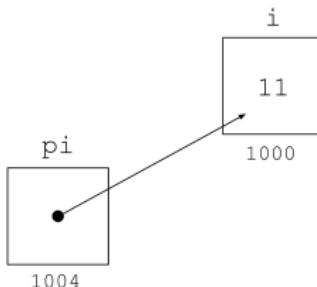
```
int i=3;
int *pi;
pi=&i; // pi contiene la dirección de memoria de i
*pi=11; // Contenido de pi es "11". Por lo tanto i = 11
```



Operadores de punteros (2/2)

- Suponiendo que `i` está en la posición de memoria 1000 y que `pi` está en la 1004:

```
int i=11;
int *pi;
pi=&i;
cout << pi << endl; // Muestra "1000"
cout << *pi << endl; // Muestra "11"
cout << &pi << endl; // Muestra "1004"
```



Declaración con inicialización

- Como cualquier otra variable, podemos inicializar un puntero en el momento de su declaración:

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

- Cuando queremos indicar que un puntero no apunta a ningún dato válido le asignamos el valor `NULL`:

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

- `NULL` es una constante entera con valor cero. A partir del estándar C++ 2011 se puede usar la constante `nullptr` que representa el cero como una dirección de memoria (tipo puntero)

Ejercicios

Ejercicio 1

Indica cuál sería la salida por pantalla de estos 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 (1/2)

- El operador `new` permite reservar memoria de manera dinámica durante la ejecución del programa
- Devuelve la dirección de inicio de la memoria reservada
- Si no hay suficiente memoria para la reserva, devuelve `NULL`
- Se debe usar un puntero para almacenar la dirección que devuelve `new`:

```
double *pd;  
pd=new double; // Reserva memoria para un double  
if(pd!=NULL) { // Comprueba que se ha podido reservar  
    *pd=4.75;  
    cout << *pd << endl; // Muestra "4.75"  
}
```

pd				
2000			4.75	

1000 1002 ... 2000 2002

Reserva y liberación de memoria (2/2)

- El operador `delete` permite liberar memoria reservada con `new`:

```
double *pd;  
pd=new double; // Reserva memoria  
...  
delete pd; // Libera la memoria apuntada por pd  
pd=NULL; // Conveniente si vamos a seguir usando pd
```

- Siempre que se reserva con `new` hay que liberar con `delete`
- Un puntero se puede reutilizar tras liberar su contenido y reservar memoria otra vez con `new`:

```
double *pd;  
pd=new double; // Reserva memoria  
...  
delete pd; // Libera la memoria apuntada por pd  
pd=new double; // Reservamos de nuevo memoria  
...
```

Punteros y arrays (1/3)

- Existe una estrecha relación entre los punteros y los arrays
- La variable de tipo array es en realidad un puntero al primer elemento del array:

```
int vec[5]={4,5,2,8,12};  
cout << vec << endl; // Muestra la dirección de memoria  
                      // del primer elemento del array  
cout << *vec << endl; // Muestra "4"
```

- Siempre apunta al primer elemento del array y no se puede modificar

Punteros y arrays (2/3)

- Los punteros se pueden usar como accesos directos a componentes de arrays:

```
int vec[20];
int *pVec=vec; // Ambos son punteros a entero
*pVec=58; // Equivalente a vec[0]=58;
pVec=&(vec[7]);
*pVec=117; // Equivalente a vec[7]=117;
```

Punteros y arrays (3/3)

- Los punteros también pueden usarse para crear *arrays dinámicos*
- Para reservar memoria para un array dinámico hay que usar corchetes y especificar el tamaño
- Para liberar toda la memoria reservada es necesario también usar corchetes (vacíos):

```
int *pv;  
pv=new int[10]; // Reserva memoria para 10 enteros  
pv[0]=585; // Accedemos como en un array estático  
...  
delete [] pv; // Liberamos toda la memoria reservada
```

Punteros definidos con `typedef`

- Como vimos en el *Tema 1*, se pueden definir nuevos tipos de datos con `typedef`:

```
typedef int entero;  
entero a,b; // Equivalente a int a,b;
```

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

```
typedef int *tPunteroEntero;  
tPunteroEntero pi; // Variable de tipo puntero a entero  
// No hay que poner * al declararla
```

Punteros a registros

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

```
struct TRegistro{  
    char c;  
    int i;  
};  
typedef TRegistro *TPunteroRegistro;  
  
TPunteroRegistro pr;  
pr=new TRegistro;  
pr->c='a'; // Equivalente a (*pr).c='a';  
pr->i=88; // Equivalente a (*pr).i=88;
```

Punteros como parámetros de funciones (1/2)

- Un puntero, como cualquier otra variable, se puede pasar como parámetro por valor o por referencia a una función:

```
void funcValor(int *p){ // Paso por valor
    ...
    p=NULL;
}

void funcReferencia(int *&p){ // Paso por referencia
    ...
    p=NULL;
}

int main(){
    int i=0;
    int *p=&i;
    funcValor(p);
    // p sigue apuntando a i
    funcReferencia(p);
    // p vale NULL
}
```

Punteros como parámetros de funciones (2/2)

- El mismo ejemplo de antes usando `typedef`:

```
typedef int* tPunteroEntero;
void funcValor(tPunteroEntero p) {
    ...
    p=NULL;
}
void funcReferencia(tPunteroEntero &p) {
    ...
    p=NULL;
}
int main(){
    int i=0;
    tPunteroEntero p=&i;
    funcValor(p);
    funcReferencia(p);
}
```

Errores comunes (1/2)

- No liberar la memoria reservada dinámicamente:

```
void func() {
    int *pEntero=new int;
    *pEntero=8;
    return; // ;Error! Falta delete pEntero;
}
```

- Utilizar un puntero que no apunta a nada:

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

Errores comunes (2/2)

- Usar un puntero tras haberlo liberado:

```
int *p,*q;  
p=new int;  
...  
q=p;  
delete p;  
*q=7; // ;Error! La memoria ya se había liberado
```

- Liberar memoria no reservada con new:

```
int *pEntero=&i;  
delete pEntero; // ;Error! Apunta a memoria estática
```

Ejercicio 2

Dado el siguiente registro:

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

Realiza 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.

