LAL-Info Documentation

Release 1.14-npac

LAL

February 12, 2014

spot

The primary purpose of this lecture is to explore various computer science technologies which are exercized through:

- the conception,
- the building,
- and the development

of a complex application like those developed in an actual scientific collaboration.

Throughout this lecture, the following items will be investigated:

- configuration management tools for software building (how to structure a project and automate its construction),
- version control management tools to track and log modifications to a piece of code or a whole project,
- automated documentation generator,
- object oriented programming and leveraging `C++` and its `Standard Template Library`
- definition of data structures to handle the data generated by an application,
- interactions between various components using themselves various and different technologies.

A graphical application loosely inspired from an actual scientific imaging application, will be used throughout this lecture to back our exploration.

CHAPTER

ONE

AGENDA

Day 1	Bootstrapping the tools, first steps with C++ programmation: • bootstrap C++ development *environment* • versions management, discover *subversion* (svn) • read a structured data file containing a *picture*	• Introduction (En) • Outils - Tools • Subversion • Good practices • I/O (En)
Day 2	• read a structured data file containing a *picture* • discover the *Qt* framework	• Doxygen • STL (En) • C++ classes (En) • Memory handling (En) • C++ ops, fcts (En) • Qt
Day 3	• classes and graphical shapes • development within *DrawQt*	• DrawQt • Inheritance, polymorphism (En)
Day 4	• structured data: reading geometrical *shapes*	
Day 5 *and more*	• *projects*	

The various steps which will be followed on during this hands-on:

1. presentation of development tools and tools to ease the development: make, CMT, C++, ...
2. build of a small C++ application to introduce the use of input/output (I/O) and of the standard library (STL)
3. introduction to a graphical framework
4. interconnect the user interface and the C++ user code. Study the whole connection chain (create a shared library, declare C++ modules in the GUI, automatization with CMT)
5. definition of data structures to store graphical scenes. Definition of classes, creation of collections (leveraging STL)
6. definition of a file storage format. Elaborate save-and-reload functions for the graphical scenes.

CHAPTER

TWO

BOOTSTRAPPING

The first steps on how:

- to login
- to change your password
- to launch applications

are described on this *page*.

CHAPTER

THREE

GRADING

Your work will be graded considering the following 3 items:

1. quality of the produced code (8 pts):
   - choice and respect of coding conventions
   - choice of variables, methods, classes, ... naming
   - code clarity
   - usage of `Doxygen`
2. usage of tools (8 pts)
   - `CMT`
   - `Subversion` to manage and monitor the evolution of your work (regular, logical commits with sensible comments)
   - `Doxygen` to document your code
   - tests of the code in a dedicated directory (with `svn` use)
3. the quality of work invested in the hands-on sessions as well as its progression (4 pts)

CHAPTER

FOUR

CS HANDS-ON AT LAL

4.1 Introduction to C++

spot

This section will introduce:

- the software configuration management,
- the C++ environment,
- how to put in place a work flow.

The red line of this section is the implementation - in C++ - of an application reading files holding either a complete image or the definition of shapes which will be used later on during the hands-on session.

Such an image file can be found `there` while a file storing shapes can be found `here`. You should look at these files to familiarize yourself with their structure.

- We'll proceed in a step-by-step fashion: it is therefore very important to make sure you go through all these stages.
- You should be **extremely** conscious about your **style**, that is, code **clarity**:
  - define, use and follow **coding conventions** (i.e. code presentation)
  - give **sensible names** to variables and functions
  - properly **document** your code.
- Finally, it is paramount to regularly test your code to ensure a proper progression thru the exercize(s). In practice, this means:
  - systematically test your code after having implemented each and every feature
  - once the feature has been verified to work properly, do not hesitate to improve the clarity of your code (by e.g. refactor it or reformat it) and thus re-test it systematically after such a modification ("if it is not tested, it's broken")

4.1.1 Step 1: prepare your workarea (15 mins)

This first steps allows you to define and create your workarea with a tool to automatize tasks.

- `CMT`: we use a software configuration management tool - *CMT* - to automatize the necessary and needed steps to rebuild an application (which is bound to evolve and become more and more complicated and involved).
- Recommanded editors: available in `/Applications` such as `Emacs`: 
- `Terminal`: available from the **dock** (bottom of the screen), click on the `Terminal` icon: 
- Structure and layout of your workarea:
  - create a work directory under your *home directory*, called `Project` (Don't choose another name in order to stay consistent with this documentation)
  - in the following, the `$>` stands for the *prompt* after which one can issue `UNIX` commands:

```
$> cd
$> mkdir Project
```

Throughout this hands-on session, you should make sure to properly organize your workarea by creating -if necessary- proper directories. This allows to isolate and decouple development areas from one another, minimize naming conflicts (e.g. files automatically generated by some tools with the same name) and this dramatically eases groking the overall structure of the project.

One could also create a directory `Project/Tests` in which one would quickly evaluate and test a snippet of code extracted from the documentation.

4.1.2 Step 2: building a simple application (1h)

This step illustrates the simple development cycle of an application

---

After having bootstrapped the workarea environment, this cycle holds the following steps:

1. source code editing
2. compilation and linking steps
3. execution of the application
4. test the application (or a subset of its components)

Generally speaking, the life cycle of an application can be described by cyclically iterating through the above mentionned steps. For example, the typical cycle consists in doing 1 (once) and then the sequence (2,3,4) in a loop.

Bootstrapping the workarea environment

Firstly, go under your work directory `Project`.

```
$> cd ~/Project
```

Warning: all your work shall happen under this `Project` directory (or one of its subdirectory.)

We will then setup a workspace with `CMT` dedicated to this exercize. This workspace shall be named `Hello`.

```
$> cmt create Hello v1
-----------------------------------------
Configuring environment for package Hello version v1.
CMT version v1r20p20090520.
Root set to /Users/visiteur/tests.
System is Darwin
-----------------------------------------
Installing the package directory
Version directory will not be created due to structuring style
Installing the cmt directory
Installing the src directory
Creating setup scripts.
Creating cleanup scripts.
```

Warning: a `cmt` directory has been created as a side-effect of running the `cmt create` command. All the remaining work of this exercize shall be caried from under that directory.

```
$> cd Hello/cmt
```

Editing

We will copy the program given in example below under the directory `../src` of our workspace previously created. This will became a `hello.cpp` file.

After a close inspection of the sources, we notice it is the classical program which displays a simple message on the screen. For the moment, you shouldn't try too hard to understand what is performed on each line of this program.

```
/**
 * @file hello.cpp
 * @author LAL ens <ens@lal.in2p3.fr>
 * @date   March 2007
 *
 * First steps with a development environment.
 * First program hello.cpp: display "Hello!" on standard output.
 */

#include <iostream>

int main(int argc, char **argv)
{
  std::cout << "Hello!" << std::endl;

  return 0;
}
```

Nevertheless, you should notice:

- the `#include` command, whose character `#` shall be placed at column number 1 of your source file. This command allows to include the content of another file in your program. Here, we include the content of the file

`iostream` whose location is known to the compiler and which holds the declaration of the **prototypes** of the functions used of inputs/outputs.

- the comments following the `Doxygen` format, labelled with the `/**` and `*/` tokens

Compilation and links edition

The application building proper will be handed over to `CMT` which will be tasked with correctly configuring and activating a *tool* named `make`.

The means to this end is the `requirements` file which describes the application we want to build. Edit this file and add the following line:

```
application hello hello.cpp
```

We can now ask `CMT` to run `make`. `CMT` will automatically take care of producing the needed configuration components used by `make`.

`CMT` configuration w.r.t your environment:

```
# issue this only once, for each new project
$> cmt config
```

Initialize `CMT`:

```
# issue this everytime you modify the CMT environment
$> source setup.sh
```

Building:

```
$> cmt make
#CMT---> Info: Execute action make => make bin=../Darwin/
#CMT---> (Makefile.header) Rebuilding ../Darwin/Darwin.make
#CMT---> (Makefile.header) Rebuilding ../Darwin/setup.make
#CMT---> (Makefile.header) Rebuilding ../Darwin/constituents.make
#CMT---> (constituents.make) Rebuilding library links
#CMT---> (constituents.make) all done
#CMT---> (constituents.make) Building hello.make
#CMT---> Info: Application hello
#CMT---> (constituents.make) Starting hello
#CMT---> (hello.make) Rebuilding ../Darwin/hello_dependencies.make
#CMT---> compiling ../src/hello.cpp
#CMT---> building application ../Darwin/hello.exe
#CMT---> hello ok
#CMT---> (constituents.make) hello done
#CMT---> all ok.
```

Execution and tests

If all went well, we can now run and test our new application:

```
$> ../Darwin/hello.exe
Hello!
```

4.1.3 Step 3: Familiarization with C++ (2h)

We'll modify the `hello.cpp` file by introducing a `while` loop which will display the previous welcome message until a correct answer is given. This correct answer will terminate the program.

A few more precisions, in addition to these *informations*:

- right now don't use `using namespace std;` but declare explicitly the namespace for each STL object;
- `std::cout` is the standard output. One uses the << operator to write into `std::cout`;
- `std::cin` is the standard input. One uses the >> operator to read from `std::cin`;
- `std::endl` is the manipulator allowing to insert an end-of-line character (insertion of such a character isn't implicit, you have to do it manually);
- use a boolean variable (of type `bool` whose values are either `true` or `false`) to manage the `while` loop;
- use variable of type `std::string` for the array of characters (`std::string` is defined in the `<string>` header file)
- the `std::string` equality comparison operator in C++ is ==
- the operator to test if 2 `std::strings` are not equivalent is `!=`;
- the logical operators AND and OR are, respectively, `&&` and `||`.

The building and testing of the application is performed as in the previous stage.

You should make sure to update the comments inside the program.

Typically, one should get the following output:

```
$> ../Darwin/hello.exe
Hello!
continue? yes
Hello!
continue? yes
Hello!
continue? yes
Hello!
continue? yes
Hello!
continue? no
Bye.
```

4.1.4 A few references

These might help in case you are a bit lost with C++:

1. A C++ reference site
2. Lectures from J.F Rabasse.
3. An introduction to the C++ language is available.

   It is by no means exhaustive but is a short introduction to a few elements of C++ needed throughout this hands-on.

4.2 First steps with subversion (svn)

spot

With this exercize we'll investigate:

- version(s) management
- the tool svn

4.2.1 Reminder

You should have read the slides about subversion to be able to complete this exercize.

Warning: When issuing an svn commit command, **always** make sure to verify the commit was successful. A typical error is to run the command from the wrong directory. No modification is then committed but this isn't considered as an error by subversion. For example:

```
$> svn commit -m "svn doc modification" .
Sending        trunk/src/svn.rst
Transmitting file data .
Committed revision 666.
$>
# ==> the commit was successful

$> svn commit -m "svn doc modification" .
$>
# ==> no message nor ack. from svn: OOPSIE !
```

You can check the state of your repository in the Trac browser or directly via http .

4.2.2 Creating your project repository (15 min)

The first step consists in creating your project in the Etudiants svn repository laying out it the right way *Subversion Best Practices* recommends to structure a repository with three subdirectories: /trunk, /branches, and /tags.

In this very first step we will act directly onto the svn directory getting an URL as argument for svn commands. Then in the later steps we will work in our local workarea getting local file arguments.

- verify you can talk to the svn server and verify that your repository is empty (replace ens<n> with the correct team name, reply 'p' if asked about the certificate, the password is the one specified during the svn lecture, this isn't the one from your local account):

  ```
  $> svn ls https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>
  ```

- create your project Hello in your repository:

  ```
  $> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Hello \
              -m "Creation of Hello project"
  ```

- create the standard layout:

```
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Hello/trunk \
             https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Hello/branches \
             https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Hello/tags \
             -m "Proper layout for Hello project"
```

- verify:

```
$> svn ls -R https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>
```

- finally import your Hello **code** into the `trunk` directory:

```
$> cd ~/Project/Hello
$> svn import . https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Hello/trunk \
       -m "Initial import"
```

- verify again:

```
$> svn ls -R https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>
```

4.2.3 Initializing the workarea environment (15 min)

The next step consists in initializing your workarea directory which will hold a copy of your previously imported project *Hello*.

- go under your `Project` directory and rename your unversioned directory:

```
$> cd ~/Project
$> mv Hello Hello-unversioned
```

- create a working copy (`checkout`) of the `trunk` branch of your previously imported project *Hello* - notice that the local name is your project name:

```
$> svn checkout https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Hello/trunk Hello
```

- verify that your workarea is well connected to the right repository and notice the `svn` metadata directories `.svn` at each level of your hierarchy project:

```
$> cd Hello
$> svn info
$> ls -alR
```

- clean the directory:

```
$> rm -r ~/Project/Hello-unversioned
```

- check the state of your workarea - you should be synchronized:

```
$> svn status .
```

4.2.4 Modifying files (5 min)

During this exercize, we'll modify one or more files and save (or `commit` in `svn`-speak) these modifications into the repository. We'll first use the traditional `cp` and `rm` commands and then their `svn` counterparts in order to compare

their different behaviours. This operation can be repeated many times to artificially increase the history of commands (for the next step.)

When we save the modifications to your repository, you have to fill in a message - with no accentual letters - justifying the modifications.

Go under the `src` directory which we just added to our `svn` repository.

- modify an existing file (e.g. add a comment to `hello.cpp`), check its effect and save the modification into the repository:

```
$> # edit hello.cpp
$> svn status
M       hello.cpp
$> svn commit -m "first modification applied to hello.cpp"
```

- copy an already existing file (without using `svn cp`), check its effect and save the modification into the repository:

```
$> cp hello.cpp hello_world.cpp
$> svn status
?       hello_world.cpp
$> svn add hello_world.cpp
$> svn commit -m "adding hello_world.cpp"
```

  The `?` means the file isn't (yet) tracked or known to the repository but does exist in your local directory.

- delete the file previously added (but without using `svn rm`), check its effect and save the modification in the repository:

```
$> rm hello_world.cpp
$> svn status
!       hello_world.cpp
$> svn rm hello_world.cpp
$> svn commit -m "deleting hello_world.cpp"
```

  The `!` means the file exists in the repository but not in your local area.

- copy an already existing file (using `svn cp`), compare its effect with previous commands and save the modification in the repository:

```
$> svn cp hello.cpp hello_world.cpp
$> svn status
A  +    hello_world.cpp
$> svn commit -m "adding hello_world.cpp with svn-cp"
```

- delete the file (using `svn rm`), compare its effect with previous commands and save the modification in the repository:

```
$> svn rm hello_world.cpp
$> svn status
D       hello_world.cpp
$> svn commit -m "deleting hello_world.cpp with svn-rm"
```

- now, it is up to you to suppress the directory `Hello/Darwin`...

```
$> svn rm <????>
```

4.2.5 Listing modifications (15 min)

During the following steps, we'll experiment with 2 usages of the versions' history which is stored in the repository: visualizing the differences between versions and rolling back to an older version. To use the history, one usually begins with dumping the list of revisions available for inspection together with their (hopefully descriptive and informative) commit message.

- display the modifications (default mode):

```
$> svn log
```

- dump the modifications (detailed/verbose output):

```
$> svn log -v
```

- display the modifications for a particular file:

```
$> svn log hello.cpp
```

- compare the differences.

4.2.6 Visualizing the differences (15 min)

We'll investigate 2 use-cases for displaying differences:

1. between the local workarea and the repository,
2. between 2 revisions in the repository.

- dump the differences between the local workarea and the last revision in the repository:

```
$> svn diff hello.cpp
```

- display the differences between the local workarea and an arbitrary revision in the repository. Compare with the previous result:

```
# select a revision 'n' in the history...
$> svn diff -r n hello.cpp
```

- display the differences between 2 revisions `m` and `n` in the repository (the last revision is always named **HEAD**). The file `hello.cpp` should exist in both revisions (check the effect if this wasn't the case):

```
$> svn diff -r m:n hello.cpp
```

- repeat the previous step using the web interface `WebSVN`. Go to this `URL`: https://trac.lal.in2p3.fr/Etudiants/browser and browse to the `hello.cpp` file from your branch. Click on the *"Journal des révisions"*, select 2 revisions in the column *"diff"* and click on *"Voir les différences"*.

