

Unit 1: Introduction

Programming 2

Degree in Computer Engineering
University of Alicante
2025-2026



1. Algorithms and program design
2. Compilation
3. Basic elements of C++
4. Debugging
5. Exercises

Algorithms and program design

How to make a program

1. Study the problem and the possible solutions
2. Algorithm design on paper
3. Writing the program on the computer
4. Program compilation and error correction
5. Program execution
6. ... and testing (almost) all the possible cases

The process of writing, compiling, executing and testing has to be iterative, including individual tests on functions and modules.

Recommended methodology for programming

- Study the problem and explore the solutions
- Design the algorithm on paper
- Design the program trying to write many functions with little code (about 30 lines per function)
- Avoid repeated code by properly using functions
- `main` should be like the index of a book and allow us to understand what the program does at a glance
- Compile and test the functions separately: do not wait until having the whole program written before compiling and testing

Compilation

The compilation process

- The *compiler* converts a source code into an object code
- In Programming 2 we use the GNU C++ compiler to transform the source code in C++ into an executable program
- The GNU compiler is invoked with the `g++` program and admits numerous arguments:*
- `-Wall`: shows all the *warnings*
 - `-g`: adds information for the debugger
 - `-o`: sets the name of the executable
 - `-std=c++11`: uses the 2011 C++ standard
 - `--version`: shows the current version of the compiler
- Example of use:

Terminal

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

*The complete list of arguments can be accessed by running `man g++` on the Linux terminal

Basic elements of C++

Structure of a program

```
#include <standard header files>
...
#include "own header files"
...
using namespace std; // Allows using cout, string...
...
const ... // Constants
...
typedef struct enum ... // Definition of new types
...
// Global variables: FORBIDDEN in Programming 2!!
...
functions ... // Declaration of functions
...
int main() { // Main function
...
}
```

We keep some rules from Programming 1

- No usage of `global variables` is allowed
- No `warnings` should appear when compiling the source files for assignments and exams
- No use of `break` and `continue` is allowed within loop structures
- No `multiple return` statements are allowed in a single function

Identifiers

- *Identifiers* are names of variables, constants and functions
- They must begin with lowercase or uppercase letters, or the underscore character
- C++ distinguishes between upper and lower case letters:

```
int group,Group; // These are two different variables
```

- Identifiers must describe their use:

```
int numStudents=0;  
void visualiseStudents(){...}
```

- Bad examples:

```
const int EIGHT=8;  
int p,q,r,a,b;  
int counter1,counter2; // More usual: int i,j;
```

Reserved words

- In C++ *reserved words* exist that cannot be used as user-defined identifiers:

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

- If we use them as identifiers we will get a compilation error:

```
int friend=10;
```

Terminal

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

- This type of error messages is not easy to interpret

Variables > Definition and types

- *Variables* may store different types of data
- The type of a variable must be indicated when declared
- *Basic data types* in C++:

Type	Size (in bits)*
int	32
char	8
float	32
double	64
bool	8
void	Not a type

- `unsigned` can be used with `int` to get positive numbers (without sign):

```
int i=3; // Values from -2.147.483.648 to 2.147.483.647
unsigned int j=3; // Values between 0 and 4.294.967.295
```

*In the x86 architecture

Variables > Initialisation

- When a variable is declared, it should be *initialised*:

```
int numTeachers=11;
```

- It is not necessary to initialise it if the first thing done after declaring the variable is assigning it a value:

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

Variables > Scope (1/3)

- The *scope* of a variable (or constant) is the part of the program in which the variable can be accessed
- A variable can be used from the moment it is declared and within the block between curly braces that contains it:

```
int numBoxes=0;

if(i<10){
    // numBoxes can be used here
    int numBoxes=100; // Same name but different scope
    cout << numBoxes << endl; // Output is 100
}

cout << numBoxes << endl; // Output is 0
```

Variables > Scope (2/3)

- *Local variable* to a function:
 - A variable that is declared within a function
 - Usually declared at the beginning of the function, although it can be introduced at an intermediate point as well:

```
void print() {  
    int i=3,j=5; // At the beginning of the function  
    cout << i << j << endl;  
    ...  
    int k=7; // At an intermediate point  
    cout << k << endl;  
}
```

- *Global variable*:
 - A variable declared out of the function scope
 - It is recommended not to use global variables (they are dangerous)
 - **In Programming 2 it is forbidden to use global variables**

