CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Compilation & Testing of Object Systems

adelaide.edu.au

seek LIGHT

Review

- You had a review on important concepts of OOP
- Most important topics you had so far:
  - Pointers
  - Class Hierarchies/Inheritance
  - Polymorphism

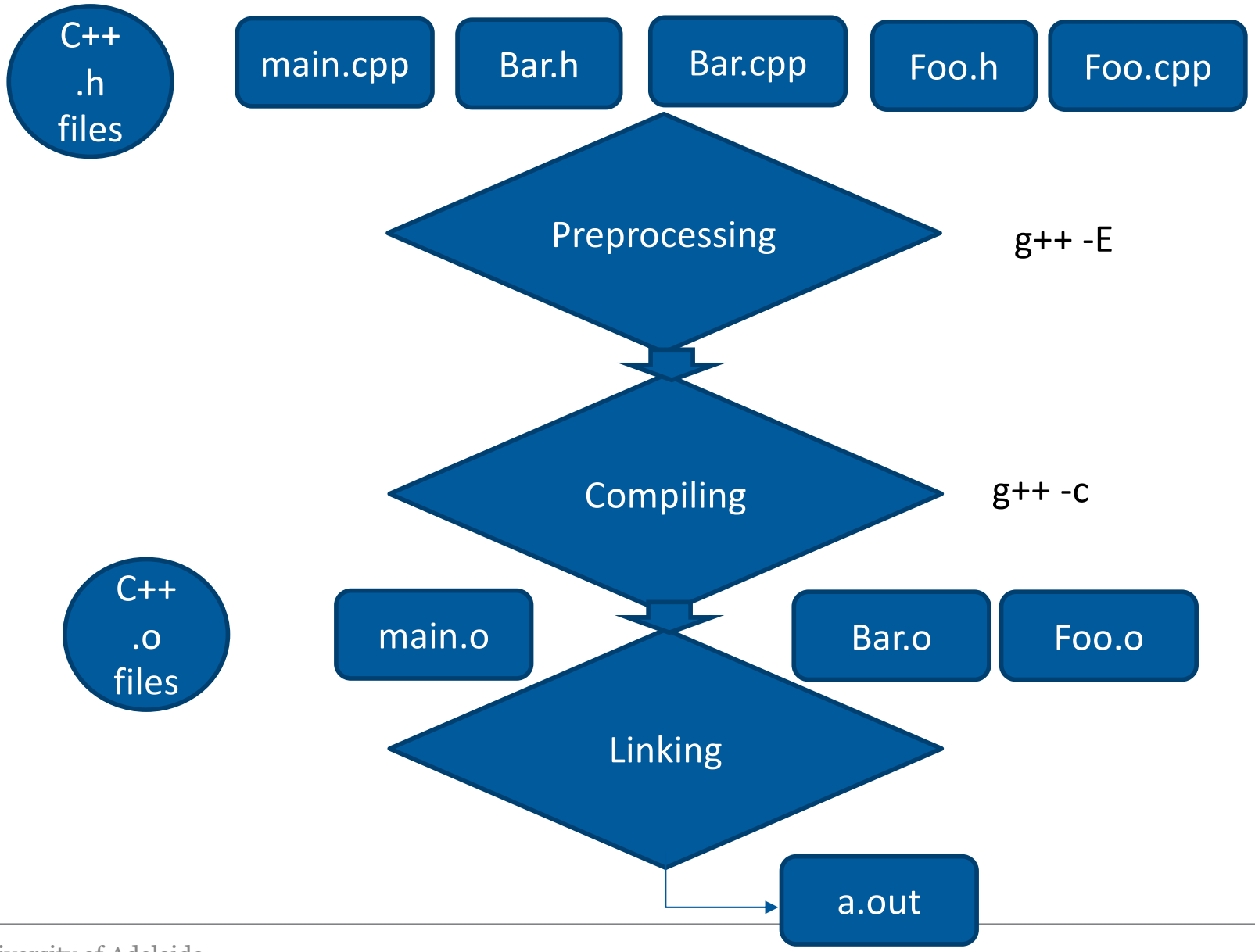
Overview

- In this lecture we will discuss:
  - How the compilation process of a multi-object program occurs and how this impacts what code is included in each file
  - Testing object based programs and building a test suite

Separate Compilation

- You can place all the classes and main into one file. So why separate them?
    - Abstraction and Encapsulation – provide libraries for other people
        - More important if you are developing an ADT
    - Easier to split the job in a large project
    - Easier for version control
    - Easier debugging
    - Change only one file and just compile that one
- We divide a program into separate parts.
- These are compiled separately and linked together when you need to build the final running application.
    - How?

