

Tema 1: Introducció

Programació 2

Grau en Enginyeria Informàtica
Universitat d'Alacant
Curs 2025-2026



1. Disseny d'algorismes i programes
2. Compilació
3. Elements bàsics de C++
4. Depuració
5. Exercicis

Disseny d'algorismes i programes

Com es fa un programa

1. Estudi del problema i de les possibles solucions
2. Disseny de l'algorisme **en paper**
3. Escriptura del programa **en l'ordinador**
4. Compilació del programa i correcció d'errors
5. Execució del programa
6. ... i prova de tots els casos possibles (o quasi)

El procés d'escriure, compilar, executar i provar ha de ser iteratiu, fent proves de funcions o mòduls del programa per separat.

Metodologia recomanada per a programar

- Estudi del problema i de la solució
- Disseny de l'algorisme en paper
- Dissenyar el programa intentant fer moltes funcions amb poc codi (unes 30 línies per funció)
- Evitar codi repetit utilitzant adequadament les funcions
- El `main` hauria de ser com l'índex d'un llibre i permetre entendre el que fa el programa d'una ullada
- Compilar i provar les funcions per separat: no esperar a tindre tot el programa per a començar a compilar i provar

Compilació

El procés de compilació

- El *compilador* permet convertir un codi font en un codi objecte
- En Programació 2 usem el compilador GNU C++ per a transformar el codi font en C++ en un programa executable
- El compilador de GNU s'invoca amb el programa `g++` i admet nombrosos arguments:
 - `-Wall`: mostra tots els *warnings*
 - `-g`: afegir informació per al depurador
 - `-o`: per a indicar el nom de l'executable
 - `-std=c++11`: per a usar l'estàndard de C++ de 2011
 - `--version`: mostra la versió actual del compilador
- Exemple d'ús:

Terminal

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

*Podeu veure la llista completa d'arguments executant `man g++` en el terminal de Linux

Elements bàsics de C++

Estructura d'un programa

```
#include <fitxers de capçalera estàndard>
...
#include "fitxers de capçalera propis"
...
using namespace std; // Permet usar cout, string...
...
const ... // Constants
...
typedef struct enum ... // Definició de nous tipus
...
// Variables globals: PROHIBIT en Programació 2!!
...
funcions ... // Declaració de funcions
...
int main() { // Funció principal
...
}
```

Mantenim algunes normes de Programació 1

- No es permet utilitzar **variables globals**
- No han d'aparèixer **warnings** en compilar els fitxers font de les pràctiques i els exàmens
- No es permet l'ús de **break** i **continue** en estructures de repetició
- No es permeten **múltiples return** en una mateixa funció

Identificadors

- Els *identificadors* són noms de variables, constants i funcions
- Han de començar per lletra minúscula, majúscula o guió baix
- C++ distingeix entre lletres majúscules i minúscules:

```
int grup,Grup; // Són dues variables diferents
```

- L'identificador ha d'indicar per a què s'utilitza:

```
int nombreAlumnes=0;  
void visualitzarAlumnes(){...}
```

- Mals exemples:

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

Paraules reservades

- En C++ hi ha *paraules reservades* que no es poden utilitzar com a noms definits per l'usuari:

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

- Si les usem com a identificadors ens donarà un error de compilació:

```
int friend=10;
```

Terminal

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

- Aquest tipus de missatges d'error no és de vegades fàcil d'interpretar

Variables > Definició i tipus

- Les *variables* permeten emmagatzemar diferents tipus de dades
- S'ha d'indicar el tipus de la variable quan es declara
- *Tipus bàsics* (o primitius) de dades en C++:

Tipus	Grandària (en bits)*
int	32
char	8
float	32
double	64
bool	8
void	No és un tipus

- Es pot usar `unsigned` amb `int` per a tindre només números positius (sense signe):

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

*En l'arquitectura x86

- Sempre que es declara una variable cal *inicialitzar-la*:

```
int nombreProfessors=11;
```

- No cal inicialitzar-la si el primer que es farà després de declarar la variable és assignar-li valor:

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

Variables > Àmbit (1/3)

- L'àmbit d'un variable (o constant) és la part del programa on es pot accedir a aquesta variable
- Una variable es pot usar des que es declara i dins del bloc entre claus que la conté:

```
int numCaixes=0;

if(i<10){
    // numCaixes es pot usar ací
    int numCaixes=100; // Mateix nom però un altre àmbit
    cout << numCaixes << endl; // Imprimeix 100
}

cout << numCaixes << endl; // Imprimeix 0
```

