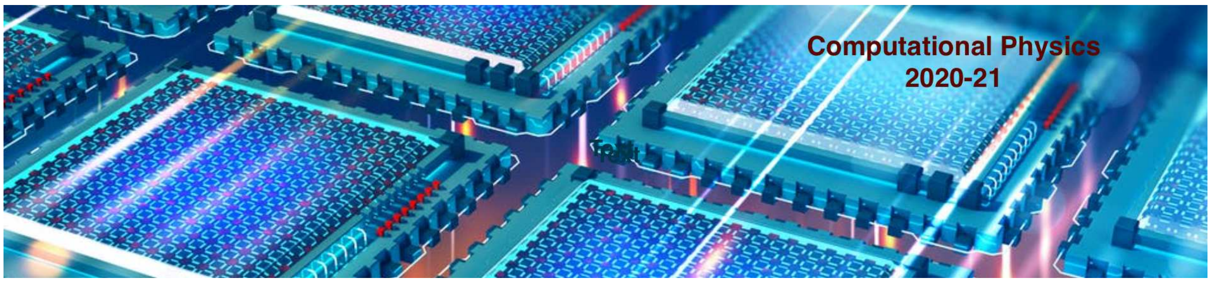
Computational Physics

numerical methods with C++ (and UNIX)

2020-21



Fernando Barao
Instituto Superior Tecnico, Dep. Fisica
email: fernando.barao@tecnico.ulisboa.pt

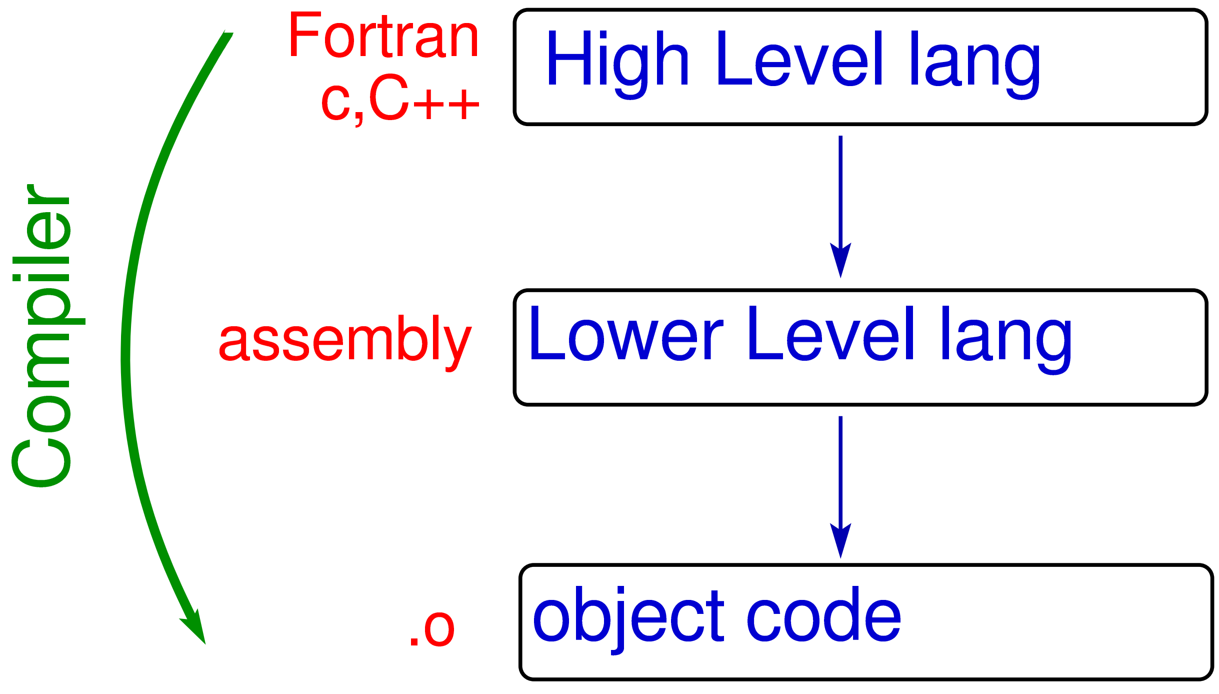
Computational Physics
Compiling a C++ program