Variables > Scope (3/3)

- Example of collateral effect when using a global variable:

```
#include <iostream>
using namespace std;
int counter=10; // Global variable

void countDown(void){
    while(counter>0){
        cout << counter << " ";
        counter--;
    }
    cout << endl;
}

int main(){
    countDown();
    countDown(); // Prints nothing
}
```

Constants

- *Constants* have a fixed value (that cannot be changed) during the whole execution of the program
- They are declared by adding `const` before the data type:

```
const int MAXSTUDENTS=600;  
const double PI=3.141592;  
const char FAREWELL[]="Goodbye";
```

- Useful to define values that are used in multiple points of a program, and which do not change their values (such as the size of an array or a chess board)

Type	Examples
int	123 017* 1010101
float/double	123.7 .123 1e1 1.231E-12
char	'a' '1' ';' '\n' '\0' '\\'
char[] (string)	"" "hello" "double: \"
bool	true false

*A constant value starting with a zero is treated as an octal number

Data types > Conversion (1/2)

- *Implicit type conversion*: automatically done by the compiler

Types	Examples
char → int	int a='A'+2; // a is 67
int → float	float pi=1+2.141592;
float → double	double halfPi=pi/2.0;
bool → int	int b=true; // b is 1
int → bool	bool c=77212; // c is true

- *Explicit type conversion*: defined by the programmer using the *cast operator* (writing the data type in parentheses)

```
char theC=(char) ('A'+2); // theC is 'C'  
int integerPi=(int)pi;    // integerPi is 3
```

Data types > Conversion (2/2)

- Sometimes, if the *cast* is not done, the compiler displays a *warning* complaining about the comparison of different data types
- **It is important not to ignore *warnings***
- When comparing an integer (`int`) with an unsigned integer (`unsigned int`) a *warning* occurs:

```
int num=5;
char str[]="Hello";

if(num<strlen(str)){ // strlen returns an unsigned int
    // The warning can be avoided with a cast:
    // if((unsigned)num<strlen(str))
}
```

Terminal

warning: comparison between signed and unsigned integer...

Data types > Definition of new types

- In C++ new types can be defined using `typedef`:

```
typedef int integer;  
integer i,j;  
  
// logic and boolean are equivalent to bool  
typedef bool logic,boolean;
```

- It is possible to declare an array as a new data type:

```
typedef char tStr[50]; // tStr is an array of chars
```

- In C++ names that appear after `struct`, `class`, and `union` are also types

Data Types > Checks

- In C++, you can check whether a variable is alphanumeric (a digit or a letter) using the `isalnum()` function:

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

- It returns `true` if it is, and `false` otherwise
- The `ctype` library must be included in the code to use it:

```
#include <ctype.h>
...
if(isalnum(c)) { // Check if 'c' is alphanumeric
    cout << c << " is alphanumeric";
}
else {
    cout << c << " is not alphanumeric";
}
```

Increment and decrement operators

- The ++ and -- operators are used to increase or decrease the value of an integer variable by one unit
- *Preincrement/predecrement*: increases/decreases the variable before considering its value

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

- *Postincrement/postdecrement*: increases/decreases the variable after considering its value

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

- It is recommended that these operators are used isolated:

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

Arithmetic expressions (1/2)

- *Arithmetic expressions* are formed by operands (`int`, `float` and `double`) and arithmetic operators (`+` `-` `*` `/`):

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

- `char` and `bool` operands are implicitly converted to integer:

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

- If two integers are divided, the result is an integer:

```
cout << 7/2; // Output is 3
```

- If we want the result of the integer division to be a real value, we must make a *cast* to `float` or `double`:

```
cout << (float)7/2; // Output is 3.5  
cout << (float)(7/2); // Watch out! Output is 3
```


Arithmetic expressions (2/2)

- The `%` operator (modulus) returns the remainder after integer division:

```
cout << 30%7; // Output is 2
```

- Operators precedence.*

++ (increment) -- (decrement) ! (negation) - (unary minus)
* (product) / (division) % (modulus)
+ (sum) - (subtraction)

- If in doubt, use brackets:

```
cout << 2+3*4; // Output is 14
               // * has higher precedence than +
cout << 2+(3*4); // Output is 14
cout << (2+3)*4; // Output is 20
```

*From highest to lowest precedence. The operators of a row have the same precedence