Variables > Àmbit (2/3)

- *Variable local* a una funció:
 - Aquella que es declara dins d'una funció
 - Normalment es declara al principi, encara que poden introduir-se en un punt intermedi:

```
void imprimir(){  
    int i=3,j=5; // Al principi de la funció  
    cout << i << j << endl;  
    ...  
    int k=7; // En un punt intermedi  
    cout << k << endl;  
}
```

- *Variable global*:
 - Es declara fora de les funcions
 - Es recomana no utilitzar variables globals (són perilloses)
 - En Programació 2 està prohibit usar variables globals

Variables > Àmbit (3/3)

- Exemple d'efecte col·lateral en usar una variable global:

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

void compteEnrrere(void){
    while(comptador>0){
        cout << comptador << " ";
        comptador--;
    }
    cout << endl;
}

int main(){
    compteEnrrere();
    compteEnrrere(); // Ací no imprimeix res
}
```

Constants

- Les *constants* tenen un valor fix (no pot ser canviat) durant tota l'execució del programa
- Es declaren anteposant `const` al tipus de dada:

```
const int MAXALUMNES=600;  
const double PI=3.141592;  
const char COMIAT[]="ADEU";
```

- Són útils per a definir valors que s'usen en múltiples punts d'un programa i que no canvien de valor (com la grandària d'un vector o d'un tauler d'escacs)

Tipus	Exemples
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 constant amb un zero al principi es tracta com un número octal

Tipus de dades > Conversió (1/2)

- *Conversió de tipus implícita*: la fa el compilador de manera automàtica

Tipus	Exemple
char → int	int a='A'+2; // a val 67
int → float	float pi=1+2.141592;
float → double	double piMig=pi/2.0;
bool → int	int b=true; // b val 1
int → bool	bool c=77212; // c val true

- *Conversió de tipus explícita*: la defineix el programador utilitzant l'operador *cast* (posant el tipus de dada entre parèntesi)

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

Tipus de dades > Conversió (2/2)

- De vegades, si no es fa *cast*, el compilador dóna un avís (*warning*) que s'estan comparant tipus que no són iguals
- És important no ignorar els *warnings*
- Quan comparem un enter (`int`) amb un enter sense signe (`unsigned int`) es produeix un *warning*:

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

if(num<strlen(cad)){ // strlen retorna un enter sense
    signe
    // Es pot evitar el warning amb un cast:
    // if((unsigned)num<strlen(cad))
}
```

Terminal

```
warning: comparison between signed and unsigned integer...
```

Tipus de dades > Definició de nous tipus

- En C++ es poden definir nous tipus mitjançant `typedef`:

```
typedef int enter;  
enter i,j;  
  
// logic i boolean són equivalents al tipus bool  
typedef bool logic,boolean;
```

- És possible declarar un vector com un tipus:

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

- A més, en C++ els noms que apareixen després de `struct`, `class` i `union` són també tipus

Tipus de dades > Comprovacions

- En C++, es pot comprovar si una variable és alfanumèrica (un dígit o una lletra) utilitzant la funció `isalnum()`:

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

- Retorna `true` si ho és i `false` en cas contrari
- Cal incloure la biblioteca `ctype` al codi per poder utilitzar-la:

```
#include <ctype.h>
...
if(isalnum(c)){ // Comprovar si és alfanumèric
    cout << c << " és alfanumèric";
}
else{
    cout << c << " no és alfanumèric";
}
```

Operadors d'increment i decrement

- Els operadors ++ i -- s'usen per a incrementar o decrementar el valor d'una variable entera en una unitat
- *Preincrement/predecrement*: s'incrementa/decrementa abans de prendre el valor

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

- *Postincrement/postdecrement*: s'incrementa/decrementa després de prendre el valor

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

- És recomanable que apareguen sols en la instrucció:

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

Expressions aritmètiques (1/2)

- Les *expressions aritmètiques* estan formades per operands (int, float i double) i operadors aritmètics (+ - * /):

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

- Si apareix un operand de tipus char o bool es converteix a enter implícitament:

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

- Si dividim dos enters el resultat és un enter:

```
cout << 7/2; // L'eixida és 3
```

