These strange mystical beasts
Compiler, linker, preprocessor, object code, debugger and makefiles.

Speaker: Carlos Beltrán-González

October 18, 2007

Introduction

In this seminar we will introduce some of these mystical beasts that we use when programming. These are basically, the compiler, the preprocessor, the linker and the debugger. The main goals are:

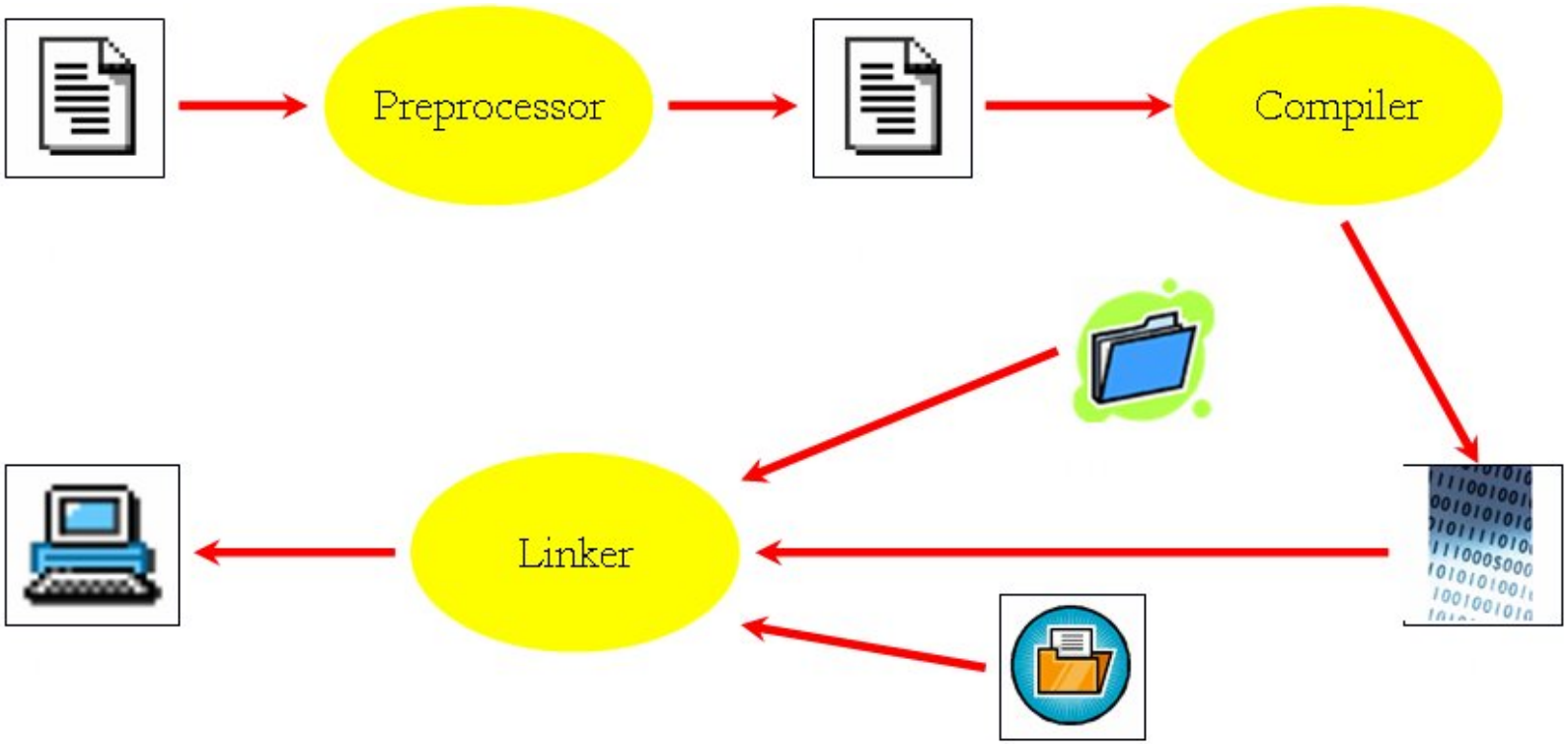
1. To understand completely the tools involved in the creation of programs (executables)
2. To become completely autonomous in the use of these tools

Introduction

In this seminar we will introduce some of these mystical beasts that we use when programming. These are basically, the compiler, the preprocessor, the linker and the debugger. The main goals are:

1. To understand completely the tools involved in the creation of programs (executables)
2. To become completely autonomous in the use of these tools

From the code to the executable (a bird's eye)



Source Code

The source code (sometimes called just *source*) is a set of instructions,data, used to implement an algorithm in machine code. This, to build a program that can be executed by the computer.

