CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

Separate Compilation & Introduction to Computational Complexity

adelaide.edu.au

seekLIGHT

Review

- You had a review on important concepts of OOP + recursion
- Most important topics you had so far:
  - Pointers
  - Polymorphism
  - Tail recursion

Overview

- In this lecture we will discuss:
  - Making use of separate compilation
  - Software Life Cycle
  - Start the topic of computational complexity

Separate Compilation

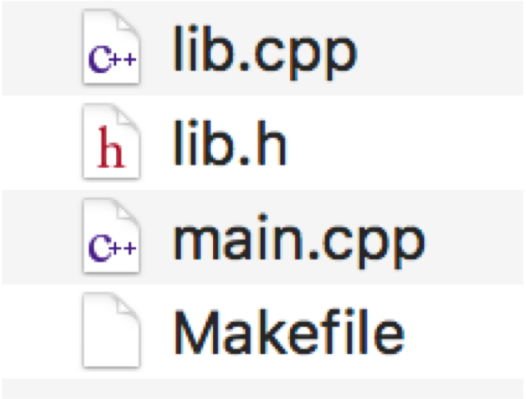
- You can place each class into its own files. A header and a cpp file.
  - Why is this good?
    - Abstraction and Encapsulation – provide libraries for other people
      - More important if you are developing an ADT
    - Easier to split the job in a large project
    - Easier for version control
    - Easier debugging
    - Change only one file and just compile that one
- So, we can divide a program into separate parts.
- These are compiled separately and linked together when you need to build the final running application.
    - How?

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
  - One object file
- Both these cpp files includes lib.h (why?)
  - `#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
- Is it allowed to have one declaration repeated in two object files?
  - How about the definition of that?




```
#ifndef __LIB_H__
#define __LIB_H__

//The declaration of your
class/library

#endif
```

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.
- 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>`

Makefile

Include a header

- Makefile:
  - Command: make or make labelName
  - What if only the main changes
  - What if only the Implementation of lib changes?

```
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
```

- Both main.cpp and lib.cpp include the header file
- Both object files will include that declaration
- What if the header has the definition as well?
- Linker gets confused -> error
  - This is why you should not include definitions there!
- It works if you only have one object file but that means that you are not using the header file, as an interface to be used at different places.

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.

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

Designing for OOP

- Whenever we design a solution using this programming model we have to:
  - identify the objects and their variables
  - identify the behaviors
  - work out how to hide things correctly
  - work out how to re-use code effectively
  - use polymorphism to our benefit

- But there are many ways to put these systems together.

Design structures

- Design phase includes designing structures

- The way that we put our structures together will have a very strong impact on:
  - Ease of addition
  - Ease of deletion
  - Ease of insertion
  - Ease of searching/sorting

Example – Library System

- How can we implement this?
- The most common solution is to keep an array of books.
- What are the problems with arrays, from the previous criteria?
  - Add a new book in the middle
    - Shift all the data after that
    - If no free space, move the whole array to a new position
  - Delete
    - Shift all the data after that
  - A linked list (you will learn about it more in near future) is better for these operations than an array
  - We can still do everything in both structural organisations but the efficiency of operations are dictated by how well the structure fits our purpose.

Search in a sorted array

- Given a sorted array ***arr*** of size ”n”, and one search item ***itm***, I suggest this for finding the item:
- For(i=1 to n)
  - If arr[i]=itm return i

- Does my pseudo code return the correct index?
- What do you think about my work?
- Have you ever looked up a word in a dictionary?