- Si volem que el resultat de la divisió entera siga un valor real cal fer un *cast* a float o double:

```
cout << (float)7/2; // L'eixida és 3.5  
cout << (float)(7/2); // Ull! L'eixida és 3
```


Expressions aritmètiques (2/2)

- L'operador % retorna la resta de la divisió entera:

```
cout << 30%7; // L'eixida és 2
```

- Precedència d'operadors:*

++ (increment) -- (decrement) ! (negació) - (menys unari)
* (multiplicació) / (divisió) % (mòdul)
+ (suma) - (resta)

- En cas de dubte useu parèntesi:

```
cout << 2+3*4; // L'eixida és 14
               // * té més precedència que +
cout << 2+(3*4); // L'eixida és 14
cout << (2+3)*4; // L'eixida és 20
```

*De major a menor precedència. Els operadors d'una fila tenen la mateixa precedència

Expressions relacionals (1/3)

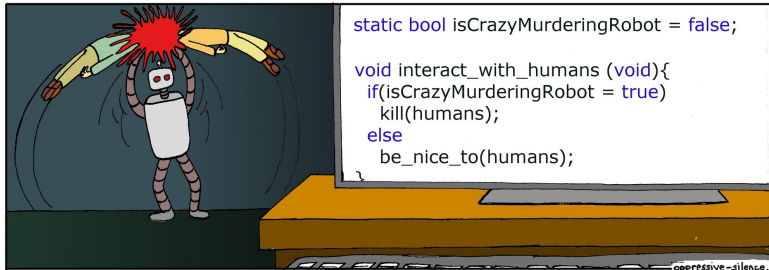
- Les *expressions relacionals* permeten realitzar comparacions entre valors
- Operadors: == (igual), != (diferent), >= (major o igual), > (major estricta), <= (menor o igual) i < (menor estricta)
- Si els tipus dels operands no són iguals es converteixen (implícitament) al tipus més general:

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

- Els operands s'agrupen de dos en dos per l'esquerra. Per a fer $a < b < c$ cal posar `a<b && b<c`
- El resultat és 0 si la comparació és falsa i diferent de 0 si és certa*

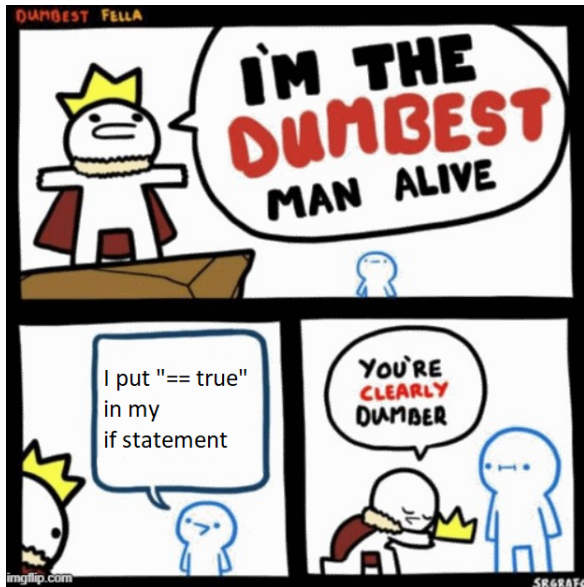
*En el compilador GCC és 1, però l'estàndard de C++ no obliga a això

Expressions relacionals (2/3)



oppressive-silence.com

Expresiones relacionales (3/3)



Expressions lògiques

- Les *expressions lògiques* permeten relacionar valors booleans i obtindre un nou valor booleà
- Operadors: ! (negació), && (i lògic) i || (o lògic)
- Precedència: ! > && > ||

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

- *Avaluació en curtcircuit*:
 - Si l'operand esquerre de && és fals, l'operand dret no s'avalua (false && elquesiga és sempre false)
 - Si l'operand esquerre de || és cert, l'operand dret no s'avalua (true || elquesiga és sempre true)

- Eixida per pantalla amb `cout`:

```
int i=7;  
cout << i << endl; // Mostra 7 i salt de línia (endl)
```

- Eixida d'error (per pantalla) amb `cerr`:

```
int i=7;  
cerr << i << endl; // Mostra 7 i salt de línia (endl)
```

- Entrada per teclat amb `cin`:

```
int i;  
cin >> i; // Guarda en i un número escrit per teclat
```

