# CSE 143

## Modules:

## Specification, Implementation, and C/C++ Source Files

[Chapter 1]

## Modules

- Large software systems need to be broken into modules if there is any hope of managing their complexity.
- Module examples:
  - Table of bank accounts (including procedures to examine and modify)
  - Spelling checker part of word processor
  - Graphical User Interface (GUI)

## General Design Goals

- Subdivide large software into smaller units
- Group related operations and data together
- Isolate implementation details in one place
- Restrict interaction between module and clients to small, well-defined interfaces

We will revisit design issues later; for now we will focus on how to build modules in C++

## Specification vs. Implementation

Two parts of each module

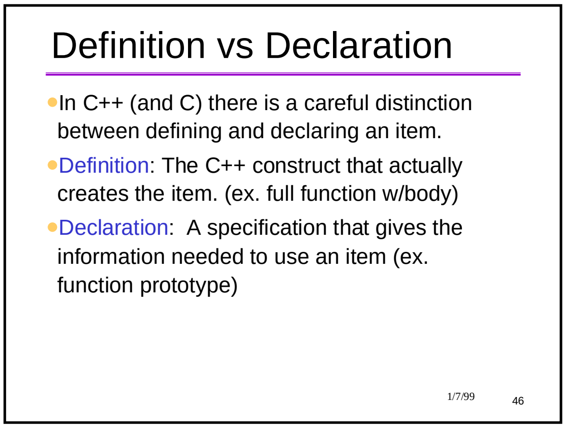
- Specification (*what*)
  - Also known as “interface”
  - Describes the services that the module provides to clients (users)
  - **Publicly** visible
- Implementation (*how*)
  - Parts of the module that actually do work
  - **Private**, hidden behind module interface

## Modules in C++

- Modules represented by a pair of files
  - *specification* (`.h`) file
  - *implementation* (`.cpp`, `.cc`, `.c++`, `.C`, etc) file
- Client’s only interaction with module is through the interface defined in the `.h` file

## Imports and Exports

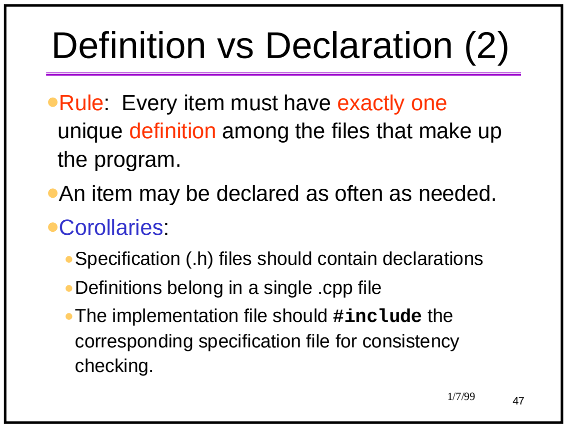
- Specification (.h) file declares which items are *exported*
  - constants, function prototypes, and data types
- Client program must *import* features of a module to use them
  - Use the `#include` directive



## Definition vs Declaration

- In C++ (and C) there is a careful distinction between defining and declaring an item.
- Definition: The C++ construct that actually creates the item. (ex. full function w/body)
- Declaration: A specification that gives the information needed to use an item (ex. function prototype)

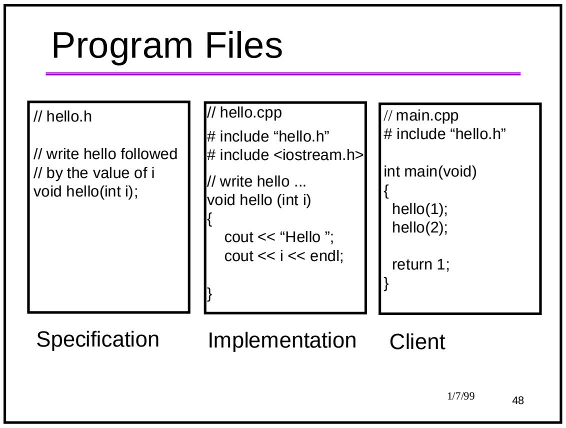
1/7/99 46



## Definition vs Declaration (2)

- Rule: Every item must have exactly one unique definition among the files that make up the program.
- An item may be declared as often as needed.
- Corollaries:
  - Specification (.h) files should contain declarations
  - Definitions belong in a single .cpp file
  - The implementation file should **`#include`** the corresponding specification file for consistency checking.