Referencias

Referencias (1/4)

- Las referencias de C++ son como punteros pero con una sintaxis menos recargada (*azúcar sintáctica*)
- No hay nada que puedas hacer con referencias que no puedas hacer con punteros

```
int a=10;
int *b=&a; // Variable puntero
*b=20;
cout << a << " " << *b; // Muestra "20 20"
int &c=a; // Variable referencia
c=30;
cout << a << " " << c; // Muestra "30 30"
```

- En el código anterior, `c` se puede considerar como un segundo nombre para `a`

Referencias (2/4)

- Las referencias no pueden ser `NULL`, siempre están conectadas a un dato
- Una vez se ha inicializado una referencia, no se puede hacer que se refiera a una posición de memoria diferente, pero esto sí es posible con punteros
- Al crear una referencia siempre hay que inicializarla, pero los punteros se pueden inicializar en cualquier momento tras su declaración

Referencias (3/4)

- Las referencias simplifican el código de las funciones que tienen parámetros pasados por referencia
- La siguiente función usa punteros para pasar dos parámetros por referencia:

```
void swap(int *x,int *y){  
    int temp=*x;  
    *x=*y;  
    *y=temp;  
}  
  
int main(){  
    int a=10,b=20;  
    swap(&a,&b);  
    cout << a << " " << b; // Muestra "20 10"  
}
```

Referencias (4/4)

- La siguiente función es equivalente a la anterior, pero usa referencias en lugar de punteros:

```
void swap(int &x,int &y){  
    int temp=x;  
    x=y;  
    y=temp;  
}  
  
int main(){  
    int a=10,b=20;  
    swap(a,b);  
    cout << a << " " << b; // Muestra "20 10"  
}
```

- Esta es la sintaxis que hemos estado usando en la asignatura
- Es más sencilla y cómoda que la del ejemplo anterior

Implementación de una pila

Implementación de una pila (1/6)

- Una *pila* es una estructura de datos muy usada en programación
- Consiste en una lista de elementos
- Se puede añadir o eliminar elementos a una pila con una restricción: el último elemento añadido (*push*) será el primer elemento en ser sacado (*pop*)
- Ejemplos de pila en el mundo real
 - En una pila de platos, el plato que está encima y que acaba de ser apilado siempre será el primero en ser desapilado
 - Los carritos de la compra del supermercado, donde siempre se coge el último que hayan dejado

Implementación de una pila (2/6)

- Una pila puede implementarse usando vectores de tamaño fijo, pero esto limitará el número de elementos que pueden apilarse
- Podemos solucionarlo (parcialmente) usando un vector muy grande, pero si apilamos pocos elementos estaremos desperdiciando memoria
- Usar punteros para implementar la pila permite usar solo la memoria que necesitemos en cada momento
- Se puede implementar usando la idea de *lista enlazada*
 - Al apilar un nuevo elemento se reserva dinámicamente espacio en memoria para un registro
 - Este registro contiene los datos a guardar y un puntero al último elemento de la pila
 - Tendremos un puntero (llamado `head`) que siempre apuntará a la cima de la pila

Implementación de una pila (3/6)

- En la siguiente implementación el puntero `head` se pasa como parámetro a las diferentes funciones
- Se pasa por referencia cuando alguna de estas funciones puede cambiar el puntero para que apunte a otro registro
- Estructura de un elemento (nodo) de la pila:

```
struct Node{  
    int data; // Información que queremos almacenar  
    struct Node *next; // Puntero al siguiente elemento  
};
```

Implementación de una pila (4/6)

- Funciones para apilar (push) y desapilar (pop) elementos:

```
void push(Node *&head, int newData){  
    Node *newNode=new Node; // Reservamos memoria  
    newNode->data=newData; // Guardamos los datos  
    newNode->next=head; // Apuntamos al último nodo  
    head=newNode; // head apunta al nuevo nodo  
}  
  
void pop(Node *&head) {  
    Node *ptr;  
    if(head!=NULL){ // Nos aseguramos que hay elementos  
        ptr=head->next; // Segundo elemento de la pila  
        delete head; // Borramos la cima  
        head=ptr; // head apunta ahora al segundo elemento  
    }  
}
```

Implementación de una pila (5/6)

- Funciones para mostrar (`display`) y vaciar (`destroy`) la pila:

```
void display(Node *head) {
    Node *ptr;
    ptr=head;
    while(ptr!=NULL){ // Hasta recorrer toda la pila
        cout << ptr->data << " "; // Mostramos los datos
        ptr=ptr->next; // Pasamos al siguiente elemento
    }
}

void destroy(Node *&head) {
    Node *ptr,*ptr2;
    ptr=head;
    while(ptr!=NULL){ // Hasta recorrer toda la pila
        ptr2=ptr; // Vamos a eliminar el nodo actual
        ptr=ptr->next; // Apuntamos al siguiente elemento
        delete ptr2; // Borramos el nodo actual
    }
    head=NULL; // La pila ya está vacía
}
```

Implementación de una pila (6/6)

- Ejemplo de función principal usando dos pilas:

```
int main() {
    // Declaramos e inicializamos las dos pilas
    Node *head1=NULL;
    Node *head2=NULL;
    // Añadimos tres elementos a la primera pila
    push(head1,3);
    push(head1,1);
    push(head1,7);
    display(head1); // Muestra "7"
    pop(head1); // Eliminamos la cima
    display(head1); // Muestra "1"
    destroy(head1); // Vaciamos la primera pila
    // Añadimos un elemento a la segunda pila
    push(head2,9);
    display(head2); // Muestra "9"
    destroy(head2); // Vaciamos la segunda pila
}
```



@prog2ua

Tema 5: Introducción a la programación orientada a objetos

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2024-2025



Índice

1. Introducción
2. Conceptos básicos
3. POO en C++
4. Objetos y gestión de memoria
5. Relaciones
6. Compilación
7. Ejercicios

Introducción

Definición

- La *programación orientada a objetos* (POO) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas informáticos
- La aplicación entera se reduce a un conjunto de objetos y sus relaciones
- C++ es un lenguaje orientado a objetos, aunque también permite programación imperativa (*procedimental*)
- Cambia el enfoque a la hora de diseñar los programas...
- ... ¡pero todo lo que has aprendido hasta ahora te sigue valiendo!