Control de flux > if

- Les *estructures de control de flux* avaluen una expressió condicional (`true` o `false`) i seleccionen la següent instrucció a executar depenent del resultat
- `if` avalua una condició i agafa un camí o un altre:

```
int num=0;
cin >> num; // Llegim un nombre per teclat

if(num<5){
    cout << "El nombre és menor de 5";
}
else if(num<9){ // Si no es compleix el primer if
    cout << "El nombre està entre 5 i 8";
}
else{ // Si no, executa aquesta altra
    cout << "El nombre és major o igual que 9";
}
```

Control de flux > while

- `while` executa instruccions mentre es complisca la condició:

```
int i=10;
while(i>=0){
    cout << i << endl; // Farà un compte enrere del 10 a 0
    i--; // Si no decrementem tindrem un bucle infinit
}
```

- Aneu amb compte en utilitzar `||` dins de la condició, perquè les dues parts han de ser falses perquè acabe el bucle:

```
while(i<grandaria || !trobat){
    // Les dues condicions han de ser falses per a terminar
}
```

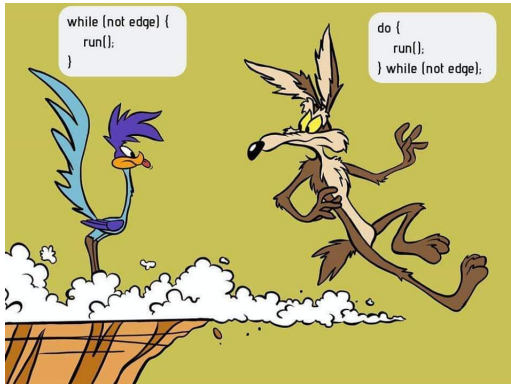
- Normalment necessitarem `&&` en lloc de `||`:

```
while(i<grandaria &&!trobat){
    // Acaba quan alguna de les condicions és falsa
}
```


Control de flux > do-while

- do-while executa el cos del bloc almenys una vegada:

```
int i=0;  
do{ // Mostra el valor de i almenys una vegada  
    cout << "i val: " << i << endl;  
    i++;  
}while(i<10);
```



Control de flux > for

- for equival a un while:

```
for(inicialització;condició;finalització){  
    // Instruccions  
}
```

```
inicialització;  
while(condició){  
    // Instruccions  
    finalització;  
}
```

- Té una sintaxi més elegant i compacta que while:

```
for(int i=10;i>=0;i--){  
    cout << i << endl; // Farà un compte enrreere del 10 al  
    0  
}
```

- switch permet seleccionar entre diverses opcions:

```
char opcio;  
cin >> opcio; // Llegim un caràcter de teclat  
  
switch(opcio){  
    case 'a': cout << "Opció A" << endl;  
               break; // Ix del switch  
    case 'b': cout << "Opció B" << endl;  
               break;  
    case 'c': cout << "Opció C" << endl;  
               break;  
    default: cout << "Una altra opció" << endl;  
}
```

- L'expressió en el switch (opcio en l'exemple anterior) ha de ser int o char (donarà error de compilació en cas contrari)

Vectors i matrius (1/3)

- Els *vectors* (o *arrays*) emmagatzemen múltiples valors en una única variable en posicions de memòria contigües
- Aquests valors poden ser de qualsevol tipus que desitgem, fins i tot tipus de dades pròpies
- En declarar un vector cal especificar-ne la grandària (quants elements emmagatzema) mitjançant constants o variables:

```
// Grandària definida mitjançant constants
const int MAXALUMNES=100;
int alumnes[MAXALUMNES]; // Pot emmagatzemar 100 enters
bool grupsPlens[5]; // Pot emmagatzemar 5 booleans

// Grandària definida mitjançant variables (no
    recomanable)
int numElements;
cin >> numElements; // No sabem quin número introduirà
float llistaNotes[numElements];
```

Vectors i matrius (2/3)

- Quan s'inicialitza un vector en declarar-lo no fa falta indicar-ne la grandària:

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

- Assignació i accés a valors mitjançant l'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 té grandària TAM, el primer element es troba en la posició 0 i l'últim en la posició TAM-1
- Podem tindre una errada en temps d'execució si intentem llegir o escriure en un element fora del vector:

```
int vec[5];  
vec[5]=7; // Possible errada en temps d'execució  
          // L'últim element vàlid està en vec[4]
```

Vectors i matrius (3/3)

