@prog2ua

Unit 1: Introduction

Programming 2

Degree in Computer Engineering
University of Alicante
2023-2024

Index

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
...
}
```

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 its 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* 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

- *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:

```c
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...
```

- 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

Increment and decrement operators

- The `++` and `--` operators are used to increase or decrease the value of an integer variable in 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) `-` (substraction)

- 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 lower precedence. The operators of a row have the same precedence

Relational expressions (1/3)

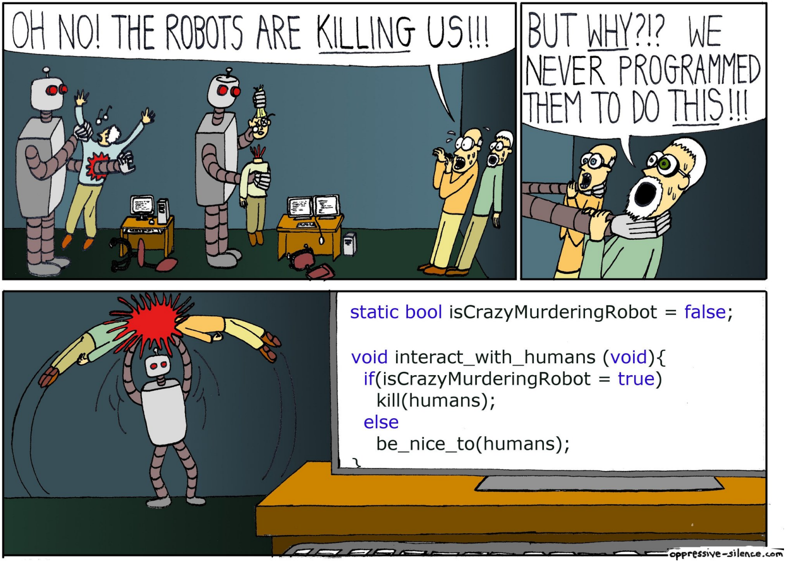
- *Relational expressions* allow comparisons between values
- Operators: $==$ (equal), $!=$ (different), $>=$ (greater or equal), $>$ (greater), $<=$ (less or equal) and $<$ (less)
- If operand types are not equal, they are (implicitly) converted to the most general type:

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

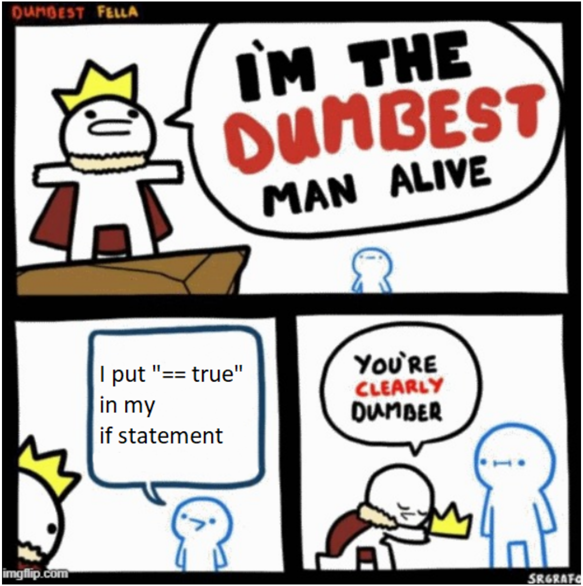
- Operands are grouped two by two from left to right: $a < b < c$ must be coded as `a<b && b<c`
- The result is `0` if the comparison is false and different from `0` if it is true*

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

Relational expressions (2/3)



Expresiones relacionales (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
```

*More details in Unit 2

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){ // If num is lower than 5 then execute this
  cout << "The number is lower than five";
}
else{ // If not, execute this one
  cout << "The number is greater or equal than five";
}
```

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
}
```

- `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` (1/2)

- `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)

Flow control > `switch` (2/2)

- `break` can be used to exit a loop if a specific condition is met:

```
int vec[]={1,2,5,7,6,12,3,4,9};
int i=0;
// Exits the loop when finding 6 in vec
while(i<9){
  if(vec[i]==6)
    break;
  else
    i++;
}
// Equivalent code without using break
bool found=false;
while(i<9 && !found){
  if(vec[i]==6)
    found=true;
  else
    i++;
}
```

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; // May occur a runtime error
          // 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

- *Character arrays* are vectors containing a sequence of characters ending with the null character '\0':*

```
char str[]="hello"; // The compiler adds '\0'
```

"hello" →

h	e	l	l	o	\0

- If not initialised, it is necessary to specify its size:

```
const int SIZE=10;
char str[SIZE]; // Ok
char str2[]; // Compilation error
```

- Remember: `"a"` is a character array and `'a'` is a single character

```
char str[]="a"; // Ok
char str2[]='a'; // Compilation error
```

*More details about character arrays in Unit 2

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)

- More than one `return` can be used in the body of a function if this simplifies the code:

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

```
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
}
```

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

STL vectors (1/2)

- The *Standard Template Library* (STL) is a library of C++ functions
- It provides different data structures and algorithms
- It includes the class `vector`, which allows us to store elements of any type, similarly to regular arrays, but without having to worry about the size:

```
#include <vector> // Include this whenever vector is used
vector<int> vec; // Declares an integer vector
                 // Not necessary to indicate its size
```