Fernando Barao, Phys Department IST (Lisbon)

Computer programming

- ✔ Symbolic languages use words (“add”, “move”, ...) instead of operation codes
- ✔ High-level symbolic languages:
  - ▶ FORTRAN FORmula TRANslator mid 1950’s
  - ▶ BASIC Beginner’s All-purpose Symbolic Instruction Code mid 1960’s
  - ▶ PASCAL early 1970’s
  - ▶ C mid 1970’s
  - ▶ C++, Java, ... mid 1980’s on
- ✔ C and C++ allow the manipulation of bits and bytes and memory addresses (some people tag it as mid-level languages)
- ✔ Other languages like Mathematica, Matlab or Maple: very rapid coding up but...code is interpreted (slower)
- ✔ The lowest level symbolic language is called the *assembly language*
- ✔ The **assembler** program translates the assembly into **machine code (object code)** that will be understood by the CPU

Computer programming



Creating an executable

- ✔ An executable file contains binary code encoding machine-language instructions
- ✔ To create it, we need to start by writing a program in a symbolic language, **the source code**
  - ► use some Unix editor like, **pico, gedit, emacs, atom, sublime**
- ✔ Next, we produce the object code, by compiling the source code and eventually linking with other pieces of code located in libraries or being compiled at the same time
  - ► compilers: **C++** → **g++**, **c** → **gcc**, **FORTRAN** → **gfortran**
  - ► the compiler assigns memory addresses to variables and translates arithmetic and logical operations into machine-language instructions
- ✔ The object code is loaded into the memory (RAM) and it is runned by the CPU (no further need of the compiler)
  - ► the object files are specific to every CPU and are not necessarily portable across different versions of the operating system

5-2

Computational Physics

aulas práticas

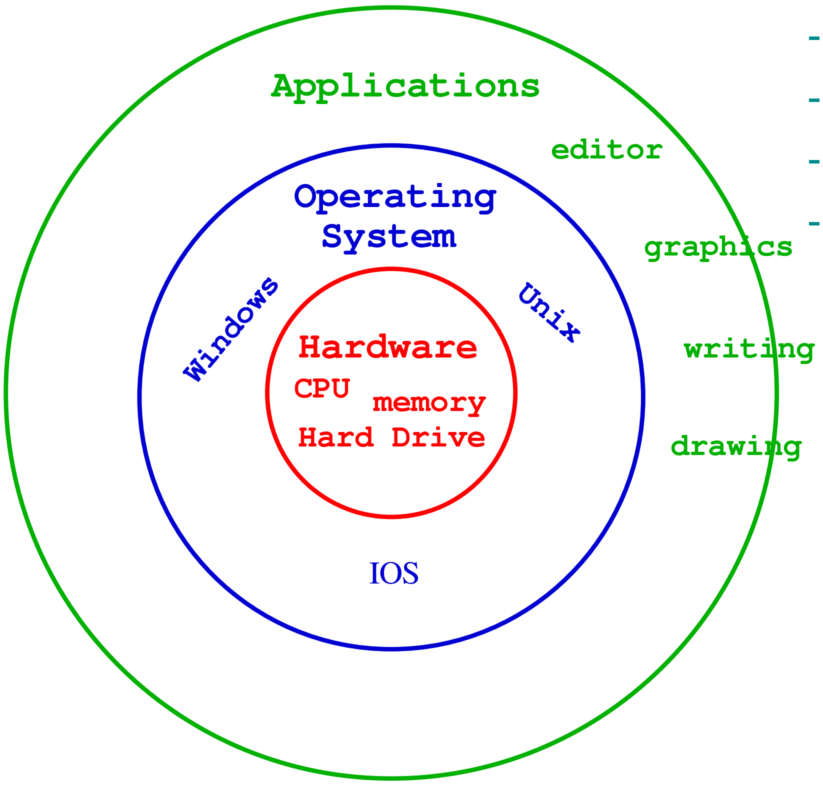
Computational Physics

Operating systems

UNIX (linux)

