

# Tema 1: Introducción

## Programación 2

---

Grado en Ingeniería Informática  
Universidad de Alicante  
Curso 2022-2023



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
...
}
```

# 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

# 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:

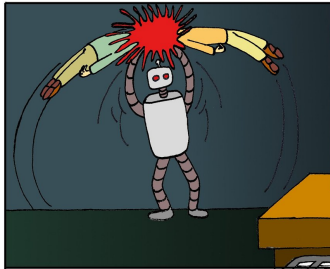
```
if(2<3.4){...} // Se transforma en: if(2.0<3.4)
```

- 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)

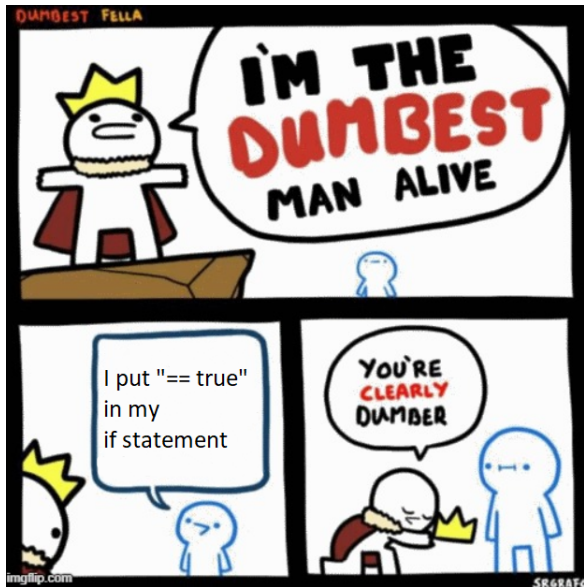


```
static bool isCrazyMurderingRobot = false;
```

```
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
```

\*Más detalles en el Tema 2

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

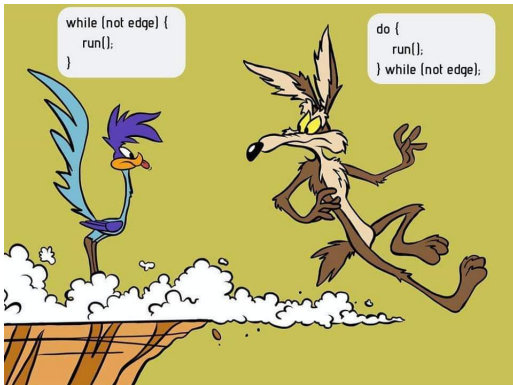
- Normalmente necesitaremos `&&` en lugar de `||`:

```
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  
}
```



## Control de flujo > switch (1/2)

- 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)

## Control de flujo > switch (2/2)

- Se puede utilizar `break` para salir de un bucle si se cumple una condición:

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Salimos del bucle al encontrar el 6 en vec
while(i<9){
    if(vec[i]==6)
        break;
    else
        i++;
}
// Código equivalente sin usar break
bool encontrado=false;
while(i<9 && !encontrado){
    if(vec[i]==6)
        encontrado=true;
    else
        i++;
}
```

## 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;  
  
    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)

- Se puede utilizar más de un `return` en el cuerpo de una función si eso simplifica el código:

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

```
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
}
```



## 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

## 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/>

# 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
```

## 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

---

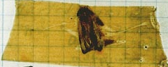
# Depuración de código en C++ (1/3)

- 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 Antarm started  
1000 " stopped - antarm ✓ { 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 2.130676415  
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.

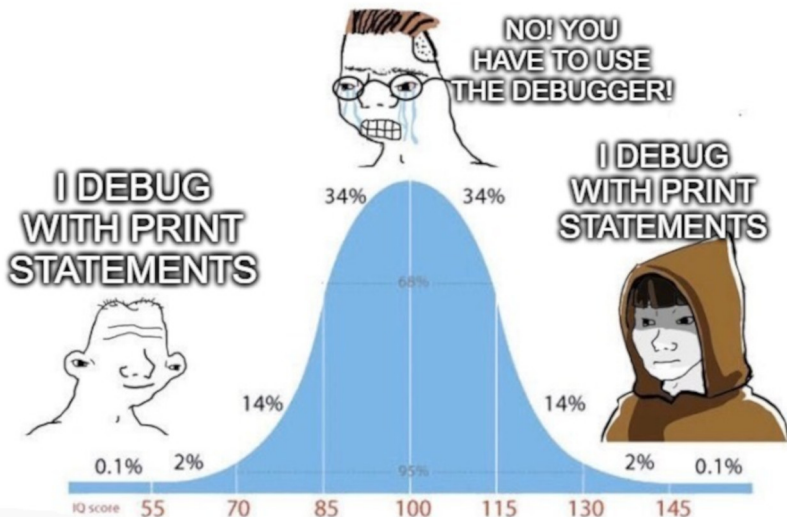
1545  Relay #70 Panel F  
(noth) in relay.

First actual case of bug being found.  
16100 Antarm 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

---



## Ejercicio 1

Implementa un programa que contenga una función con el siguiente prototipo: `int primeNumber(int n)`. Esta función devolverá el *n*-ésimo 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]
```