Separate Compilation

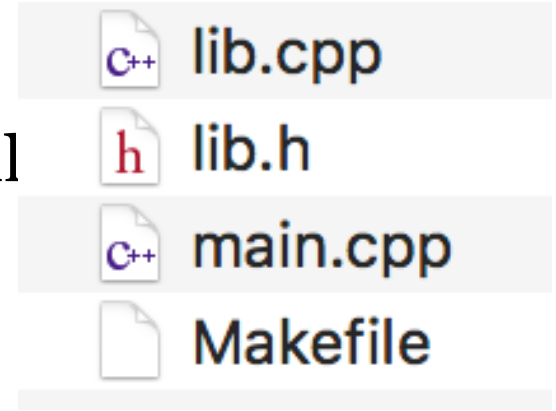


Separate File Compilation

- We need to compile more than one file.
- You should already know that a piece of code without a `main()` method won't compile. This is because C++ doesn't know where to start the execution.
- We can, however, compile any file to its **object form**, by using the -c flag.
- Technically, a C++ program is a set of code components, compiled through to objects, and then linked together with the implementing libraries and a `main()` method.
- Thus, we can compile all of our individual files separately and then link them together, with a driver, to make a program.

What does a compiler do?

- g++ -c main.cpp lib.cpp
  - Makes two object files
- What if a cpp files includes lib.h as well as something else that includes lib.h
  - `#include "lib.h"`
  - Copy the whole header there
  - Should be declared once;
    If you are including that in more than one file use `ifndef`
- g++ command also **links** the object files and makes an exe




```
#ifndef __LIB_H__
#define __LIB_H__

//The declaration of your class/library

#endif
```

- Is it allowed to have one declaration repeated in two object files?
  - How about the definition of that?

Interface and implementation

- All the users need to know in order to make use of our classes are just the interfaces
- **interface file :** Exposed behaviors, with their usage comments
  - Usually has a name <something>.h
  - Contains private members of the class as well, to keep the entire class declaration in one place.
  - We have to do this because C++ won't allow the class declaration across two files.
- The implementation is stored in the **implementation file**.
  - Usually has the name <something>.cpp

The Interface File

- We've been using interfaces the whole time.
- Required for compile!
  - The `include` directive tells the **pre-processor** to use a particular interface file. This is then used to check all of your code that depends on public member variables and functions.
  - Otherwise, the compiler can't check to see if the function you are calling or the variable you are using are really **declared**.
  - The linker links the declarations with their implementation. If they are not **defined**, the linker gives an error. (the linker starts working after the compiler, but when we say compile, we usually mean compile and link. The errors that the linker gives are also considered compile time error)

Where do I use it?

- The implementation of the class must include the interface file - to allow checking.
- **Wherever** you use the class, you must also include the interface file.
- If you are not using the class, you should **NOT** include the interface file.
- The **application file** or **driver file**, the one that contains the `main()` method, often includes many of these as it creates most of the objects.

Two ways to include s header file

- We use a well-established short-hand to tell the C++ compiler where to find the interface file.
- Using “ and “ means that the header file may be found in this directory (usually):
  `#include “myInterface.h”`
- Using < and > means that the header file is stored wherever the pre-defined header files are kept:
  `#include <iostream>`

Build automation software – can we make the build process easier to manage?

makefiles

- Format

```
label : dependencies
        action
```

- Label is used to specify what you want to do.
  - *all* usually means build everything
  - *clean* usually means remove any executables and object files
- The dependencies indicate what files you need
  - If these don't exist, make will look for a label to make them

```
all: main.o lib.o
        g++ main.o lib.o -o out

main.o: main.cpp lib.h
        g++ -c main.cpp -o main.o

lib.o: lib.h lib.cpp
        g++ -c lib.cpp -o lib.o
```

File: `makefile`

Demo

- Let's build a makefile for a program

Building a program

- Technically, a C++ program is a set of code components, compiled through to objects, and then linked together with the implementing libraries and a `main()` method.
- Thus, we can compile all of our individual files separately and then link them together, with a driver, to make a program.

Testing

- Things you should already know:
  - Tests should include boundary cases (empty strings, full arrays, max value, etc)
  - Tests should include at least one instance of each possible equivalence class that should have ***different*** behavior (positive numbers, negative numbers, characters, integers, even numbers, odd numbers, etc)
- It's tiresome to keep typing all the tests. We can use the makefile to make testing easier.

```
int main (void)
{
   // unit tests for Lib
   Lib myLib;

   // test: even values
   int expected = 27;
   test1=myLib.function1(2);
   if (test1 == expected)
      cout << “even: PASSED”;
   else {
       cout << “even: FAILED ”;
       cout << expected “<< expected
       cout << ” returned “ << test1;
   }
   // test: 0
   int expected = -1;
   ...
```

test.cpp

```
all: main.o lib.o
        g++ main.o lib.o -o out

main.o: main.cpp lib.h
        g++ -c main.cpp -o main.o

lib.o: lib.h lib.cpp
        g++ -c lib.cpp -o lib.o

test: test.o lib.o
        g++ test.o lib.o –o test

test.o: test.cpp lib.h
        g++ -c test.cpp –o test.o
```

```
>> make test
>> test
```

