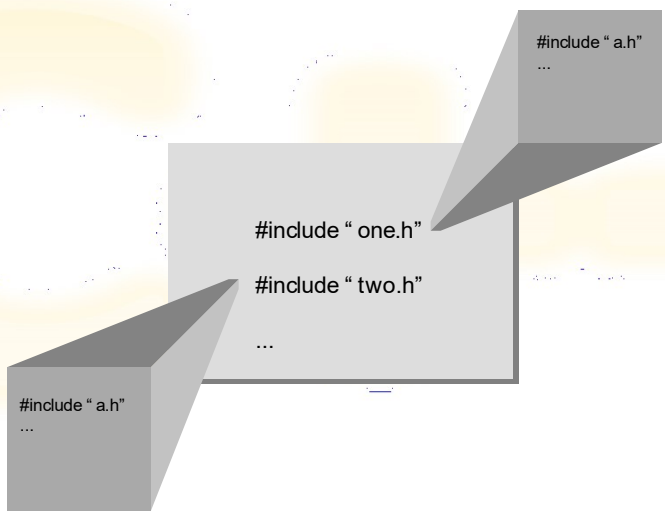
Architectural Issues

Contents

- Objectives
  - To be able to use namespaces
  - To be able to avoid the use of the preprocesor
- Contents
  - Header & Source Files
  - Include Guards
  - #defined ‘constants’
  - Preprocessor Macros

Header & Source Files



3

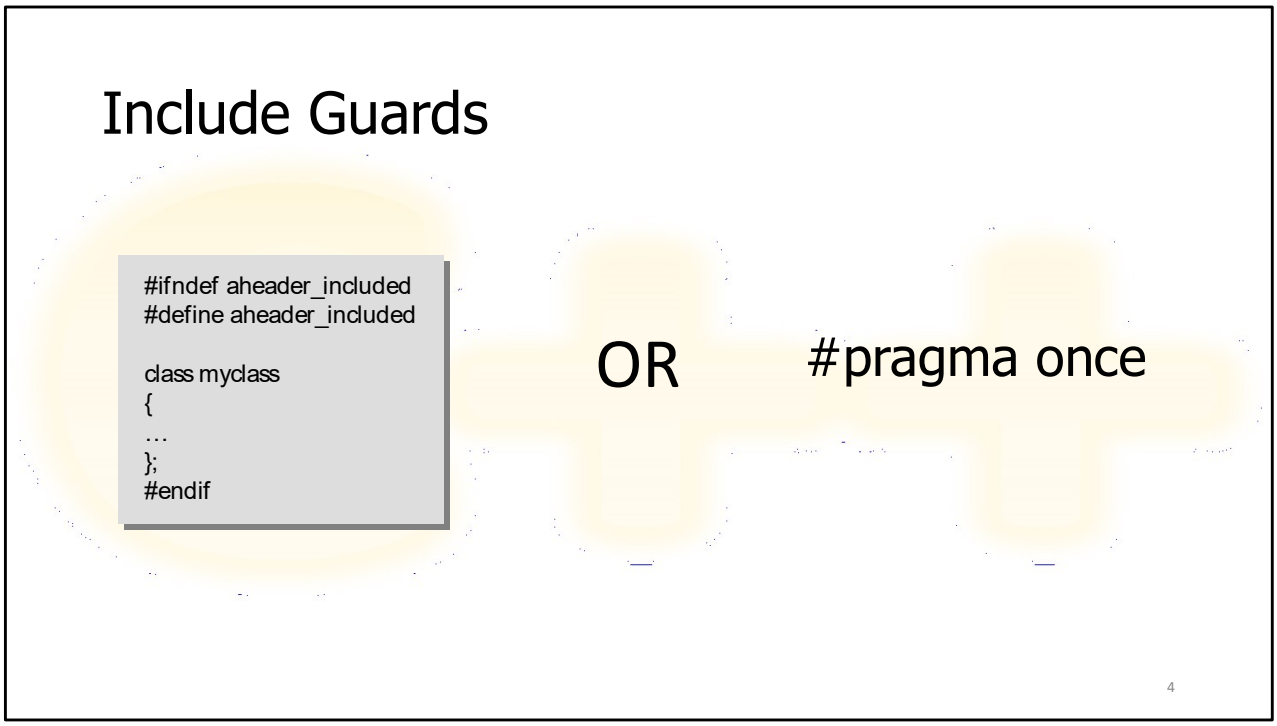
Header and source files have already been discussed and used extensively throughout this module. However there are potential problems when using header files.

The two header files one.h and two.h both include the file a.h. You are likely to end up with compiler errors if this problem occurs. Worse, you may not even be aware of what files are included by other header files.

There are a couple of possible solutions.

Let the programmer sort it out themselves! Not very satisfactory!!

Make header files safe for multiple inclusion – recommended! This is done by using include guards.



There is a feature called conditional compilation. This is implemented by using co

the first time the file was included the aheader_included item would not exist so the code would be included to define aheader_included and declare the class myclass. Subsequent inclusions would detect that the aheader_included symbol was defined and the result of the #ifndef would be false. In this case everything up to the #endif would be skipped.mpiler directives.

To some extent the above method of coding include guards has been superseded by a newer pre-processor directive. #pragma is a relatively new directive that was introduced to allow for platform or compiler specific directives. Since error & warning messages are compiler dependent they have been placed in the #pragma directive.

#pragma has many options which may vary from compiler to compiler and platform to platform.

#defined 'constants'

There is often a need for particular constant values that are used repeatedly throughout a program. If the value changes for some reason, then it is necessary to change all instances of that constant - a process which is fraught with possibilities for introducing errors. To overcome this, C provides a preprocessor directive that allows constants to be given a symbolic name with the #define directive.

#define CONST_NAME constant_value

5

Wherever the preprocessor finds the symbol CONST_NAME, it will substitute the constant_value.

HOWEVER, #defined constants are NOT constant. The definition of a constant is something that cannot change. In C / C++ there is nothing to prevent you from doing the following:

#define PIE 3.142
#define PIE 666

It is FAR better to use the C++ const statement than #defined symbols!

Exercise