Fernando Barao, Phys Department IST (Lisbon)

Operating systems



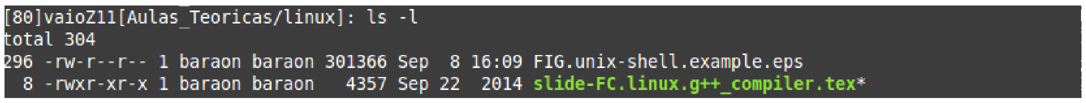
Many linux distributions:
- Debian
- Ubuntu (& derivatives)
- Linux Mint
- Fedora

Linux installation

- ✔ Directly in the computer using one of the many available flavors: ubuntu, fedora, ...
  ubuntu: http://www.ubuntu.com/desktop
  fedora: http://fedoraproject.org
  linuxMint: http://www.linuxmint.com
- ✔ Through a virtual machine
  virtual box: www.virtualbox.org
  VMware: available at IST

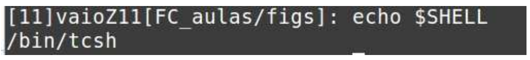
UNIX shell

- ✔ the shell, the command line interface, is a program like any other one
- ✔ it takes commands from the user and transmit to the operating system the corresponding actions
- ✔ most shell commands are actually small programs, accepting options and arguments
  example: ls -l <dirname>



```
[80]vaioZ11[Aulas_Teoricas/linux]: ls -l
total 304
296 -rw-r--r-- 1 baraon baraon 301366 Sep  8 16:09 FIG.unix-shell.example.eps
  8 -rwxr-xr-x 1 baraon baraon   4357 Sep 22  2014 slide-FC.linux.g++_compiler.tex*
```

- ✔ unix shells
  bash: the default shell on most linux systems
  csh: C shell (similar to C programming language)
  ksh: korn shell
  tcsh: enhanced but compatible with C shell
- ✔ to find your current shell: *echo $SHELL*



```
[11]vaioZ11[FC_aulas/figs]: echo $SHELL
/bin/tcsh
```

- ✔ to find your available shells: *cat /etc/shells*

Computational Physics

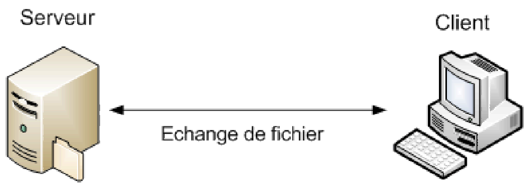
SVN

managing file versions

Fernando Barao, Phys Department IST (Lisbon)

SVN introduction

- ✔ the subversion (SVN) management system is used for for *file version control*
- ✔ Software projects having several cooperating developers need a common source repository (SVN server) keeping a sinchronized copy of all file versions
- ✔ source control tools track all prior versions of all files, allowing developers to “time travel” backward and forward in their software to determine when and where bugs are introduced
- ✔ these tools also identify conflicting simultaneous modfications made by two (poorly-communicating) team members, forcing them to work out the correct solution (rather than blindly overwriting one or the other original submission)



SVN directories organization

- ✔ SVN project: **FC**
- ✔ checkout:
  svn co –username=<numero> svn://fcomp.tecnico.ulisboa.pt/FC
- ✔ directories and files

```
FC/2020/A01/96000 .......... pasta de trabalho provada do aluno
         |-----> grupo
FC/2020/DOCs ............... eventual documentação
```