4.2.7 Rolling back to a previous revision (45 min)

A rather interesting usage for keeping an history of revisions is the ability to roll back to an earlier version of a file or a set of files.

- revert the modifications previously applied to your workarea and roll back to the last revision in the repository:

```
$> svn status
# ... check that hello.cpp has indeed been modified
$> svn revert hello.cpp
```

- roll back to a particular revision `n` in the repository in order to start again from a 'right version' - you first revert the modifications applied to your workarea - `HEAD` means the latest revision in the repository:

```
$> svn revert hello.cpp
$> svn merge -r HEAD:n hello.cpp
$> svn status
```

- roll back to revision `n` of the repository, keeping the local modifications:

```
# ... edit and modify
$> svn merge -r HEAD:n hello.cpp
$> svn status
# check for conflicts and, if any, resolve them
$> svn commit -m "revert to revision n for hello.cpp"
```

4.2.8 Resurrecting a deleted file (20 min)

Some times one want to get back a deleted file the way the file is added back to the repository together with it's history. Hence future '`svn log`' on this file will traverse back through the file's resurrection and through all the prior history.

- add a file, edit it and check in several times

```
$> echo "Test 1" > foo.txt
$> svn add foo.txt
$> svn ci -m "Added foo.txt"
$> echo "Test 2" >> foo.txt
$> svn commit -m "First modification of foo.txt"
$> echo "Test 3" >> foo.txt
$> svn commit -m "Another modification of foo.txt"
$> svn update
$> svn log
```

- then delete it

```
$> svn delete foo.txt
$> svn commit -m "Deleted foo.txt"
```

- edit hello.cpp and check in several times too

```
$> echo "// Test 4" >> hello.cpp
$> svn commit -m "Added some comments" hello.cpp
$> echo "// Test 5" >> hello.cpp
$> svn commit -m "Added some comments" hello.cpp
```

- now look for the revision *N* in which was deleted the file to be resurrected

```
$> svn update
$> svn log --verbose
```

- add the file to the repository

```
$> export REPO=https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>
$> svn copy $REPO/Hello/trunk/src/foo.txt@<N-1> ./foo.txt
$> svn status
$> svn commit -m "Resurrected foo.txt from revision N-1"
```

Note that the same technique works just as well for resurrecting deleted directories.

4.2.9 Using a branch (30 min)

A branch is a "*cheap copy*" of a subtree (ie, the trunk or another branch) of a SVN repository. It works a little bit like symbolic links on UNIX systems, except that once you make modifications to files within a SVN branch, these files evolve independently from the original files which were "copied". When a branch is completed and considered stable, it must be merged back to its original copy.

Now you will create an experimental branch in order to test a new implementation without betting your entire project. Experimental branches may be abandonned when the experiment fails. But if they succeed you can easily merge that branch with the trunk.

In our case our refactoring will be simple: we want to test the instruction `'using namespace std;'` in order to get rid of all `std::` ...

- define a environment variable to ease the repository calls and verify:

```
$> export REPO=https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>
$> svn ls $REPO
```

- be careful to start from a well synchronized state:

```
$> cd ~/Project/Hello
$> svn st
# if needed check in or revert
```

- create your branch:

```
$> svn copy $REPO/Hello/trunk \
            $REPO/Hello/branches/TRY-namespace-std \
            -m "Branch from trunk to test namespace std"
```

- transforms your current working copy to reflect your experimental branch:

```
$> svn switch $REPO/Hello/branches/TRY-namespace-std
$> svn info
```

- edit and check in as often as needed:

```
$> svn ci -m "Use of 'using namespace std;'" .
$> svn info
```

- switch back to the trunk:

```
$> svn switch $REPO/Hello/trunk
$> svn info
```

- finally merge your experimental branch to the trunk:

```
$> svn merge $REPO/Hello/trunk \
             $REPO/Hello/branches/TRY-namespace-std
$> svn ci -m "Merge TRY-namespace-std branch with trunk" .
```

- you may delete your experimental branch, you don't need it anymore. Of course, your branch isn't really gone, it's simply missing from the `HEAD` revision. If you use `svn checkout`, `svn switch` or `svn list` with earlier revision number, you will still see your branch.

```
$> svn delete $REPO/Hello/branches/TRY-namespace-std \
              -m "Removed branch TRY-namespace-std"
# Get the log with additional information from merge history and spot the branch revisions
$> svn up && svn log --use-merge-history .
$> svn ls $REPO/Hello/branches/ -r<nn>
```

4.2.10 Using a tag (10 min)

A tag is just a '*snapshot*' of a project in time, named in a human-friendly way.

- if not done define a environment variable to ease the repository calls and verify:

```
$> export REPO=https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>
$> svn ls $REPO
```

- be careful to start from a well synchronized state:

```
$> cd ~/Project/Hello
$> svn st
# if needed check in or revert
```

- create your tag:

```
$> svn copy $REPO/Hello/trunk \
            $REPO/Hello/tags/SVN-exercise
            -m "Tagging the end of the SVN exercise."
```

- verify:

```
$> svn ls -R $REPO/Hello
```

Normally and **by convention** one never works in a tag branch.

4.2.11 Parting notes

Do not forget to check, at the last session, that all your work is indeed correctly committed and saved in your `svn` repository.

i.e.:

- create an empty directory in which you'll export -with the *ad hoc* command- all of the performed exercizes,
- recompile everything,
- check they all run smoothly and correctly.

4.3 Reading an image from a file

spot

This exercize describes in a step-by-step fashion how to write a program reading an image from a file.

You should strive for systematically using all the tools you've learned so far:

- CMT
- svn

Do not forget to also systematically test (many times) and after each step the behaviour of your program. Do not hesitate to decompose a step into smaller intermediate ones.

This section should allow you to remember the basics of object oriented programming and of C++ before tackling the DrawQt project. All the steps should be rigorously followed, if, however, some detail is escaping you, do not hesitate to call for help.

Note: Concerning the compilation errors, remember to head toward the *page* listing the main and usual issues.

4.3.1 Step 1: environment configuration (20 min)

This first step -a variant of *hello world*- installs, tests and validates the projects within your development environment. In our case, the project Image is managed with *CMT* and subversion.

We'll start with configuring our svn environment:

```
$> cd ~/Project
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Image -m "Added Image project"
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Image/trunk -m "Added Image project"
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Image/branches -m "Added Image projec
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Image/tags -m "Added Image project"
            -m "Added Image project"
```

We'll need the Interfaces package to access and use the system libraries. Fetch it from the svn repository called Enseignement:

```
$> cd ~/Project
$> svn export https://svn.lal.in2p3.fr/projects/Enseignement/LAL-Info/tags/head/Interfaces \
             Interfaces
```

Now, create your new CMT package:

```
$> cmt create Image v1
$> svn co https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/Image/trunk Image
$> cd Image
```

Edit the requirements file like so:

```
1 package Image
2
3 # for basics system libraries
4 use Platform v1r* ${HOME}/Project/Interfaces
5
6 # to build an application readImage.exe
```

```
application readImage readImage.cpp

# define an action 'read' to execute readImage.exe
action read $(bin)/readImage.exe
```

Then, edit the file `src/readImage.cpp` along these lines:

```
/**
 * @file   readImage.cpp
 * @author LAL ens <ens@lal.in2p3.fr>
 * @date   February 2006
 *
 * @brief  Read an image from a file image.txt
 *
 *
 */
#include <iostream>

int main (int argc, char **argv)
{
  // print an informative message on the screen
  // ...

  return (0);
}
```

Now we are left with building and running/testing the application:

```
$> cmt make
$> cmt read
```

At this point, you can commit and save your first version of the program in `svn`, after having removed the not-important files (history-wise, *i.e.* the files you do not wish to track in `svn`):

```
$> rm -r ../Darwin
$> rm ../cmt/Makefile ../cmt/*.csh ../cmt/*.sh
$> cd ..
```

Note: You can also instruct `svn` to ignore certain files or directories by editing the svn configuration file `~/.subversion/config` along the section *[miscellany]* :

- uncomment the `global-ignores` variable (has to be defined on one line)
- add in it definition all patterns describing the files you won't track in `svn`; i.e. those generated by `CMT`:
  - `Makefile`, `setup.*` and `cleanup.*`, the `Darwin` directory

Finally, add the current tree to `svn`, save and synchronize your workarea and the repository:

```
$> svn add .
$> svn commit -m "Added Image package"
$> svn update
```

You can check the repository is up-to-date using the by now well known `svn status` command in the correct directory.

4.3.2 Step 2: Accessing a data file (20 min)

Create a directory to hold the input data:

```
$> mkdir ./data
```

Retrieve the `Image` file with `Ctrl-click` (and then *Save the link* or *Enregistrer la cible* or *Telecharger le fichier...*). Save the file under the `../data` directory.

The structure and format of this file is described over *there*.

Commit the data file in `svn`:

```
$> svn add data
$> svn commit -m "adding data directory" ./data
```

Here are the snippets of code which deal with opening the data file:

```
...
#include <fstream>
...

const std::string filename = "../data/image.txt";

std::ifstream f;
f.open(filename.c_str());

if (!f.is_open()) {
  std::cout << "error. file [" << filename << "] could not be found."
            << std::endl;
  f.close();
  return (0);
 }

std::cout << "file: [" << filename << "] open." << std::endl;
f.close();
...
return (0);
```

More detailed informations about input/output functions can be found *there*.

Now, we can build and test our application like so:

```
$> cmt make
$> cmt read
```

Eventually, we commit and save into `svn`.

Note: remember we work in the `src` directory...

4.3.3 Step 3: reading the data file token-by-token (30 min)

The following snippet of code shows how to loop over the content of a file, simply reading it word by word:

```
#include <iostream>
#include <fstream>
#include <string>
```

```

int main (int argc, char **argv)
{
  std::cout << "Application readImage." << std::endl;

  const std::string filename = "../data/image.txt";
  std::ifstream f;
  f.open( filename.c_str() );

  if (!f.is_open()) {
    std::cout << "error. file [" << filename << "] could not be found."
              << std::endl;
    f.close();
    return (0);
  }

  std::string word;
  std::cout << "file: [" << filename << "] open." << std::endl;

  // put the word-by-word reading loop here and
  // print out the word which has been read

  while (!f.eof()) {
    //...
  }


  return (0);
}
```

Warning: The reading function will only return a word back to you as long as the end of the file is not reached. Ergo, you shouldn't forget to test for that edge case.

Build this application, test it and commit your changes in `svn` once everything is behaving correctly.

4.3.4 Step 4: creating a `class` named `Image` (15 min)

Before going further in the implementation of our reading application, we'll create a `C++` class `Image`. This class will allow us to properly isolate the data members from the operations associated with or applied on our images. This is one of the pillars of object-oriented programming.

A presentation on `C++` classes is available here.

In order to cleanly separate the structure of the class itself from its implementation, we'll create 2 files:

- the header file `include/image.h` (create the `include` directory):

```
/**
 * @file  image.h
 * @brief File holding the description of the class Image
 * @author LAL ens <ens@lal.in2p3.fr>
 * @date   February 2009
 */

#ifndef PROJECT_IMAGE_H
#define PROJECT_IMAGE_H 1
```

```

/**
 * @brief
 */
class <Enter the name of your class>
{
public:
  /** @brief ...
   */
  Image();

  /** ...
   */
  ~Image();
};

#endif // !PROJECT_IMAGE_H
```

- the implementation file `src/image.cpp`:

```
/**
 * @file  image.cpp
 * @brief ...
 * @date   February 2009
 */

#include "image.h"

Image::Image()
{
}

Image::~Image()
{
}
```

You need to modify the `cmt/requirements` file for the build system to be aware of these new files:

```
package Image

# add the 'include' directory to the list of directories
# the compiler should know about
include_dirs $(IMAGEROOT)/include

# for basics system libraries
use Platform v1r* ${HOME}/Project/Interfaces

# to build an application readImage.exe
application readImage readImage.cpp image.cpp

# define an action 'read' to execute readImage.exe
action read $(bin)/readImage.exe
```

Rebuild everything, test and commit in `svn`.

4.3.5 Step 5: isolate the reading of a file in a dedicated method (1h)

In this step, we'll isolate the reading proper of a file into a method of the class `Image`. (Don't forget to add all `#include` if necessary) This method will later on be integrated and leveraged during the hands-on session on `DrawQt`.

You shall add the method `ReadFile()` with the following signature:

> bool **`ReadFile`**(const std::string& *filename*)

to your new class `Image`; the method `ReadFile()` will return a boolean in order to indicate whether the file scan ran into an error or not.

When this work is completed, the `main()` function of your program **readImage** should only contain:

- the instantiation of an object of the class `Image`
- the call to the method `ReadFile()`

Note that the error handling is not mandatory yet (this will be for a later step.)

Build, test and commit in `svn`.

4.3.6 Step 6: interpreting the data from the image file (1h)

Using the informations about the image file format *documentation*, we'll re-implement the reading of the file. Previously, in *step 3* we were reading the input file 'word by word' (or 'token by token'.)

But now that the format of the image file and its grammar is known, we build upon it to greatly simplify the reading of such files.

1. there won't be any need for the *while* loop anymore
2. the loop will be replaced by a serie of:

```
read a word
if not an expected word:
   return an error
read next word
```

3. memorize the values of `HEIGHT` and `WIDTH` in new data member variables called, respectively, `m_nX` and `m_nY`
4. iterate through the content of pixels of this image and display on screen the values, in a line-by-line fashion in a new method `Image::Print()`.

Note: To read next token into a word you have already seen the `f >> my_word` method; if you want to read the next token into an `int`, you could also use the same method `f >> my_int`

4.3.7 Step 7: data storage (45 min)

1. store the data in a one-dimension `private` array you'd call `m_data`. you should use the *std::vector* class.
2. then, change to a 2-D array (use a *std::vector of std::vector*)

4.3.8 Step 8: documenting the package `Image` (45 min)

This exercize will ask you to go back to your `Image` package and comment it as best as possible with the tool `Doxygen`.

We will now improve our application with some documentation generated with doxygen. In order to ease the process of documenting the code, we'll first add a few rules to *CMT* so it can steer `doxygen`.

Open the `cmt/requirements` file and add the following lines at the end:

```
# define an action to generate the documentation
document doxygen doc -group=documentation TO=../doc
```

Now, we have to tell `CMT` that it has to read is requirement file :

```
$> cmt config
```

We'll also have to create a `doc` directory to store this documentation. Create it as: `image/doc`.

Now, we have to configure **doxygen** thanks to a dedicated file named `Doxyfile`. This file is available `here`, save it under this new `doc` directory.

We can build and browse the documentation like so:

```
$> cmt make doc
$> open ../doc/html/index.html
```

We can now modify the documentation of our code and inspect these modifications once the documentation has been regenerated. Once the result looks satisfying, save and commit the modifications.

Note: It isn't necessary to add **ALL** the documentation files to `svn`. Indeed, these output files can be automatically generated from the `Doxyfile`. It is thus sufficient to just add that file to the *repository*.

```
$> svn add ../doc
$> svn revert --depth infinity ../doc/html
$> svn commit -m "added the doc generation with Doxygen management"
```

You could also update your `~/.subversion/config` file in order to definitely ignore the `html` directory.

4.4 Managing a graphical interface with `Qt`

---

spot

This session will detail in a step-by-step fashion how to write an application leveraging the `Qt` framework and then how to modify a graphical user interface such as `DrawQt` in order to add buttons which select various and different graphical shapes.

You should strive for systematically using all the tools you know, such as:

- `CMT`
- `svn`
- `Doxygen` and the proper syntax to decorate your code with comments (which are then picked up by `Doxygen`)

4.4.1 Step 1: a first Qt application (5 min)

This first exercize is a variation over *hello world*.