Object Code

- The object code (or object file) is the transformation/translation of source code into *machine code* (bynary). This object code is only understandable by the computer.
- The object codes are normally grouped in *libraries* that contain a number of *functions* intimately related. For example, a set of mathematical operations.
- The object code consist in a executable code plus information that permits the linker to join the object code with other object codes to generate a working program.

Preprocessor

What is the preprocessor

A preprocessor is a program (or part of a program) that makes textual substitutions in the source code of a program.

The more common substitutions are:

- Macro's expansions
- Inclusion of other files
- The conditional code selection

The instructions inserted in the source code to be elaborated by the preprocessors are called *directives*; in the C/C++ preprocessor, these directives are the lines starting with the character #.

Preprocessor

What is the preprocessor

A preprocessor is a program (or part of a program) that makes textual substitutions in the source code of a program.

The more common substitutions are:

- Macro's expansions
- Inclusion of other files
- The conditional code selection

The instructions inserted in the source code to be elaborated by the preprocessors are called *directives*; in the C/C++ preprocessor, these directives are the lines starting with the character #.

Preprocessor

What is the preprocessor

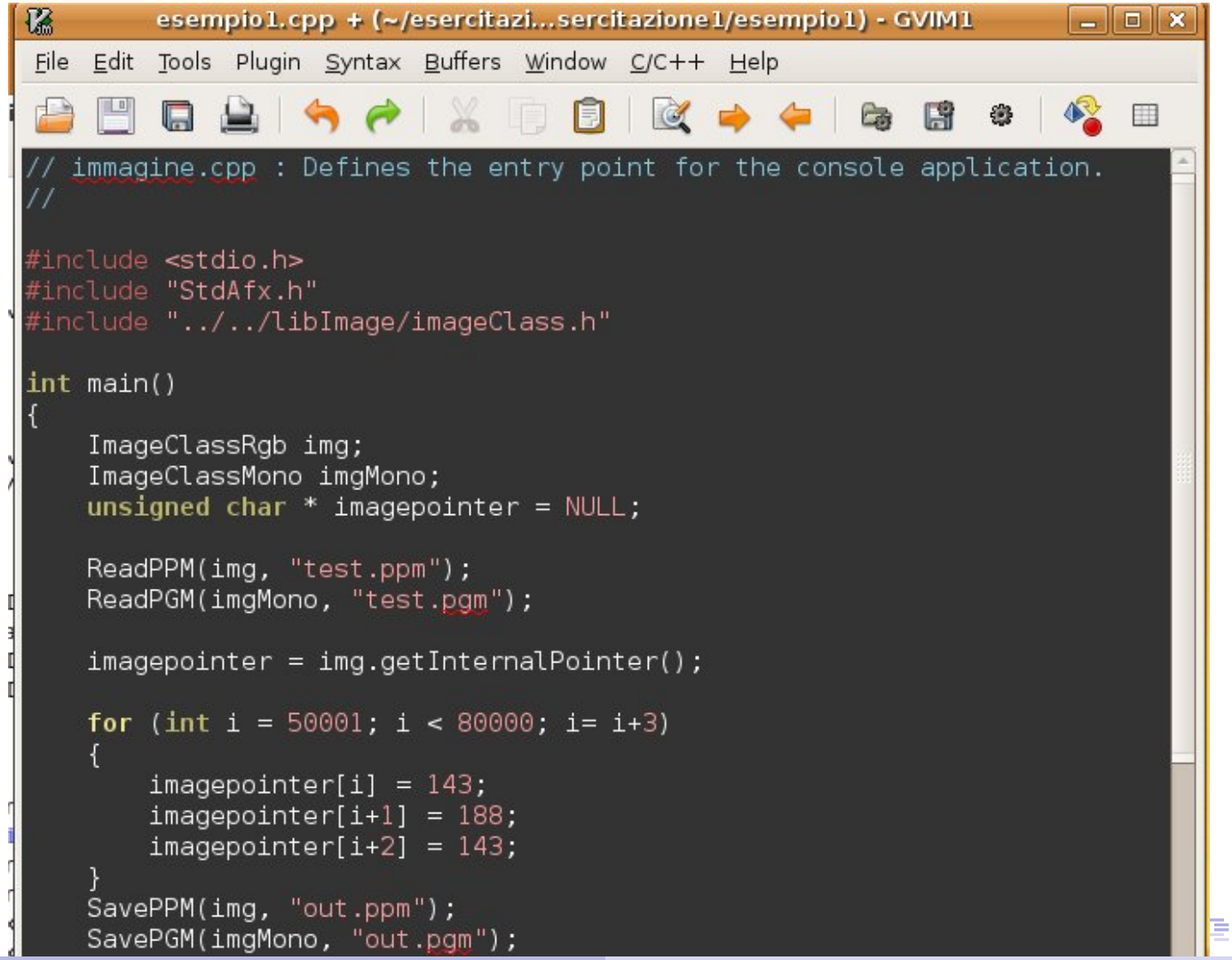
A preprocessor is a program (or part of a program) that makes textual substitutions in the source code of a program.

