CRICOS PROVIDER 00123M

School of Computer Science

COMP SCI 1103/2103 Algorithm Design & Data Structure

More about pointers and memory management

adelaide.edu.au

seek LIGHT

Previously on ADDS

- Pointers

```
int *ptr = new int;
*ptr = 6;
ptr = new int;
```

- Stack and Heap
- Global, Automatic and Dynamic variables

Using heap

- *new* without *delete* will continue to use more and more memory.
- Losing track of memory is referred to as a memory leak - eventually, something's going to happen.
- In order to prevent memory leak, delete the memory of a pointer whenever you want to set it to another part of the memory by new.
- But deleting (returning memory to the freestore) doesn't solve all of our problems. What do I mean?

Overview

- Segmentation fault
- Dynamic Array
- Multi-Dimensional Array
- Pointer to functions
- ADT

Segmentation Faults

- Your program will crash if you try to:
  - Use memory that isn't allocated to you
    - Use a deleted variable
    - Use uninitialized objects;
    - Go outside arrays
  - Try to write to memory that's read only
  - Use up all the memory
- In some cases, this crash will also leave a copy of the computer's memory state – referred to as a core dump

Debugging Segmentation Fault

- Try to avoid this by taking care of the memory management by following a good programming style.
- Compile your code with –g option and use gdb to get stacktrace of the segmentation fault.
- “cout” what you expect for Debugging

Dynamic Array

- It is illegal/meaningless to change the pointer value in an array variable.

```
int a[10];
int b[20];
int *ptr;
ptr = b;

a = ptr; // Illegal
```

- For ordinary arrays you must specify the size of the array when you write the program.

- A dynamic array is an array whose size is not specified when you write the program, but is determined while the program is running.

Dynamic Arrays

- Dynamic arrays are created using the *new* operator.

```
double *dArray = new double[array_size];
```

- Dynamic arrays are used like ordinary arrays.

- Remember to call *delete[]* when your program is finished with the dynamic arrays.
- *delete dArray;*
  - Undefined behaviour

Multi-Dimensional Array

- Two-Dimensional Array

```
dataType arrayName[rowSize][columnSize];
```

- An object of array type contains a contiguously allocated non-empty set of N subobjects of type T.
- How does it look like in the memory?
- How can you create a two-dimensional array in the heap?

  int * dArray2= new int[size1][size2];
- How does it look like in the memory?