- ✔ group directories

```
FC/2020/A01/
           main/ ........ directory containing main programs
                 P01.C    solution of problem 1
                 P02.C    solution of problem 2
                 (...)
            src/ ........ directory containing classes and header files
                 point.C  class point code
                 point.h  class point header
            lib/ ........ group libraries
            bin/ ........ binaries (.o) and executables
           Makefile
```

SVN operations

- ✔ SVN project: **FC**
- ✔ SVN server: **fcomp.tecnico.ulisboa.pt**
- ✔ clients: **TortoiseSVN (windows), terminal (mac, linux)**
- ✔ **checkout:** get a local copy of the server repository to a local repository directory

```
svn co --username=<numero> svn://fcomp.tecnico.ulisboa.pt/FC <localdir>
```

- ✔ the timeline of the project, i.e. the versions, is characterized by revision numbers; for getting a local copy corresponding to a given revision number:

```
svn co -r <revison number> svn://fcomp.tecnico.ulisboa.pt/FC
```

- ✔ getting a detailed information of file changes in the server

```
svn log -v
```

- ✔ making a new directory in local copy

```
svn mkdir <dir name>
```

SVN operations (cont.)

- ✔ check local copy status (which files have been modified - M, are not under control of the subversion tool, etc); do it before a commit!

```
svn status --verbose
```

- ✔ adding files to local SVN repository

```
svn add <file name>
```

- ✔ removing files from local SVN repository

```
svn delete <file name>
```

- ✔ renaming files in local copy

```
svn mv <file name> <new file name>
```

- ✔ synchronizing the local copy to the server

```
svn ci -m "some comments about the changes you made"
```

SVN operations (cont.)

- ✔ updating the local copy (sinchronize the local copy from server)

```
$ svn update
$ svn up                            #short command
$ svn update -r <revision number> #update to this revision number
```

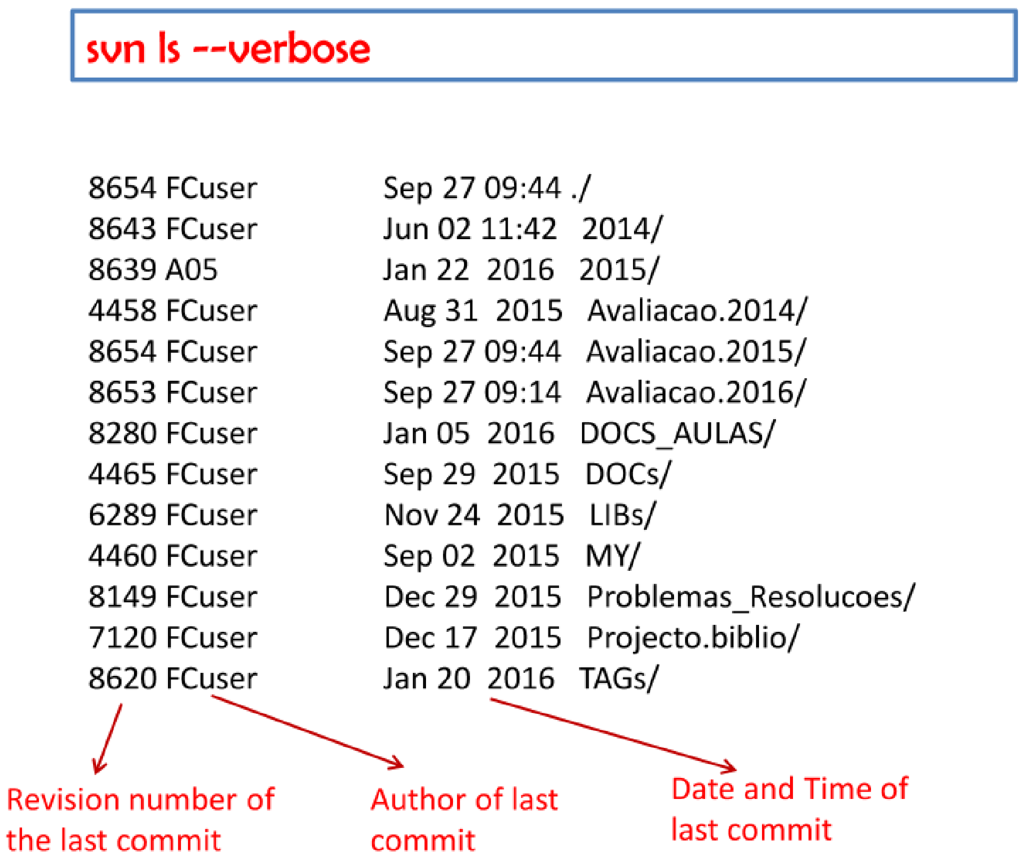
- ✔ get information about SVN server

```
$ svn info
```