We start with configuring the `svn` environment (replace the `ens<n>` with your team id. *e.g.* `ens2`)

```
$> cd ~/Project
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/TpQt
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/TpQt/trunk \
             https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/TpQt/branches \
             https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/TpQt/tags \
             -m "Create TpQt project"
$> svn co https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/TpQt/trunk TpQt
```

For this exercize, we'll need the `Interfaces` package to gain access to the system libraries as well as the `Qt` environment. If the directory `Project/Interfaces` isn't already present, create it like so:

```
$> cd ~/Project
$> svn export https://svn.lal.in2p3.fr/projects/Enseignement/LAL-Info/tags/head/Interfaces \
              Interfaces
```

Then, create a new project with `CMT`:

```
$> cd ~/Project
$> cmt create TpQt v1
$> cd TpQt
$> svn add .
```

The last piece of boilerplate is the `cmt/requirements` file which should look like:

```
1  package TpQt
2
3  # to gain access to doxygen
4  use Platform v1r* ${HOME}/Project/Interfaces
5
6  # for the Qt environment
7  use Qt v2r*   ${HOME}/Project/Interfaces
8
9  # tools for the build (shared libraries, rules, ...)
10 use dld v2r*   ${HOME}/Project/Interfaces
11
12 # add our directory of headers for the compiler
13 include_dirs $(TPQTROOT)/include
14
15 # rule to create a 'moc' file (needed for the code generation for
16 # the signal/slots events)
17 document moc moc_myWindow   \
18          FROM=../include/myWindow.h \
19          TO=../src/moc_myWindow.cpp
20
21 # describe how the library holding our class shall be built
22 library TpLib \
23         ../src/myWindow.cpp \
24         ../src/moc_myWindow.cpp
25
26 # arguments and options for the compilation and link of 'myWindow' class
27 macro lib_TpLib_cflags   " ${Qt_cflags}"
28 macro lib_TpLib_cppflags " ${lib_TpLib_cflags}"
29 macro TpLib_shlibflags   " ${Qt_linkopts} ${dld_linkopts}"
30 macro TpLib_linkopts     " -L${TPQTROOT}/$(Platform_bin) -lTpLib" \
```

```
    WIN32                   " ${TPQTROOT}\$(Platform_bin)\TpLib.lib"

macro_append QtTestlinkopts " ${TpLib_linkopts}"

# describe how the QtTest.exe application shall be built
application QtTest QtTest.cpp

# boilerplate needed to package the application for MacOS
document darwin_app TpQt    FROM=QtTest TO=../app/QtTest

# update the DYLD_LIBRARY_PATH environment variable
# (needed by the MacOS dynamic linker to find our new shared library)
path_append DYLD_LIBRARY_PATH "" \
            Darwin "${TPQTROOT}/$(Platform_bin)"

# rule to create the documentation
document doxygen doc -group=documentation TO=../doc

## EOF ##
```

Note: Notice the line:

`use Qt v2r* Interfaces`

at the beginning of the `requirements` file. This line tells `CMT` to retrieve (and use) the definitions and configurations relevant for the `Qt/v2r*` package which can be located thanks to the `$CMTPATH` environment variable. (this env. variable has been automatically defined for you when the `Terminal` is launched – thanks to the configuration put in the `.zshrc` profile file)

Now comes the task of creating the `myWindow` class. First the header file `../include/myWindow.h`:

```
#ifndef TPQT_MYWINDOW_H
#define TPQT_MYWINDOW_H 1

// qt related includes
#include <QtGui/QMainWindow>
#include <QtGui/QPushButton>

/**
 * @file   myWindow.h
 * @author LAL ens <ens@lal.in2p3.fr>
 * @date   March 2007
 *
 * @brief  first class HelloWorld in Qt
 *
 *
 */
class myWindow : public QMainWindow
{
  // the macro Q_OBJECT is mandatory to hint Qt with the signal/slot
  // communication between (instances of) classes.
  Q_OBJECT

  public:

  /** @brief Constructor of our main class
   * @param parent : Parent widget of the class. In our case this will be the
   * main window (so the parent will be "NULL") "NULL"
```

```
    * @param fl : Creation flags for the window. This is useful to create a
    * window which can't be resized, or without a 'quit' button, etc...
    */
  myWindow( QMainWindow* parent = 0, Qt::WFlags fl = Qt::Window );

  /** @brief Destructor.
   */
  virtual ~myWindow();

private:

  /** @brief The 'hello' PushButton
   */
  QPushButton* m_hello;

};


#endif // !TPQT_MYWINDOW_H
```

Then the implementation, which we save under `../src/myWindow.cpp`:

```
/**
 * @file   myWindow.cpp
 * @author LAL ens <ens@lal.in2p3.fr>
 * @date   March 2007
 *
 * @brief  first class HelloWorld in Qt
 *
 *
 */

// Interface
#include <myWindow.h>

/** @brief Constructor for the myWindow class
 *
 * Our class, in order to be a display window, needs to inherit from the
 * @c QMainWindow class of the Qt framework.

 * @param parent : Parent widget of the class. In our case, this will be the
 * main window, so the parent is actually a NULL pointer.
 * @param fl : Creation flags for the window. This is useful to create a
 * window which can't be resized, or without a 'quit' button, etc...
 */
myWindow::myWindow( QMainWindow* parent, Qt::WFlags fl )
  : QMainWindow( parent, fl )
{
  // Create the push button
  m_hello = new QPushButton(...);
  // To complete the above line, heed towards the Qt documentation
  // the documentation of QPushButton is available here:
  // http://doc.qt.digia.com/4.0/qpushbutton.html


  // display our button (in a central position)
  setCentralWidget( m_hello );
}
```

```
37
38
39 /**
40  *  @brief Destroy an instance of myWindow (and all the objects it may hold)
41
42  *  i.e. the button we created previously.
43  */
44 myWindow::~myWindow()
45 {
46   delete m_hello;
47 }
```

Eventually comes the `main` function, which is put in `QtTest.cpp`:

```
1  /**
2   * @file   QtTest.cpp
3   * @author LAL ens <ens@lal.in2p3.fr>
4   * @date   March 2007
5   *
6   * @brief  first HelloWorld en Qt
7   *
8   *
9   */
10
11 // qt-related includes
12 #include <QtGui/QApplication>
13 #include <QtGui/QPushButton>
14
15 // local includes
16 #include "myWindow.h"
17
18 int main(int argc, char *argv[])
19 {
20   QApplication app(argc, argv);
21   myWindow * mw = new myWindow;
22   mw->show();
23   return app.exec();
24 }
```

Before being able to build and test our new application, we need to execute the `cmt/setup.sh` script which will correctly configure a few environment variables (`Qt` environment, libraries handling, ...) Heed towards the `cmt` directory and issue:

```
$> cmt config
$> source ./setup.sh
```

Now you can modify the `myWindow.cpp` file to make it compilable... Then:

```
$> cmt make
$> cmt make doc
```

Important: During the execution of these 2 commands and during this whole session (and all the following ones!) do **NOT** just look at their output without trying to analyze the messages logged on the screen. Do not forget that if a syntax error was inserted in your code or if a file is missing, these commands won't clobber (nor replace) the previously built executable (or documentation), thus one might (wrongly!) be led to believe no modifications were taken into account by the build system...

Finally, we can test the application (*via* 2 methods):

- from the `Finder` application (*i.e.* the file browser), launch the file `../app/QtTest.app`, or
- from your `Terminal`:

```
$> open ../app/QtTest.app
```

> **Warning:** The text output from the standard output `std::cout` are not shown in a graphical application. Indeed, such an application works in a completely decoupled fashion from the `Terminal`. However, it is possible to retrieve these outputs *via* the system console:
>
> ```
> $> open /Applications/Utilities/Console.app
> ```
>
> This console is used for all the *system* outputs so it is hardly surprising if you find a great number of messages coming from other applications. It is possible to filter the messages (top-right), though. (beware: it isn't case sensitive.)

We can then save and commit this *first* and *initial* version in our `svn` repository, after having removed the files we don't care to track and version:

```
$> rm -r ../Darwin ../doc ../app
$> rm Darwin.make Makefile *.csh *.sh
$> cd ..
$> svn add .
$> svn commit -m "initial version" .
```

We can then go back under the `src` directory where all the source files we modify are located:

```
$> cd ./src
```

4.4.2 Step 2: attaching a `signal/slot` to a button (10 min)

As you may have noticed, no action was associated with the button of the previous application. In order to fix this issue, we'll use the `signal/slot` mechanism of `Qt`.

- the emitting object will be the `hello` button,
- the signal being sent will be the mouse-click action,
- the receiving object will be the `myWindow` class,
- the slot to which the signal will be sent will be the `close()` method of the class `QMainWindow`.

To leverage all of this machinery, it's sufficient to add the following line, just after the creation of the button, in `myWindow.cpp`:

```
1  // Connection of the signal 'clicked()' (from the button)
2  // to the slot 'close()' (of the QMainWindow)
3  connect( m_hello, SIGNAL(clicked()), this, SLOT(close()) );
```

As usual, rebuild, test and then commit in the `svn` repository:

```
$> cmt make
$> cmt make doc
$> open ../app/QtTest.app
$> cd ..
$> svn commit -m "adding a signal/slot communication"
```

Note: It sometimes so happens that the compiler doesn't completely take into account your last modification. In this case, the application will compile successfully but will crash at runtime (when you execute it.) This is because the automatically generated `moc_myWindow` file hasn't been regenerated nor recompiled. To fix this, just delete it (in `src`) and recompile (it should be regenerated anew.)

4.4.3 Step 3: adding a menu (15 min)

We'll now add a menu to our window.

Warning: On a Mac, menus are not displayed together with the window (at the top of the window as for most of the other graphical operating systems) but at the *top of the screen*.

Add the following `includes` in your `myWindow.cpp` file:

```
#include <QMenu>
#include <QMenuBar>
```

The following code is to be put in place of the previous version of the constructor of `myWindow`:

```
/** @brief Constructor for the myWindow class
 *
 * Our class, in order to be a display window, needs to inherit from the
 * @c QMainWindow class of the Qt framework.

 * @param parent : Parent widget of the class. In our case, this will be the
 * main window, so the parent is actually a NULL pointer.
 * @param fl : Creation flags for the window. This is useful to create a
 * window which can't be resized, or without a 'quit' button, etc...
 */
myWindow::myWindow( QMainWindow* parent, Qt::WFlags fl )
  : QMainWindow( parent, fl )
{
  // Create the push button
  m_hello = new QPushButton("Hello world!");
  connect( m_hello, SIGNAL(clicked()),
           this,    SLOT(close()) );

  // display our button (in a central position)
  setCentralWidget( m_hello );

  // create a menu bar
  QMenuBar *menubar = new QMenuBar(this);

  // create a "File" menu
  QMenu *fileMenu = new QMenu("File");

  // add this file menu to the main menu bar
  menubar->addMenu(fileMenu);

  // add a few items to the file-menu
  //  - an 'open' item
  fileMenu->addAction("Open");

  //  - a separator
```

```
36    fileMenu->addSeparator();
37
38    //  - a 'quit' item which we connect to the 'close' slot of the main window
39    // Do not use the world "Quit" on mac OSX, see here why :
40    // http://doc.qt.nokia.com/4.7-snapshot/qmenubar.html#qmenubar-on-mac-os-x
41    fileMenu->addAction("Bye application", this, SLOT(close()));
42
43    // finally, add the menu bar to the main window
44    setMenuBar(menubar);
45  }
```

Recompile, test and commit in `svn`.

4.4.4 Step 4: adding a new `slot` (15 min)

We wish now to add our own action when someone clicks on a menu.

Thus, we'll first start by adding an action to our menu `Open` which we'll connect to a method `modify()` (defined later on.)

After the creation of the menu `Open`, add the following lines:

```
1   // add a new item "Modify" which we connect to the method modify()
2   fileMenu->addAction( "Modify", this, SLOT(modify()) );
```

Then, we add the method `modify()` which will be called whenever the menu `Modify` is clicked on. This method will be called by an event (a click on the menu), so it isn't a regular `C++` method but really a new `slot` which we need to define.

In the `myWindow.h` header file, add the following lines in the body of the class:

```
1   public slots:
2     void modify();
```

Now, we need to implement it, in the `myWindow.cpp` source file:

```
1   /** @brief This method modifies the text of the 'hello' button.
2
3    * Documentation about @c QPushButton is available here:
4    *   http://doc.qt.digia.com/4.0/qpushbutton.html
5
6    * @c QPushButton is a special kind of button.
7    * Indeed, buttons can be PushButtons, CheckButtons, RadioButtons, etc...
8    * It inherits the properties of an abstract button type: QAbstractButton.
9
10   * Documentation about @c QAbstractButton is available here:
11   *   http://doc.qt.digia.com/4.0/qabstractbutton.html
12
13   */
14  void myWindow::modify()
15  {
16    // Complete this line according to the documentation
17    m_hello->setText(...);
18  }
```

Rebuild, test and commit in `svn`.

4.5 DrawQt

4.5.1 Step 1: installation (30 min)

As for the previous sessions (*reading an image* and *Qt*,) we need to:

- create a new workarea `DrawQt` under your work directory `Project`,
- fetch the initial version of the `DrawQt` code – called `DrawQtBase` – from the `Enseignement` repository:

```
$> cd Project
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/DrawQt/trunk -m "Added DrawQt pr
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/DrawQt/branches -m "Added DrawQt
$> svn mkdir https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/DrawQt/tags -m "Added DrawQt pro
$> svn co https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/DrawQt/trunk DrawQt

$> svn export https://svn.lal.in2p3.fr/projects/Enseignement/LAL-Info/tags/2011/DrawQtBase \
              DrawQt
```

Note: `svn export` exports a project from a repository **without retrieving** the revisions management. Indeed, we'll modify this code base and commit it back into **our own** `svn` repository.

- commit your `DrawQt` directory into `svn`.

Head toward the presentation of the application.

4.5.2 Step 2: reading the documentation (10 min)

During this initial step, paramount for the proper understanding of the overall workings of the application, we'll crawl thru some documentation.

1. first, build the documentation of `DrawQt` in the same fashion than for the previous sessions: using `doxygen`,
2. read the documentation in your browser,
3. look for the `WFrame` class. It will play a central role in this session. Navigate thru the classes connected to this `WFrame` and try to grok the relationships between those.

4.5.3 Step 3: building the application (20 min)

In order to build the application, we need to properly configure `CMT` so the tool knows where to find the sources, headers, etc...

Note: This configuration step wasn't mandatory for the other sessions because our workarea was a directory created by `CMT` itself hence autoconfigured. (and this configuration step wasn't necessary either for the documentation generation of *step 2*)

```
$> cmt config
$> source ./setup.sh
```

Then rebuild and run the application:

```
$> open ../app/DrawQt.app
```

This application is – for the moment – just a skeleton we'll complete as we go along.

4.5.4 Step 4: reading an image (1h30)

Note: This step works you through the image reading implementation which has been written during the previous session.

To ease the implementation task, speed up the process and help you start on sane foundations, you are provided with a skeleton function. It is up to you to (correctly) complete it OR to re-use your own `ReadFile` code (copy paste it in place of the skeleton).

1. The image reading functionality in `DrawQt` is named as in the *image reading session*: `ReadFile(const std::string&)`. Helped by the `DrawQt` documentation, look for the class hosting that method and which method or function is calling that method. Then, compare its implementation with the one you wrote during the *image reading session* (and complete/amend your skeleton if necessary.)
2. Test your application with the images stored under the `Data` directory. Some of the input files are purposedly corrupted: you'll need to amend the `Image::ReadFile(const std::string&)` method to correctly handle (or print an error message) when dealing with such input data.
   - test 1: bad keyword
   - test 2: issue with the array filling (beware: program may crash!)

   To display a message in a small dialog box, use the very well documented method QMessageBox.
3. once the tests are satisfactory, commit, remove all spurious printout messages and then commit again.

4.5.5 Step 5: support for an elliptical shape (1h30)

Note: The code you were provided with only handles rectangular shapes. In this exercize, a new `Ellipse` class will be implemented to deal with a new type of shape. This class will inherit from the class `Shape`.

---

1. Create the files `ellipse.h` and `ellipse.cpp`. This class inherits from `Shape` so it also contains a data member `m_box` of type `BoundingBox`. This variable is sufficient to encode the properties of an ellipsis. It is therefore not needed to add any new data member to the class `Ellipse`.
2. Update the `cmt/requirements` file so it knows about the `ellipse.cpp` file.
3. Implement the methods of the class `Shape` which aren't suitable for the class `Ellipse`. You can look at the `Rectangle` class for some inspiration. The `Qt` method allowing to draw an ellipse is documented here.
4. Open the `include/defs.h` header file and modify it accordingly.