- Correctness goes first
- Then think about efficiency – This week we start talking about complexity

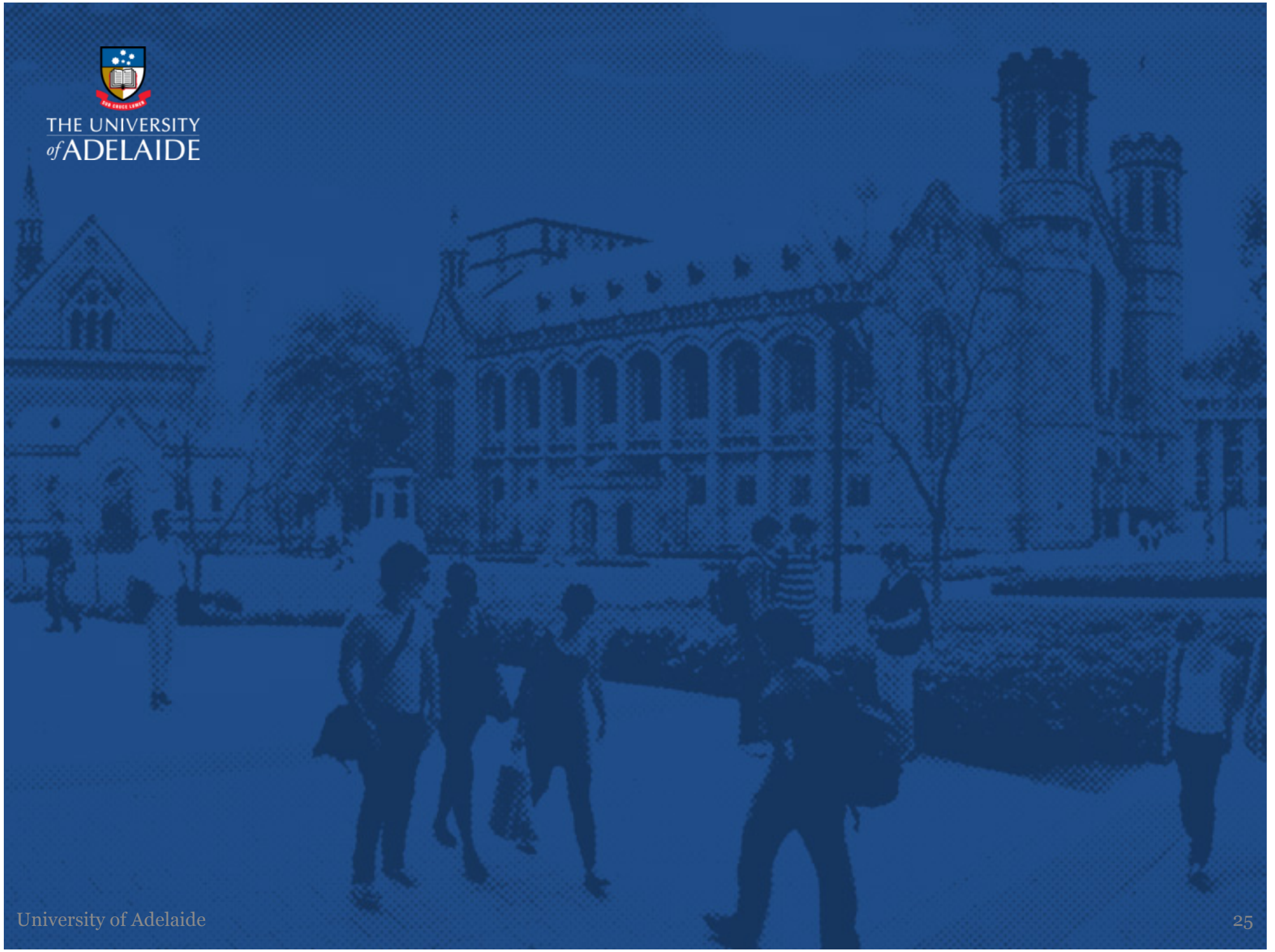
How many comparisons, at most!

- If I'm lucky, the first thing I'm comparing might be the answer
- But who cares if I'm lucky! We should think of worst cases
- For(i=1 to n)
  - If arr[i]=itm return i

- begin=1, end= n
- While begin<= end
  - middle= begin+(end-begin)/2
  - If arr[middle]=itm
    - Return middle
  - If arr[middle]> item
    - end= middle-1
  - Else begin= middle+1

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.
- Complexity
  - The efficiency of your code is related to the structures AND algorithms that you use.
    - As we'll see, these are heavily bound together.


THE UNIVERSITY
of ADELAIDE