

Tema 1: Introducción

Programación 2

Grado en Ingeniería Informática
Universidad de Alicante
Curso 2025-2026



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

- 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);
```

- Devuelve `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:*

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

- 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)

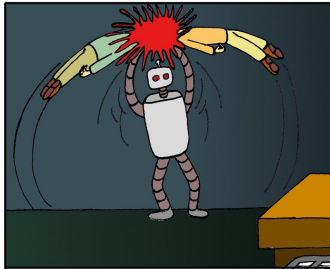
- Las *expresiones relacionales* permiten realizar comparaciones entre valores
- Operadores: == (igual), != (distinto), >= (mayor o igual), > (mayor estricto), <= (menor o igual) y < (menor estricto)
- Si los tipos de los operandos no son iguales se convierten (implícitamente) al tipo más general:

```
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)

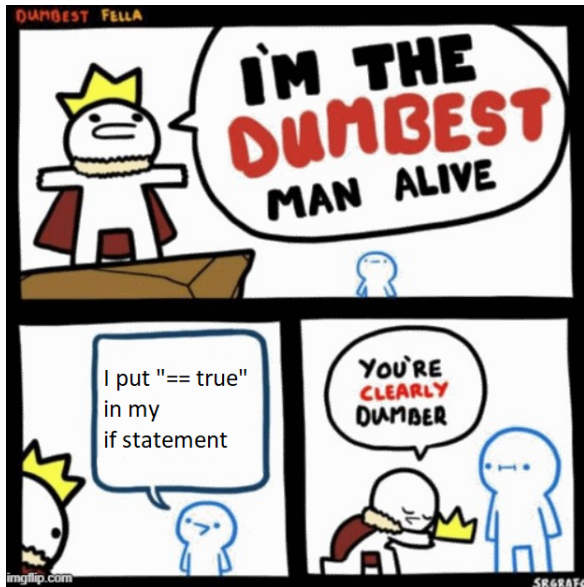


```
static bool isCrazyMurderingRobot = false;
```

```
void interact_with_humans (void){  
    if(isCrazyMurderingRobot = true)  
        kill(humans);  
    else  
        be_nice_to(humans);  
}
```

oppressive-silence.com

Expresiones relacionales (3/3)



- 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)

- 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
```

- 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){
    cout << "El número es menor que 5";
}
else if(num<9){ // Si no se cumple el primer if
    cout << "El número está entre 5 y 8";
}
else{ // Si no se cumple nada de lo anterior
    cout << "El número es mayor o igual que 9";
}
```

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<tamanyo || !encontrado){
    // Las dos condiciones han de ser falsas para terminar
}
```

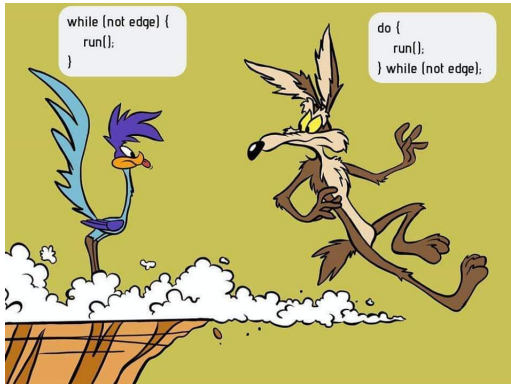
- Normalmente necesitaremos `&&` en lugar de `||`:

```
while(i<tamanyo && !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  
}
```

- `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 > 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[]="hola";  
// 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

Cadenas de caracteres > 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++

Cadenas de caracteres > 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 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"
```

Cadenas de caracteres > Entrada con 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 ')

Cadenas de caracteres > Entrada con 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
```

Cadenas de caracteres > Entrada con `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)

Cadenas de caracteres > Entrada con `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 << "Leído 1: " << cad << endl;  
cout << "Cadena 2: ";  
cin.getline(cad,10);  
cout << "Leído 2: " << cad << endl;
```