- ✔ get list of files in svn server

```
$ svn ls
```

SVN operations (cont.)



```
svn ls --verbose
```

```
8654 FCuser        Sep 27 09:44 ./
8643 FCuser        Jun 02 11:42  2014/
8639 A05           Jan 22  2016  2015/
4458 FCuser        Aug 31  2015  Avaliacao.2014/
8654 FCuser        Sep 27 09:44  Avaliacao.2015/
8653 FCuser        Sep 27 09:14  Avaliacao.2016/
8280 FCuser        Jan 05  2016  DOCS_AULAS/
4465 FCuser        Sep 29  2015  DOCs/
6289 FCuser        Nov 24  2015  LIBs/
4460 FCuser        Sep 02  2015  MY/
8149 FCuser        Dec 29  2015  Problemas_Resolucoes/
7120 FCuser        Dec 17  2015  Projecto.biblio/
8620 FCuser        Jan 20  2016  TAGs/
```

Revision number of the last commit
Author of last commit
Date and Time of last commit

SVN operations: conflicts

1. João and Pedro make a checkout of a file **t.txt** version *revision 1*
2. They make independently changes in the file
3. João commit its file and create *revision 2*
4. Pedro after having changed its file, try to commit it and a conflict arises! (we need allways to have un updated copy!!)

```
$ svn commit -m ``I made the following changes on t.txt: ...''
Sending     t.txt
Transmitting file data .svn: Commit failed (details follow):
svn: Out of date: '/myproject/t.txt'
```

5. Subversion has detected that the file you want to commit has changed since you last updated it
what can we do for solving the conflict? You need to update the file again...

SVN operations: conflicts (cont.)

- **automatic merging**
  if the changes are "independent"(not on the same place of the file) subversion do the merge for you

```
$ svn update
```

  after this operation the merge of the several modifications is done

- **manual merging**

```
$ svn update

Conflict discovered in 't.txt'.
Select: (p) postpone, (df) diff-full, (e) edit,
(mc) mine-conflict, (tc) theirs-conflict,
(s) show all options:

answer: p (postpone)

C t.txt
Updated to revision 6.
```

SVN operations: conflicts (cont.)

- update failed and the **C** means there is a conflict in **`t.txt`**! In the local copy several files were created:

```
t.txt.mine (my file that I tried to commit)
t.txt.r4 (file version before my change)
t.txt.r5 (file version after the changes of Mr. X)

t.txt (this file contains all changes and must to be edited!)

<<<<<<<<< .mine
What I introduced
=========
What is now here on new version 6
>>>>>>>>> .r5
the remaining changes on version 5
```

SVN operations: conflicts (cont.)