   Note: `enum SHAPE_TYPE` is a type used for printing the name of a `shape` in a statistics box. There are rules to follow if you want the name of your shape to be correctly displayed. These rules are described in lines 75-100 of the file `wstatistics.ui`. This file is a graphical interface file generated by `QtDesigner`, a tool to create graphical user interfaces. Its format is `XML`. During the compilation, `CMT` takes this file as input and produces a proper `C++` file (`ui_wstatistics.h`) which is fed back to the compiler.

   ---

5. Implement the `bool IsInside(int x, int y)` method which will test if a given point falls inside the ellipse or not.

Note: Some formulae about ellipses can be found here.

Note: The origin point `(0,0)` of the frame is the top-left corner. Positive `x` go to the right. Positive `y` go down.

```
    (0,0) +-----------+ (xmax, 0)
          |           |
          |           |
          |           |
          |           |
          |           |
(0, ymax) +-----------+ (xmax, ymax)
```

---

6. Implement the method `SumValues(Image *pData)` and the other methods which need special treatment for an ellipse.
7. Complete the `Selection::AllocateShape(SHAPE_TYPE type)` method so it can correctly handle the ellipse case.
8. Add a button for the ellipse in `WFrame`. You can look at the `Rectangle` class for inspiration. An icon exists in the directory `data/icons`: `Ellipse.bmp`.
9. Update the statistics, on the right in `DrawQt` if not already done.
10. Do not forget to start with the correct ellipse formula and check the result...

4.5.6 Step 6: support for a square shape (1h)

Note: A square is a special case of a rectangle. The class `Square` will thus rather naturally inherit from the `Rectangle` class. We'll just amend the method to update the shape to force the rectangle to have a width and a height of identical sizes.

---

1. Create the files `square.h` and `square.cpp`. This class won't need any new data member.
2. Update the `cmt/requirements` file so it knows about `square.cpp`.
3. Implement the methods of the class `Rectangle` which aren't suitable for `Square`.
4. Redefine `ModifyEndShape(int lastMousePosition_x, int lastMousePosition_y)` from `Shape` in the class `Square` to force `m_box` to stay square. This method is called for each and every mouse move during the painting of a shape. Look at the `BoundingBox` documentation, it could help you.

Note: You may want to use some mathematical functions in order to draw in all ways your square. `max` function is defined in std template algorithm, the `abs` function is defined in math library.

---

1. Update the `Selection::AllocateShape(SHAPE_TYPE type)` method.
2. Add a button for the square in `WFrame`. An icon is available: `Square.bmp`
3. Update the statistics on the right in `DrawQt` if it hasn't already been done.

4.5.7 Step 7: support for a circular shape (1h)

Note: A circle is a special case of an ellipse. The class `Circle` will thus rather naturally inherit from the `Ellipse` class. We'll just amend the method to update the shape to force the ellipse to have a major and a minor axis of identical lengths.

1. Create the files `circle.cpp` and `circle.h`. This class won't need any new data member.
2. Update the `cmt/requirements` file so it knows about `circle.cpp`.
3. Implement the methods of the class `Ellipse` which aren't suitable for `Circle`.
4. Redefine `ModifyEndShape(int x, int y)` from `Shape` in the class `Circle` to force `m_box` to stay round. This method is called for each and every mouse move during the painting of a shape.
5. Update the `Selection::AllocateShape(SHAPE_TYPE type)` method.
6. Add a button for the square in `WFrame`. An icon is available: `Circle.bmp`
7. Update the statistics on the right in `DrawQt` if it hasn't already been done.

4.5.8 Step 8: make a tag for your current work

Note: At this step, you should have a simple drawQt application that your will improve after. Let's Make a tag with this version. For more information about tag and branches, see the svn exercice

1. Go back into your Project/DrawQt folder

```
$> svn cp . https://svn.lal.in2p3.fr/projects/Etudiants/ens<n>/DrawQt/tag/<enter your tag name h
```

4.6 Reading a file of shapes

4.6.1 Bootstrapping

Note: When a region of interest is identified in an image, one often needs to find it back later on.

In this session, we'll implement a few `I/O` methods for the already existing shapes to be able to save and restore them. We'll of course build atop the code we already developed during the course of the previous sessions.

In order to easily read back a set of shapes, our strategy to load a shape from disk and its implementation will consist of:

- a class `Selection`:
  - holding **a set of** objects of type `Shape` (only one object for the time being)
  - holding a method `bool Selection::Read(std::ifstream& f)` which will read the **WHOLE** file of shapes.
- a class `Shape`:
  - holding **only one** shape.

4.6.2 Step 1: adding the declaration of the new methods (45 min)

No data handling is asked for at this stage, it will be performed later on.

For the moment, we'll only read a file holding just one shape. The grammar for the shape files is available and documented on this *page*.

- Create a file `shape.txt` holding a `square` centered in `(x=50, y=90)` and of size `100`. During the "projects session(s)", we'll augment this file with a set of shapes.
- Commit this new file in `svn` under `data` directory of your `DrawQt` package:

```
$> cd data
$> svn add shape.txt
$> svn commit -m "adding my first shape file" shape.txt
```

- When clicking on the bottom-right `read` button of `DrawQt`, we wish to be able to read a file holding **one** shape. The connection of this button to the `void Visu2D::ReadSelection()` method has already been implemented for you, you can nevertheless have a look at how this has been performed in the `WFrame` class.
- Implement the `void Visu2D::ReadSelection()` method. It shall be able to:
  1. open a file selector (look at `bool WFrame::ReadImage()` for inspiration)

Note: The return type of the QFileDialog will be a QString, but that the reading method of `Selection` is an `ifstream`. Thereafter how to convert this QString into a `ifstream`:

```
//  First, create an ifstream pointer
std::ifstream f;

// Then convert the QString into an std::string using QString::toStdString() method
std::string my_name = fname.toStdString();

// Open the file
f.open(my_name.c_str());
```

Now you could call the reading method of `Selection`

When a region of interest is identified in an image, one often needs to find it back later on.

  1. call the reading method of `Selection`
  2. print a message in the class `Selection` to check everything is correct

> **Warning:** Take care when you call an object method, that the object has been created and initialized before using it. Ex:
>
> ```
> Selection my_selection = NULL;
> ....
> my_selection->Read();
> ```
>
> In this case `my_selection` is initialized to `NULL`. Using a method on a `NULL` pointer has disastrous effects on a program at runtime... You MUST protect your method calls by "if (my_selection)" statement.

4.6.3 Step 2: implementing the keywords reading method (45 min)

Note: Similar to the reading of `Image` files, we'll implement a method to read back `Shape` files in this exercize as described there.

The method to implement is given below. It can be used to cross-check with your implementation for the reading of `Image` files.

```
/**
   The grammar for a shape-file is the following:
      shapes ::= shape | shape shapes | comment shapes

      shape ::= 'SHAPE' <number>
                'TYPE'  type_name data
                'SHAPE_END'
      comment   ::= 'COMMENT' ... 'COMMENT_END'
      type_name ::= 'SQUARE' | 'RECTANGLE' | 'CIRCLE' | 'ELLIPSE' | 'POLYGON'
      data      ::= envelope | list
      envelope  ::= 'ORIGIN' <x> <y>
                    'WIDTH'  <width>
                    'HEIGHT' <height>
      list   ::= 'NB_POINTS' <number> points
      points ::= point points | point
      point  ::= 'POINT' <x> <y>

 */

bool Selection::Read(std::ifstream &f)
{
  bool status = false;

  if ( !f.is_open() ) {
    std::cout << "Error. File ..." << std::endl;
    return false;
  }

  std::string word = "";
  std::string type = "";
  std::string tmp  = "";

  unsigned int id_nbr = 0;
  unsigned int origin_x = 0;
  unsigned int origin_y = 0;
  unsigned int height = 0;
  unsigned int width  = 0;
  unsigned int nb_points = 0;
  unsigned int pt_x = 0;
  unsigned int pt_y = 0;

  // define a state variable to identify words located inside the definition
  // of a given Shape.
  /**
   *  @brief Enum state
   *  Identifies the state of the analysis w.r.t the Shape
   */
  enum {
    void_state,
    shape,
```

```
    comment,
    type_name,
    data,
    envelope,
    list,
    points
  } state;

  // state initialization
  state = void_state;

  while ( !f.eof() ) {

    if ( void_state == state ) {
      f >> word;

      // grammar reminder:
      // -----------------
      // shapes ::= shape | shape shapes | comment shapes
      // shape ::= 'SHAPE' <number>
      //           'TYPE'  type_name data
      //           'SHAPE_END'
      // comment   ::= 'COMMENT' ... 'COMMENT_END'
      // in such a state, the allowed keywords are:
      if ( "COMMENT" == word ) {
        state = comment;
      } else if ("SHAPE" == word) {
        f >> id_nbr;
        state = shape;
      }

    } else if (comment == state) {
      // comment   ::= 'COMMENT' ... 'COMMENT_END'
      f >> tmp;
      if ( "COMMENT_END" == tmp ) {
        // go to next state...
        state = void_state;
      }

    } else if (shape == state) {
      // shape ::= 'SHAPE' <number>
      //           'TYPE'  type_name data
      //           'SHAPE_END'
      // => need to read the tokens following the word 'SHAPE': <number>
      f >> word;
      if ("TYPE" == word) {
        state = type_name;
      } else if ("SHAPE_END" == word) {
        // end of shape OK.
        status = true;
        state = void_state;
      } else {
        QMessageBox msgBox;
        msgBox.setText("Read shapes\n Error: invalid shape file");
        msgBox.exec();

        return false;
        // error case(s) to be better handled...
```

```
    }

  } else if (type_name == state) {
    // type_name ::= 'SQUARE' | 'RECTANGLE' | 'CIRCLE' | 'ELLIPSE' | 'POLYGON'
    f >> type;
    if ("SQUARE" == type) {
    } else if ("RECTANGLE" == type) {
    } else if ("CIRCLE"    == type) {
    } else if ("ELLIPSE"   == type) {
    } else if ("POLYGON"   == type) {
    } else {
      QMessageBox msgBox;
      msgBox.setText("Read shapes\n Error: invalid shape file");
      msgBox.exec();

      return false;
      // error case(s) to be better handled...
    }
    state = data;

  } else if (data == state) {
    // data      ::= envelope | list
    f >> word;
    if ("ORIGIN" == word) {
      state = envelope;

    } else if ("NB_POINTS" == word) {
      state = list;

    } else {
      QMessageBox msgBox;
      msgBox.setText("Read shapes\n Error: invalid shape file");
      msgBox.exec();

      return false;
      // error case(s) to be better handled...
    }

  } else if (envelope == state) {
    // envelope  ::= 'ORIGIN' <x> <y>
    //               'WIDTH'  <width>
    //               'HEIGHT' <height>
    f >> origin_x;
    f >> origin_y;
    f >> word;
    if ("WIDTH" != word) {
      // handle error
    }
    f >> width;
    f >> word;
    if ("HEIGHT" != word) {
      // handle error
    }
    f >> height;
    state = shape;

  } else if (list == state) {
    // list   ::= 'NB_POINTS' <number> points
```

```
      f >> nb_points;
      state = points;

    } else if (points == state) {
      // points ::= point points | point
      // point  ::= 'POINT' <x> <y>
      for (unsigned int a = 1; a <= nb_points; ++a) {
        f >> word;
        if ("POINT" == word) {
          f >> pt_x;
          f >> pt_y;
        } else {
          // handle error
        }
      }
      state = shape;
    }
  } //> while-loop

  return status;
}
```

As we don't handle (yet) the shapes parameters, nothing is visible because no shape is painted. The only means to check everything is working properly is to print out messages (so we know the reading is performing well).

As usual, build, test and commit to `svn`:

```
$> cmt make
$> svn commit -m "implemented data file reading method with keywords"
```

4.6.4 Step 3: completing the reading method (1h30)

Note: The reading method being implemented, we're still left with actually completing its skeleton (allocating shapes, handling of errors, ...)

For each shape we read from disk, we need to actually create an instance of the according class. To this end, a method `Shape* AllocateShape(SHAPE_TYPE type)` needs to be called with the correct `type` argument.

1. In the reading method, add a new variable `SHAPE_TYPE type_id`. Assign it the `type` the method has just finished reading (see the header file `defs.h`)
2. Allocate a new object of type `Shape` using the method `AllocateShape(...)`

Warning: Do not forget to handle all the possible **error cases** (*e.g.* a non existing shape name).

Note: Do not bother (for the moment) with the type `POLYGON` which will be tackled during a dedicated project later on.

Build, test and commit to `svn`:

```
$> cmt make
$> svn commit -m "type shape reading and shape allocation"
```

4.6.5 Step 4: reading the content of a shape (25 min)

Note: Last step for the `Shape` class: we need to store the properties of each shape (`WIDTH`, `HEIGHT`, ...)

The class `Shape` owns an object of type `BoundingBox`. This is the object which will hold the properties of a shape.

1. Carefully inspect the documentation of the `BoundingBox` class.
2. In the class `Selection`, the parameters read from the file will need to be properly handed over to the new `Shape` object (`m_shape`). Leverage the methods available in the `Shape` class to address this issue. If needed, implement new helper methods within the `Shape` class.

Note: Do not bother (for the moment) with the type `POLYGON` which will be tackled during a dedicated project later on.

Note: In order to draw the new shape read from the file, the method update() needs to be called at the end of the reading (in the class `Visu2D`). This method will send the `repaint` signal, connected to the method paintEvent().

Rebuild, test and commit into `svn`:

```
$> cmt make
$> svn commit -m "reading of the shape parameters"
```

4.6.6 Step 5: error handling (2-3h)

1. Comments handling. It isn't mandatory to store them but simply to be able to print them on screen when reading a shape-file, by means of *e.g.* a `QMessageBox`.

Additional cross checks and errors handling can be implemented. The most usual types of errors are listed below:

- check the file is open and/or the stream is indeed valid,
- check no keyword is missing,
- check a circle has indeed identical height and width,
- check the dimensions aren't negative
- ...

A file `error-shapes` is provided to test a few of these frequent errors.

4.6.7 Step 6: saving shapes (3-4h)

For this exercize, we'll save objects in a file. We'll strive to correctly format the output file so it closely matches the original file format: ideally, the output file should be readable by another application using the `Shape` class (as we did it for our application).

Therefore, we'll implement a couple of new `save` methods which will deal with the details of the file formatting.

We wish to be able to save a file holding a shape when clicking on the button `save` at the bottom right of the `DrawQt` window. In a similar fashion than for the reading, connect this button to the method `void Visu2D::SaveSelection()`

This method will:

- open a file selector,
- call the writing method of the `Selection` class: `Selection::Save(std::ofstream&)`

In the method `Selection::Save(std::ofstream&)`:

1. Complete this method as we did for its reading counterpart. To this end, add a writing method in the `Shape` class.
2. Check everything is working correctly by saving a file and reading it back.

Note: At this point, you are reading to start with the mini projects. But before that, please call us so we can check everything is correct and compatible with the projects you'll tackle.

4.7 Projects

4.7.1 Displaying the average intensity of a shape

The computation of the intensities' integral in a shape is only sensible when the displayed image corresponds to an event count (*e.g.* gamma rays detection in images from a POCI (*Perioperative Compact Imager*) gamma camera.) In the case of the *rat's olfactory bulb*, the image is computed from a movie (*i.e.* a serie of images) and the interesting information is then the average of the intensities in a shape.

The main modifications will consist in:

- in the dialog box `WStatistics`, add a widget to display the intensities' average. You should modify the `wstatistics.ui` file (thru the `Qt-designer`) to do such a thing.
- accordingly amend the `void WStatistics::UpdateStats(...)` method to correctly update this item.

4.7.2 Reading/writing an image from/to a binary format

In order to ease the data access, the `image.txt` file provided at the beginning of the session and holding the images, was an `ASCII` file. In practice, this kind of file format is very seldomly used because it isn't very efficient, both in terms of memory/disk space and in terms of access/reading speed. Instead, one usually resorts to binary format(s) which consists in writing out data values byte by byte without any particular formatting.

Note: In their simplistic incarnations, binary formats aren't portable, that is: *e.g.* you can not write a binary file on a `MacOS` machine and expect to read it correctly on a `PC` machine. There are of course ways to write portable binary formats.

