Programmazione Avanzata per il Calcolo Scientifico
Advanced Programming for Scientific Computing
Lecture title: Organization of a C++ program

Luca Formaggia

MOX
Dipartimento di Matematica
Politecnico di Milano

A.A. 2014/2015

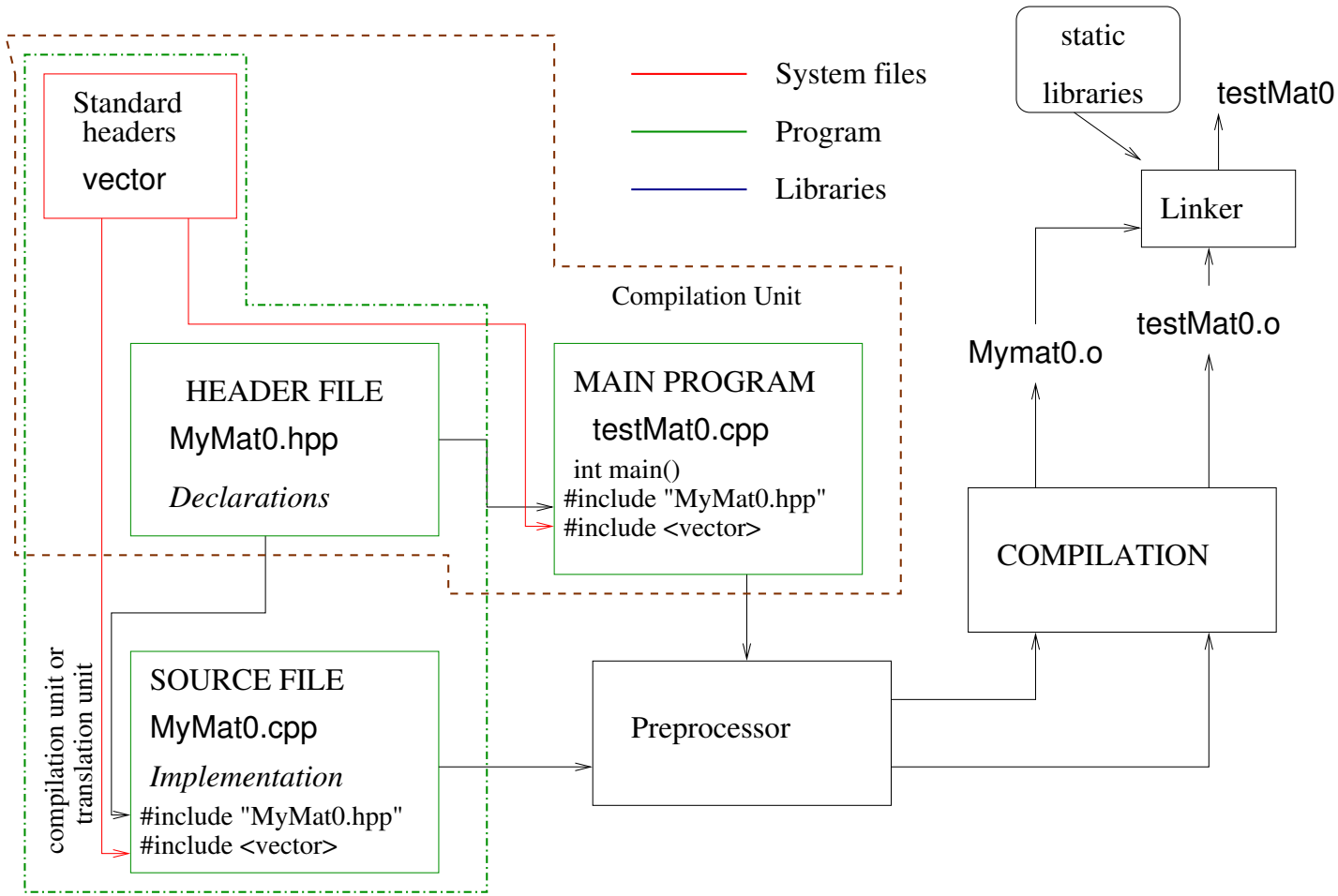
Organization of a C++ program

- Header e source files
- What should a header file contain
- The preprocessor
- main cpp directives
- Header guard

Software organization

A typical C++ program is organised in header and source files.
Each source file typically encompass related functionalities: class and functions
The header files are used to specify the information that is needed to make use of the code: either to compile it or, if the software is in the form of a library, to develop programs that use the library.
A library can be static of dynamic (shared) (we will have a special lecture on them). For the moment let's think the library as a collection of compiled object that can be used (linked) by an external program.
Header files are normally stored in special directories with name `include`.

A possible layout



What should a header file contain

Named namespaces	namespace LinearAlgebra
Type declarations	class Matrix{...}
Extern variables declatations	extern double a;
Constant variables	const double pi=4*atan(1.0)
Constant expressions	constexpr double h=2.1
Constexpr functions	constexpr double fun(double x)
Enumerations	enum bctype {...}
Function declarations	double norm(...);
Forward declarations	class Matrix;
Includes	#include<iostream>
Preprocessor directives	#ifdef AAA
Template declarations	template <class T> class A;
Template definitions	template <class T> class A{...};
Definitions of inline functions/methods	inline fun(){...}

What a header file should not contain

Function definitions	`double norm(){ ..}`
Definition of non-const variables	`double bb=0.5;`
Definition of class static members	`double A::b;`
Array definitions	`int aa[3]={1,2,3};`
Unnamed namespaces	`namespace { ...}`

Which type of information a header file provides?

A header file contains all the information needed by the compiler to verify type consistency, calculate the dimension of an object to be stored in the stack memory, instantiate templates, define static objects (i.e. objects that are defined at compilation stage, and not at linking stage).