The more common substitutions are:

- Macro's expansions
- Inclusion of other files
- The conditional code selection

The instructions inserted in the source code to be elaborated by the preprocessors are called *directives*; in the C/C++ preprocessor, these directives are the lines starting with the character #.

An example of preprocessor lines



The compiler

What is a compiler?

A compiler is a "translator" program in charge of producing the *object code* (in machine language) from a source code written in a given programming language (in our case C++). The source code is considered to be in a more human readable level. This process translation/transformation process is called *compilation*

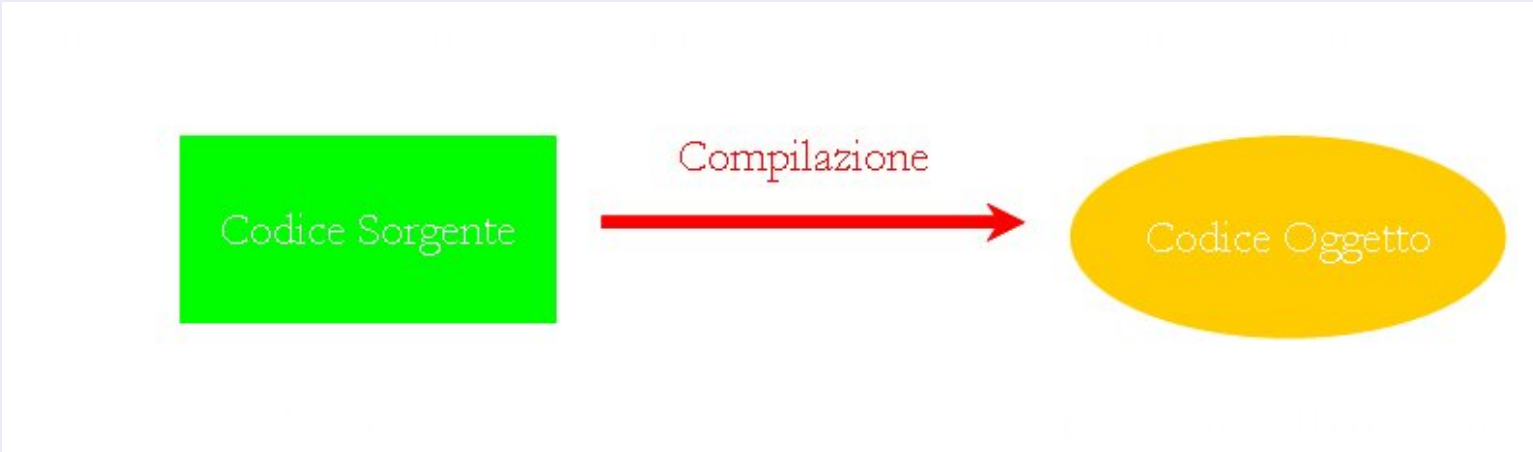
What does the compiler do?