Relational expressions (1/3)

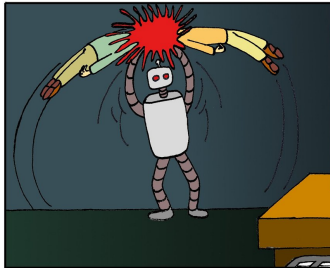
- *Relational expressions* allow comparisons between values
- Operators: `==` (equal), `!=` (different), `>=` (greater or equal), `>` (greater), `<=` (less or equal) and `<` (less)
- If operand types are not equal, they are (implicitly) converted to the most general type:

```
if(2<3.4){...} // Converted to: if(2.0<3.4)
```

- Operands are grouped two by two from left to right: $a < b < c$ must be coded as `a<b && b<c`
- The result is 0 if the comparison is false and different from 0 if it is true*

*In the GCC compiler it is 1, but the C++ standard does not impose this

Relational expressions (2/3)

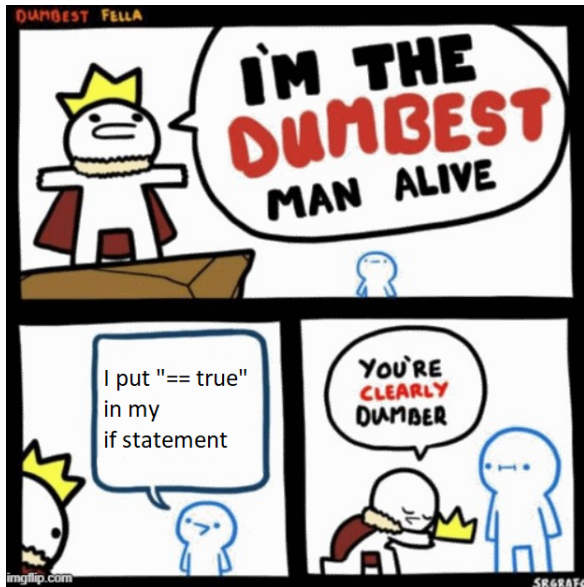


```
static bool isCrazyMurderingRobot = false;
```

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

oppressive-silence.com

Relational expressions (3/3)



Logical expressions

- The *logical expressions* allow to operate boolean values and obtain a new boolean value
- Operators: `!` (negation), `&&` (logical *and*) and `||` (logical *or*)
- Precedence: `! > && > ||`

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

- *Short-circuit evaluation*:
 - If the left operand of `&&` is false, the right operand is not evaluated (false `&&` whatever is always false)
 - If the left operand of `||` is true, the right operand is not evaluated (true `||` whatever is always true)

Input and output

- Screen output with `cout`:

```
int i=7;  
cout << i << endl; // Outputs 7 and a line break (endl)
```

- Error (screen) output with `cerr`:

```
int i=7;  
cerr << i << endl; // Outputs 7 and a line break (endl)
```

- Keyboard input with `cin`:

```
int i;  
cin >> i; // Stores in i a number written with the  
          keyboard
```

Flow control > if

- *Flow control structures* evaluate a conditional expression (`true` or `false`) and select the following instruction to execute depending on the result
- `if` evaluates a condition and takes one path or another:

```
int num=0;
cin >> num; // Read a number

if(num<5){
    cout << "The number is lower than 5";
}
else if(num<9){ // The first if condition is not met
    cout << "The number is between 5 and 8";
}
else{ // If not, execute this one
    cout << "The number is greater or equal than 9";
}
```

Flow control > while

- `while` executes instructions as long as the condition is true:

```
int i=10;
while(i>=0){
    cout << i << endl; // Does a countdown from 10 to 0
    i--; // Forgetting to decrease implies an infinite loop
}
```

- Caution when using `||` within the condition, because the two parts must be false to finish the loop:

```
while(i<length || !found){
    // The two conditions must be false to finish the loop
}
```

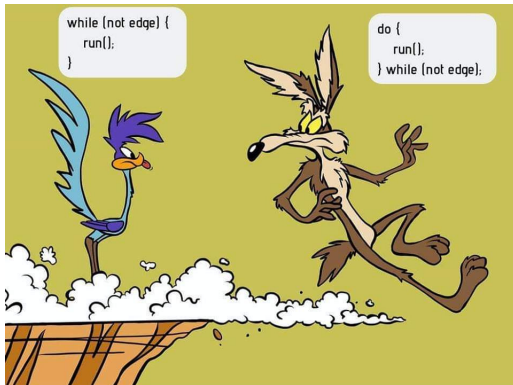
- Usually we will need `&&` instead of `||`:

```
while(i<length && !found){
    // Finishes the loop when either condition is false
}
```