Clases y objetos (1/4)

- En Programación 2 ya hemos usado clases y objetos:

```
int i; // Declaramos una variable i de tipo int  
string s; // Declaramos un objeto s de clase string
```

- Una *clase* (o tipo compuesto) es un modelo para crear objetos de esa clase
- Un *objeto* de una determinada clase se denomina una *instancia* de la clase
- En el ejemplo anterior, s es una instancia/objeto de la clase string
- Las clases son similares a los tipos simples, aunque permiten muchas más funcionalidades

Clases y objetos (2/4)

- Un registro o `struct` es un tipo simple
- Se puede considerar como una clase “ligera” que sólo almacena datos visibles desde fuera:

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

Clases y objetos (3/4)

- Una clase contiene datos y una serie de funciones que manipulan esos datos, llamadas *funciones miembro* o *métodos*
- Se puede controlar qué datos/métodos son visibles (`public`) y cuáles están ocultos (`private`)
- Las funciones miembro pueden acceder a los datos públicos y privados de su clase
- Clase “cutre” equivalente a `struct Fecha`:*

```
class Fecha{  
    public: // Datos públicos  
        int dia;  
        int mes;  
        int año;  
};
```

*Decimos que es “cutre” porque no ofrece ninguna ventaja con respecto a `struct Fecha`

Clases y objetos (4/4)

- Acceso directo a elementos del objeto, como en un registro:

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

- En un buen diseño orientado a objetos, normalmente no se accede directamente a los datos: para modificar los datos se usan métodos
- En el ejemplo anterior, `f.dia=100` no daría error
- Con métodos podemos controlar qué valores se dan a los datos:

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

Conceptos básicos

Conceptos básicos

- Principios en los que se basa el diseño orientado a objetos:
 - Abstracción
 - Encapsulación
 - Modularidad
 - Herencia
 - Polimorfismo

Abstracción

- La *abstracción* denota las características esenciales de un objeto y su comportamiento
- Cada objeto puede realizar tareas, informar y cambiar su estado, comunicándose con otros objetos en el sistema sin revelar cómo se implementan estas características
- El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevas clases
- El proceso de abstracción tiene lugar en la fase de diseño

Encapsulación

- La *encapsulación* significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad al mismo nivel de abstracción
- La *interfaz* es la parte del objeto que es visible (pública) para el resto de los objetos: conjunto de métodos y datos de los cuales disponemos para comunicarnos con un objeto
- Cada objeto oculta su implementación (cómo lo hace) y expone una interfaz (qué hace)
- La encapsulación protege a las propiedades de un objeto contra su modificación: solamente los propios métodos del objeto pueden acceder a su estado

Modularidad (1/2)

- Se denomina *modularidad* a la propiedad que permite subdividir una aplicación en partes más pequeñas (*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 también pueden compartir módulo

Modularidad (2/2)

- Una clase `miClase` se implementaría con dos ficheros fuente:
 - `miClase.h`: contiene constantes que se usen en este fichero, la declaración de la clase y la de sus métodos
 - `miClase.cc`: contiene constantes que se usen en este fichero, la implementación de los métodos y puede que tipos internos que use la clase
- El programa principal (`main`) usa y comunica las clases
- Se incluirá en un fichero aparte (por ejemplo, `prog.cc`)
- Para compilar todos los módulos y obtener un único ejecutable:

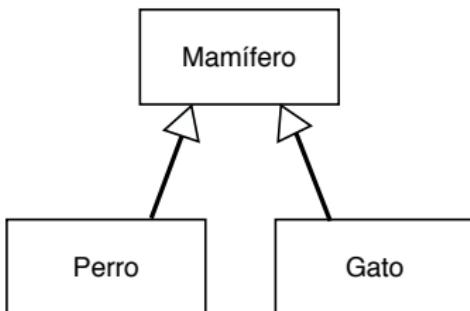
Terminal

```
$ g++ miClase1.cc miClase2.cc prog.cc -o prog
```

- Este método es adecuado sólo si tenemos pocas clases (en este ejemplo habría dos: `miClase1` y `miClase2`)
- Al final del tema veremos cómo compilar adecuadamente programas con múltiples clases utilizando la herramienta *make*

Herencia (1/2)

- No la vamos a trabajar en Programación 2
- Las clases se pueden relacionar entre sí formando una jerarquía de clasificación
- La *herencia* permite definir una nueva clase a partir de otra
- Se aplica cuando hay suficientes similitudes y la mayoría de las características de la clase existente son adecuadas para la nueva clase
- En este ejemplo, las *subclases* Perro y Gato heredan los métodos y atributos especificados por la *superclase* Mamífero:



Herencia (2/2)

- La herencia nos permite adoptar características ya implementadas por otras clases
- Facilita la organización de la información en diferentes niveles de abstracción
- Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen
- 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*

Polimorfismo

- No lo vamos a trabajar en Programación 2
- El *polimorfismo* es la propiedad según la cual una misma expresión hace referencia a distintas acciones
- Por ejemplo, un método `desplazar` puede referirse a acciones distintas si se trata de un avión o de un coche
- Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre
- Las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos:

```
Mamifero *a=new Perro;  
Mamifero *b=new Gato;  
Mamifero *c=new Gaviota;
```