- The initial size of a STL vector is `0` and it grows dynamically as required
- Use `push_back` to add elements at the end of the vector:*

```
vec.push_back(12); // Adds 12 at the end of the vector
vec.push_back(8); // Adds 8 after 12
```

*Since it is a class, its methods are called by putting a point after the name of the variable

STL vectors (2/2)

- Access to elements by means of the `[]` operator:

```
vec[10]=23; // Similar to regular arrays
cout << vec[8] << endl;
```

- With `size` we obtain the number of elements of the vector:

```
// Traverse all the elements of the vector
for(unsigned int i=0;i<vec.size();i++){
  vec[i]=10;
}
```

- With `clear` we can delete all the elements and with `erase` a specific one:

```
vec.erase(vec.begin()+3); // Deletes the fourth element
vec.clear(); // Deletes all the elements of the vector
```

- There are many other functions to work with STL vectors*

*More information at `http://www.cplusplus.com/reference/vector/vector/`

Structures

- 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
  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
```

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
```

Arguments (1/4)

- The *arguments* of a program are used to provide information (usually options) from the command line
- Their use is very common and allows us to modify the behaviour of the program:

Terminal

```
$ ls          // Lists the files in a directory
$ ls -a       // Lists also hidden files (option "-a")
$ ls -a -l    // Adds extra information from each file (option "-l")
```

Arguments (2/4)

- `main` is a function and as such it can receive two parameters: `argc` and `argv`
- These parameters allow us to manage arguments passed to the program through command line:

```
// Always in this order
int main(int argc,char *argv[]){
  ...
  return 0;
}
```

- `int argc`: number of arguments passed to the program (including also the program name)
- `char *argv[]`: array of character arrays with the arguments passed to the program

Arguments (3/4)

- Example of use:

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

Terminal

```
$ ./myProgram -a -h X     // Example of call with three parameters
Arg. 0 : ./myProgram
Arg. 1 : -a
Arg. 2 : -h
Arg. 3 : X
```

- Arguments do not have to start with a hyphen (–) but it is quite a common practice

Argument (4/4)

- It seems easy to manage arguments, but sometimes things can become complicated
- Users do not always use the same order when introducing arguments:

Terminal

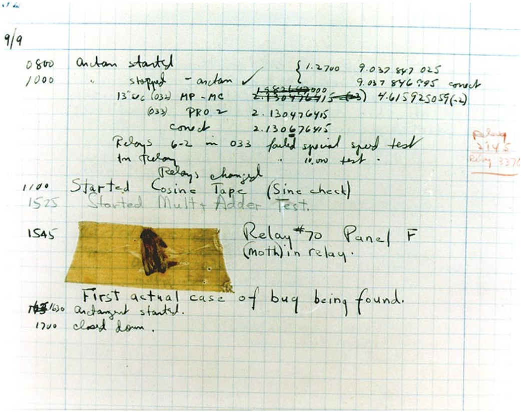
```
$ g++ -Wall -o prog prog.cc -g
$ g++ -g -Wall prog.cc -o prog
```

- There may be errors in the command-line parameters and help information should be shown to the user
- It is recommended to use a dedicated function to manage the arguments

Debugging

Debugging code in C++ (1/3)

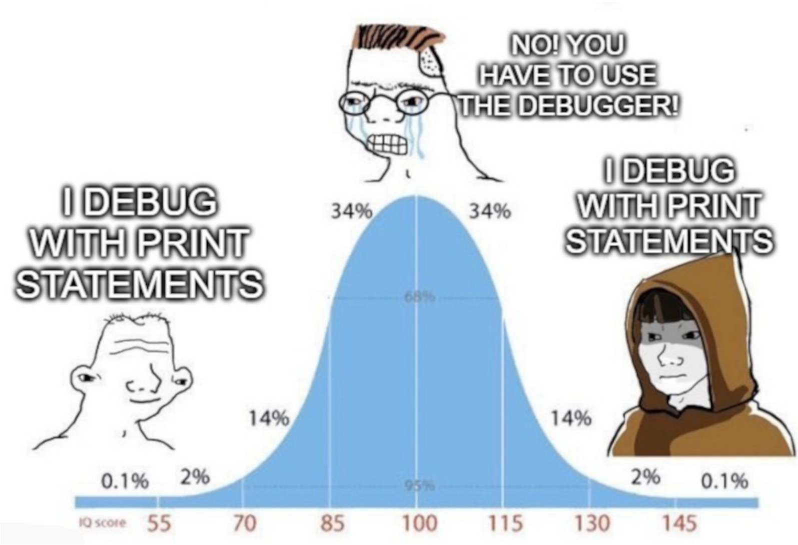
- When there is a runtime error in our code it is sometimes difficult to locate where the error is
- A *debugger* is a program that helps to find and correct runtime errors in the code (*bugs*)



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

Exercise 1

Implement a program containing a function with the following prototype: `int primeNumber(int n)`. This function will return the n-th prime number. The program must print prime numbers on the screen with the following options:

- `-L` prints each number on a separate line (by default they are all printed on the same line)
- `-N n` prints the `n` first prime numbers (10 by default)

Execution examples:

Terminal

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