School of Computer Science

COMP SCI 2103/7103 Algorithm Design & Data Structure

More about pointers and memory management

Previously on ADDS

- Pointers

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

- Stack and Heap
  - Heap fragmentation
  - Memory leak

- Segmentation fault
- Global, Automatic and Dynamic variables

Overview

- Dynamic Array
- Multi-Dimensional Array
- Pointer to functions

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

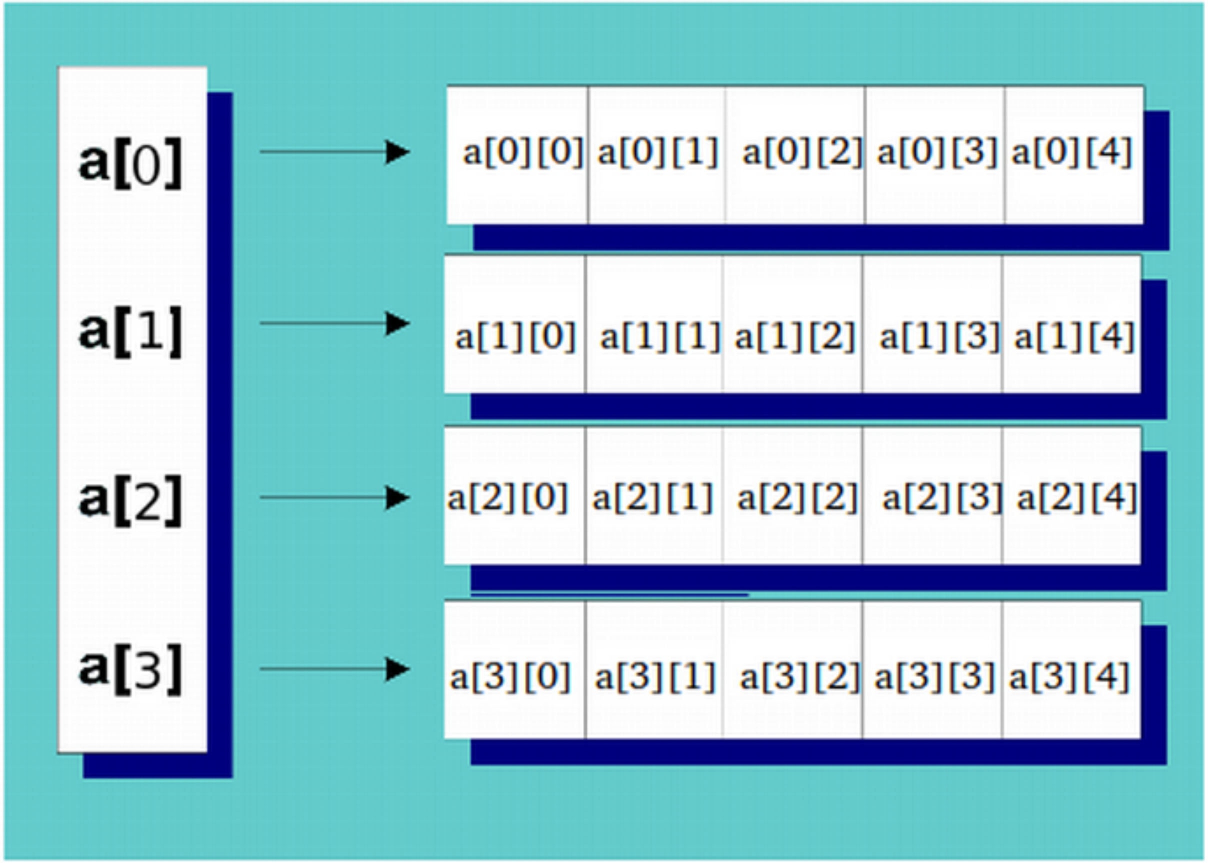
Multi-Dimensional Array

- 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.