For the file accesses, one will either use the `STL iostream` or the `QFile` class from `Qt`. More informations about `Qt` can be found here.

We'll save:

- `m_nX` as a `short` (`2` bytes)
- `m_nY` as a `short` (`2` bytes)
- the `m_nX * m_nY` values as `int` (`4*m_nX*m_nY` bytes)
- *Writing.* Add the item `Save image` (in `WFrame::InitMenus()`) in the `File` menu. Implement the associated method `Image::SaveImage()`. The file extension shall be `.bin`. To handle the files' extensions, one could leverage the `QFileInfo` class and its `suffix()` method.

One could either save the image value-by-value or percolate through the `QVector` class whose method `QVector::data()` returns the address of the beginning of the vector and thus allows to save line-by-line (it is more efficient because there are fewer seeks so fewer disk accesses.) The conversions `std::vector <-> QVector` can be done by means of `QVector::toStdVector()` and `QVector::fromStdVector()`.

- *Reading back.* Modify the `WFrame::ReadImage()` so as to be able to seamlessly load images from `ASCII` files (`.dat`) and binary image files (`.bin`). To achieve such a goal, one will have to pass as a third argument to `QFileDialog::getOpenFileName(...)` the string `"Images (*.dat *.bin)"`. The files' extensions will be discovered thanks to the `QFileInfo` class and its `QFileInfo::suffix()` method. Implement the corresponding `Image::ReadBinaryFile()`.
- The program will be then tested with the `bulbe-olfactif-hexana.bin` and `bulbe-olfactif-pentylacetate.bin` files which contain images from the rat's olfactory bulb acquired by intrinsic optical imaging while under olfactory stimulii.

4.7.3 Handling many shapes

Note: To begin with, the `Selection` class will need to be modified so each and every new `Shape` type will be added to the already existing shapes on screen.

Then, an option allowing the user to choose between the `Replace shape` and `Add shape` modes shall be implemented.

---

1. Modify the `Selection` class to handle an `std::vector<Shape*>` in lieu of a `Shape*`.
   (a) in the method:

   ```
   void Selection::HandleNewShape(Shape *s);
   ```

   use `std::vector::push_back(Shape *s)` to add a new `Shape`.
   (b) in the method:

   ```
   void Selection::Draw(QPainter *);
   ```

   write a loop to handle and represent all cached shapes.
   (c) all the methods of the type `GetWidth()`, `GetHeight()`,... aren't sensible. At this stage, one could just have them return the values associated with the last shape having been drawn.

   At this stage, it is possible to concurrently represent many different shapes on screen.
2. It is now time to add the option allowing the user to choose between the modes `Replace shape` and `Add shape`.
   (a) Create a button in `WFrame` to select the mode `Add`. An icon is available in the directory `data/icons`: `Add.bmp`
   (b) Add a boolean data member in `Visu2D` to know which mode is active. This variable needs to be updated each time the user clicks on the *ad hoc* button in `WFrame`.
   (c) Amend the method:

   ```
   void Selection::HandleNewShape(Shape *s);
   ```

   Add a boolean argument to know if the shape is added anew or is replacing the old one.
3. Finally, the user needs to be able to pick a shape among those already painted on screen and to display the informations related to this shape. You should associate the mouse right click (`Control-click`) within a shape to its selection.

(a) Implement a method to find and return the index of the shape which contains a given point `(x,y)`:

```
/**
 * @brief Return the index of the shape nearest to a given point (x,y)
 * @param x: x coordinate.
 * @param y: y coordinate.
 */
int GetNearestShape(int x, int y);
```

To achieve this, the method will loop over all the shapes stored in the `std::vector` and will ask the method `bool Shape::IsInside(int x, int y)` if the point `(x,y)` falls inside the shape.

(b) In the method:

```
void Visu2D::HandleMouse(int type, QMouseEvent *e);
```

add for the right-click case the search for the nearest shape and the emission of an `update` signal:

```
emit(UpdateStats(m_selection, idx))
```

where `idx` is the index of the nearest shape. If the shapes' translation has already been implemented, add as a fourth argument to the method:

```
void Visu2D::MoveShape(int type, int x, int y);
```

the index of the shape to be moved around.

(c) Implement 2 methods, to – respectively – compute the total area of all currently displayed shapes and compute the total number of counts.

(d) In the `WFrame` class, update the method:

```
void WFrame::UpdateStats(Selection *pSelection);
```

so it takes a second argument of type `int` to represent the index of the shape.

(e) Modify the connection accordingly, in the constructor of the class `WFrame`, *i.e.:* add an argument of type `int` for the index of the shape:

```
connect(m_visu2d,
        SIGNAL(UpdateStats(Selection*, int)),
        SLOT(UpdateStats(Selection*, int)))
```

4. In the `Selection` class, update the read/write methods of several shapes. One can retrieve and reuse the code of the previous sessions.
5. Check that everything is working correctly !

4.7.4 Internationalization

spot

In the context of international projects, it may be usefull to provide multiple versions of the same piece of software but in various languages.

`Qt` offers the proper tools to easily tackle such a task.

4.7.5 Moving a shape

spot

The shape is moved if the left-button of the mouse is clicked (and held down) and if the button `Move shape` is selected. The motion of the shape is identical of that of the mouse.

1. Implement the option allowing the user to select the mode `Move shape` by adding a button in `WFrame`. An icon is available in the directory `data/icons`: `Move.bmp`
2. Add a boolean data member `m_moveShapeMode` to `Visu2D` to know if the current mode is `Moving a shape` or not. This variable needs to be updated each time the user clicks on the corresponding button in `WFrame`.
3. Complete the `Shape` class with the following `virtual` method:

```
/** @brief Test if the shape is still within the visualization window after
 *         a move. If it is, then move it.
 *  @param dx: displacement along x
 *  @param dy: displacement along y
 *  @param w:  width of the window
 *  @param h:  height of the window
 */
virtual bool TestAndMove(int dx, int dy, int w, int h);
```

   `w` and `h` give the limits of the image in data coordinates to test the translation being asked for doesn't move the shape out of the visualization area (even just partially.) This method needs to be redefined for the polygon as the translation isn't restricted to just that of the `BoundingBox`.

4. In the `Visu2D` class, add a method similar to `GenerateShape()` and complete it:

```
void Visu2D::MoveShape(int type, int xData, int yData)
{
  switch (type) {
  case PRESS:
    // handle it

  case MOVE:
    // handle it

  case RELEASE:
    // handle it
  }
  update();
}
```

   The method `Selection::GetShape()` returns – for the moment – the one shape falling into the selection. In the next project, it will return the current shape.

5. Modify the method:

```
void Visu2D::HandleMouse(int type, QMouseEvent *e);
```

   to handle the mouse motion.

4.7.6 Operators overloading

spot

Reading/writing shapes via operator overloading. Heed towards the presentation about the operators overloading

---

Have a look at the `include/boundingbox.h` header file which holds the definition of the `BoundingBox` for each shape, and in which a few operators have been already defined. Overload the << and >> operators in the `Shape` and `BoundingBox` classes so they can be used to read/write shapes.

Note: You need to replace the methods `Shape::Read(std::ifstream &f)` and `Shape::Save(std::ofstream &f)` using the corresponding overloaded operator. But the reading of the `SHAPE`, `TYPE` (and perhaps `SHAPE_END`) keywords – which describe the structure of the file – will stay in `Selection::Read(std::ifstream &f)`.

The `WIDTH`, `HEIGHT` and `ORIGIN` keywords – which describe the data layout – will now be read in the `BoundingBox` class. The `Shape::Read(std::ifstream &f)` method will thus contain only the bare minimum (*i.e.* a few lines...)

The goal of this step is to delegate as best as possible the data handling to the class which manages those data – *i.e.* `BoundingBox`.

You should add a method `const std::string& BoundingBox::isValid() const` to handle errors. This method should return `""` (an empty string) when everything is OK or your error message otherwise.

The prototypes of the methods to be overloaded are below:

```
/**
 * @brief output operator overload (shapes output format)
 * @param os : output fstream
 * @param sh : shape to save
 */
friend std::ofstream& operator<<(std::ofstream& os, const Shape* sh);

/**
 * @brief input operator overload (shapes input format)
 * @param is : input fstream
 * @param sh : shape to display
 */
friend std::ifstream& operator>> (std::ifstream& is, Shape* sh);


/**
 * @brief output operator overload (boundingbox output format)
 * @param os : output fstream
 * @param bo : boundingBox to save
 */
friend std::ofstream& operator<<(std::ofstream& os, const BoundingBox& bo);

/**
 * @brief input operator overload (boundingbox input format)
 * @param is : input fstream
 * @param bo : boundingBox to save
 */
friend std::ifstream& operator>> (std::ifstream& is, BoundingBox& bo);
```

Warning:

- notice the `friend` keyword. This keyword allows to grant wider access to parts of the internals of a class to a function or to another class. In the `.cpp` source file, this keyword shouldn't appear in the declaration (nor in the definition) of the functions.
- notice how the `Shape` class is passed by-address and how the `BoundingBox` class is passed by-reference.

4.7.7 Implementing a new polygon shape

spot

A polygon is stored as a sequence of points and graphically displayed as a sequence of line segments connecting these points.

The computation of a polygon's area and of the number of counts which are enclosed within a polygon's perimeter need algorithms' developments and will be tackled in a different project.

To simplify the code, the drawing of a polygon will be performed as long as the mouse's button is pressed down. As soon as it is released, the drawing will be considered as done. Don't forget to only store the polygon's points when the current point is different from the previous one.

To better grasp the overall flow of events when a *mouse event* is triggered, here is the method call sequence:

- `Visu2D::mouseMoveEvent(...)`, `Visu2D::mouseReleaseEvent(...)`, `Visu2D::mousePressEvent(...)` will call:
- `Visue2D::HandleMouse(...)` which itself will call:
- `Visu2D::GenerateShape(...)`, in which you'll find the various actions associated with the mouse and the various methods of `Selection` being called.

You can browse the documentation of `Visu2D` to better grok this mechanism.

1. Create the files `polygon.h` and `polygon.cpp`. The list of points will be stored in a `std::vector` of `Point` during the mouse's movement.
2. Update the `cmt/requirements` file to know about the `polygon.cpp` file.
3. Overload the methods of the `Shape` class which aren't suitable for the `Polygon` class. As for the `drawEllipse` method used in a previous session, the method `drawLine` allowing to draw a line segment is documented in Qt.

   In the `bool Polygon::ModifyEndShape()` method, add the first corner of the polygon at the end of the `std::vector` in order to have a closed polygon.
4. Complete the method `Selection::AllocateShape(SHAPE_TYPE type)`
5. Add a button in `WFrame` for the polygon. An icon is provided by `Polygon.bmp`
6. Update the statistics in `DrawQt` if it hasn't been done already.
7. Update the read/write methods for the polygon.

> **Warning: For those who already tackled the ‘‘operator overloading‘‘ project**
> The `Polygon` holds parameters which are different than the other shapes. During the writing/reading of this shape, it will be unavoidable to detect when you are dealing with such a special shape. To call a specific method of `Polygon` with your `Shape` object, you will have to resort to cast this object into a `Polygon`:
>
> ```
> Shape *s = ...;
> Polygon *poly = static_cast<Polygon*>(s);
> ```

4.7.8 Estimating the mask of a polygon shape

spot

A mask is an image in which each pixel has for a value:

- `1` if it is inside the polygon or exactly on its contour,
- `0` if it is outside the polygon

The proposed solution to tackle this exercize is to use a recursive method: a function (or method) which calls itself. The main steps of such an algorithm are:

- mask initialization: the pixels corresponding to the corners of the polygon are set to `1`, the others are set to `0`,
- contour closure: discontinuities may stem from the fact that the corners aren't necessarily close to each other, which will be an issue at a later stage of the algorithm. This step is hence used to make the contour continuous.
- determination of a point within the contour (seed): simple algorithm for a polygon similar to a convex polygon (but harder and harder for more complex polygons)
- filling the inner area of the contour: starting from the seed, call a function which recursively modify the value of its 4 neighbours (changing it from `0` to `1`.)

---

1. in the class `Polygon`, declare a data member `m_mask` of type `Image`. Add a method:

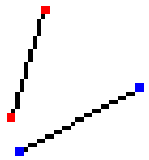
   ```
   void Polygon::InitMask();
   ```

   Instantiate the `m_mask` object with dimensions identical to the ones of the `m_box BoundingBox` (it isn't necessary for the mask to have the dimensions of the image.) Initialize all the pixels of the mask to `0`, except those stored in `m_polygon` which will be set to `1`.

2. Closing the polygon. For each pair of consecutive corners:

   - compute the slope of the segment joining them,
   - for each point of this segment, set the pixel to `1` in `m_mask`.

     Attention: if the slope is `< 1` (in absolute value,) the loop must run over the `x` (blue case.) Otherwise, it must run over the `y` (red case.)

     At this stage, the contour should be completely closed.

3. Determination of a point within the polygon (seed). A simple algorithm consists in counting – starting from a point not on the contour (*i.e.* the seed) – the **number of points on an horizontal line passing by the seed point, intersecting with the contour** in an arbitrary chosen direction (left or right.) If the count is odd, the point is inside the contour. Otherwise, it is outside.

4. Filling the contour. A recursive function is a function which calls itself. Beware to the boundary conditions to prevent such a function to indefinitely call itself (hence using all the stack memory and then crashing your program.)

   Here, if a calling pixel `(x,y)` was set to `0`, the method changes it to `1` and the method is then applied on each of the 4 direct neighbours (right, left, up and down.) Such a process is stopped when the calling pixel is already set to `1` (the contour is reached) or when the image border is reached (which shouldn't happen if the contour was correctly closed to start with and if the seed was correctly put inside the contour.)

   ```
   void Polygon::Update4Neighbours(Image &mask, int x, int y)
   {
     if (x < 0 || x >= mask.GetNX() ||
   ```

```
        y < 0 || y >= mask.GetNY() ||
        mask.GetData(x, y) ) {
      return;
    }

    mask.SetData(true, x, y);
    Update4Neighbours(mask, x+1, y  );
    Update4Neighbours(mask, x-1, y  );
    Update4Neighbours(mask, x,   y-1);
    Update4Neighbours(mask, x,   y+1);
  }
```

5. One should display the mask to check it indeed contains `ones` inside the contour and `zeroes` around:

```
for (y = 0; y < m_mask.GetNY(); ++y) {
  for (x = 0; x < m_mask.GetNX(); ++x) {
    std::cout << m_mask.GetData(x,y);
  }
  std::cout << "\n";
}
```

6. This mask must now be used to decide if a given pixel falls within or outside a polygon, and thus compute the area of the polygon (and compute the number of counts.)

The algorithm described above is rather limited (*w.r.t* the seed search) because it is rather difficult to be sure the point is inside the polygon. One can write a more complex and sophisticated algorithm but one will (almost) always find an edge case. That's why we've actually chosen a different approach in the solution below. The principle of this alternative consists in applying the recursive method on the points outside the contour (a `0` pixel at the image's border is bound to be outside the contour.) If one applies this method as long as there are `0` pixels on the image's border, then one gets a sort of negative of the mask, *i.e.* a mask whose pixels are `1` if they are outside the polygon (or on the contour) and `0` otherwise. One then just has to invert the mask except for the pixels on the contour itself (which need to stay set to `1`.) The implementation of this method needs another local variable `contour` of type `Image` to store the pixels on the contour. The code below replaces the steps `3` and `4`:

```
// initialize the mask with the closed contour
for (y = 0; y < m_mask.GetNY(); ++y) {
  for (x = 0; x < m_mask.GetNX(); ++x) {
    m_mask.SetData(contour.GetData(x, y), x, y);
  }
}

// call the recursive method for the left- and right-border pixels
for (y = 0; y < m_mask.GetNY(); ++y) {
  if (!m_mask.GetData(0, y)) {
    Update4Neighbours(m_mask, 0, y);
  }

  if (!m_mask.GetData(m_mask.GetNX()-1, y)) {
    Update4Neighbours(m_mask, m_mask.GetNX()-1, y);
  }
}

// call the recursive method for the up- and bottom-border pixels
for (x = 0; x < m_mask.GetNX(); ++x) {
  if (!m_mask.GetData(x, 0)) {
    Update4Neighbours(m_mask, x, 0);
  }
```

```
  if (!m_mask.GetData(x, m_mask.GetNY()-1)) {
    Update4Neighbours(m_mask, x, m_mask.GetNY()-1);
  }
}

// inverting the 0s and 1s, except for the contour
for (y = 0; y < m_mask.GetNY(); ++y) {
  for ( x = 0; x < m_mask.GetNX(); ++x) {
    if (!contour.GetData(x, y)) {
      m_mask.SetData(1-m_mask.GetData(x, y), x, y);
    }
  }
}
```

4.7.9 Estimating the mask of a polygon shape with `Qt` tools

spot

We wish to use the `Qt` classes to initialize the polygon shape mask. As for the *corresponding algorithm development*, the mask should hold the value:

- `1` if a pixel is inside the polygon or on its contour,
- `0` otherwise.