The compiler

What is a compiler?

A compiler is a "translator" program in charge of producing the *object code* (in machine language) from a source code written in a given programming language (in our case C++). The source code is considered to be in a more human readable level. This process translation/transformation process is called *compilation*

What does the compiler do?



How does the compiler work?

Tasks

- *Lexical Analysis:* The program is subdivided in known keywords of the programming language, for example, *while*, *for*, constants and varibles names; these keywords received the name of *tokens*.
- *Syntantic and Semantic Analysis*: In this phase, the compiler creates a syntactic tree using the *tokens* extracted in the previous phase. The tree is built based on a given *grammar* that defines the possible sequences of *tokens* accepted by the programming language.
- *Code Generation* From the syntactic tree and from the symbols tables the information is extracted to generate the executable code.

Can the compiler help the programmer?

Compiler tip!

Generally, the compiler is capable of recognising some types of errors in the source code and, in some cases, suggest the ways of fixing the code.

Compiler errors

There exits two types of errors: *errors* and *warnings*.

Errors

The presence of errors blocks the invocation of the compiler. Actually, the source code is not compiled at all.

Warnings

- The compilation proccess in NOT blocked
- Inform the programmer about ambiguity in the code
- They can represent non desirable situations as logic errors
- Some warning errors can be ignored if the compiler resolves correctly the ambiguity.

Compiler errors

There exits two types of errors: *errors* and *warnings*.

Errors

The presence of errors blocks the invocation of the compiler. Actually, the source code is not compiled at all.

Warnings

- The compilation proccess in NOT blocked
- Inform the programmer about ambiguity in the code
- They can represent non desirable situations as logic errors
- Some warning errors can be ignored if the compiler resolves correctly the ambiguity.

The Linker

What is the linker?

The *linker* is a program that takes one or more *modules* containing object codes and puts them together in a single executable program.

The modules contain the machine code plus information useful to the linker; mainly *symbols definitions*:

- *Symbols exported or defined* These are functions or variables that can be found in the object code and that can be available for other modules.
- *Symbols not defined or imported* These are functions or variables that are used or called from within a particular module but are not defined internally because defined in other modules or in the system's libraries.

The Linker

What is the linker?

The *linker* is a program that takes one or more *modules* containing object codes and puts them together in a single executable program.

The modules contain the machine code plus information useful to the linker; mainly *symbols definitions*:

- *Symbols exported or defined* These are functions or variables that can be found in the object code and that can be available for other modules.
- *Symbols not defined or imported* These are functions or variables that are used or called from within a particular module but are not defined internally because defined in other modules or in the system's libraries.

Linker errors

Why linker errors?

Even if our source code didn't produce errors when compiling, errors may appear when linking. Linking errors can block the process of building the executable code.

Possible linker errors

- *Undefined Symbol* The linker is not capable of resolving all the not defined symbols. In practice, this means that it can not find the module that contains the definitions of a variable or the implementation of a function that have been used in the source code.
- *Ambiguous Symbol* The linker can not distinguish which among the exported symbols present in the modules corresponds to a given imported symbol.

Linker errors

Why linker errors?

Even if our source code didn't produce errors when compiling, errors may appear when linking. Linking errors can block the process of building the executable code.

Possible linker errors

- *Undefined Symbol* The linker is not capable of resolving all the not defined symbols. In practice, this means that it can not find the module that contains the definitions of a variable or the implementation of a function that have been used in the source code.
- *Ambiguous Symbol* The linker can not distinguish which among the exported symbols present in the modules corresponds to a given imported symbol.

The debugger

What is the debugger

The debugger is a program that give you *runtime* information of a running program. The program has to be compiled in *debug mode* thus the debugger can get the information needed to inform the programmer about the flow of the program

What can a debugger do?