Forward declarations

If a type name is used only to specify the type of the argument of functions, or the type of a static variable, then a forward declaration is enough

```
class Triangle;// Forward declaration
class Point; ...
int fun (Triangle & a);
class MM{
static Point P;
...}
```

However:

```
class Triangle;
...
Triangle b; //NO!
```

The translation unit

A C++ translation unit is a source file, whose extension is `.cpp`, but also `.cc` is normally used for C++ files.
It contains definition of function, classes, struct part of your program. Normally you want to express with a translation unit a set of related functionalities, often just a single class, that is

The compilation Process

Compiling an executable is in fact a multistage process

- preprocessor The source code is transformed into an intermediate source by processing CPP directives;
- compilation Each translation unit is converted into an object file. Optimization is done at this stage;
- linker Translation units are assembled and unresolved symbols resolved by linking appropriate libraries;
- loader Possible dynamic (shared) library are used to complete the linking process. The program is then loaded in memory for execution.

The preprocessor

To understand the mechanism of the header files correctly it is necessary to introduce the C preprocessor (cpp). It is launched at the beginning of the compilation process and it modifies the source file producing a second source file (which is normally not shown) for the actual compilation.

The operations carried out by the preprocessor are guided by directives characterized by the symbol # in the first column. The operations are rather general, and the C preprocessor may be used non only for C or C++ programs but also for other languages like FORTRAN!.

Synopsis of cpp

Very rarely one calls the preprocessor explicitly, yet it may be useful to have a look at what it produces
To do that one may either use
`cpp [-DVAR1] [-DVAR2=xx] [-Iincdir] file >pfile`
or use the option `-E` of the compiler:
`g++ -E [-DVAR1] [-DVAR2=xx] [-Iincdir] file >pfile`

Note: `-DXX` and `-I<dirname>` compiler options are in fact cpp options.

Main cpp directives

`#include<filename>`

Includes the content of `filename`. The file is searched fist in the directories possibly indicated with the option `-Idir`, then in the system directories (like `/usr/include`).

`#include "filename"`

Like before, but first the current directory is searched for `filename` then those indicated with the option `-Idir`, then the system directories.

`#define VAR`

Defines the *macro variable* `VAR`. For instance `#define DEBUG`. You can test if a variable is defined by `#ifdef VAR` (see later). The preprocessor option `-DVAR` is equivalent to put `#define VAR` at the beginning of the file. Yet it overrides the corresponding directive, if present.

`#define VAR=nn`

It assigns value `nn` to the (*macro variable*) `VAR`. `nn` is interpreted as an alphanumeric string. Example: `#define VAR=10`. Not only the test `#ifdef VAR` is positive, but also any occurrence of `VAR` in the following text is replaced by `nn`. The corresponding cpp option is `-DVAR=10`.

```
#ifdef VAR
code block
#endif
```

If VAR is undefined code block is ignored. Otherwise, it is output to the preprocessed source.

```
#ifndef VAR
code block
#endif
```

If VAR is defined code block is ignored. Otherwise, it is output to the preprocessed source.
Note: A macro VAR can be defined globally for the translation unit under consideration using the preprocessor option -DVAR or -DVAR=VALUE .

Special macros

The compiler set special macros depending on the options used, the programming language etc. Some of the macros are compiler dependent, other are rather standard:

- `__cplusplus` Is set to a value if we are compiling with a c++ compiler. In particular, it is set to 201103L if we are compiling with a C++11 compliant compiler.
- NDEBUG is a macros that the user may set it with the `-DNDEBUG` option. It is used when compiling "production" code to signal that one DOES NOT intend to debug the program. It may change the behavior of some utilities, for instance assert() is deactivated if NDEBUG is set. Also some tests in the standard library algorithms are deactivated. Therefore, you have a more efficient program.

EXAMPLES

Some examples on the way the preprocessor works are in Preprocessor.

The header guard

To avoid multiple inclusion of a header file the most common technique is to use the header guard, which consists in checking if a macro is defined and, if not, defining it!

```
#ifndef HH_MYMATO__HH
#define HH_MYMATO__HH
... Here the actual content
#endif
```

The variable after the `ifndef` (`HH_MYMATO__HH` in the example) is chosen by the programmer. It should be a long name, so that it is very unlikely that the same name is used in another header file! Some IDEs generate it for you!

Testing for the C++ standard you are using

In Utilities/cxxversion.hpp an example of the use of cpp macro to test the C++ standard one is using to compile a program.
The file Utilities/test_cppversion.cpp show an example of its use.

Internal and external linkage

While the scope of a name is the part of a translation unit where it is visible, the linkage refers to its accessibility.
external linkage means the name (functions of variable) is accessible throughout your program and internal linkage means that it's only accessible in the translation unit which defines it.
In C++ variables in the global scope and functions have by default external linkage. Variables in a namespace or local to a translation unit have by default internal linkage.

extern and static

Linkage may be changed from default by using the `extern` (for external linkage) or the `static` or unnamed namespace (for internal linkage).
Maybe some examples explain things better than words.
Anyway, remember that in almost all cases the default behavior is what you want, so you do not have to worry too much about linkage!

See the example in Linkage/main_linkage.cpp.

You may want to change the default behavior in two cases

- Creating global variables for values that must be shared by many parts of the program. For instance, global parameters or objects that define utilities to be shared. Then you declare them with external linkage. Remember that they must be defined somewhere!
- Hide a function that is used only locally in a translation unit. In that case you use internal linkage via the `static` keyword or using unnamed namespace. Note that that the definition/declaration is only in the source (.cpp) file.

In general, try to avoid using globals, unless really necessary.