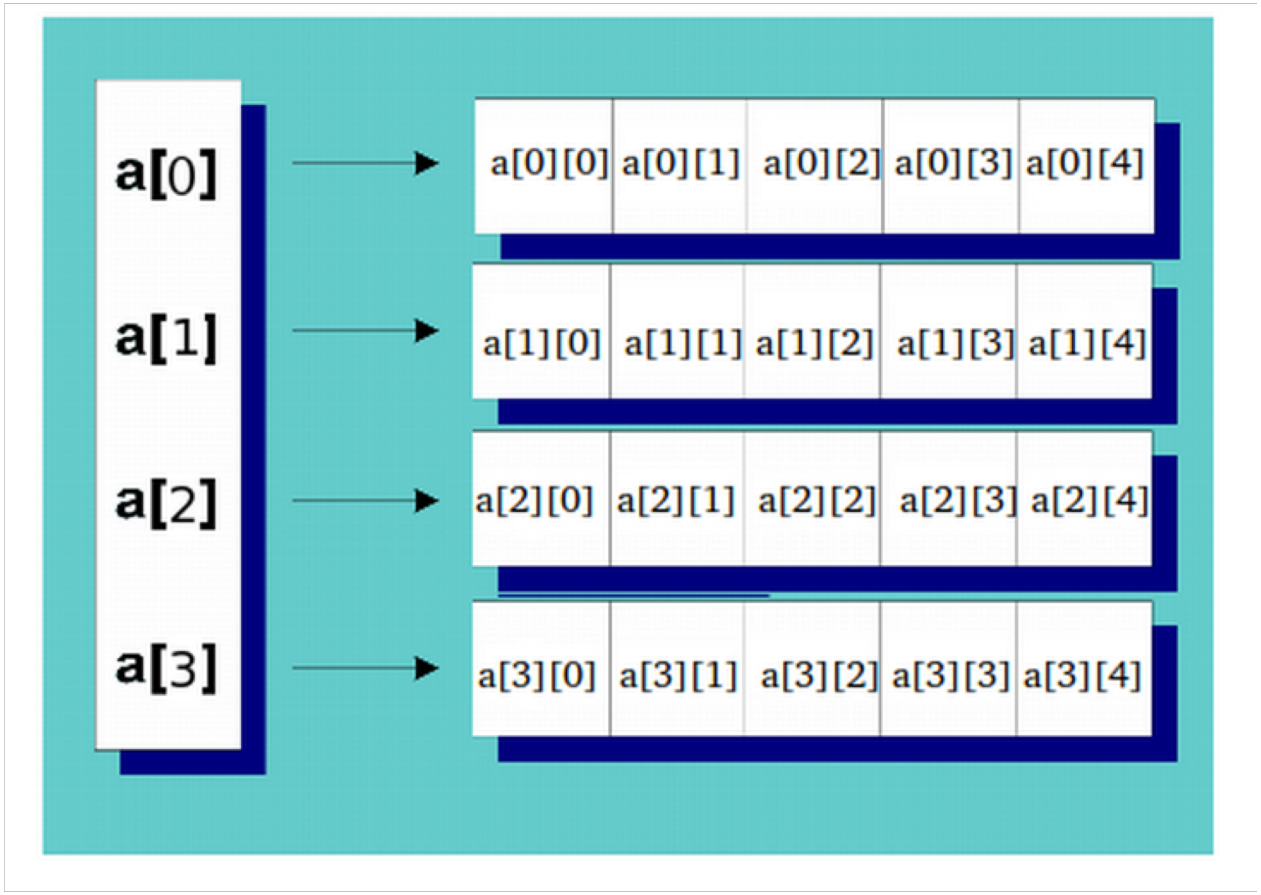
Multi-Dimensional Array

- Define int a[5][4];
- a[i], for i=0,1,2,3,4, is an array of size 4.

- If the address of a[0] is 0xXXXX then what is the address of a[1]?
- How can you represent it by means of “a” (which works like a pointer, with pointer arithmetic)?
- Haw can you represent the address of a[i][j] using a?
  a+i*NoOfComulns+j. Access the element by *(a+i*NoOfComulns+j)

- Question: With this approach, a large block in the memory is required. Can you suggest anything else?

Multi-Dimensional Array



Multi-Dimensional Array

- Pointer to Pointer (Multiple Indirection)
  - Where is a stored? How about pointers to rows?
  - How about the rows?

```
1   #include <iostream>
2   using namespace std;
3
4   main() {
5       int row = 4, col = 5;
6       int **a;
7       a = new int*[row];
8       for (int i = 0; i < row; i++)
9           a[i] = new int[col];
10  }
11
```

How do you find the value in a[i][j], using the **dereference operator?**

((a+i)+j)

Multi-Dimensional Array

- In C++, you can create n-dimensional arrays for any integer n.

```
int array[15][3][2];
```

Passing Arrays to Functions

- Pass-by-value
- Pass-by-reference

```
void modifyNumber(int number, int numbers[]){
  number 1001;
 numbers[0] = 5;
}

int main(){
  int x = 1;
  int y[10];
  y[0] = 1;

  modifyNumber(x, y);

  cout << "x is " << x << endl;
  cout << "y[0] is " << y[0] << endl;

  return 0;
}
```

x is 1
y[0] is 5

Returning Arrays From Functions

- It is not allowed in C++

- Get around by passing another array argument in the function.
- Return a pointer (Not a good idea to return the address of a local variable)