With a debugger the programmer can basically perform the next task:

- Follow the flow of the program step by step
- Visualize memory states, variables values, conditions status...etc
- Change runtime context in the case of different threads

The debugger

What is the debugger

The debugger is a program that give you *runtime* information of a running program. The program has to be compiled in *debug mode* thus the debugger can get the information needed to inform the programmer about the flow of the program

What can a debugger do?

With a debugger the programmer can basically perform the next task:

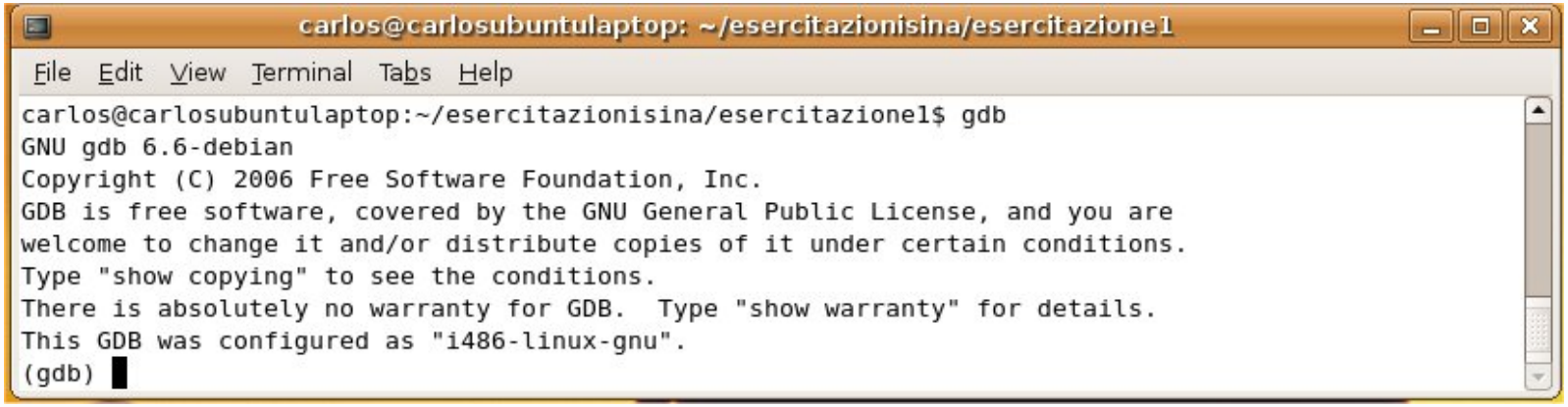
- Follow the flow of the program step by step
- Visualize memory states, variables values, conditions status...etc
- Change runtime context in the case of different threads

The debugger in Unix/Linux

GDB

The debugger is called "gdb". It is a textual debugger. Difficult to use with a long learning curve.