FYI, the `Qt` class to use is `QBitmap`.

The main steps of this exercize are:

- to instantiate a `QBitmap` object,
- to associate it with a `QPainter` object so as to draw the contour along the same principle than the on-screen display,
- fill the contour using a `QBrush`,
- initialize `m_mask` from the bitmap.

4.7.10 Editing the color of shapes

Very few instructions because very few things to actually do!

- As for when you added the ellipse, circle, ... buttons in the `DrawQt` interface, add a new button which will act as a color selector. An icon is available in the directory `data/icons`: `Colors.bmp`
- Connect this button to the action:

  ```
  void Visu2D::SelectColor();
  ```

  which you'll implement so as to display a color selector and retrieve the chosen value.

  Note: To display a color selector, use the QColordialog class. If your `SelectColor()` is bigger than **3 lines** then you probably are heeding toward the wrong direction...

- Now, it's up to you to make sure the shapes' selection is painted with the right color.

4.7.11 Deleting already displayed shapes

As for the previous project, very few instructions because not much to do!

- As for when you added the ellipse, circle, ... buttons in the `DrawQt` interface, add a new button which will act as an eraser. An icon is available in the directory `data/icons`: `Eraser.bmp`
- Connect this button to the action:

  ```
  void Visu2D::EraseShape();
  ```

  which you'll implement so as to erase all the displayed shapes. Don't forget to call the method `update()` at the end of your `EraseShape()` method (so the window can be refreshed.)

4.7.12 Shapes with different roles

To improve the signal over background ratio (SBR) of a given image, we estimate the average noise in an image region containing *a priori* only background and we subtract it to the regions containing the signal.

In order to achieve such a laudable goal, we aim at being able to discriminate between 2 types of shapes: those containing signal and those containing noise. The final information will be the intensity difference between the averaged intensity in the *signal* shapes and in the *noise* shapes.

In practice, the user will have to choose *a priori* if the shape she is drawing corresponds to a *signal* shape or to a *noise* one. A different color will be associated to each shape according to its type (or role.)

The main modifications will consist in:

- add a button in the tools bar to select the role of the shape to draw,
- modify the list of data members of the `Selection` class. You could, either:
  - create a new data member `m_noiseShapes` of type `std::vector<Shape*>`, or
  - modify the `m_shapes` variable into a 2d array (*e.g.:* index `0` to store the *signal* shapes and index `1` for the *noise* ones. For clarity's sake, one should probably use an `enum` such as: `enum SHAPE_ROLE {SIGNAL=0, NOISE, NB_ROLES};`)

    ```
    std::vector< Shape* > m_shapes[NB_ROLES];
    ```

- modify the `Selection` class to take into account this new information. Most of the methods of this class must be amended. For example:

  ```
  Selection::AllocateShape(SHAPE_TYPE type);
  ```

  should become:

  ```
  Selection::AllocateShape(SHAPE_TYPE type, SHAPE_ROLE role);
  ```

- apply the necessary modifications to allow the displayed color of a shape to be dependant on its role.
- in the `WStatistics` dialog box, add a widget to display the difference of intensities between the average intensity in the *signal* shapes and the average intensity for the *noise* shapes. Such a modification can be performed on the `wstatistics.ui` file with `Qt designer`. Finally, apply the necessary modifications to update this item.

4.7.13 Saving shapes in an `XML` file

spot

One wishes to save shapes in a somewhat more 'standard' file format in order to be able to share these files with other applications. Therefore, we'll use an XML file format. More informations on QtXML

Our file will have the following structure:

```
<!DOCTYPE Selection PUBLIC '' ''>
<!--This file is created by DrawQt-->
<selection>
 <shape number="1" >
  <type>square</type>
  <origin x="111" y="32" />
  <dimension width="19" height="19" />
 </shape>
 <shape number="2" >
  <type>rectangle</type>
  <origin x="23" y="16" />
  <dimension width="57" height="31" />
 </shape>
 <shape number="3" >
  <type>ellipse</type>
  <origin x="174" y="43" />
  <dimension width="46" height="47" />
 </shape>
 <shape number="4" >
  <type>circle</type>
  <origin x="117" y="79" />
  <dimension width="20" height="20" />
 </shape>
 <shape number="5" >
  <type>polygon</type>
  <nb_points value="5" />
  <point x="103" y="106" />
  <point x="86" y="108" />
  <point x="71" y="118" />
  <point x="67" y="133" />
  <point x="118" y="115" />
 </shape>
</selection>
```

The DOM model will be used to read/write our XML file. This means the XML file will be entirely loaded in memory as a tree. One will have to iterate on the nodes of the tree and retrieve the interesting informations (and discard the other ones.)

There is another model – called SAX – which we won't use during this exercize (*FYI*, in that model, the file is read piecewise on the fly and callbacks that you specify are called for each type of node.)

The first thing to mention is that Qt doesn't verify (yet) if the XML file is valid (*i.e.* if the file has the expected XML structure.) To address this issue, one can use external tools (and specify a DTD), but this isn't required for this exercize.

We'll hence assert the following hypothesis: the input file is well formed. It will thus be correctly read by Qt (*e.g.* there won't be any tag left open.) Moreover, the file will contain informations which are expected: we won't bother with error recovery.

A good online documentation on the Qt functions and XML can be found here.

A few words on the basics...

Take the file saving case:

```
/* file structure we want to obtain:
 <!DOCTYPE Selection PUBLIC '' ''>
  <!--This file is created by DrawQt-->
  <selection>
    <shape number="1" >
      <type>square</type>
    </shape>
  </selection>
 */

// create an instance of our DOM implementation
QDomImplementation impl = QDomDocument().implementation();

// create the document name
// <!DOCTYPE Selection PUBLIC '' ''>
QString name = "Selection";
QDomDocument doc(impl.createDocumentType(name, "", ""));

// add a comment
// <!--This file is created by DrawQt-->
doc.appendChild(doc.createComment("This file is created by DrawQt"));
doc.appendChild(doc.createTextNode("\n"));

// create the root node
// <selection>
QDomElement selectionNode = doc.createElement("selection");

// add this node to the document
doc.appendChild(selectionNode);

// create a shape-node and add an attribute
// <shape number="1" >
QDomeElement shapeNode = doc.createElement("shape");
shapeNode.setAttribute("number", idx+1);

// create a type-node and add an attribute
// <type>square</type>
QDomElement typeNode = doc.createElement("type");
typeNode.appendChild(doc.createTextNode(QString("square")));

// add this type-node to the shape-node
shapeNode.appendChild(typeNode);

// add this shape-node to the selection
selectionNode.appendChild(shapeNode);

// write the document into the file
outstream << doc.toString().toStdString();
```

Now, the reading part:

```
/* file structure after saving:
  <!DOCTYPE Selection PUBLIC '' ''>
  <!--This file is created by DrawQt-->
  <selection>
    <shape number="1" >
      <type>square</type>
    </shape>
```

```
  </selection>
 */

// create an instance of a DOM document
QDomDocument doc;

// read the tree via the file
QFile f(fileName.c_str());
if (!f.open(QIODevice::ReadOnly)) {
  return false;
}

doc.setContent(&f);
f.close();

// now, all the elements of the file are loaded into the tree.
// we just have to iterate over those.

// retrieve the root element
QDomElement root = doc.documentElement();

// check the root is indeed the "selection" element
if (root.tagName() != QString("selection")) {
  return false;
}

// fetch the first child of "selection"
QDomElement child = root.firstChild().toElement();

// iterate over all children
while (!child.isNull()) {
  // child is a "shape"
  if (child.tagName() == QString("shape")) {
    // retrieve its attribute and display it
    std::cout << "shape number ["
              << child.attribute("number","0").toInt()
              << "]" << std::endl;

    // read the next element
    QDomElement shapeNode = child.firstChild().toElement();

    // iterate over all children
    while (!shapeNode.isNull()) {
      // child is a "type" node
      if (shapeNode.tagName() == QString("type")) {
        // retrieve its attribute and display it
        std::cout << "type [" << shapeNode.text().toStdString() << "]"
                  << std::endl;
      }
      // read next element
      shapeNode = shapeNode.nextSibling().toElement();
    }
  }
  // read next element
  child = child.nextSibling().toElement();
}
```

So you'll have to add the following methods to the `Selection` class:

```
/**
 * @brief Save the coordinates of the shapes into the file 'fileName'
 * @param fileName: name of the output file
 * @return true if everything's OK.
 */
bool Selection::SaveXML(const std::string& fileName);

/**
 * @brief Read the coordinates of the shapes saved into the file 'fileName' and
 *        print them out
 * @param fileName: name of the input file
 * @return true if everything's OK.
 */
bool Selection::LoadXML(const std::string& fileName);
```

You'll also have to redefine the stream operators (<< and >>) of the `Selection`, `BoundingBox` and `Polygon` to take into account the `XML` elements overloads.

e.g. for the `BoundingBox`:

```
QDomElement& operator<< (QDomElement &shapeNode, const BoundingBox &bb);
```

An `XML` input file is available `here`.

---

spot

At this stage, various independent but complementary developments are offered.

- on the graphical side:
  - *Moving* a shape: ability to move an already existing shape. `(**)`
  - *Changing the color* of already displayed shapes. `(*)`
  - *Creating a new Polygon shape*. `(*)`
- graphics, on the side:
  - *Handling many shapes*: ability to concurrently represent many shapes on the screen. `(**)`
  - *Deleting displayed shapes* `(*)`
- a bit of algorithmics:
  - *Estimating the mask of a polygon shape*: this mask is used to compute the area of a polygon and count the number of events. The thorny issue in this development is the algorithmics development. `(***)`
- miscellaneous and optional:
  - *Displaying the average intensity* in a shape (instead of a count ratio): involves the usage of `qt-designer`. `(**)`
  - *Saving shapes in a XML format* `(***)`
  - *Saving an image in a binary file* and reading it back `(**)`
  - *Handling 2 types of shapes* corresponding to either a signal region or a background region and subtracting the average values. `(***)`
  - *Internationalization*: using the tools provided by `Qt` to easily generate various versions of `DrawQt` in different languages. `(**)`
  - *Operators overloading*: Saving/loading shapes via operators overloading. `(**)`

  - *Estimating the mask of a polygon shape with Qt tools*: it is advised to have already completed the algorithmics development before tackling this one. (**)

- (*) : easy
- (**) : average
- (***) : more difficult

4.8 Appendix

4.8.1 Using the terminal under MacOS X

First steps

In the following, the *prompt* will be represented as $>. The *prompt* may differ from one account to the other (*i.e.* it may be heavily customized by users or system administrators.)

- *Logging in*: your *login name* is ens-<n> where <n> runs from 31 to 50 depending from geographical location in the lab room. Your *password*, for the first connection, is ens<AAAA> where <AAAA> is the current year.

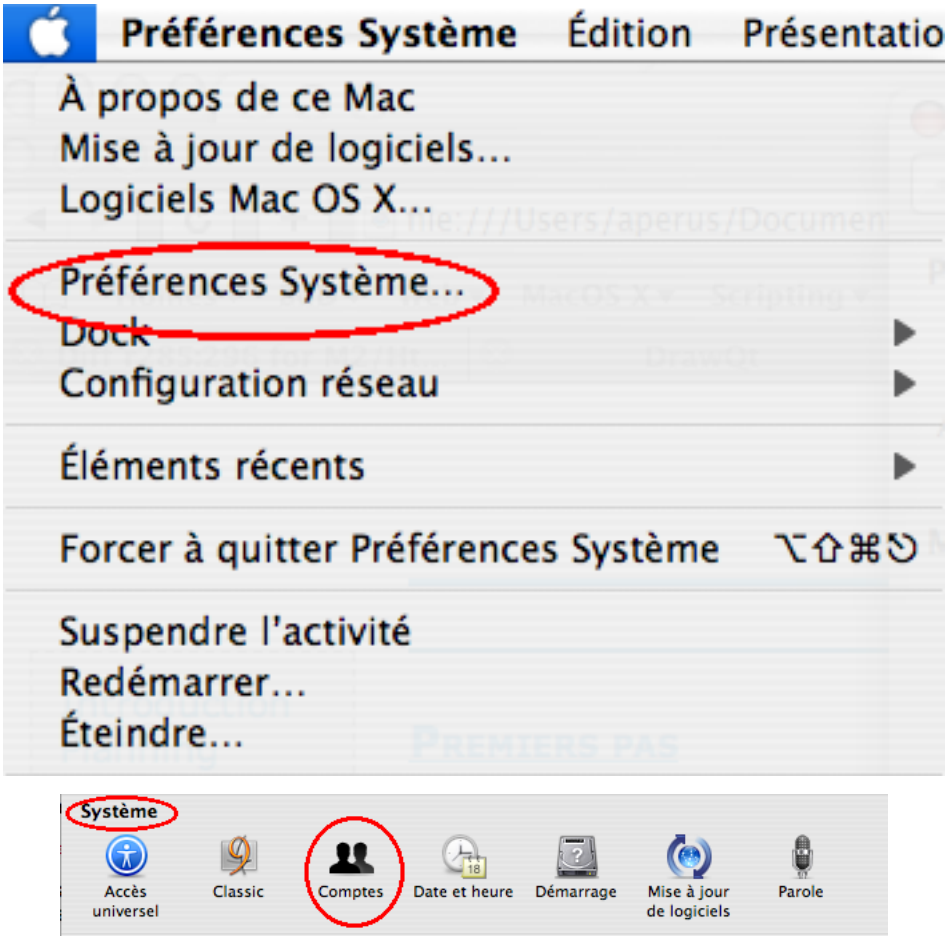
  Note: passwords are case sensitive.

- *changing your password*: the first thing to do, after the first connection, is to change your password in:

  Apple Menu > *System Preferences* > *Accounts*

  or, in french:

  Menu Pomme > *Preferences Systeme* > *Comptes*



A good password should not be found in any dictionnary (neither french nor english nor ...), contains one or more numbers and/or other non-alphanumeric characters. It must nonetheless be easy to remember (w/o being ever written down.)

- *launching the Terminal application*:

  In the `Dock` (the icons bar at the bottom of the screen,) click on the `Terminal` application icon:

- *launching the Safari webbrowser*:

  Most of the documentation is hypertext documents which will be accessed with the `Safari` webbrowser. Either by issueing in your `Terminal` window:

```
$> open -a Safari
```

  or by clicking on the `Safari` icon in the `Dock`:

- *a few characters/key-combination associations*:

character	key combination
{	Atl-(
}	Alt-)
[	Alt-Shift-(
]	Alt-Shift-)
~	Alt-n-space
|	Alt-Shift-L
button 2	Control-click
button 3	Apple-click
\	Alt-Shift-/`

A few `unix` commands from a `terminal`

- *command syntax*:

  Generally speaking, a command issued in a terminal has the following syntax:

  ```
  $> command_name parameter0 parameter1 ...
  ```

  Some parameters may be interpreted as options which may modify the behaviour of the command. These parameters-options are usually a single character prefixed with the − character. Other parameters may actually be the input arguments to the command.