Flow control > do-while

- do-while executes the body at least once:

```
int i=0;  
do{ // Shows the value of i at least once  
    cout << "i is: " << i << endl;  
    i++;  
}while(i<10);
```



Flow control > for

- for is equivalent to while :

```
for(initialisation;condition;completion){  
    // Instructions  
}
```

```
initialisation;  
while(condition){  
    // Instructions  
    completion;  
}
```

- Provides a more elegant and compact syntax than while:

```
for(int i=10;i>=0;i--){  
    cout << i << endl; // Counts down from 10 to 0  
}
```

Flow control > switch

- `switch` allows selecting between several options:

```
char option;  
cin >> option; // Reads a character from keyboard  
  
switch(option){  
    case 'a': cout << "Option A" << endl;  
               break; // Exits the switch  
    case 'b': cout << "Option B" << endl;  
               break;  
    case 'c': cout << "Option C" << endl;  
               break;  
    default: cout << "Another option" << endl;  
}
```

- The expression in the `switch` (`option` in the previous example) has to be `int` or `char` (otherwise a compilation error is emitted)

Arrays and matrices (1/3)

- *Arrays (or vectors)* store multiple values in a single variable in contiguous memory locations
- These values can be of any type, including our own data types
- When declaring an array, its size (the number of elements it stores) must be specified by means of constants or variables:

```
// Size defined by constants  
const int MAXSTUDENTS=100;  
int students[MAXSTUDENTS]; // Can store 100 integers  
bool fullGroups[5]; // Can store 5 booleans  
  
// Size defined by variables (not recommended)  
int numElements;  
cin >> numElements; // Users can input any number  
float listMarks[numElements];
```

Arrays and matrices (2/3)

- When initialising an array at declaration, it is not necessary to indicate its size:

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

- Assignment and access to values through the `[]` operator:

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

- If an array has size `SIZE`, the first element is located in position 0 and the last one in position `SIZE-1`
- A runtime error may occur when trying to read or write in an out of bounds position:

```
int vec[5];  
vec[5]=7; // A runtime error may occur  
           // The last valid element is vec[4]
```

Arrays and matrices (3/3)

- A *matrix* is an array in which each position contains another array
- It is necessary to set the size of the two dimensions (rows and columns):

```
const int SIZE=10;  
char board[SIZE][SIZE]; // Matrix of 10 x 10 elements  
int table[5][8]; // Matrix of 5 x 8 elements
```

- As with arrays, they begin in 0 and end at SIZE-1
- Assignment and access to values through the [] operator:

```
int matrix[8][10];  
matrix[2][3]=7; // Necessary to indicate row and column
```

- It is possible to use rows of matrices as if they were arrays:

```
readArray(matrix[4]); // Passes row 4 as an array
```

Character arrays > Declaration (1/3)

- *Character arrays* contain a sequence of `char` elements ending with the null character (`'\0'`):

```
// The compiler automatically puts the '\0' at the end  
char str[]="hello";  
// Another way of initialising, character by character  
char str[]={ 'h', 'e', 'l', 'l', 'o', '\0' };  
// Missing '\0': not a valid character array  
char str[]={ 'h', 'o', 'l', 'a' };
```

- Many functions that work with character arrays* look for the `'\0'` to identify where the array ends
- If there is no `'\0'` in the array, the result of these functions may not be as expected

*Such as those defined in the `cstring` library, as described later

Character arrays > Declaration (2/3)

- Character arrays in C have a fixed size and cannot be resized after being declared:

```
char str[10]; // Stores a maximum of 10 elements
```

- A space must always be reserved to store the null character ('\0');

```
char str[10]; // Maximum of 9 characters and '\0'
```

- They can be initialised when declared. In that case, it is not necessary to set the size:

```
char str[]="hello"; // Size 6 (5 letters + '\0')  
char str2[10]="hello"; // Size 10, but only 6 are used
```

- Character arrays in C can also be used in C++

Character arrays > Declaration (3/3)