GDB screenshot

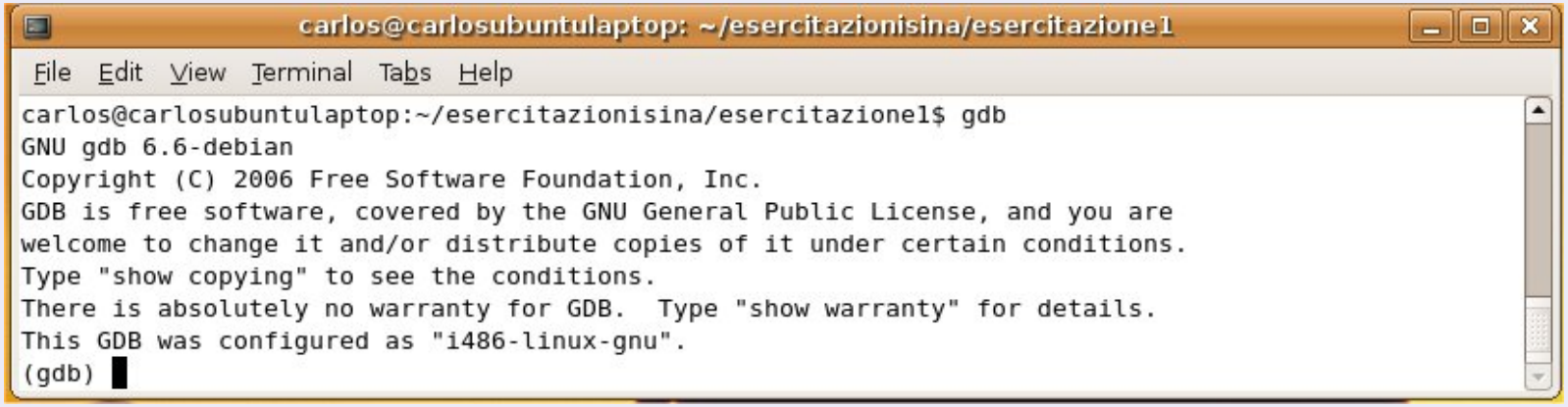


The debugger in Unix/Linux

GDB

The debugger is called "gdb". It is a textual debugger. Difficult to use with a long learning curve.

GDB screenshot



The Windows debugger

Microsoft Visual Studio

The family of Visual Studio IDE's provides one of the windows based more powerfull debuggers in the market.

An example of Compilation in Linux environment

We will show step by step how to create a library and link it to an executable. A *library*, as explained before, is nothing more than a convenient grouping of one or more object modules.

Creating an object code

```
g++ -c imageClass.cpp -o imageClass.o
```

An example of compilation in Linux environment

Creating a library

```
ar rcs libImage.a imageClass.o
```

Linking the library with an example file

```
g++ -static esempio.cpp -L. -lImage -o executable_name
```

An example of compilation in Linux environment

Creating a library

```
ar rcs libImage.a imageClass.o
```

Linking the library with an example file

```
g++ -static esempio.cpp -L. -lImage -o executable_name
```

Putting a project together

Software complexity

Some software projects can arrive to have thousands of files and hundreds of libraries (i.e. an operating system). Some tools have been developed to assist the developer in structuring and manage such a huge quantity of files.

Tip!

Today, these tools are used independently of the software project complexity. They facilitate in any case the process of compilation, linking...etc

Putting a project together

Software complexity

Some software projects can arrive to have thousands of files and hundreds of libraries (i.e. an operating system). Some tools have been developed to assist the developer in structuring and manage such a huge quantity of files.

Tip!

Today, these tools are used independently of the software project complexity. They facilitate in any case the process of compilation, linking...etc

Tools used to group code (development level)

Existing tools

Basically, these two types of tools:

- Makefile like scripting
- IDE's environments

Makefiles

What are Makefiles?

In Unix/Linux environments the so called "makefiles" are used to group code. Other tools can be used to do this job like "CMake" or "Ant". In general, all these tools are *script* like files that *describe* the software structure of a project

Makefiles facilitate

- automation of the compilation, linking process.
- Setting the compiler, linker tags.
- Make conditional compilation/linking
- Detect changes in source code and compile only *parts* of code
- Automatically handling dependencies and libraries.