- Una *matriu* és un vector les posicions del qual són, cadascuna d'elles, un altre vector
- Cal donar grandària a les dues dimensions (files i columnes):

```
const int TAM=10;  
char tauler[TAM][TAM]; // Matriu de 10 x 10 elements  
int taula[5][8]; // Matriu de 5 x 8 elements
```

- Com els vectors, comencen en 0 i acaben en TAM-1
- Assignació i accés a valors mitjançant l'operador []:

```
int matriu[8][10];  
matriu[2][3]=7; // Cal indicar fila i columna
```

- És possible utilitzar files de matrius com si foren vectors:

```
lligArray(matriu[4]); // Passem la fila 4 com un vector
```

Cadenes de caràcters > Declaració (1/3)

- Les *cadenes de caràcters* són vectors que contenen una seqüència de tipus `char` acabada amb el caràcter nul (`'\0'`):

```
// El compilador fica el '\0' al final automàticament  
char cad[]="hola";  
// Una altra manera d'inicialitzar, caràcter a caràcter  
char cad[]={ 'h', 'o', 'l', 'a', '\0' };  
// Falta el '\0': no és una cadena de caràcters vàlida  
char cad[]={ 'h', 'o', 'l', 'a' };
```

- Moltes de les funcions* que treballen amb cadenes busquen el `'\0'` per a saber on acaba la cadena
- Si no tenim el `'\0'` pot ser que el resultat d'aquestes funcions no siga l'esperat

*Com aquelles que pertanyen a la llibreria `cstring` i que veurem més endavant

Cadenes de caràcters > Declaració (2/3)

- Les cadenes de caràcters en C tenen grandària fixa i una vegada declarades no poden canviar de grandària:

```
char cad[10]; // Emmagatzema com a màxim 10 elements
```

- Cal tindre en compte que s'ha de reservar sempre un espai per a emmagatzemar el caràcter nul ('\\0'):

```
char cad[10]; // Emmagatzema com a màxim 9 lletres i el '\\0'
```

- Es poden inicialitzar en declarar-les, i en aquest cas no cal posar la grandària:

```
char cad[]="hola"; // Grandària 5 (4 lletres + '\\0')  
char cad2[10]="hola"; // Grandària 10, encara que només  
ocupa 5
```

- Les cadenes de caràcters en C es poden usar també en C++

- Errors comuns en declarar cadenes de caràcters:

```
// El vector és massa xicotet per a guardar la cadena  
char cad[5]="parallelepiped"; // Error de compilació
```

```
// S'usen cometes simples (') en lloc de dobles (")  
char cad[]='h'; // Error de compilació  
char cad[]='hola'; // Error de compilació
```

```
// No es posa la grandària i no s'inicialitza  
char cad[]; // Error de compilació
```

```
// S'intenta assignar valor amb '=' després de declarar  
char cad[10];  
cad="hola"; // Error de compilació
```

Cadenes de caràcters > Eixida per pantalla

- Eixida per pantalla amb `cout` i `cerr` com la resta de tipus simples (`int`, `float`, etc.)
- Podem combinar en l'eixida variables, constants i dades de diferent tipus:

```
char cad[]="Nota";  
int num=10;  
  
cout << cad << " -> " << num; // Mostra "Nota -> 10"
```

Cadenes de caràcters > Entrada amb operador >> (1/2)

- Podem llegir una cadena de caràcters des de teclat com amb altres tipus simples, utilitzant `cin` i l'operador `>>`
- Existeixen algunes diferències a l'hora de llegir des de teclat respecte a altres tipus de dades
- Ignora els blancs* abans de la cadena:

```
char cad[32];  
cin >> cad;  
// L'usuari escriu "  hola"  
// La variable cad emmagatzema "hola"
```

*Entenem per "blanc" un espai, tabulador o salt de línia (' \n ')

Cadenes de caràcters > Entrada amb operador >> (2/2)

- Acaba de llegir quan troba el primer blanc en la cadena. **No ens permet llegir sencera una cadena que continga blancs:**

```
char cad[32];  
cin >> cad;  
// L'usuari escriu "bona vesprada"  
// La variable cad emmagatzema "bona"
```

- No limita el nombre de caràcters que es lliguen. **L'usuari pot escriure una cadena més gran del que admet el vector:**

```
char cad[5];  
cin >> cad;  
// L'usuari escriu "esternoclidomastoidal"  
// Pot envair zones de memòria que no deuria i  
// produir una fallada de segmentació
```