  Examples:

  – a command without any parameter:

  ```
  $> ls
  test.f  metafile.ps  mytest  x.f
  ```

  – a command with a parameter:

  ```
  $> ls *.f
  test.f  x.f
  ```

  – a command with an option:

  ```
  $> ls -l
  total 10
  -rw-r--r--  1 ens   ENS    3 Jan 20 20:56 test.f
  -rw-r--r--  1 ens   ENS    9 Jan 20 20:57 metafile.ps
  -rwxr-xr-x  1 ens   ENS    5 Jan 20 20:57 myest
  -rw-r--r--  1 ens   ENS    2 Jan 20 20:57 x.f
  ```

  – a command with an option and a parameter:

  ```
  $> ls -l *.f
  total 10
  -rw-r--r--  1 ens   ENS    3 Jan 20 20:56 test.f
  -rw-r--r--  1 ens   ENS    2 Jan 20 20:57 x.f
  ```

- *inline help*:

  Under `Unix`, the inline help is called `man` (for `manual`.) Here is some online documentation about `man` on `MacOS`. Here is the equivalent for `Linux`.

  ```
  $> man command_I_wish_to_know_the_manual_of
  ```

  If one doesn't know the exact name of the command one wishes to be enlightened with, one can also run a keyword-based query in the manual using the `-k` option, followed by a keyword:

```
$> man -k compilation
```

This will provide you with the list of commands whose description line contains the word *'compilation'*.

To exit the `man` page, hit `q` or `Control-C`

If there is no `man` page for a command, one can usually try to reach the help page of that command by running the command with `-h`, `-H` or `-?`.

For example, `gs -h` can be used to know about the possible options and parameters for the `Ghostscript` tool (which interprets the `postscript` files on-screen.)

How to organize and handle files

- **Files and directories**

  A file name can contain any printable character except a slash `/` nor a space. However, it is recommanded not to use any character with special meaning for the shell, such as:

  ```
  * ? < > | ; [ ] & ( ) % # $ \
  ```

  Some files have a name with a leading dot `.`. Especially in the login directories (*i.e.* your home directory,) these files generally contain initialization parameters and configuration options. These files are granted special treatment by the file handling commands (such as `ls`, `rm`, `mv`, `cp`...)

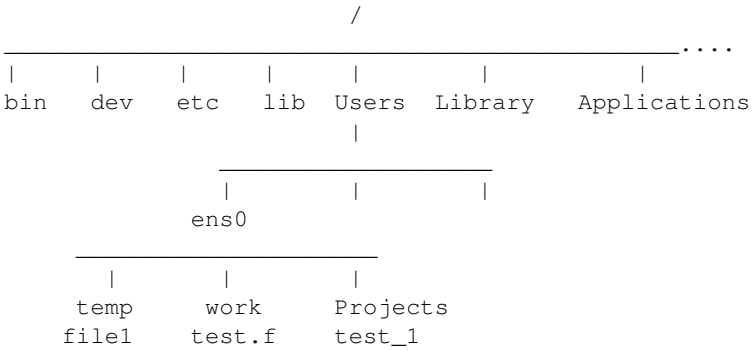
  For example, the command:

  ```
  $> ls *
  ```

  won't list these special files.

  The files' organization is hierarchical: each level in the hierarchy is called a *directory*. A directory is actually a file which contains a list of files, among which one can find other directories, called subdirectories, among which...

  The whole set of files is mounted under a unique hierarchy whose the main root is called `root` and designated by `/`. The following schema should clarify:

  The complete name of a file is built by specifying the hierarchy of directories which holds it, separated by slashes `/`.

  - if the name starts by a `/`, it is an absolute path (*i.e.* a path starting from the root of the filesystem,)
  - otherwise, it is a relative path, *i.e.* from the current directory. For each position in the hierarchy, there is a *current directory*, also called *work(ing) directory*, which is considered by default if one does not explictly qualify the access path.

    Example:

if the current directory is `/Users/ens0`, to refer to the file `file1` which is located under the subdirectory `temp`, one can either access via:

```
/Users/ens0/temp/file1
```

or via:

```
temp/file1
```

- `.` and `..` are – respectively – the current directory and the parent directory.
- ~ is the base directory of the current user (*i.e.* her home directory)

- **a few useful commands to work with files and directories**:

command name	purpose
`pwd`	print working directory
`cd rep`	change to (sub)directory `rep`
`mkdir rep`	make the (sub)directory `rep`
`rmdir rep`	remove the (sub)directory `rep`
`ls`	list the directory content
`cp`	copy
`mv`	move or rename
`rm`	remove a file
`more`	a simple pager (displays the content of a file page by page)

- **find a file**

  It is possible to find and select a file from the metadata attached to it (name, date of last access, date of last modification, size, ...) thanks to the `find` command. `MacOS` documentation can be found here. `Linux` documentation can be found there.

  This command walks thru the filesystem tree and executes an action for each and every file selected. *e.g.:* the `-print` action will print the name of each selected file.

  ```
  $> find root_directory [-criterions] [-actions]
  ```

  Example:

  ```
  $> find /Users/ens0 -name preferences -print
  ```

  The command `find` starts from the `/Users/ens0` directory, walks thru the whole sub-tree, looking for files which have as a name `preferences` and then prints out the absolute path of each such file.

  The following command will list all the files whose name starts with `pref` and are under your home directory:

  ```
  $> find ~ -name "pref*" -print
  ```

- **looking for a string in a file**

  The command `grep` prints all the lines containing a given *expression* (a string) in a file:

  ```
  $> grep [options] expression [file...]
  ```

  Example:

  ```
  $> grep 'rectangle' ~/Projects/DrawQt/src/*.cpp
  ```

  This command will return the list of the lines in all the `.cpp` files (under `~/Projects/DrawQt/src`) which contain the string `rectangle` (but **not** `Rectangle`. If you want a case insensitive search, pass `-i` as an option to `grep`.)

The `grep` command allows to precisely describe the type of string being looked for and its relative position in the lines being investigated.

`MacOS` documentation can be found here. `Linux` documentation can be found over there.

4.8.2 Compilation errors

Checking errors

If everything is ok, when you build your program, you should have something as the hereafter message with, at the end, the sequence all ok:

```
$> cmt make
------> (Makefile.header) Rebuilding constituents.make
------> (constituents.make) Rebuilding setup.make Darwin.make
CMTCONFIG=Darwin
...
...
...
all ok.
```

But sometimes a insidious bug unawares arises. In this case the build command won't end with all ok but with something as:

```
> ../src/hello.cpp: In function 'int main()':
> ../src/hello.cpp:16: error: 'cout' was not declared in this scope
```

So the first thing: check the messages sent by the build command!

An error is always reported with error. The compiler will also specify:

- in which file the error arises,
- at which line, and
- how it understand them.

Example:

src/hello.cpp:153:error: expected '}' before 'else'

> **Warning:** Sometimes a first error will be followed by a lot of other errors..

For example if a brace is missing:

```
src/hello.cpp:153: error: expected '}' before 'else'
src/hello.cpp:153: error: expected '}' before 'else'
src/hello.cpp:156: error: break statement not within loop or switch
src/hello.cpp: At global scope:
src/hello.cpp:159: error: expected declaration before '}' token
```

The first error is followed by another one, so that the best thing to do is to correct the first one before go on with the second one.

Correcting warnings

It is a good idea to correct the warning messages too. These arise as:

```
src/hello.cpp:92: warning: unused variable 'titi'
```

Some examples

- Include file missing :

```
> ../src/hello.cpp: In function 'int main()':
> ../src/hello.cpp:16: error: 'cout' was not declared in this scope
```

The incriminated line contents: `std::cout << "Hello!" << std::endl;`

The compiler doesn't find `cout()`, this method being not declared. The include of the file `iostream` is missing:

```
#include <iostream>
```

- 'ambiguous overload' :

```
> ../src/visu2d.cpp: In member function 'void Visu2D::SaveSelection()':
> ../src/visu2d.cpp:201: error: ambiguous overload for 'operator==' in 'fname == 0'
> ../src/visu2d.cpp:201: note: candidates are: operator==(const char*, const char*) <built-in>
> ../src/visu2d.cpp:201: note:                 operator==(QNoImplicitBoolCast, int) <built-in>
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:290: note:                 bool QString::
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:723: note:                 bool QString::
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:340: note:                 bool QString::
> /usr/local/qt/lib/QtCore.framework/Headers/qbytearray.h:427: note:              bool operator=
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:735: note:                 bool operator=
> ../src/visu2d.cpp: In member function 'bool Visu2D::ReadSelection()':
> ../src/visu2d.cpp:214: error: ambiguous overload for 'operator==' in 'fname == 0'
> ../src/visu2d.cpp:214: note: candidates are: operator==(const char*, const char*) <built-in>
> ../src/visu2d.cpp:214: note:                 operator==(QNoImplicitBoolCast, int) <built-in>
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:290: note:                 bool QString::
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:723: note:                 bool QString::
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:340: note:                 bool QString::
> /usr/local/qt/lib/QtCore.framework/Headers/qbytearray.h:427: note:              bool operator=
> /usr/local/qt/lib/QtCore.framework/Headers/qstring.h:735: note:                 bool operator=
> make[3]: *** [../Darwin/visu2d.o] error 1
```

This error, although more verbose, is simpler to resolve than it seems. Looking carefully, the first error is the following:

```
> ../src/visu2d.cpp:201: error: ambiguous overload for 'operator==' in 'fname == 0'
```

The corresponding line in the file is:

```
200 :  QString fname = QFileDialog::getOpenFileName(this, "Choose a file", ".", "Images (*.sel)"
201 :  if ( fname == 0)
```

The variable `fname` has the `QString` type and the compiler can not apply the operator ==. It is proposing several solutions:

```
>note:                 bool QString::operator==(const QString&) const
>note:                 bool QString::operator==(const char*) const
>note:                 bool QString::operator==(const QByteArray&) const
>note:                 bool operator==(const char*, const QByteArray&)
>note:                 bool operator==(const char*, const QString&)
```

Choose the right one ! In our case, the best solution is the following:

```
201 :  if ( fname == QString(""))
```

- 'No such file' :

```
> Headers -F/usr/local/qt/lib -DQT3_SUPPORT    ../src/myWindow.cpp
> ../src/myWindow.cpp:15:20: error: wframe.h: No such file or directory
```

The compiler didn't find the file `wframe.h`. There are several hints:

  - the file doesn't exist
  - wrong spelling
  - problem during including the file

```
#include <iostream>
```

Means that the file should be in the lookup directories scanned by the compiler by default such system libraries. For example: `iostream`, `fstream` ...

```
#include "wframe.h"
```

This file should be in one among the paths given to the compiler by the build command. There `CMT` is the manager and the include paths are specified by `CMT` with the `requirements` file and the *`include_dirs`* directive. By default the `include` directory is included.

- Example 4 :

```
> In file included from ../src/myWindow.cpp:12:
> ../src/myWindow.h:4:29: error: QtGui/QMainWindow: No such file or directory
> ../src/myWindow.h:5:29: error: QtGui/QPushButton: No such file or directory
> ../src/myWindow.cpp:13:17: error: QMenu: No such file or directory
> ../src/myWindow.cpp:14:20: error: QMenuBar: No such file or directory
```

4.8.3 `CMT`: a tool for the configuration management

Introduction

The software configuration (this is a rather crude definition) is the activity in charge of description and control of all that constitutes a software project. This includes such various informations as:

- the name of the author of the software,
- project managers,
- the structural decomposition of the project in independent or correlated parts ("packages")
- resources needed to build or use of the software,
- description of components (libraries, applications, etc. ...),
- actions to build the software (compilation, link editing, etc. ...).

For example, simply building an application, even simple, but made with some source modules, and using some libraries (eg graphics) quickly becomes tedious if one only manually chain the commands for compiling and editing links , especially memorizing options compilations that quickly become very complex.

The first tool available is the tool **make**. This tool can fully describe the components of an application in terms of source files, libraries, compiled modules to be assembled, etc. It can detect among all possible actions for the reconstruction of a software product, needed after partial modification of the source files.

However, the tool will become very complex to handle. Indeed, we must describe in a text file called traditionally `Makefile` all rules of dependency between the modules, then all actions to be taken to make each item. Detailed knowledge of all options to compile, link etc. .. is necessary, and **make** has no way to know the structure of the development (where are the sources, the compilation, packages produced by other persons of the same project) or to manage information specific to a given platform.

A number of tools (mostly commercial) exist in some environments to meet a little better on these issues. One can cite `MS Visual` in Windows environments, and in the Unix world, many commercial products provide similar services.

We will consider a tool (not commercial) called `CMT` used in our physical environment, which automates the setup process significantly. One can also read the complete documentation of this tool.

The `CMT` tool

Among the services provided by this tool, we will mainly consider here:

- steering `CMT`
- description of the source files of an application to automatically generate the necessary files needed by **make** .
- the ability to modify the compilation options or linking.
- the ability to build reusable libraries.
- the declaration of the use of external (external packages)

How to control `CMT`

Each work location (ie where we will produce an application for example) is called a package in `CMT` terminology. Each package will be described by a text file named `requirements` installed in its directory `cmt` . `CMT` will find in this file all informations needed to manage the configuration of the product. We will now describe some of this information, those most often used.

When it was installed for the first time such a file `requirements` in a directory, thus defining a package, it is necessary to configure the package with the command **cmt config** to run locally, and only once. This will produce a universal `Makefile` (that means that remain unchanged regardless of changes in the package).

In the future use of the tool **make** the command make , `CMT` will be used transparently to automatically regenerate the correct configuration settings of **make**.

Description of the source files for an application

If you want to build an application, we must give it a name (eg `foo`). The name is used among others things to name the executable file (**toto.exe**).

```
application foo foo.cpp ../somewhere/a.cpp -s=../lib x.cpp y.cpp z.cpp
```

Then we need to specify all source files that will be compiled and then assembled to make the application. These source files can be in the proper directory of the package (the current directory) or not. In the example shown here, we have:

- `foo.cpp` is a source file in the current directory
- `a.cpp` is a source file in the directory `../somewhere`
- option *-s=../lib* indicates that the next source files will be searched in this directory (until the next option *-s=*)

Changing compiler options or editing link

The configuration of the tool **make** is done through standardized macros, each one corresponding to a compiler or a special tool. For C++ (as we are concerned primarily) we can consider the macro `CPPFLAGS` and `cpplinkflags` .

Generally, this will increase these macros, as they have already received a definition used by other packages or by `CMT` itself. This is shown in the following example:

```
macro_append CPPFLAGS "-D__USE_STD_IOSTREAM"
```

Be careful to observe scrupulously the uppercase and lowercase letters, and many include a space character before option added (so here before *-D...*).

Construction of libraries

A library can pre-compile and pre-assemble C++ modules so that multiple applications can use them without having to rebuild them systematically.

We define a library with the following statement, put in the `requirements`, giving it a name and describing the source files (as for applications):

```
library event a.cpp b.cpp c.cpp
```

Then, for an application (`foo` , for example) uses this library, it is sufficient to install a dedicated macro as follows:

```
macro foolinkopts "L-.-levent"
```

which means in this example that the application foo will be linked with the library named `event` and located in the current directory (option *-L.*).

Statement by the use of external

Numerous external packages, usually providing specialized libraries can be registered to `CMT` and be referenced easily through the following instructions:

```
use OPAC v3
use Ci   v5r2
```

Each of these instructions gives automatic and transparent access to the libraries provided by these packages.

Using `CMT`

```
$> cd a_working_directory
$> cmt create MyPackage v1
$> cd MyPackage/cmt
$> vi ../src/...
$> vi requirements
$> source setup.sh
$> cmt make
```

For more information, visit the website CMT

4.8.4 Inputs/Outputs (I/O) in C++

Streams

The class `std::ifstream` implements input operations on file based streams.

- In order to use the class `std::ifstream` you need to include the file `fstream` with the directive :

```
#include <fstream>
```

- the constructor :

  `std::ifstream::`**`ifstream`**(const char* *name*)

  constructs the named file stream:

```
{
  std::ifstream f ("figures.dat");

  if (!f)
  {
    std::cerr << "Cannot open the file figures.dat" << std::endl;
    return (1);
  }
  ...
}
```

- the `eof()` method

  bool `std::ifstream::`**`eof`**`() const`

  returns `true` if the end of the given file stream has been reached.

```
{
  std::ifstream f ("figures.dat");

  while (!f.eof ())
  {
    ... // Loop on file reading
  }

  f.close ();
  ...
}
```

- the `close()` method

  void `std::ifstream::`**`close`**`()`

  closes the file.