Terminal

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

Cadenas de caracteres > Entrada con `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 leído es: " << cad << endl;
```

Terminal

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

- ¿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  
...
```


Cadenas de caracteres > 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
```

Cadenas de caracteres > 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

Cadenas de caracteres > 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"
```

Cadenas de caracteres > 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
```

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;  
  
    instruccion1;  
    instruccion2;  
    ...  
  
    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 (1/2)

- 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
    unsigned 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
```

- Los campos de un registro se pueden inicializar al declararlo:

```
struct Equipo{  
    unsigned int id=0;  
    char nombre[100]="";  
    unsigned int victorias=0;  
    unsigned int derrotas=0;  
    unsigned int empates=0;  
    Jugador jugadores[20];  
};
```

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
```

Depuración

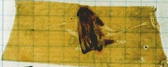
Depuración de código en C++ (1/3)

- 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 Antism started
1000 " stopped - antism ✓ { 1.2700 9.037 847 025
1300 (032) MP-MC 1.48260000 9.037 846 595 correct
033 PRO 2 2.130476415 4.615925059 (-2)
correct 2.130476415
Relays 6-2 in 033 failed speed test
in relay " 11.000 test.

1100 Relays changed
Started Cosine Tape (Sine check)
1525 Started Multi Adder Test.

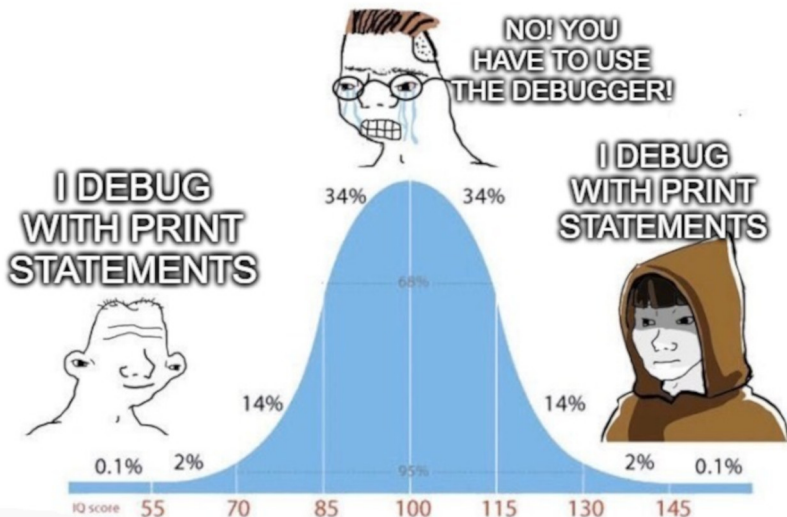
1545  Relay #70 Panel F
(noth) in relay.

1610 First actual case of bug being found.
Antism started.
1700 closed down.

Relay 3145
Relay 3376

- 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 (1/2)

Ejercicio 1

Diseña una función `mostrarMedia` que reciba como entrada un array con 10 valores enteros, calcule el valor medio de todos ellos y muestre por pantalla únicamente aquellos valores que estén por encima de la media.

Ejercicio 2

Diseña una función `contarVocales` que reciba como entrada una cadena de caracteres y devuelva cuántas vocales contiene.

```
contarVocales("HOLA") // Devuelve 2  
contarVocales("Hoy es el dia menos pensado") // Devuelve 10
```

Ejercicios (2/2)

Ejercicio 3

Crea un registro llamado `Jugador` con los campos: `nombre` (array de 50 caracteres) y `goles` (entero). Haz una función que lea los datos de 4 jugadores por teclado y muestra el nombre y la cantidad de goles del máximo goleador.

La entrada por teclado tendrá el siguiente formato, con los datos de cada jugador en una línea:

```
Bobby Charlton|34  
Gary Lineker|23  
Miroslav Klose|36  
Lukas Podolski|12
```