Makefiles

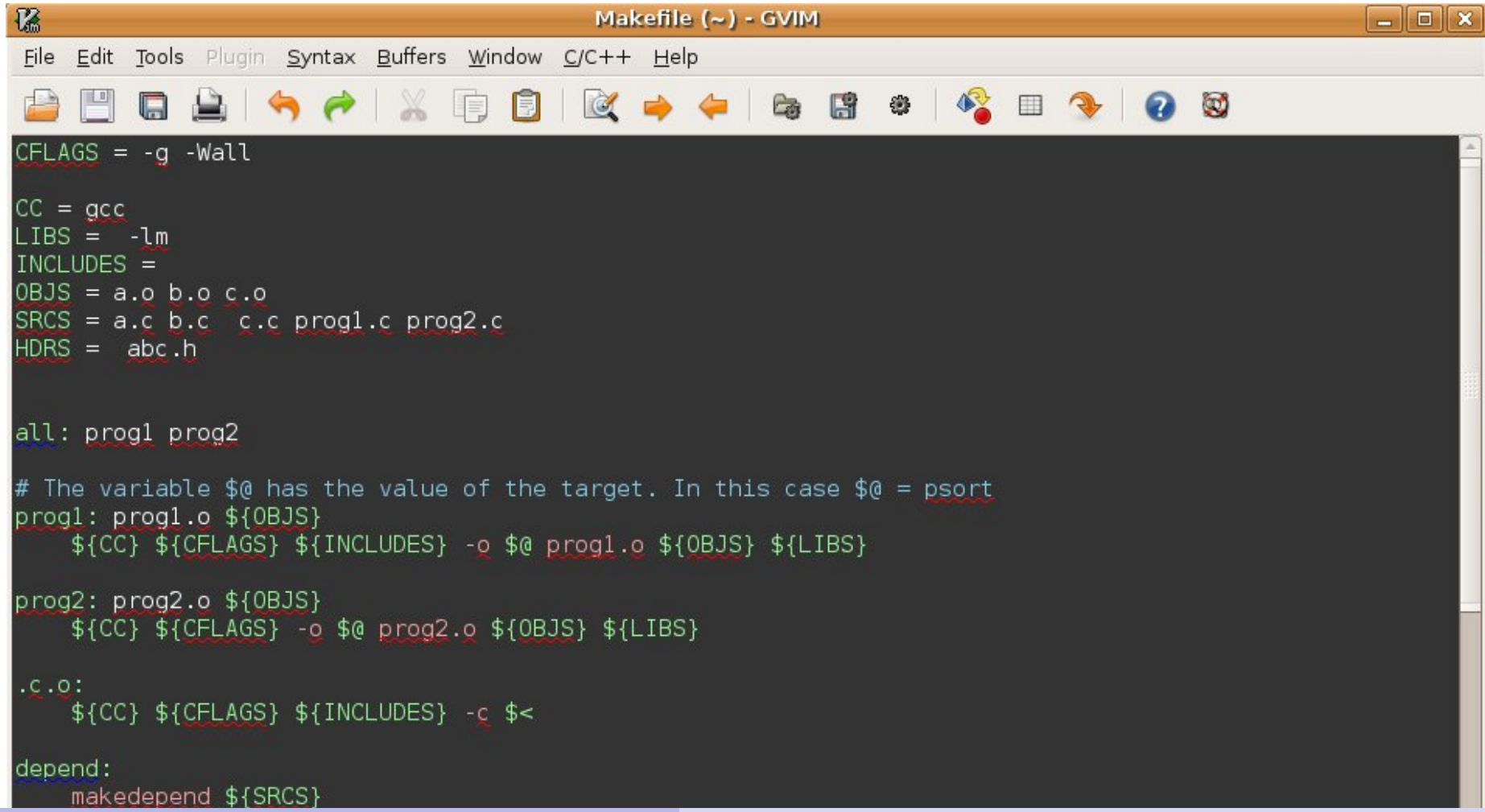
What are Makefiles?

In Unix/Linux environments the so called "makefiles" are used to group code. Other tools can be used to do this job like "CMake" or "Ant". In general, all these tools are *script* like files that *describe* the software structure of a project

Makefiles facilitate

- automation of the compilation, linking process.
- Setting the compiler, linker tags.
- Make conditional compilation/linking
- Detect changes in source code and compile only *parts* of code
- Automatically handling dependencies and libraries.

Makefile Example



Development environments

IDEs

IDE stands for Integrated Development Environment

Available IDEs are:

- Microsoft Visual Studio (Visual Basic, C++, C#)
- KDevelop (Linux-KDE)
- Eclipse (All platforms)

Development environments

IDEs

IDE stands for Integrated Development Environment

Available IDEs are:

- Microsoft Visual Studio (Visual Basic, C++, C#)
- KDevelop (Linux-KDE)
- Eclipse (All platforms)

What IDEs do

They provide

- Visual control of the software project.
- Editor, visual classes manager
- Integrated debugger
- Integration with other tools or software modules

Some editors in Linux

- gedit (recomended) easy to use
- emacs (hard to use professional editor)
- gvim (harder to use professional editor)modal

The end

Questions to cbeltran@dist.unige.it