Cadenes de caràcters > Entrada amb `getline` (1/4)

- També podem llegir una cadena de caràcters de teclat mitjançant `cin` i la funció `getline`
- Aquesta funció permet llegir cadenes amb blancs i limitar el nombre de caràcters llegits:

```
const int TAM=100;
char cad[TAM];
// cad: variable on emmagatzemem la cadena
// TAM: nombre de caràcters a llegir
cin.getline(cad,TAM);
// Si l'usuari introdueix "bona vesprada"
// en cad s'emmagatzema "bona vesprada"
```

- Llig com a màxim `TAM-1` caràcters o fins que arribi al final de línia
- El `'\n'` del final de línia es llig però no es guarda en la cadena
- La funció afig `'\0'` al final del que ha llegit (per això només llig `TAM-1` caràcters)

Cadenes de caràcters > Entrada amb `getline` (2/4)

- Si l'usuari introdueix més caràcters dels que caben, aquests es queden en el *buffer* de teclat i la següent lectura falla:

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

Terminal

```
$ elMeuPrograma
Cadena 1: hola a tothom
Llegit 1: hola a to
Cadena 2: Llegit 2:
```

Cadenes de caràcters > Entrada amb `getline` (3/4)

- Poden haver-hi problemes quan llegim de `cin` combinant l'operador `>>` i la funció `getline`:

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

cout << "Num: ";
cin >> num;
cout << "Escriu una cadena: " ;
cin.getline(cad,100);
cout << "El que he llegit és: " << cad << endl;
```

Terminal

```
$ elMeuPrograma
Num: 10
Escriu una cadena: El que he llegit és:
```

Cadenes de caràcters > Entrada amb `getline` (4/4)

- Per què succeeix això?
 - Amb l'operador `>>` es llig 10, però es deixa de llegir quan es troba el primer caràcter no numèric ('`\n`' en aquest cas)
 - El primer que troba en el *buffer* la funció `getline` és un '`\n`', per la qual cosa acaba de llegir i no guarda res en `cad`
- Solució:

```
...  
cin >> num;  
cin.ignore(); // Afegim aquesta línia  
              // Saca el '\n' del buffer  
// Ja es pot usar getline sense problema  
...
```


Cadenes de caràcters > La llibreria `cstring` (1/3)

- La llibreria `cstring` conté una sèrie de funcions que faciliten el treball amb cadenes de caràcters
- Per a poder utilitzar-la cal incloure la llibreria en el codi:

```
#include <cstring>
```

- `strlen` retorna la longitud (nombre de caràcters) d'una cadena:

```
char cad[10]="adeu";  
cout << strlen(cad); // Imprimeix 4
```

- `strcpy` còpia una cadena en una altra. **Cal anar amb cura de no superar la grandària del vector de destinació:**

```
char cad[5];  
strcpy(cad,"hola"); // Cap: 4 + '\0' = 5 caràcters  
strcpy(cad,"adeu!") // No cap!! Violació de segment
```

Cadenes de caràcters > La llibreria `cstring` (2/3)

- `strcmp` compara dues cadenes en ordre lexicogràfic*, retornant 1 si `cad1 > cad2`, 0 si `cad1 == cad2` i -1 si `cad1 < cad2`:

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

- `strcat` afegeix el contingut d'una cadena al final d'una altra. **Ha d'haver-hi prou espai en la cadena destí:**

```
char cad[10]="hola";  
strcat(cad," , mu"); // En total 9 caràcters (cabe)  
strcat(cad,"ndo"); // Afegeix 3 més (ja no cap!)
```

*Ordre que segueixen les paraules en un diccionari

Cadenes de caràcters > La llibreria `cstring` (3/3)

- Les funcions `strncpy`, `strncpy` i `strncat` comparen, copien o concatenen només els `n` primers caràcters:

```
char cad[8];  
strncpy(cad, "hola, mon", 4); // Només copia "hola"  
cad[4]='\0'; // No afegeix el '\0' de manera automàtica  
// L'hem d'afegir nosaltres al final
```

```
char cad1[8]="adios";  
char cad2[8]="adeu";  
// Només compara els dos primers caràcters  
cout << strncmp(cad1, cad2, 2) << endl; // Imprimeix 0
```