POO en C++

Declaración e implementación (1/2)

- La clase SpaceShip se implementará como un módulo usando dos ficheros: SpaceShip.h y SpaceShip.cc

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

Declaración e implementación (2/2)

```
// SpaceShip.cc (implementación de los métodos)
#include "SpaceShip.h"

SpaceShip::SpaceShip(int ms, string nm){ // Constructor
    maxSpeed=ms;
    name=nm;
}

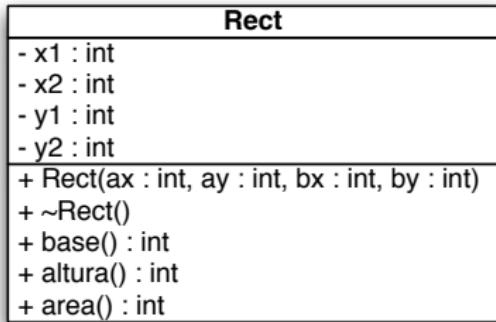
SpaceShip::~SpaceShip(){} // Destructor

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

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

Diagrama UML (1/3)

- Un *diagrama UML* permiten describir las clases y relaciones entre clases en un diseño orientado a objetos:



- El – delante de un atributo o método indica que es privado
- El + indica que es un atributo o método público
- La línea horizontal separa los atributos (parte superior) de los métodos (parte inferior)

Diagrama UML (2/3)

- Traducción a código del diagrama UML anterior:

```
// Rect.h (declaración 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();
};
```

Diagrama UML (3/3)

```
// Rect.cc (implementación de los métodos)
Rect::Rect(int ax,int ay,int bx,int by) {
    x1=ax;
    y1=ay;
    x2=bx;
    y2=by;
}
Rect::~Rect(){}
int Rect::base(){ return (x2-x1); }
int Rect::altura(){ return (y2-y1); }
int Rect::area(){ return base()*altura(); }
```

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

Accesores

- No es conveniente acceder directamente a los datos miembro de una clase (principio de encapsulación)
- Lo normal es definirlos como `private` y acceder a ellos implementando métodos `set/get/is` (llamados **accesores**):

Fecha	
-	dia : int
-	mes : int
-	anyo : int
+	getDia () : int
+	getMes () : int
+	getAnyo() : int
+	setDia (d : int) : void
+	setMes (m : int) : void
+	setAnyo (a : int) : void
+	isBisiesto () : bool

- Los accesores `set` nos permiten controlar que los valores de los atributos sean correctos

Forma canónica

- Todas las clases deben implementar estos cuatro métodos:
 - Constructor
 - Destructor
 - Constructor de copia
 - Operador de asignación
- Si alguno no ha sido definido en la clase, el compilador lo crea por defecto

Constructor (1/7)

- El *constructor* se invoca automáticamente cuando se crea un objeto de la clase
- Las clases deben tener al menos un método constructor
- Si no definimos un constructor, el compilador creará uno por defecto sin parámetros (los datos miembros de los objetos creados así estarán sin inicializar)
- Una clase puede tener varios constructores con parámetros distintos (el constructor puede *sobrecargarse*)
- La sobrecarga es un tipo de polimorfismo

Constructor (2/7)

- Ejemplos de constructor:

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

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

- Llamadas al constructor:

```
Fecha f;
Fecha f(10,2,2010);
Fecha f(); // ;Error de compilación!
```

Constructor (3/7)

- Los constructores (al igual que otras funciones) pueden tener parámetros por defecto
- Estos valores por defecto sólo se ponen en el fichero de cabecera (.h):

```
// Fecha.h
class Fecha{
    ...
    Fecha(int d=1,int m=1,int a=1900);
    ...
}
```

- Con este constructor podríamos crear objetos de varias formas:

```
Fecha f; // dia = 1, mes = 1, anyo = 1900
Fecha f(10,2,2010); // dia = 10, mes = 2, anyo = 2010
Fecha f(10); // dia = 10, mes = 1, anyo = 1900
Fecha f(18,5); // dia = 18, mes = 5, anyo = 1900
```

Constructor (4/7)

- Los parámetros por defecto del ejemplo anterior se mostrarían de la siguiente manera en un diagrama UML:

Fecha
- dia: int
- mes: int
- anyo: int
+ Fecha (dia: int=1, mes: int=1, anyo: int=1900)
...

Constructor (5/7)

- Si los parámetros que se le pasan al constructor son incorrectos no debería de crearse el objeto
- Esto se puede controlar mediante el uso de *excepciones*:
 - Podemos lanzar una excepción con `throw` para indicar que se ha producido un error
 - Podemos capturar una excepción con `try/catch` para reaccionar ante el error
- Si se produce una excepción y no la capturamos, el programa terminará inmediatamente
- Las excepciones sólo deben usarse cuando no hay otra opción (por ejemplo, en los constructores)

Constructor (6/7)

- Ejemplo de uso de excepciones:

```
int root(int n){  
    if(n<0)  
        throw exception(); // Lanza la excepción y termina  
    return sqrt(n);  
}  
  
int main(){  
    try{ // Intentamos ejecutar estas instrucciones  
        int result=root(-1); // Provoca una excepción  
        cout << result << endl; // Esta línea no se ejecuta  
    }  
    catch(...){ // Si hay una excepción la capturamos aquí  
        cout << "Negative number" << endl;  
    }  
}
```

Constructor (7/7)

- 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 llega a crearse
    }
    catch(...){
        cout << "Coordenada incorrecta" << endl;
    }
}
```

Destructor (1/2)

- El *destructor* de la clase debe liberar los recursos (normalmente memoria dinámica) que el objeto esté usando
- Una clase sólo tiene una función destructor que no tiene argumentos y no devuelve ningún valor
- Es un método con igual nombre que la clase y precedido por el carácter ~:

```
// Declaración
~Fecha();
// Implementación
Fecha::~Fecha() {
    // Liberar la memoria reservada (si fuera necesario)
}
```

