

# Tema 1: Introducció

## Programació 2

---

Grau en Enginyeria Informàtica  
Universitat d'Alacant  
Curs 2024-2025



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)

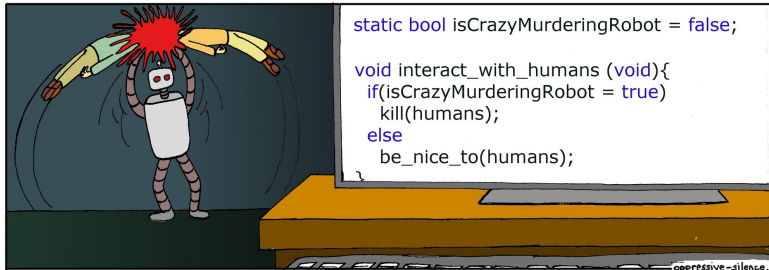
- Les *expressions relacionals* permeten realitzar comparacions entre valors
- Operadors: == (igual), != (diferent), >= (major o igual), > (major estricta), <= (menor o igual) i < (menor estricta)
- Si els tipus dels operands no són iguals es converteixen (implícitament) al tipus més general:

```
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\*

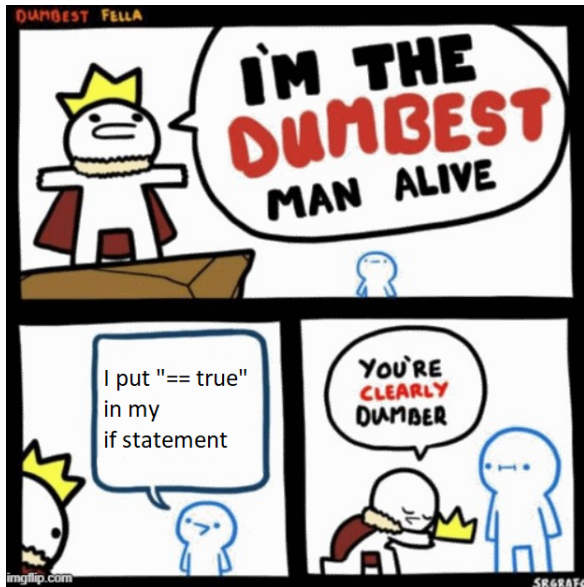
\*En el compilador GCC és 1, però l'estàndard de C++ no obliga a això

## 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)

# Entrada i eixida

- 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
```

\*Més detalls en el Tema 2

- 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 número per teclat

if(num<5){ // Si num és menor que 5 executa aquesta part
    cout << "El número és menor de cinc";
}
else{ // Si no, executa aquesta altra
    cout << "El número és major o igual que cinc";
}
```

## 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 `||`:

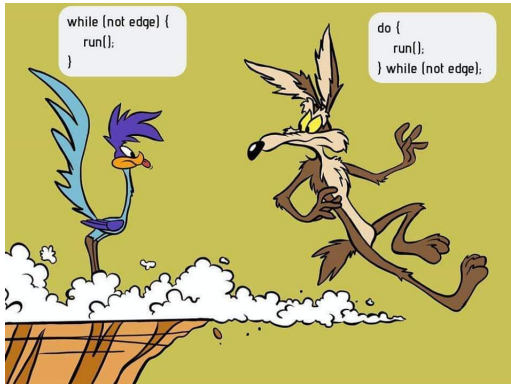
```
while(i<grandaria &&!trobat){
    // Acaba quan alguna de les condicions és falsa
}
```



# 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

- Les *cadenes de caràcters* són vectors que contenen una seqüència de caràcters acabada en el caràcter nul '\0':\*

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

"hola" → 

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

- Si no la inicialitzem cal especificar-ne la grandària:

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

- Recorda: "a" és una cadena i 'a' és un caràcter

```
char cadena[]="a"; // Ok  
char cadena2[]='a'; // Error de compilació
```

Més detalls sobre cadenes de caràcters en el Tema 2

## 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

- 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
    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
```

# 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
```

## Vectors STL (1/2)

- La *Standard Template Library* (STL) és una llibreria de funcions per a C++
- Proporciona diferents estructures de dades i algorismes
- Inclou la classe `vector`, que permet emmagatzemar elements de qualsevol tipus, com un vector normal, però sense haver de preocupar-nos de la grandària:

```
#include <vector> // Sempre que usem vector
vector<int> vec; // Declara un vector d'enters
                // No és necessari indicar la grandària
```

- La grandària inicial d'un vector STL és 0 i creix de manera dinàmica en funció de les necessitats
- Per a afegir elements al final del vector usem `push_back`:\*