```
char cad1[50]="Hola, ";  
char cad2[]="mon meravellos";  
strncat(cad1, cad2, 3); // cad1 valdrà "Hola, mon"
```

- Per a passar una cadena de caràcters a `int` o `float` es poden usar les funcions `atoi` o `atof`
- Aquestes funcions pertanyen a la llibreria `cstdlib`:

```
#include <cstdlib> // Sempre que s'usen les funcions

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

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

Funcions > Definició (1/2)

- Una funció és un bloc de codi que fa una tasca
- Permet agrupar operacions comunes en un bloc reutilitzable
- Pot opcionalment tindre paràmetres d'entrada i retornar un valor com a eixida:

```
tipusRetorn nomFuncio(parametre1,parametre2,...) {  
    tipusRetorn ret;  
  
    instruccio1;  
    instruccio2;  
    ...  
  
    return ret;  
}
```

- Una funció no hauria de tindre molt de codi
- Si he de fer *copy-paste* en el codi és perquè necessite una funció

Funcions > Definició (2/2)

- Sempre es pot trobar la manera d'utilitzar un únic `return` en el cos d'una funció:

```
// No permès a Programació 2
bool buscar(int vec[], int n){
    for(int i=0;i<TAM;i++){
        if(vec[i]==n)
            return true; // Primer return
    }
    return false; // Segon return
}
```

```
// Versió alternativa amb un return
bool buscar(int vec[],int n){
    bool trobat=false;
    for(int i=0;i<TAM &&!trobat;i++){
        if(vec[i]==n)
            trobat=true;
    }
    return trobat; // Un únic return
}
```

Funcions > Paràmetres (1/2)

- Es permet pas de paràmetres per *valor* o per *referència* (amb &)

```
// a i b es passen per valor, c per referència  
void funcio(int a,int b,bool &c){  
    c=a<b; // c manté aquest valor en acabar la funció  
}
```

- Quan es passa un paràmetre per valor, el compilador en fa una còpia local per a usar-lo dins de la funció
- Si és un tipus de dada molt gran, és convenient passar-ho per referència amb `const` per eficiència:

```
void funcio(const string &s){  
    // El compilador no fa còpia de s, però si  
    // intentem modificar-la ens dóna un error  
}
```

- En P2 no es permet passar paràmetres per referència si no seran modificats, excepte si és amb `const`, com s'ha explicat

Funcions > Paràmetres (2/2)

- Els vectors i matrius es passen implícitament per referència (no cal posar & davant)
- El nom d'un vector o matriu, sense claudàtors, conté l'adreça de memòria on està emmagatzemat*
- En passar una matriu com a paràmetre no cal posar la grandària de la primera dimensió en la declaració de la funció:

```
void sumar(int v[],int m[][TAM]){  
    // En m no es posa la grandària de la primera dimensió  
    ...  
}  
...  
// No es posen claudàtors en la crida a la funció  
sumar(v,m);
```

*Més informació en el Tema 4

Funcions > Prototips

- De vegades és necessari utilitzar una funció abans que aparega el seu codi (o una funció el codi de la qual estiga en un altre mòdul)*
- En aquests casos cal posar el *prototip* de la funció:

```
void laMeuaFuncio(bool,char,double[]); // Prototip

char unaAltraFuncio(){
    double vr[20];
    // Encara no s'ha declarat laMeuaFuncio
    // però podem usar-la gràcies al prototip
    laMeuaFuncio(true,'a',vr);
}

// Declaració de la funció
void laMeuaFuncio(bool exist,char opt,double vec[]){
    ...
}
```

Registres (1/2)

- Un *registre* és una agrupació de dades, les quals no han de ser del mateix tipus
- Es defineixen amb la paraula `struct`:

```
struct Alumne{ // Defineix un nou tipus de dada Alumne
    unsigned int dni;
    float nota;
};
```

- Per a accedir als camps s'ha d'indicar el nom de la variable i del camp, separats per un punt:

```
Alumne a,b;
a.dni=123133; // Assignació de dades a un camp
b=a; // Assignació d'un registre complet bit a bit
```

- Els camps d'un registre es poden inicialitzar en declarar-lo:

```
struct Equip{  
    unsigned int id=0;  
    char nom[100]="";  
    unsigned int victories=0;  
    unsigned int derrotes=0;  
    unsigned int empats=0;  
    Jugador jugadors[20];  
};
```