1/7/99 47



## Program Files

```
// hello.h

// write hello followed
// by the value of i
void hello(int i);
```

Specification

```
// hello.cpp
# include “hello.h”
# include <iostream.h>
// write hello ...
void hello (int i)
{
    cout << “Hello ”;
    cout << i << endl;

}
```

Implementation

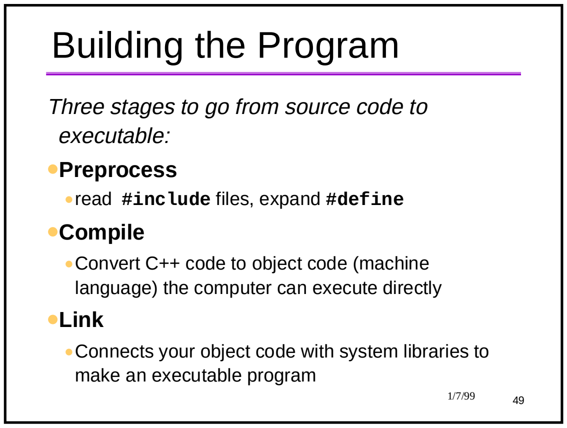
```
// main.cpp
# include “hello.h”

int main(void)
{
  hello(1);
  hello(2);

  return 1;
}
```

Client

1/7/99 48

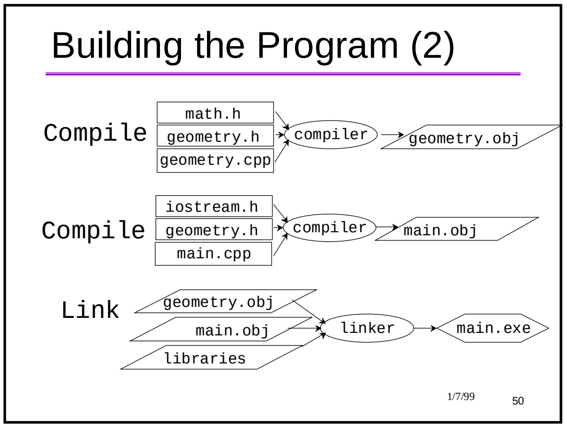


## Building the Program

*Three stages to go from source code to executable:*

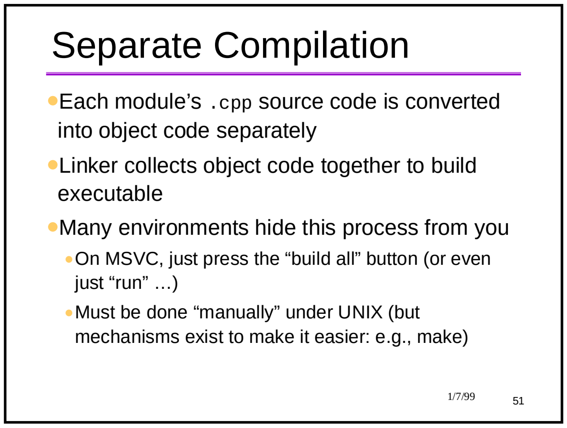
- **Preprocess**
  - read **`#include`** files, expand **`#define`**
- **Compile**
  - Convert C++ code to object code (machine language) the computer can execute directly
- **Link**
  - Connects your object code with system libraries to make an executable program

1/7/99 49



## Building the Program (2)

1/7/99 50



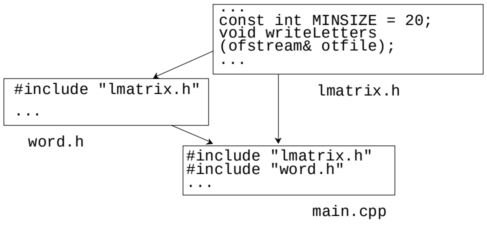
## Separate Compilation

- Each module’s `.cpp` source code is converted into object code separately
- Linker collects object code together to build executable
- Many environments hide this process from you
  - On MSVC, just press the “build all” button (or even just “run” …)
  - Must be done “manually” under UNIX (but mechanisms exist to make it easier: e.g., make)

1/7/99 51

## Multiple Inclusion Problem

Compile-time error if identifiers (function names, constants, etc.) are defined multiple times:



```
...
const int MINSIZE = 20;
void writeLetters
(ofstream& otfile);
...
```

lmatrix.h

```
#include "lmatrix.h"
...
```

word.h

```
#include "lmatrix.h"
#include "word.h"
...
```

main.cpp

## Multiple Inclusion Hack

- To avoid this problem, use preprocessor directives:

Preprocessor directive

```
// lmatrix.h
#ifndef _LMATRIX_H_
#define _LMATRIX_H_
...
const int MINSIZE = 20;
void writeLetters (ofstream& otfile);
...
#endif
```

- Read the above as:

  If `_LMATRIX_H_` undefined, compile the code through `#endif`

## Function Scope

- Normally, a function defined in a .cpp implementaiton file is visible to (can be called from) all other parts of the program.
  - Appropriate for functions that are part of the module’s interface.
  - Not good for functions only used in the module as part of the implementation (i.e., a function whose existence should be a private mater, not visible to clients).

## `static` Functions

- A function definition may be preceded by the keyword **`static`**. Such functions are said to have *file scope* and are not visible outside the .cpp file containing the definition. Use for functions that are not part of the module’s interface. Example:

```
// yield the value 17
static int xvi( ) {
   return 17;
}
```