```
vec.push_back(12); // Afig 12 al final del vector
vec.push_back(8); // Afig 8 darrere del 12
```

\*Com és una classe, els mètodes s'invoquen posant un punt després del nom de la variable

## Vectors STL (2/2)

- Accés a elements mitjançant l'operador []:

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

- Amb size obtenim el nombre d'elements del vector:

```
// Recorrem tots els elements del vector  
for(unsigned int i=0;i<vec.size();i++){  
    vec[i]=10;  
}
```

- Amb clear podem esborrar tots els elements i amb erase un en concret:

```
vec.erase(vec.begin()+3); // Elimina el quart element  
vec.clear(); // Elimina tots els elements del vector
```

- Existeixen moltes altres funcions per a treballar amb vectors STL\*

\*Més informació en <http://www.cplusplus.com/reference/vector/vector/>



## Arguments del programa (1/4)

- Els *arguments* d'un programa s'usen per a proporcionar-hi informació (normalment opcions) des de línia d'ordres
- Aquest ús és molt habitual i permet modificar el comportament del programa:

### Terminal

```
$ ls           // Mostra els fitxers d'un directori
$ ls -a        // Mostra també els fitxers ocults (opció "-a")
$ ls -a -l     // Afig informació extra de cada fitxer (opció "-l")
```

## Arguments del programa (2/4)

- El `main` és una funció i per tant pot rebre dos paràmetres: `argc` i `argv`
- Aquests paràmetres permeten gestionar el pas d'arguments per línia d'ordres al programa:

```
// Sempre en aquest ordre  
int main(int argc, char *argv[]){  
    ...  
    return 0;  
}
```

- `int argc`: nombre d'arguments passats al programa (comptant també el nom del programa)
- `char *argv[]`: vector de cadenes de caràcters amb els arguments passats al programa

## Arguments del programa (3/4)

- Exemple d'ús:

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

### Terminal

```
$ ./elMeuPrograma -a -h X    // Exemple de crida amb tres paràmetres  
Arg. 0 : ./elMeuPrograma  
Arg. 1 : -a  
Arg. 2 : -h  
Arg. 3 : X
```

- Els arguments no han de començar amb un guió (-) necessàriament però és una pràctica prou habitual

## Arguments del programa (4/4)

- Sembla fàcil gestionar els arguments del programa, però de vegades pot ser complicat
- L'usuari no sempre usa el mateix ordre a l'hora d'introduir els arguments:

### Terminal

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

- Pot haver-hi errors en la introducció i cal mostrar missatges d'ajuda a l'usuari
- És recomanable usar una funció a banda per a gestionar els arguments

# 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 Antenn started  
1000 stopped - antenna ✓

1300 (032) MP - MC 2.130476415  
(033) PRO 2 2.130476415  
conv 2.130676415

Relays 6-2 in 033 failed speed speed test  
in relay 10,000 test.

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

1545

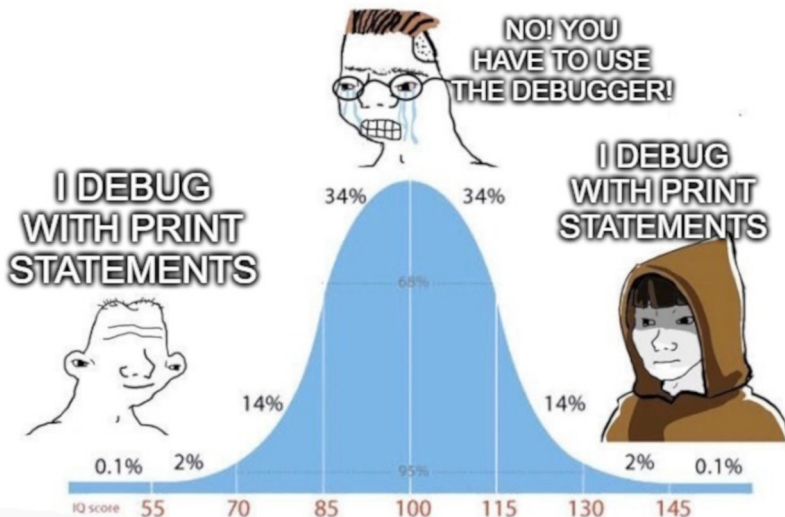
Relay #70 Panel F  
(moth) in relay.

First actual case of bug being found.  
1650 Antenn started.  
1700 closed down.

Relay 2145  
Relay 3370

- 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

---

## Exercici 1

Implementa un programa que continga una funció amb el següent prototip: `int primeNumber(int n)`. Aquesta funció tornarà el  $n$ -èsim nombre primer. El programa haurà d'imprimir nombres primers per pantalla amb les següents opcions:

- `-L` imprimir cada nombre en una línia diferent (per defecte s'imprimeixen tots a la mateixa línia)
- `-N n` imprimir els  $n$  primers nombres primers (per defecte 10)

Exemples d'execució:

### Terminal

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