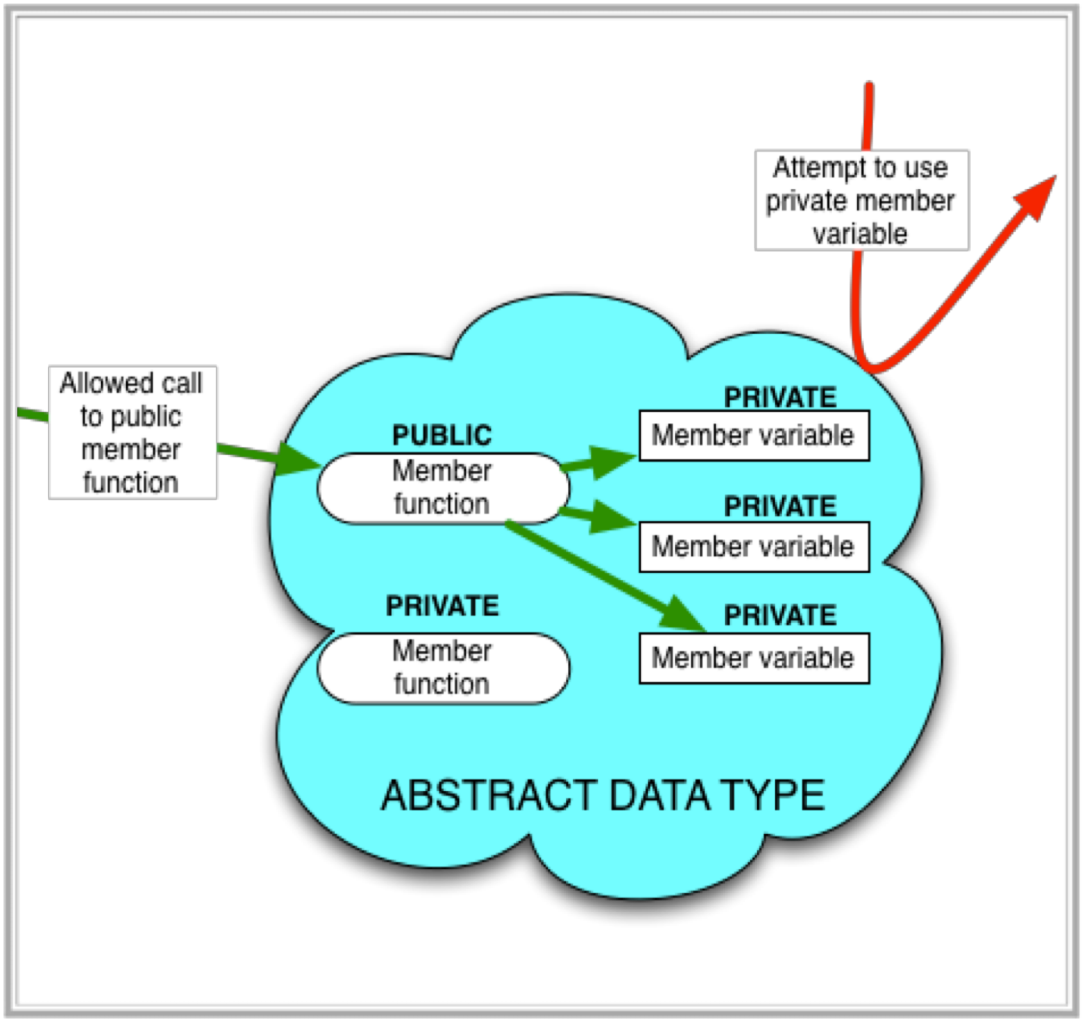
Pointer to Functions

```
1   // pointer to functions
2   #include <iostream>
3   using namespace std;
4
5   int addition (int a, int b) {
6       return (a+b);
7   }
8
9   int subtraction (int a, int b) {
10      return (a-b);
11  }
12
13  int operation (int x, int y, int (*functocall)(int,int)) {
14      return (*functocall)(x,y);
15  }
16
17  int main () {
18      int m,n;
19      int (*minus)(int,int) = subtraction;
20
21      m = operation (7, 5, addition);
22      n = operation (20, m, minus);
23      cout << n << endl;
24      return 0;
25  }
26
```

Summary

- Pointers allow you to access storage but it's extremely manual. Misjudging your pointer arithmetic will have strange results.
- C++ arrays and pointers are very easy to cause problems!
- Heap vs Stack
- There are global, automatic (ordinary) and dynamic variables in C++.
- You need to understand how these work to make the best use of them.
  - Space is finite - management is important.
  - Runtime problems cause segmentation faults and crash. Core dumps help you with finding the problem.
  - If you mismanage the stack, or the heap, your program will fail.
  - Using memory without deallocating it will compile but will generally crash at runtime.

Abstract Data Types



Data Type

- Types are more than just **values**, they also come with a valid set of **operations**.
- A data type is the values AND the set of operations defined over these values.