Destructor (2/2)

- Todas las clases necesitan un destructor y si no se especifica, el compilador crea uno por defecto
- El compilador llama automáticamente al destructor del objeto cuando acaba su ámbito
- También se invoca al destructor al hacer `delete`
- El destructor de un objeto invoca implícitamente a los destructores de todos sus atributos

Constructor de copia (1/2)

- Un *constructor de copia* crea un objeto a partir de otro objeto existente:

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

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

Constructor de copia (2/2)

- El constructor de copia se invoca automáticamente cuando:
 - Una función devuelve un objeto
 - Se inicializa un objeto cuando se declara:

```
Fecha f2(f1); // Constructor de copia
Fecha f2=f1; // Constructor de copia
f2=f1; // Aquí no se invoca al constructor sino a =
```

- Un objeto se pasa por valor a una función:

```
void funcion(Fecha f);
funcion(f);
```

- Si no se especifica ningún constructor de copia, el compilador crea uno por defecto que hace una copia atributo a atributo del objeto

Operador de asignación

- No lo vamos a ver en Programación 2
- El *operador de asignación* (=) permite una asignación directa de dos objetos:

```
Fecha f1(10,2,2011); // Constructor  
Fecha f2; // Constructor  
f2=f1; // Operador de asignación
```

- Por defecto, el compilador crea un operador de asignación que copia atributo a atributo
- Podemos redefinirlo para nuestras clases si lo consideramos necesario

Declaraciones *inline* (1/2)

- Los métodos con poco código se pueden implementar directamente en la declaración de la clase (declaración *inline*):

```
// Rect.h
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();
};
```

Declaraciones *inline* (2/2)

- Es más eficiente declarar funciones *inline*
- Cuando se compila el código generado para las funciones *inline*, se inserta en el punto donde se invoca a la función (en lugar de hacerlo en otro lugar y hacer una llamada)
- Las funciones *inline* también se pueden implementar fuera de la declaración de clase, en el fichero .cc, usando la palabra reservada `inline`:

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

Métodos constantes (1/2)

- Los métodos que no modifican los atributos del objeto se pueden declarar como *métodos constantes*:

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

- En un objeto constante sólo se pueden invocar métodos constantes:

```
int Fecha::getDia(){ // No se ha declarado como const
    return dia;
}
int main(){
    const Fecha f(10,10,2011);
    cout << f.getDia() << endl; // Error de compilación
}
```

- Los métodos `get` deben declararse constantes, ya que se limitan a devolver valores y no modifican nunca al objeto

Métodos constantes (2/2)

- Se representan poniendo <<const>> delante del nombre del método en los diagramas UML
- En este ejemplo hay cuatro métodos constantes (getSubtotal, getCantidad, getPrecio y getDescription):

Línea
- cantidad: int - precio: float - descripcion: string
+ Linea() + <<const>> getSubtotal(): float + <<const>> getCantidad(): int + <<const>> getPrecio(): float + <<const>> getDescription(): string + setCantidad(cant: int): void + setPrecio(precio: float): void + setDescripcion(descripcion: string): void

Funciones amigas

- Una *función amiga* no pertenece a la clase pero puede acceder a su parte privada
- Se declara usando la palabra reservada `friend` en su declaración:

```
class MiClase{  
    friend void unaFuncionAmiga(int,MiClase &);  
    public:  
        ...  
    private:  
        int datoPrivado;  
};
```

```
void unaFuncionAmiga(int x,MiClase &c){  
    c.datoPrivado=x; // Correcto, porque es amiga  
}
```

Sobrecarga de la entrada/salida (1/4)

- Podemos sobrecargar las operaciones de entrada/salida de cualquier clase:

```
Fecha f;  
cin >> f;  
cout << f;
```

- El problema es que no pueden ser funciones miembro de una clase porque el primer operando (`cin/cout`) no es un objeto de esa clase (es un `stream`)
- Los operadores se sobrecargan usando funciones amigas:

```
friend ostream& operator<<(ostream &o, const Fecha &f);  
friend istream& operator>>(istream &i, Fecha &f);
```

Sobrecarga de la entrada/salida (2/4)

- Declaración:

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

Sobrecarga de la entrada/salida (3/4)

- Implementación:

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

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

Sobrecarga de la entrada/salida (4/4)

- En un diagrama UML se pondrá la palabra <<friend>> delante del operador, ya que se trata de una función amiga
- En este ejemplo, la clase tiene sobrecargado el operador de salida (`operator<<`):

Factura
<u>- nextId: int = 1</u>
<u>+ IVA: const int = 21</u>
- fecha: string
- id: int
<u>+ Factura(c: Cliente*, fecha: string)</u>
<u>+ anyadirLinea(cant: int, desc: string, prec: float): void</u>
<u>- getNextId(): int</u>
<u>+ <<friend>> operator<<: ostream &</u>

Atributos y métodos de clase (1/4)

- Los *atributos de clase* tienen el mismo valor para todos los objetos de la clase (son como variables globales para la clase)
- Los *métodos de clase* producen la misma salida para todos los objetos de la clase y sólo pueden acceder a atributos de clase
- También se llaman atributos y métodos **estáticos**
- Se declaran mediante la palabra reservada `static` al definir la clase:

```
class Fecha{
    public:
        static const int semanasPorAnyo=52;
        static const int diasPorSemana=7;
        static const int diasPorAnyo=365;
        static string getFormato();
        static bool setFormato(string);
    private:
        static string cadenaFormato;
};
```

Atributos y métodos de clase (2/4)

- En el fichero donde se implementan los métodos (.cc) no se debe poner la palabra `static` al definirlos
- Para la clase `Fecha` del ejemplo anterior, tendríamos el siguiente código:

```
// Fecha.cc
string Fecha::getFormato() { // No se pone static
    ...
}

bool Fecha::setFormato(string s) { // No se pone static
    ...
}
```