- the operator >> enables reading the next word.

```
{
  std::ifstream f ("figures.dat");

  // Reading word by word
  while (!f.eof ())
  {
```

```
    std::string word;

    f >> word;
  }

  f.close ();
  ...
}
```

- the `getline()` method offers two versions:

  1. std::istream& `std::`**`getline`**(char* *str*, int *count*)

     reads characters from in input stream until delimiting character (*end-of-line per default*) is found and saves them to the given string str. If delimiter is found, it is discarded.

     ```
     {
       std::ifstream f ("figures.dat");

       // Line by line reading
       while (!f.eof ())
       {
         char ligne[256];
         std::string s;

         f.getline (ligne, sizeof (ligne));
         s = ligne;
       }

       f.close ();
       ...
     }
     ```

  2. std::istream& `std::`**`getline`**(std::istream& *stream*, std::string& *line*)

     reads characters from in input stream until delimiting character (*end-of-line per default*) is found and saves them to the given string str. If delimiter is found, it is discarded.

     ```
     {
       std::ifstream f ("figures.dat");

       // Line by line reading
       while (!f.eof ())
       {
         std::string s;

         std::getline (f, s);
       }

       f.close ();
       ...
     }
     ```

For more information, see : streams

4.8.5 Structure and format of the image and shapes files

Structure of an `Image` file

Description

`Image` files are `ASCII` files holding the description of one image.

Syntax

For more informations about the grammar below, heed toward this wikipedia page

The complete syntax used in these text files can be described like so:

```
image   ::=   'IMAGE' data 'IMAGE_END'
data    ::=   'WIDTH' <value>
              'HEIGHT' <value>
              'PIXELS' pixels
pixels  ::=  pixel pixels | pixel
pixel   ::=  <value>
```

Note: This type of grammar is a standard. For example, the first line means:

An 'image' is composed of the word `IMAGE` followed by 'data' and finally the token `IMAGE_END`.

To know what 'data' means, one just has to go to the next line.

Structure of a `Shape` file

Description

A shape file is a structured `ASCII` file holding the description of graphical shapes.

- each individual description of a shape is flanked by 2 keywords `SHAPE` and `SHAPE_END`.
- between these 2 keywords, the description of the shape itself is given by:
  - its type (`TYPE`,)
  - the geometrical data of the shape, dependent on the type of the shape.

Syntax

For more informations about the grammar below, heed toward this wikipedia page

The complete syntax used in these text files can be described like so:

```
shapes      ::=   shape | shape shapes | comment shapes
shape       ::=   'SHAPE' <number> 'TYPE' type_name data 'SHAPE_END'
comment     ::=   'COMMENT' ...  'COMMENT_END'
type_name   ::=   'SQUARE' | 'RECTANGLE' | 'CIRCLE' | 'ELLIPSE' | 'POLYGON'
data        ::=   envelope | list
envelope    ::=   'ORIGIN' <x> <y> 'WIDTH' <width> 'HEIGHT' <height>
list        ::=   'NB_POINTS' <number> points
points      ::=   point points | point
point       ::=   'POINT' <x> <y>
```

4.8.6 STL: the (C++) standard template library

C++ itself has very few tools for managing sequences of characters, inputs/outputs and collections.

The STL library provides C++ with a standardized answer using the C++ proper mechanisms as:

- the object approach and capacity for abstraction with the operators
- use of templates
- operators overdefinition

The effective normalization of STL library make it a whole part of C++, and we will always use STL library for the management of strings, IO and collections.

In particular, the overwhelming use of this library has proven its reliability and its level of optimization, both in the use of memory or in performance.

Below we will describe only a few of services offered by STL.

Strings

The `std::string` class manages sequences of characters.

```
#include <string>

int main (int argc, char **argv)
{
  std::string text = "abc defg";

  std::string words[] = { "aaa", "bbb", "ccc", "ddd", "eee" };

  text += words[0];

  return (0);
}
```

Main operations with strings

- Assigning strings

```
std::string s = "abcd";
std::string t = s;

t += "abcd";
```

- String length

```
std::string s = "abcd";

int size = s.size ();
```

- Find content in string

```
std::string s = "abcd";

int pos = s.find ("bc");
```

- Getting C string equivalent for C functions

```
std::string s = "abcd";

if (!strcmp (s.c_str (), "abcd")) ...
...
```

- Use with `std::getline()` fonction

```
std::ifstream f;
std::string s;

std::getline (f, s);
...
```

Vectors

The `std::vector` class provides a linear sequence container for any type of object (which can be copied).

- base types:

```
#include <vector>

int main (int argc, char **argv)
{
  std::vector<int> v;

  for (int i = 0; i < 10; i++) {
    v.push_back (i);
  }
  return (0);
}
```

- user objects:

```
#include <vector>

class A
{
public:
  A (int value) : m_value(value) {}

private:
  int m_value;
};

int main (int argc, char **argv)
{
```

```
  std::vector<A> v;

  for (int i = 0; i < 10; i++) {
    v.push_back (A(i));
  }
  return (0);
}
```

- references or pointers to user objects:

```
#include <vector>

class A
{
public:
  A (int value) : m_value(value) {}

private:
  int m_value;
};

int main (int argc, char **argv)
{
  std::vector<A*> v;

  for (int i = 0; i < 10; i++) {
    v.push_back (new A(i));
  }
  // somehow use v...

  // clean-up: we need to call delete on each
  // of the pointers, otherwise the memory will
  // be leaked...
  for (std::vector<A*>::iterator
         itr = v.begin(),
         iend= v.end();
       itr != iend;
       ++itr) {
    delete *itr;
    // also put the pointer to NULL. useful for debugging
    // and to prevent inadvertant double deletes...
    *itr = 0;
  }

  return (0);
}
```

- vectors of vectors:

```
#include <vector>

int main (int argc, char **argv)
{
  std::vector< std::vector< int > > v;

  for (int x = 0; x < 10; x++) {

    std::vector< int >  column;
    for (int y = 0; y < 10; y++) {
```

```
        column.push_back (y*2);
      }
      v.push_back (column);
    }
    return (0);
```

Main operations acting on a vector

- Adding an element at the end of a vector

  ```
  v.push_back (12);
  ```

- Removing the last element from a vector

  ```
  v.pop_back ();
  ```

- Removing all elements from a vector

  ```
  v.clear ();
  ```

- Getting the first et the last element from a vector

  ```
  int first = v.front ();
  int last  = v.back ();
  ```

- Getting an element by its position

  ```
  int i = v[2];
  ```

- Setting an element localized by its position

  ```
  v.at(2) = 3;
  ```

Iterations in a vector

One can iterate through the elements of a vector using a category of iterator:

```
std::vector<T>::iterator

std::vector<T>::const_iterator

std::vector<T>::reverse_iterator

std::vector<T>::const_reverse_iterator
```

```
#include <iostream>

#include <vector>

typedef std::vector<int> int_vector;

int main (int argc, char **argv)
{
  int_vector v;

  for (int i = 0; i < 10; i++) {
    v.push_back (i);
  }
```

```
  int_vector::iterator it;

  for (it = v.begin (); it != v.end (); ++it) {

    int i = *it;

    std::cout << "i = " << i << std::endl;
  }

  return (0);
}
```

Some operations with iterators on vectors

- Removes an element from a vector

```
std::vector<int> v;
std::vector<int>::iterator it;

it = ...;

v.erase (it);
```

- Insert an element into a vector

```
std::vector<int> v;
std::vector<int>::iterator it;

it = v.begin ();

// Insert before the element pointed by it
v.insert (it, 24);
```

Lists

Lists given by `list` class almost act as vectors, but offer an implementation based on doubly-linked lists providing efficient inserts and removes.

```
#include <list>
#include <iostream>

typedef std::list<int> int_list;

int main (int argc, char **argv)
{
  int_list lst;

  for (int i = 0; i < 10; i++) {

    lst.push_front (i);
    lst.push_back (i);
  }

  int_list::iterator it;
```

```
  for (it = lst.begin (); it != lst.end (); ++it) {

    int i = *it;

    std::cout << "i = " << i << std::endl;
  }

  return 0;
}
```

Maps

The `map` class provides an indexed collection.

```
#include <iostream>

#include <map>
#include <string>

typedef std::map<int,std::string> dictionary;

int main (int argc, char **argv)
{
  dictionary d;

  std::string words[] = { "aaa", "bbb", "ccc", "ddd", "eee" };

  for (int i = 0; i < sizeof (words) / sizeof (std::string); i++) {

    d[i] = words[i];
  }

  dictionary::iterator it;

  for (it = d.begin (); it != d.end (); ++it) {

    int key = (*it).first;

    std::string word = (*it).second;

    std::cout << "word = " << word << std::endl;
  }

  return 0;
}
```

Some references

- SGI: Standard Template Library Programmer's Guide
- Dinkumware
- STL Overview (from Rob Kremer - Calgary)

4.8.7 Introduction to `make`

---

spot

make can manage the:

- maintenance,
- update,
- regeneration and
- installation

of a set of interconnected files by testing their respective dates of last changes.

It is used in two steps:

1. a file usually named `Makefile` or `makefile` describes:
   - the dependencies between different files;
   - the production rules, or how to update, how to rebuild;
2. after any change in one source file, just type *make*;

   Then **make**:
   - examines the dependencies;
   - finds files that are not up-to-date;
   - executes the only necessary commands.

Structure of a Makefile:

A configuration file for **make** may contain essentially three types of lines:

- dependencies
- command lines
- macros

A dependency and the command lines associated is called a rule.

> **Caution:** Do not consider a configuration file `Makefile` as a program. In particular, there is no "sequential instructions" from beginning to end of the file.

The rules:

They are composed of:

- a rule with dependencies

  this is expressed by a non-empty list of targets to be rebuilt, separated by a `space`
  - followed by `:` or `::`,
  - followed by the possibly empty list dependencies;

- a production rule

  expressed by a `TAB`, followed by shell commands to be executed in order to update the target.

Example:

(The characters of a line following a # are ignored by **make**)

```
# Prog.exe is generated from object files
prog.exe: prog.o test.o
      cc -o prog.exe prog.o test.o
# ^^^ This should be a TAB
# the previous line is an object files link instruction
```

In the above example, `prog.exe` is the **target**, and its last modification date is compared to those dependencies, `prog.o` and `test.o`. The **production rule**, preceded by a `TAB`, is the link command **cc -o prog.exe prog.o test.o**.

It is possible to define rules without dependencies, as rules always executed; example:

```
# To delete files no longer needed ...
clean::
    rm -f *.o core *~
```

used the following way:

```
$> make clean
```

Macros:

- These are lines like: `string1 = string2`
- They can be defined anywhere in the `Makefile`
- Any occurrence of `$(string1)` will be replaced by `string2`
- The definition of a macro can refer to another macro
- Environment variables are used in the `Makefile` as macros

There are some predefined widely used macros; most important are:

- dependencies in the rule and production:
  - `$?` : lists the names of dependent files newer than the target;
  - `$@` contains the name of dependent file without its suffix, if any;
- in the production rules only:
  - `$*` : contains the name of the dependent file without its suffix, if any;
  - `$<` : contains the filename of the dependency list in process (source file);

An example:

```
LPR   = lpr -Ppegase
FILES = preface chap1 chap2 chap3 appendix
#
print : $(FILES)
  $(LPR) $?
  touch print
#
```

```
printall :
  $(LPR) $(FILES)
```

Some tips:

- Do not forget the `tab` (note the confusion with a `space`) before each production rule. Think about it when you get this message (usual) from **make**:

  `'Make: Must be a separator on rules line #. Stop'`

- It should not be any space after the backslash continuation at the end of a line;

  The following command:

  ```
  $> cat -t -e Makefile
  ```

  can display tabs and the end of each line in the configuration file `Makefile` (with `^I` instead of `tabs` and a `$` at the end of each line);

- In some systems, the last line of the makefile must contain a character "new line";
- Each command line is executed in its own shell:

  ```
  cd myTrash
  rm *
  ```

  is not the same as:

  ```
  cd myTrash; \
  rm *
  ```

  equivalent to:

  ```
  cd myTrash; rm *
  ```

Entry point make in the online manual.

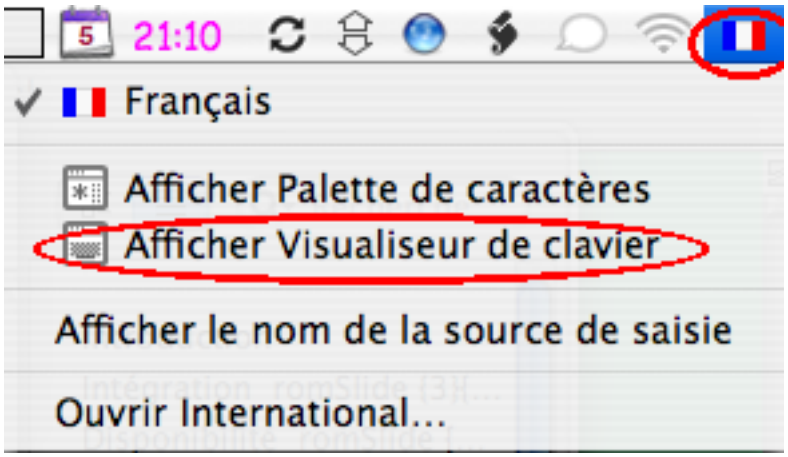
4.8.8 FAQ

MacOS X how to

... Where is that damn character? ("{" or "~" or "]" etc.) Most common hidden characters:

character	key combination
{	Atl-(
}	Alt-)
[	Alt-Shift-(
]	Alt-Shift-)
~	Alt-n-space
|	Alt-Shift-L
button 2	Control-click
button 3	Apple-click
\	Alt-Shift-/`

One can also load the *Visualiseur de clavier* from the *Saisie* menu in the menu bar at the top right:



Unix questions

... My executable is there, and yet I have the following error:

```
$> myExecutable.exe
zsh: command not found: myExecutable.exe
```

The simplest is to designate the executable by its relative path:

```
$> ./monExecutable.exe
```

where "." means the current directory.

C++/STL questions

... What is this story about `std::`? All the C++ standard library is defined in its own namespace, the `namespace std`. Thus, we must always use the prefix `std::` before all the elements that are drawn.

... How to convert a `std::string` into an integer? Using `istringstream`:

```
#include <sstream>

int main()
{
  std::istringstream is( "1" );
  int nombre;
  is >> nombre;

  return 0;
}
```

... And convert an integer to a `std::string` ? Using `ostringstream`:

```
#include <sstream>

int main()
{
  std::ostringstream os;
  os << 1;
  std::string nbre_str = os.str();
```

```
  return 0;
}
```

... And convert a `std::string` into a `const char*` ? By using the method `c_str()` . For example:

```
#include <string>

int main ()
{
  std::string filename = "../data/formes.txt";
  std::ifstream f
  f.open( filename.c_str() );
  ...
  return 0;
}
```

... I have more unanswered questions! Some of the answers given above are taken from the FAQ C++ - Club d'entraide des développeurs francophones (licensed under the GNU FDL)

The 4 basic commands for CMT

... cmt create <package> <version> creates and configures a new package

... cmt config creates `setup` and `cleanup` files

... cmt show uses shows all packages used by `CMT`

... source setup.sh/.csh in `cmt` directory updates the `CMT` environment variables

For more informations, see **cmt help**

The 5 basic commands for Subversion

... svn checkout URL[@REV] [PATH] / svn co URL[@REV] [PATH] recopies localy in the directory `PATH` a repository located at `URL`.

... svn status / svn st compares the content of the current directory with regard to the repository and displays the differences.

... svn update [PATH] / svn up [PATH] brings up the modifications from the repository `PATH` into the local directory.

> **Warning:** Every single **svn update** must follow an **svn status** to check the state of our directory with regard to the repository.

... svn add [PATH] adds the file(s) or directory `PATH` to the repository.

> **Warning:** Every single **svn add** must follow an **svn commit** so the repository is up-to-date.

... svn commit [PATH] -m"[msg]" / svn ci [PATH] -m"[msg]" sends over to the repository all the local modifications applied to the directory (or file) `PATH`.

> **Warning:** Every single **svn commit** must follow an **svn status** to check the state of our local copy with regard to the repository.

For more informations, see **svn help**

- Work in progress ...
- Slides
  - `Intro-en`

S

T

V

W