1. solve the conlict by scraping my changes and go on with João version

```
$ svn revert t.txt
Reverted 't.txt'
$ svn update t.txt
At revision 6.
```

2. Keep my changes, and dump whatever João did

```
$ cp t.txt.mine t.txt
$ svn resolved t.txt
Resolved conflicted state of 't.txt'
```

3. edit **t.txt** file

```
# remove <<<<<, >>>>>>>, ======= marks!
svn resolved t.txt
```

✔ be sure to have the latest version and commit your changes

```
svn update
svn ci -m "conflict solved!"
```

SVN operations: conflicts (cont.)

✔ how to avoid conflicts? **lock** the file you are changing...

```
svn update
svn lock t.txt
```

returns: *'t.txt' locked by user 'Joao'*

✔ now if Mr. Y asks for locking the file (because he wants also change it...) he gets *svn: warning: Path '/t.txt' is already locked by user 'Joao' in filesystem ...*

✔ do not forget, after your changes ended, to **unlock** the file!

```
svn unlock t.txt
```

✔ **check the differences in detail**

```
svn diff t.txt
```

✔ update my local copy to a given revision number

```
svn update -r <number>
```

SVN operations (cont.)

✔ recover a file (*ta.txt*) deleted in a previous revision (deleted at revision 7)

```
svn copy -r 6 svn://fcomp.ist.utl.pt/FC/ta.txt ta.txt
```

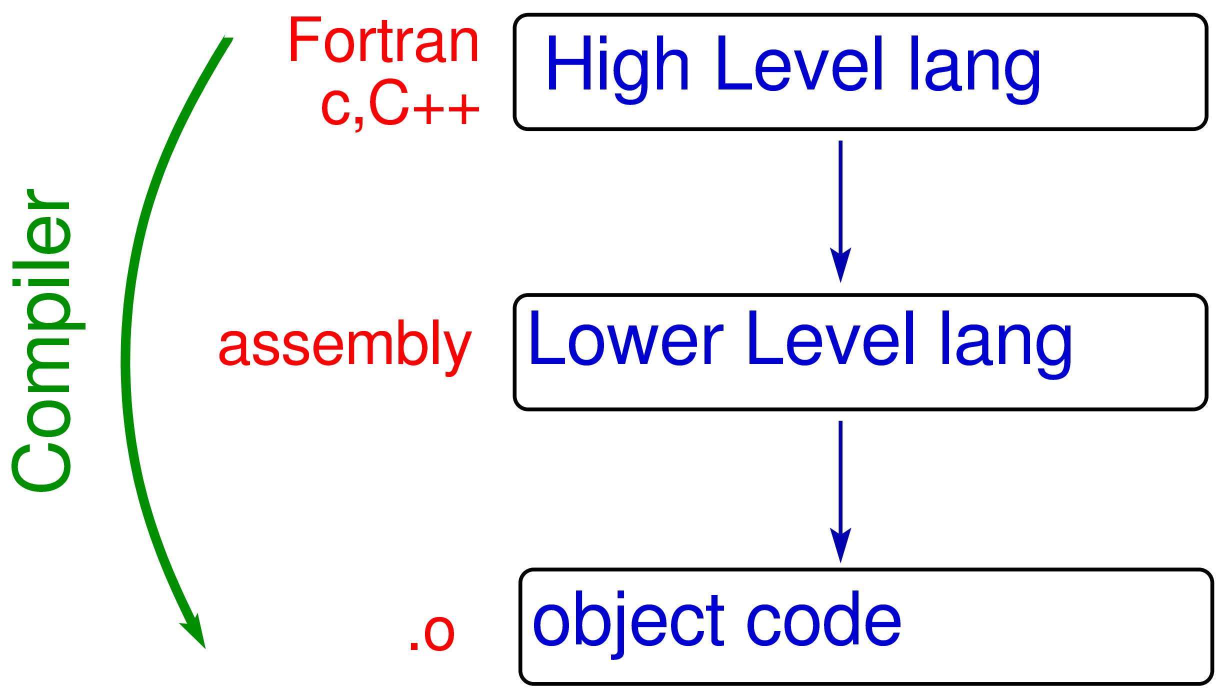
Computational Physics
Compiling a C++ program