Atributos y métodos de clase (3/4)

- Se representan subrayados en los diagramas UML
- En este ejemplo hay dos atributos estáticos (IVA y nextId) y un método estático (getNextId):

Factura
<u>- nextId: int = 1</u>
<u>+ IVA: const int = 21</u>
<u>- fecha: string</u>
<u>- id: int</u>
<u>+ Factura(c: Cliente*, fecha: string)</u>
<u>+ anyadirLinea(cant: int, desc: string, prec: float): void</u>
<u>- getNextId(): int</u>
<u>+ <<friend>> operator<<: ostream &</u>

Atributos y métodos de clase (4/4)

- Si 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
class Fecha{
    ...
    static const string finMundo;
    ...
};
```

```
// Fecha.cc
const string Fecha::finMundo="2020"; // No se pone static
```

- Acceso a atributos o métodos estáticos desde fuera de la clase:

```
cout << Fecha::diasPorAnyo << endl; // Atributo estático
cout << Fecha::getFormato() << endl; // Método estático
```

El puntero this

- El puntero `this` es una pseudovariable que no se declara ni se puede modificar
- Es un argumento implícito que reciben todos los métodos (excluyendo los estáticos) y que apunta al objeto receptor del mensaje
- Es necesario cuando queremos desambiguar el nombre del parámetro o cuando queremos pasar como argumento el objeto a una función anidada:

```
void Fecha::setDia(int dia){  
    // dia=dia; Ojo: le asigna a dia su propio valor  
    this->dia=dia;  
    cout << this->dia << endl;  
}
```

Objetos y gestión de memoria

Objetos automáticos y objetos dinámicos (1/5)

- Teniendo en cuenta su permanencia en la memoria, los objetos pueden ser *automáticos* o *dinámicos*
- Los objetos *dinámicos* se crean en tiempo de ejecución utilizando el operador `new`
- Permanecen en memoria (generalmente en *heap*) hasta que se eliminan explícitamente a través del operador `delete`*
- Los objetos *automáticos* se crean (automáticamente) en la memoria (generalmente en *stack*) en tiempo de ejecución cuando se entra en su ámbito
- Se destruyen (automáticamente) cuando se sale de dicho ámbito

*Todos estos conceptos fueron descritos en el Tema 4

Objetos automáticos y objetos dinámicos (2/5)

- Ejemplo de creación de un objeto automático y de uno dinámico:

```
void func() {
    Fecha f1; // Objeto automático
    Fecha *f2=NULL;
    f2=new Fecha; // Objeto dinámico
}
int main(){
    func();
    ...
}
```

- Al terminar la función `func` el objeto `f1` ya no está en memoria, pero el objeto apuntado por `f2` sí, aunque ya no es accesible
- Problema: el espacio de memoria al que apunta `f2` no se borra y ocupa memoria hasta que el programa acabe

Objetos automáticos y objetos dinámicos (3/5)

- En el ejemplo anterior, podemos devolver el puntero para que el objeto apuntado por f2 se pueda seguir usando:

```
Fecha* func(){ // func devuelve un puntero a Fecha
    Fecha f1; // Objeto automático
    f1.setDia(10);
    Fecha *f2=NULL;
    f2=new Fecha; // Objeto dinámico
    f2->setDia(20);
    return f2;
}
int main(){
    Fecha *f=NULL;
    f=func();
    cout << f->getDia(); // Muestra "20"
    f->setMes(1);
    delete f; // Objeto dinámico borrado de memoria
}
```

Objetos automáticos y objetos dinámicos (4/5)

- Así quedaría el ejemplo anterior si devolvíéramos un objeto automático en lugar de uno dinámico:

```
Fecha func() {  
    Fecha f1; // Objeto automático  
    f1.setDia(10);  
    return f1; // Llama al constructor de copia  
}  
int main(){  
    Fecha f=func(); // Objeto automático  
    cout << f.getDia(); // Muestra "10"  
    f.setDia(1);  
}
```

- El constructor de copia será invocado en este caso para inicializar el objeto `f` del `main`

Objetos automáticos y objetos dinámicos (5/5)

- Asignación de objetos automáticos y dinámicos:

```
Fecha f1, f2, *f3=new Fecha, *f4=new Fecha;  
f2.setDia(15);  
f3->setDia(10);  
f4->setDia(20);  
f1=f2; // Se llama al operador de asignación  
Fecha f5=f1; // Llamada al constructor de copia  
f3=f4; // El objeto asignado originalmente a f3 ya  
        // no se puede borrar  
        // f3 y f4 apuntan ahora al mismo objeto  
cout << f3->getDia(); // Muestra "20"  
delete f3;  
f4->setDia(5); // Igual de incorrecto que f3->setDia(5);  
                // f3 y f4 apuntan a memoria no válida
```

- La última línea es incorrecta, pero podría llegar a funcionar
- Al eliminar un objeto se marcan sus posiciones de memoria para ser reasignadas (no se pone justo en ese momento la memoria a cero o se reasigna a nuevos datos)

Copia profunda y copia superficial

- Dada una clase A con un campo B que es un objeto dinámico:

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

- Decimos que el constructor de copia de A realiza una *copia profunda* del objeto si reserva nueva memoria para él:

```
A::A(const A &a) {  
    ...  
    b=new B(a.b);  
}
```

- Decimos que el constructor de copia de A realiza una *copia superficial* del objeto si hace que el nuevo puntero apunte a donde el anterior:

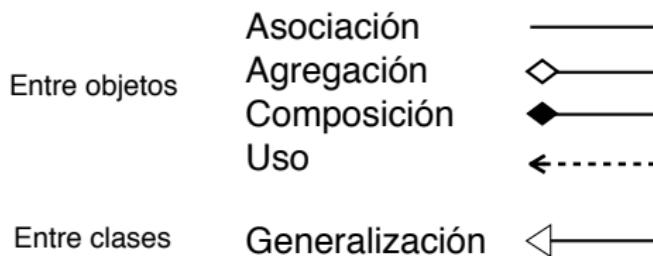
```
A::A(const A &a) {  
    ...  
    b=a.b;  
}
```

- Las dos opciones pueden ser convenientes según el problema

Relaciones

Relaciones entre objetos

- Principales tipos de relaciones entre objetos y clases:



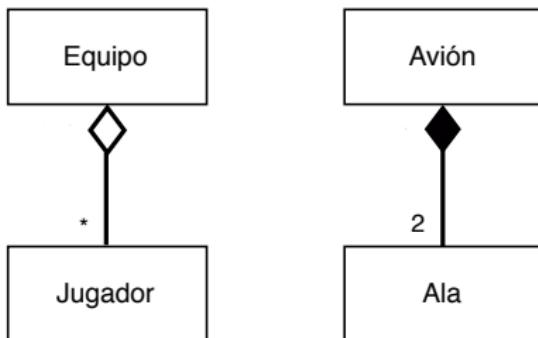
- La mayoría de las relaciones posee cardinalidad:
 - Uno o más: $1..*$ ($1..n$)
 - Cero o más: *
 - Número fijo: m
- En Programación 2 vamos a trabajar solo la agregación y la composición

Agregación y composición (1/6)

- *Agregación* y *composición* son relaciones todo-parte en las que un objeto forma parte de la naturaleza de otro
- Son relaciones asimétricas
- La diferencia entre agregación y composición es la fuerza de la relación: la agregación es una relación más débil que la composición

Agregación y composición (2/6)

- En la composición, cuando se destruye el objeto contenedor también se destruyen los objetos que contiene
 - Ej: el ala forma parte del avión y no tiene sentido fuera del mismo (si vendemos un avión, lo hacemos incluyendo sus alas)
- En el caso de la agregación, no ocurre así
 - Ej: podemos vender un equipo, pero los jugadores pueden irse a otro club (no desaparecen con el equipo)

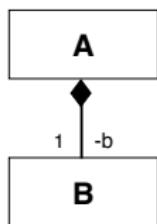


Agregación y composición (3/6)

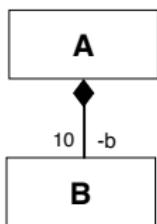
- Algunas relaciones pueden ser consideradas como agregaciones o composiciones en función del contexto en que se utilicen
 - Ej: la relación entre bicicleta y rueda
- Algunos autores consideran que la única diferencia entre ambos conceptos radica en su implementación: una composición sería una “agregación por valor”

Agregación y composición (4/6)

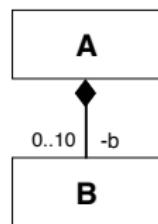
- Implementación de la composición:



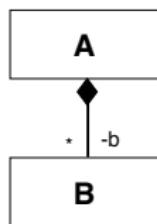
```
class A {  
private:  
    B b;  
...  
};
```



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



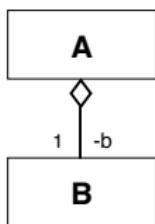
```
class A {  
private:  
    vector<B> b;  
    static const int N=10;  
...  
};
```



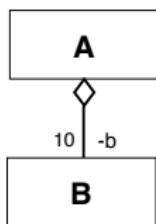
```
class A {  
private:  
    vector<B> b;  
...  
};
```

Agregación y composición (5/6)

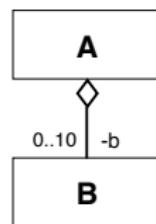
- Implementación de la agregación:



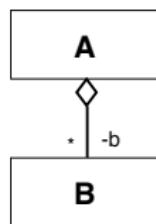
```
class A {  
private:  
    B *b;  
...  
};
```



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



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



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

Agregación y composición (6/6)

- Ejemplo de implementación de la agregación:

```
class A{
    private:
        B *b;
    public:
        A(B *b){ this->b=b; } // Constructor
};
```

```
int main(){ // Dos formas de implementar la agregación
    // 1- Mediante un puntero
    B *b=new B;
    A a(b); // Llamada al constructor
    // 2- Mediante un objeto
    B b;
    A a(&b); // Llamada al constructor
}
```

Compilación

El proceso de compilación

- La tarea de traducir un programa fuente en ejecutable se realiza en dos fases:
 - *Compilación*: el compilador traduce un programa fuente en un programa en código objeto (no ejecutable)
 - *Enlace*: el enlazador (*linker*) junta el programa en código objeto con las librerías del lenguaje (C/C++) y genera el ejecutable
- En C++ se realizan las dos fases con la siguiente instrucción:

Terminal

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

- Con la opción `-c` sólo se compila, generando código objeto (`.o`), pero sin hacer el enlace:

Terminal

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

Compilación separada (1/2)

- Cuando un programa se compone de varios ficheros fuente (.cc), lo que debe hacerse para obtener el ejecutable es:
 1. Compilar cada fuente por separado, obteniendo varios ficheros en código objeto (.o):

Terminal

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

2. Enlazar los ficheros en código objeto con las librerías del lenguaje y generar un ejecutable:

Terminal

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

- Si tiene pocos ficheros fuente, se puede hacer todo de una vez:

Terminal

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

Compilación separada (2/2)

- Problema: tenemos un fichero de cabecera `.h` que se usa en varios ficheros fuente `.cc`
- ¿Qué hay que hacer si se cambia algo en el `.h`?
 - Opción 1: lo recompilo todo (a lo “bestia”)
 - Opción 2: busco “a mano” dónde se usa y sólo recompilo esas clases
 - Opción 3: busco automáticamente dónde se usa y sólo recompilo esas clases
- La mejor es la “Opción 3” y hay un programa llamado *make* que nos ayuda a hacerlo

La herramienta *make* (1/6)

- La herramienta *make* ayuda a compilar programas grandes
- Permite establecer *dependencias* entre ficheros
- Compila un fichero cuando alguno de los ficheros de los que depende cambia
- El fichero de texto *makefile* especifica las dependencias entre los ficheros y qué hacer cuando algo cambia

La herramienta *make* (2/6)

- La herramienta *make* busca por defecto un fichero llamado *makefile*
- En este fichero se describe un *objetivo* principal (normalmente el programa ejecutable) y una serie de objetivos secundarios
- El formato de cada objetivo del fichero *makefile* es:

```
<objetivo> : <dependencias>
[tabulador]<instrucción>
```

- El algoritmo del programa *make* es muy sencillo: “*Si la fecha de alguna dependencia es más reciente que la del objetivo, ejecutar instrucción*”

La herramienta *make* (3/6)

- Imaginemos que tenemos los siguientes ficheros:

```
// C1.cc
#include "C1.h"
...
```

```
// C2.cc
#include "C2.h"
#include "C1.h"
...
```

```
// prog.cc
#include "C1.h"
#include "C2.h"
...
int main() {
...
}
```

La herramienta *make* (4/6)

- El fichero makefile sería:^{*}

```
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
```

*La opción `-Wall` muestra todos los *warnings* y `-g` añade información para el depurador

La herramienta *make* (5/6)

- En el ejemplo anterior, si se modifica C2.cc y se ejecuta *make*:

Terminal

```
$ make
g++ -Wall -g -c C2.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

- Y si se modifica C2.h y se ejecuta *make*:

Terminal

```
$ make
g++ -Wall -g -c C2.cc
g++ -Wall -g -c prog.cc
g++ -Wall -g C1.o C2.o prog.o -o prog
```

La herramienta *make* (6/6)

- Ejemplo anterior usando constantes (más “profesional”):*

```
CC = g++
```

```
CFLAGS = -Wall -g
```

```
OBJS = C1.o C2.o prog.o
```

```
prog : $(OBJS)
```

```
    $(CC) $(CFLAGS) $(OBJS) -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 $(OBJS)
```

*Más información en: <http://es.wikipedia.org/wiki/Make>

Directivas del preprocesador

- Se pueden producir errores de compilación cuando un fichero de cabecera se incluye en múltiples ficheros de nuestro código
- El compilador piensa que se está declarando múltiples veces la clase que hay en ese fichero de cabecera
- Hay que utilizar las instrucciones `#ifndef`, `#define` y `#endif` en nuestros ficheros de cabecera para evitarlo

```
// C1.h
#ifndef _C1_H_
#define _C1_H_
...
class C1{
    ...
};
#endif
```

Ejercicios

Ejercicios (1/3)

Ejercicio 1

Implementa la clase del siguiente diagrama:

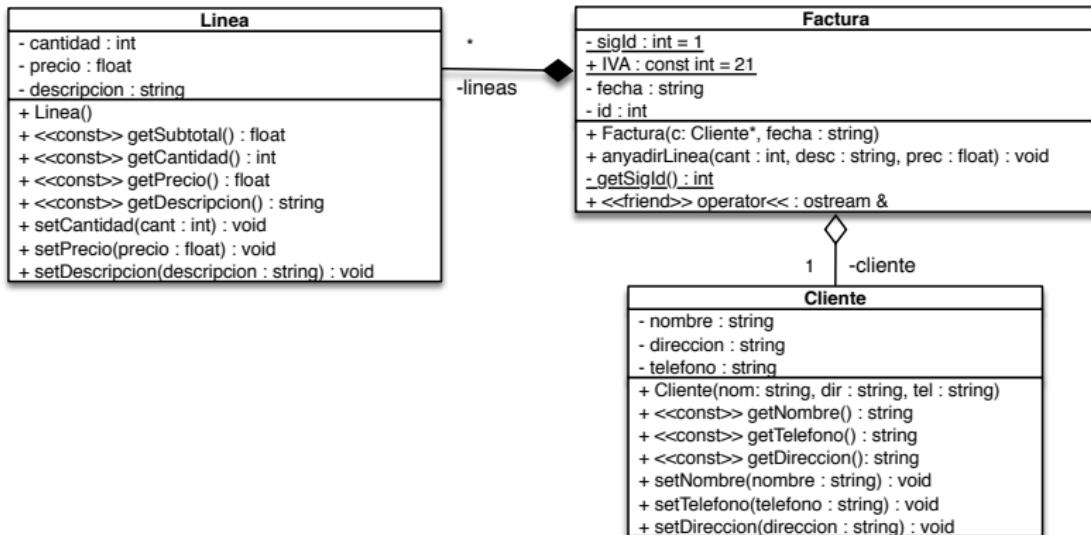
Coordenada
- x : float
- y : float
+ Coordenada (cx: float=0, cy: float = 0)
+ Coordenada (const Coordenada &)
+ ~Coordenada()
+ <<const>> getX() : float
+ <<const>> getY() : float
+ setX (cx:float) : void
+ setY (cy:float) : void
+ <<friend>> operator << : ostream &

Debes crear los ficheros `Coordenada.cc` y `Coordenada.h`, además de un *makefile* para compilarlos junto 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.

Ejercicios (2/3)

Ejercicio 2

Implementa el código correspondiente al siguiente diagrama UML:



Ejercicios (3/3)

Ejercicio 2 (continuación)

Se debe hacer un programa que cree una nueva factura, añada un producto y lo imprima. Desde el constructor de Factura se llamará al método `getSigId`, que devolverá el valor de `sigId` y lo incrementará. Ejemplo de salida al imprimir una factura:

Factura nº: 12345

Fecha: 18/4/2011

Datos del cliente

Nombre: Agapito Piedralisa

Dirección: c/ Rio 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

Subtotal: 63.39 €

IVA (21%): 13.3119 €

TOTAL: 76.7019 €