Abstract Data Types

- Suppose we have a type where the public member functions provide a large number of increasingly more complex operations.
- Now, think about that we can do something with the type, but have no idea how it is doing it - or change how it is being done.
- **The details have been abstracted away from us**.

A data type is called an ADT if the programmers who use the type have no access to the details of the implementation.

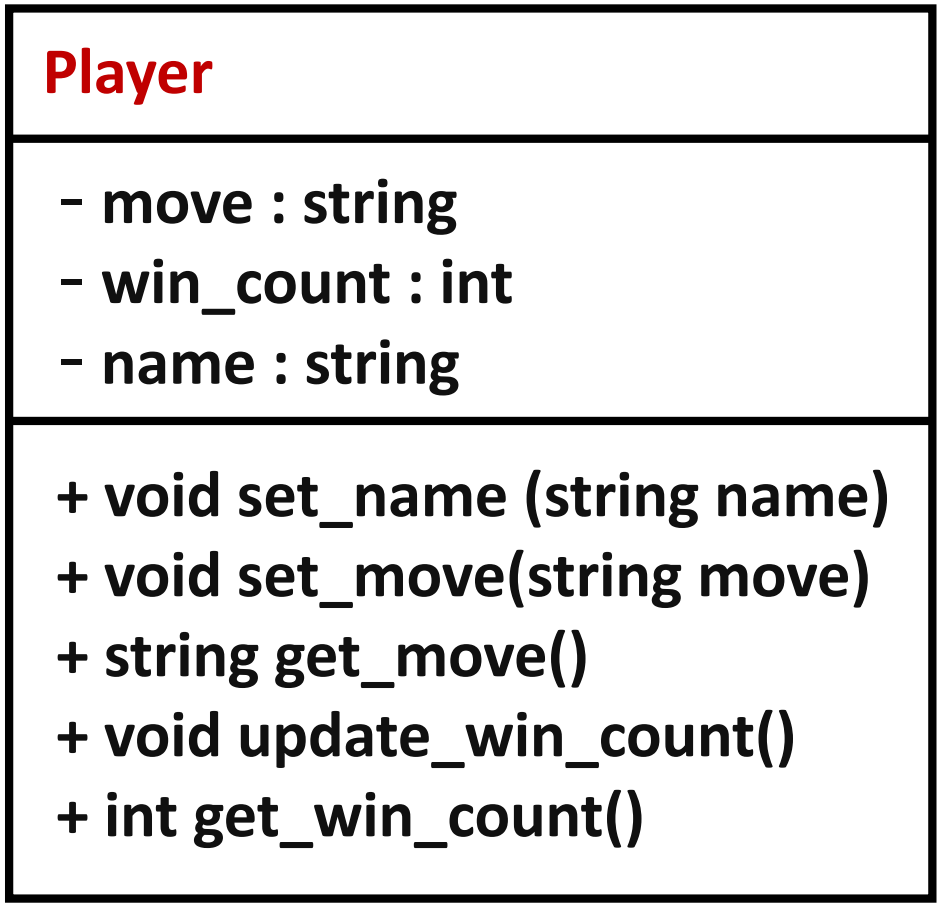
Not all classes are ADTs

- Programmer-defined types are not automatically ADTs.

- Unless defined and used with care, the programmer-defined types can make a program difficult to understand and modify.
- We need to control access to make sure that only part of the behaviour is available to others.
- How do we define the behaviour?

Example

- Recall the definition of class in C++
- How can we create a Player object?
- Do we need to know the implementation of the functions?



Separation

- We need to separate the specification of how the type is used by the users from the details of how the type is implemented.
- Class abstraction is the separation of class implementation from the use of a class

- Rules:
  - Make all member variables private
  - Make the basic operations public and specify how to use them
  - Make any helping functions private