Tipus enumerats

- Els *tipus enumerats* poden declarar-se amb un conjunt de possibles valors (*enumeradors*):

```
// Creem un nou tipus de dada color  
enum color{black,blue,green,red}; // Quatre enumeradors
```

- Les variables d'aquest tipus poden prendre qualsevol valor d'entre aquests enumeradors:

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

- Els valors dels tipus enumerats es converteixen internament en `int` i viceversa:

```
enum animal{cat,dog,monkey,fish};  
cout << monkey << endl; // Mostrará 2 per pantalla  
// És la posició que ocupa monkey en els enumeradors
```

Depuració

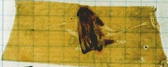
Depuració de codi en C++ (1/3)

- Quan hi ha un error en temps d'execució en el nostre codi és difícil a vegades localitzar en quin punt està la fallada
- Un *depurador* (*debugger*) és un programa que ens ajuda a trobar i corregir errors d'execució en el codi (*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 (033) 4.615925059 (-2)
correct 2.130476415
Relays 6-2 in 033 failed speed test
in relay " 11.000 test.

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

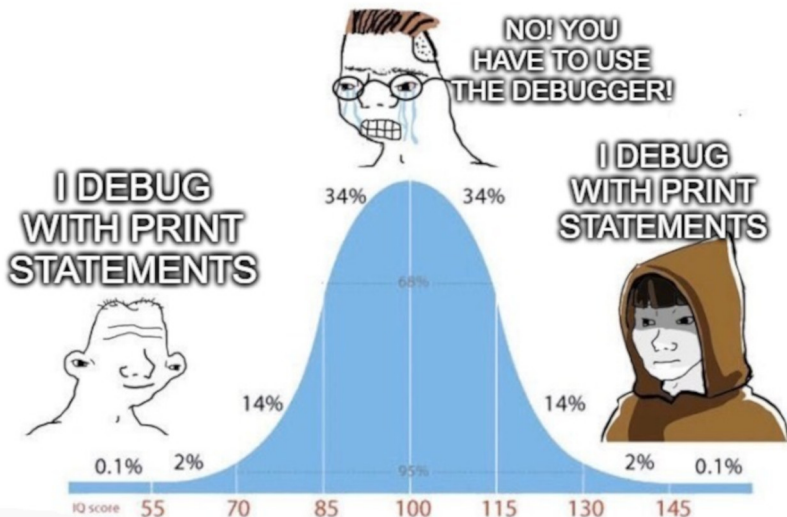
1545  Relay #70 Panel F
(noth) in relay.

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

Relay 3145
Relay 3376

- Un depurador permet, per exemple, executar el codi línia a línia o veure quins valors tenen les variables en un determinat punt d'execució
- Existeixen nombrosos programes que faciliten la tasca de localitzar errors en el codi:
 - *GDB*: inicia el nostre programa, l'atura quan ho demanem i mira el contingut de les variables. Si el nostre executable dóna una fallada de segmentació, ens diu la línia de codi on està el problema
 - *Valgrind*: detecta errors de memòria (accés a components fora d'un vector, variables usades sense inicialitzar, punters que no apunten a una zona reservada de memòria, etc.)
 - Altres exemples en Linux: *DDD*, *Nemiver*, *Electric Fence* i *DUMA*

Depuració de codi en C++ (3/3)



Exercicis

Exercicis (1/2)

Exercici 1

Dissenya una funció `mostrarMedia` que rep com a entrada un array amb 10 valors enters, calcula el valor mitjà de tots ells i mostra per pantalla únicament aquells valors que estiguen per sobre de la mitjana.

Exercici 2

Dissenya una funció `contarVocals` que rep com a entrada una cadena de caràcters i retorna quantes vocals conté.

```
contarVocals("HOLA") // Retorna 2  
contarVocals("Hoy es el dia menos pensado") // Retorna 10
```

Exercicis (2/2)

Exercici 3

Crea un registre anomenat `Jugador` amb els camps: `nom` (array de 50 caràcters) i `gols` (enter). Fes una funció que lligui les dades de 4 jugadors per teclat i mostri el nom i la quantitat de gols del màxim golejador.

L'entrada per teclat tindrà el següent format, amb les dades de cada jugador en una línia:

```
Bobby Charlton|34
```

```
Gary Lineker|23
```

```
Miroslav Klose|36
```

```
Lukas Podolski|12
```