Fernando Barao, Phys Department IST (Lisbon)

Computer programming

- ✔ Symbolic languages use words (“add”, “move”, ...) instead of operation codes
- ✔ High-level symbolic languages:
  - ► FORTRAN FORmula TRANslator mid 1950’s
  - ► BASIC Beginner’s All-purpose Symbolic Instruction Code mid 1960’s
  - ► PASCAL early 1970’s
  - ► C mid 1970’s
  - ► C++, Java, ... mid 1980’s on
- ✔ C and C++ allow the manipulation of bits and bytes and memory addresses (some people tag it as mid-level languages)
- ✔ Other languages like Mathematica, Matlab or Maple: very rapid coding up but...code is interpreted (slower)
- ✔ The lowest level symbolic language is called the *assembly language*
- ✔ The **assembler** program translates the assembly into **machine code (object code)** that will be understood by the CPU

Computer programming



Creating an executable

- ✔ An executable file contains binary code encoding machine-language instructions
- ✔ To create it, we need to start by writing a program in a symbolic language, **the source code**
  - ► use some Unix editor like,
    pico, gedit, emacs, atom, sublime
- ✔ Next, we produce the object code, by compiling the source code and eventually linking with other pieces of code located in libraries or being compiled at the same time
  - ► compilers: **C++** → **g++**, **c** → **gcc**, **FORTRAN** → **gfortran**
  - ► the compiler assigns memory addresses to variables and translates arithmetic and logical operations into machine-language instructions
- ✔ The object code is loaded into the memory (RAM) and it is runned by the CPU (no further need of the compiler)
  - ► the object files are specific to every CPU and are not necessarily portable across different versions of the operating system

g++ compiler

compiler flags that can be used in the compilation process:

generic

-c - output an object file (.o)
-o <name> - name of the output file
-g - turn on debugging (so GDB gives more friendly output)
-I<include path> - specify an include directory
-L<library path> - specify a lib directory
-l<library> - link with library lib<library>.a

warnings

-Wall - turns on most current warnings
-Wextra - turns on extra warnings (indicates unitialized variables)
-pedantic - it checks if it is C++ standard code
-Wfloat-equa - checks if one tests an equality between reals (common error)
-Woverloaded-virtual - message signaling that virtual function implemented is different from base class

g++ compiler (cont.)

compiler flags that can be used in the compilation process:

-Wshadow - two similar variables in the same block code

variables conversion

-Wconversion - warns when automatic variable conversions are done

-Wdouble-promotion - warns when a float is converted into double

-Wold-style-cast - warns when conversion a la "c"is done (C++: static_cast<type>())

optimization

Optimization saves disk space because the program size will be smaller and and saves CPU time (less time to run)

-O1, -O2 - turn on optimizations

g++ compiler (cont.)

produce object code and check syntax of test.C (.o) (-v verbose)

```
> g++ -v -c test.C
```

produce executable code of test.C (.exe)

```
> g++ -o text.exe test.C
```

optimizating compiled code and count nb of bytes : (-O0= no optimization, -O1, -O2)

```
> g++ -O1 -o text.exe test.C
> wc -c test.exe # count bytes
```

compilink + linking for debugging (no optimization and good code)

```
> g++ -g -Wall -Wextra -o text.exe test.C
```

g++ compiler (cont.)

compilink + linking with static libraries (libm.a)

```
> g++ -o text.exe test.C -L/usr/local/LIB -lm
```

code macro definitions (#define BUFFER 512) can be defined at the command line

```
> g++ -DBUFFER=512 -o test.exe test.C
```

display de preprocessed version of your C++ code

```
> g++ -E test.C > test.i
```

COMMON ERRORS TO AVOID!!!!!!! WARNING!

```
> g++ -o test.C test.C #program disappears
> g++ -E test.C > test.C
```