Interfaces

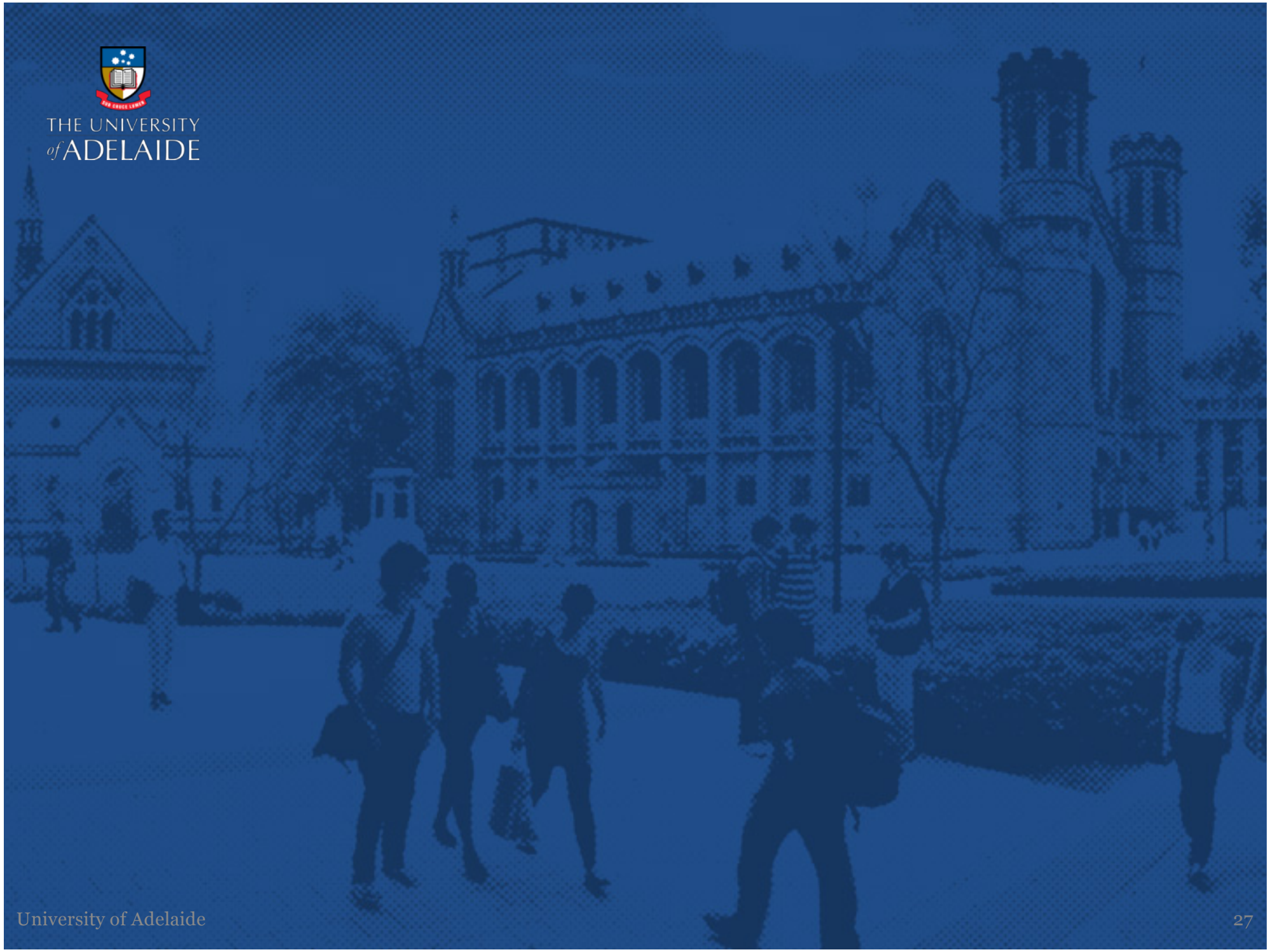
- The set of public member functions in our class, along with a description of what they do, make up the *interface* of the ADT.
- This should be all that someone needs to know to use your ADT.
- At the moment, we're writing the declaration and the implementation in the same file. But we won't always do that.

Implementation

- The implementation of the ADT tells how this interface is realized as C++ code, including:
  - definitions of public functions
  - any private or public variables
  - any private ‘helper’ functions

ADTs and Black Boxes

- From a design point of view, the implementation of an ADT is like a Black Box - you can't see inside it.
- **All a programmer can see is your interface**.
- A programmer shouldn't NEED to know about the implementation to make the ADT work.
  - Do you know how std::string or '+' are implemented?
- This is also known as ***information hiding***.


THE UNIVERSITY
of ADELAIDE