- Common errors when declaring character arrays:

```
// Array too small to store the string  
char str[5]="parallelepiped"; // Compilation error  
  
// Single quotes (') used instead of double quotes (")  
char str[]='h'; // Compilation error  
char str[]='hello'; // Compilation error  
  
// Size not set and variable not initialised  
char str[]; // Compilation error  
  
// Attempt to assign a value with '=' after declaration  
char str[10];  
str="hello"; // Compilation error
```

Character arrays > Screen output

- Screen output with `cout` and `cerr` as with any of the other basic data types (`int`, `float`, etc.)
- Output can combine variables, constants and different data types:

```
char str[]="Mark";  
int num=10;  
  
cout << str << " -> " << num; // Output is "Mark -> 10"
```

Character arrays > Input with operator >> (1/2)

- Character arrays can be read from the keyboard, as in other basic data types, using `cin` and the operator `>>`
- There are some differences when reading from the keyboard with respect to other data types
- Blanks* before the string are ignored:

```
char str[32];  
cin >> str;  
// User writes "    hello"  
// The str variable stores "hello"
```

*We mean with "blank" a space, tab or new line (`'\n'`)

Character arrays > Input with operator >> (2/2)

- Reading finishes as soon as the first blank is found. **Therefore, an entire string containing blanks cannot be read:**

```
char str[32];  
cin >> str;  
// The user writes "good afternoon"  
// The str variable stores "good"
```

- There is no limit in the number of characters that are read. **User can type a string larger than the array size:**

```
char str[5];  
cin >> str;  
// The user writes "sternocleidomastoid"  
// Could overlap memory cells not belonging to the  
variable and produce a segmentation fault
```

Character arrays > Input with `getline` (1/4)

- Keyboard input can be also read using `cin` and the `getline` function
- This function allows reading strings with blanks, limiting the number of characters to be read:

```
const int SIZE=100;  
char str[SIZE];  
// str: variable where the characters are stored  
// SIZE: number of characters read  
cin.getline(str,SIZE);  
// If the user enters "good evening"  
// the variable str stores "good evening"
```

- Reads a maximum of `SIZE-1` characters or until reaching the end of the line
- The `'\n'` at the end of the line is read but not stored in the variable
- The function adds `'\0'` to the end of what has been read (therefore only reads `SIZE-1` characters)

Character arrays > Input with `getline` (2/4)

- If the user types more characters than indicated, they remain in the keyboard *buffer* and the next reading fails:

```
char str[10];  
cout << "String 1: ";  
cin.getline(str,10);  
cout << "Read 1: " << str << endl;  
cout << "String 2: ";  
cin.getline(str,10);  
cout << "Read 2: " << str << endl;
```

Terminal

```
$ myProgram  
String 1: hello everybody  
Read 1: hello eve  
String 2: Read 2:
```

Character arrays > Input with `getline` (3/4)

- There can be problems when reading from `cin` combining the `>>` operator and the `getline` function:

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

cout << "Num: ";
cin >> num;
cout << "Input string: " ;
cin.getline(str,100);
cout << "What I read is: " << str << endl;
```

Terminal

```
$ myProgram
Num: 10
Input string: What I read is:
```

Character arrays > Input with `getline` (4/4)

- Why is this happening?
 - The `>>` operator reads `10`, but stops reading when the first non-numeric character is found (`'\n'` in this case)
 - The first thing that `getline` finds in the *buffer* is a `'\n'`, so it finishes reading and does not store anything in `str`
- Solution:

```
...  
cin >> num;  
cin.ignore(); // Add this line  
              // Gets '\n' out of buffer  
// getline can now be used without issues  
...
```


Character arrays > The `cstring` library (1/3)

- The `cstring` library contains a set of functions that facilitate working with character arrays
- The library must be included in the code using it:

```
#include <cstring>
```

- `strlen` returns the length (number of characters) of a character array:

```
char str[10]="hello";  
cout << strlen(str); // Prints 5
```

- `strcpy` copies one character array into another. **Be careful not to exceed the size of the target array:**

```
char str[5];  
strcpy(str, "cool"); // The string fits into str: 4 + '\0'  
                    = 5 characters  
strcpy(str, "house"); // No fitting! Segmentation fault!
```

Character arrays > The `cstring` library (2/3)

- `strcmp` compares two strings in lexicographical order*, returning 1 if `str1>str2`, 0 if `str1==str2`, and -1 if `str1<str2`:

```
char str1[]="root";
char str2[]="river";
cout << strcmp(str1,str2) << endl; // Prints 1
cout << strcmp(str2,str1) << endl; // Prints -1
cout << strcmp(str1,str1) << endl; // Prints 0
```

- `strcat` appends the content of one string to the end of another.
There must be enough space in the destination string:

```
char str[10]="hello";
strcat(str," wo"); // Total 10 characters (fits)
strcat(str,"rld"); // Adds 3 more (no longer fits!)
```

*Order followed by words in a dictionary

Character arrays > The `cstring` library (3/3)

- The functions `strncmp`, `strncpy`, and `strncat` compare, copy, or concatenate only the first `n` characters:

```
char str[8];  
strncpy(str, "hello, world", 5); // Only copies "hello"  
str[5]='\0'; // Does not add the '\0' automatically  
// We need to add it manually
```

```
char str1[8]="help";  
char str2[8]="hello";  
// Only compares the first two characters  
cout << strncmp(str1, str2, 2) << endl; // Prints 0
```

```
char str1[50]="Hello, ";  
char str2[]="wonderful world";  
strncat(str1, str2, 9); // str1 will be "Hello, wonderful"
```

Character arrays > Conversion to `int` and `float`

- To transform a character array to `int` or `float` the functions `atoi` or `atof` can be used
- These functions are defined in the library `cstdlib`:

```
#include <cstdlib> // Required when atoi/atof are used  
  
char str[]="100";  
int num=atoi(str); // num is 100  
  
char str2[]="10.5";  
float num2=atof(str2); // num2 is 10.5
```

Functions > Definition (1/2)

- A function is a block of code that performs a particular task
- They allow us to group common operations in a reusable block
- They can optionally have input parameters and return a value:

```
returnType functionName(parameter1,parameter2,...){  
    returnType ret;  
  
    instruction1;  
    instruction2;  
    ...  
  
    return ret;  
}
```

- A function should not have much code
- Rule of thumb: if you have to do *copy-paste* in the code then you probably need a function

Functions > Definition (2/2)

- You can always find a way to use a single `return` in the body of a function:

```
// Not allowed in Programming 2
bool search(int vec[], int n){
    for(int i=0;i<SIZE;i++){
        if(vec[i]==n)
            return true; // First return
    }
    return false; // Second return
}
```

```
// Alternative version with one return
bool search(int vec[],int n){
    bool found=false;
    for(int i=0;i<SIZE && !found;i++){
        if(vec[i]==n)
            found=true;
    }
    return found; // A single return
}
```

Functions > Parameters (1/2)

- Parameters can be passed by *value* or by *reference* (with `&`)

```
// a and b are passed by value, c by reference  
void function(int a,int b,bool &c){  
    c=a<b; // c keeps this value after the function ends  
}
```

- When passing a parameter by value, the compiler makes a local copy of it within the function
- If the parameter is a large data type, it is more efficient to pass it by reference with `const`:

```
void function(const string &s){  
    // The compiler does not copy s, but if  
    // we try to modify it we get an error  
}
```

- In Programming 2 it is not allowed to pass parameters by reference if they are not going to be modified, except if it is done with `const` as explained above

Functions > Parameters (2/2)

- Arrays and matrices are implicitly passed by reference (it is not necessary to prepend &)
- The name of an array or matrix, without square brackets, contains the memory address where it is stored*
- When passing an array as a parameter, do not include the size of the first dimension in the declaration of the function:

```
void sum(int v[],int m[][SIZE]){  
    // In m the size of the first dimension is not included  
    ...  
}  
...  
// No brackets are used in the call to the function  
sum(v,m);
```

*More information in Unit 4

Functions > Prototypes

- Sometimes it is necessary to use a function before its code appears (or a function whose code is in another module)*
- In these cases the *prototype* of the function must be included:

```
void myFunction(bool, char, double[]); // Prototype

char anotherFunction() {
    double vr[20];
    // myFunction has not yet been declared
    // but we can use it thanks to the prototype
    myFunction(true, 'a', vr);
}

// Declaration of the function
void myFunction(bool exist, char opt, double vec[]) {
    ...
}
```

*More information about creating modules in Unit 5

Structures (1/2)

- A *structure* is a collection of data, which may be of different types
- It is defined with the reserved word `struct`:

```
struct Student{ // Defines a new data type Student
    unsigned int dni;
    float mark;
};
```

- Fields are accessed indicating the name of the variable and the field, separated by a period (member access operator):