Software Life Cycle

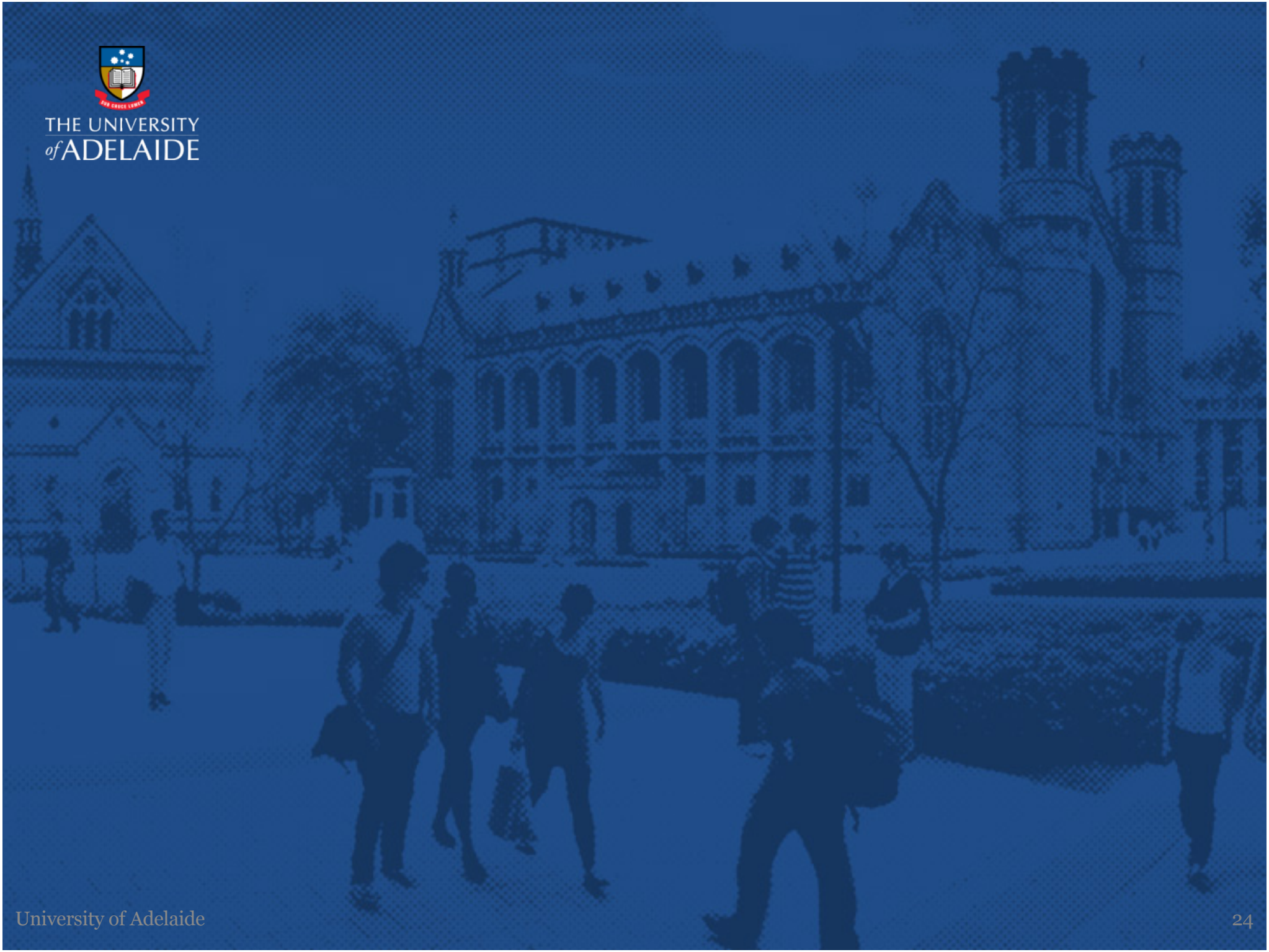
- The way that we construct our hierarchies strongly affects the efficiency and expandability of our final design.
- A good design is extensible, adaptable and reusable.

Steps for software development:

- Identify the problem
- Analyse and specify the task you want to achieve
- Design the software (classes, structures and algorithms)
- Code it
- Test it
- Maintain and develop it
- Wait for it to become obsolete

Summary

- Building programs from separate files:
  - Makes design more modular
  - Makes providing libraries easier
  - Makes testing easier
  - Makes debugging easier
  - Makes version control easier
  - Makes group activities easier.
- Makefiles can help us manage the build process
- Test Drivers can help us manage testing


THE UNIVERSITY
of ADELAIDE