```
Student a,b;
a.dni=123133; // Assignment to a field
b=a; // Assignment of a complete structure bit by bit
```

Structures (2/2)

- Struct fields can be initialised at the time of declaration:

```
struct Team{  
    unsigned int id=0;  
    char name[100]="";  
    unsigned int wins=0;  
    unsigned int losses=0;  
    unsigned int draws=0;  
    Player players[20];  
};
```

Enumerated types

- *Enumerated types* can be declared with a set of possible values (*enumerators*):

```
// Create a new data type colour  
enum colour{black,blue,green,red}; // 4 enumerators
```

- Variables of this type can take any value from this set of enumerators:

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

- The values of the enumerated types are converted internally to `int` and vice versa:

```
enum animal{cat,dog,monkey,fish};  
cout << monkey << endl; // Displays 2 on the screen  
// It is the position of monkey in the enumerators
```

Debugging

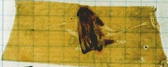
Debugging code in C++ (1/3)

- When there is a runtime error in our code it is sometimes difficult to locate where the error is
- A *debugger* is a program that helps to find and correct runtime errors in the code (*bugs*)

9/9

0800 Antenn started
1000 " stopped - antenn ✓ { 1.2700 9.037 847 025
1300 (032) MP-MC 2.130476415 ~~2.130476415~~ 9.037 846 995 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.

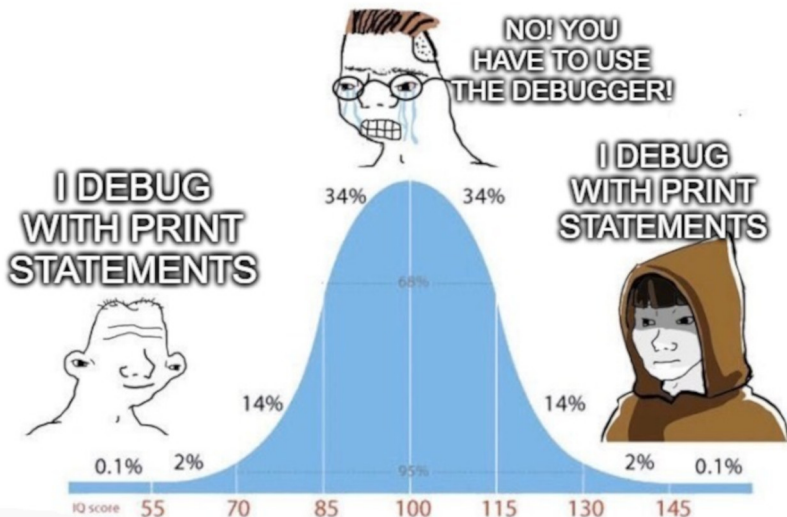
First actual case of bug being found.
16100 Antenn started.
1700 closed down.

Relay 3145
Relay 3376

Debugging code in C++ (2/3)

- A debugger allows us, for example, to execute the code line by line or to see what values the variables have at a certain execution point
- There are numerous programs that facilitate the task of locating errors in the code:
 - *GDB*: starts the program, stops it when asked for, and supervises the content of the variables. If the program has a segmentation fault, it shows the line of code where the problem is
 - *Valgrind*: detects memory errors (access to components outside an array, variables used without initialising, pointers that do not point to an allocated memory area, etc.)
 - Other Linux examples: *DDD*, *Nemiver*, *Electric Fence* and *DUMA*

Debugging code in C++ (3/3)



Exercises

Exercises (1/2)

Exercise 1

Design a function `showAverage` that receives an array of 10 integer values as input, calculates the average value of all of them, and displays on the screen only those values that are above the average.

Exercise 2

Design a function `countVowels` that receives a character array as input and returns how many vowels it contains.

```
countVowels("HOLA") // Returns 2
```

```
countVowels("Hoy es el dia menos pensado") // Returns 10
```

Exercises (2/2)

Exercise 3

Create a record called `Player` with the fields: `name` (array of 50 characters) and `goals` (integer). Create a function that reads the data of 4 players from the keyboard and displays the name and the number of goals of the top scorer.

The keyboard input will have the following format, with each player's data on a single line:

```
Bobby Charlton|34
```

```
Gary Lineker|23
```

```
Miroslav Klose|36
```

```
Lukas Podolski|12
```