Multi-Dimensional Array



Multi-Dimensional Array

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

```
#include <iostream>
using namespace std;

main() {
    int row = 4, col = 5;
    int **a;
    a = new int*[row];
    for (int i = 0; i < row; i++)
        a[i] = new int[col];
}
```

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

```
#include <iostream>
using namespace std;


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

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

   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

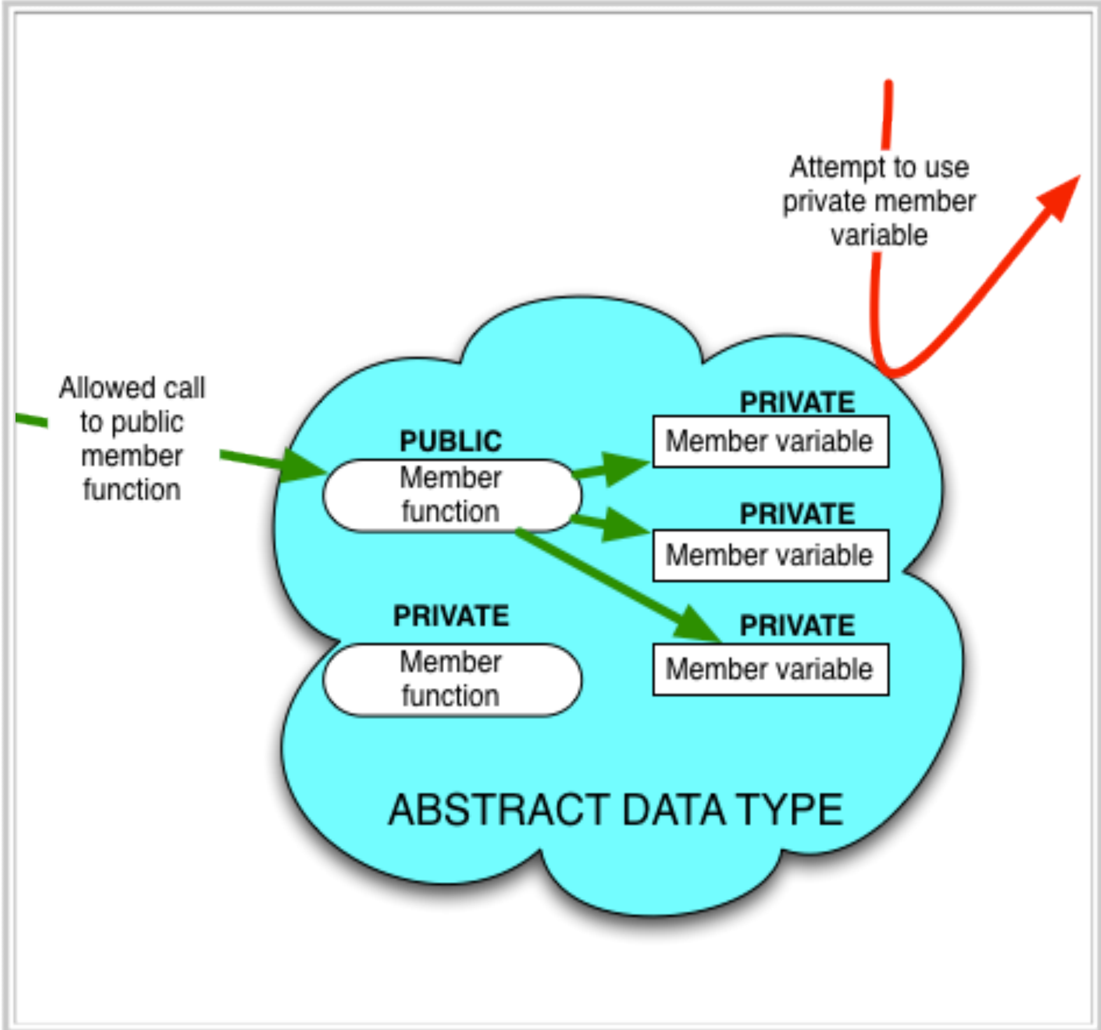
- It is not allowed in C++

- Two options
  - passing another array argument in the function.
  - return a pointer
    - Not a good idea to return the address of a local variable!!
    - But OK if we are creating a new array to return dynamically

Pointer to Functions

```
// pointer to functions
#include <iostream>
using namespace std;

int addition (int a, int b) {
    return (a+b);
}

int subtraction (int a, int b) {
    return (a-b);
}

int operation (int x, int y, int (*functocall)(int,int)) {
    return (*functocall)(x,y);
}

int main () {
    int m,n;
    int (*minus)(int,int) = subtraction;

    m = operation (7, 5, addition);
    n = operation (20, m, minus);
    cout << n << endl;
    return 0;
}
```

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?

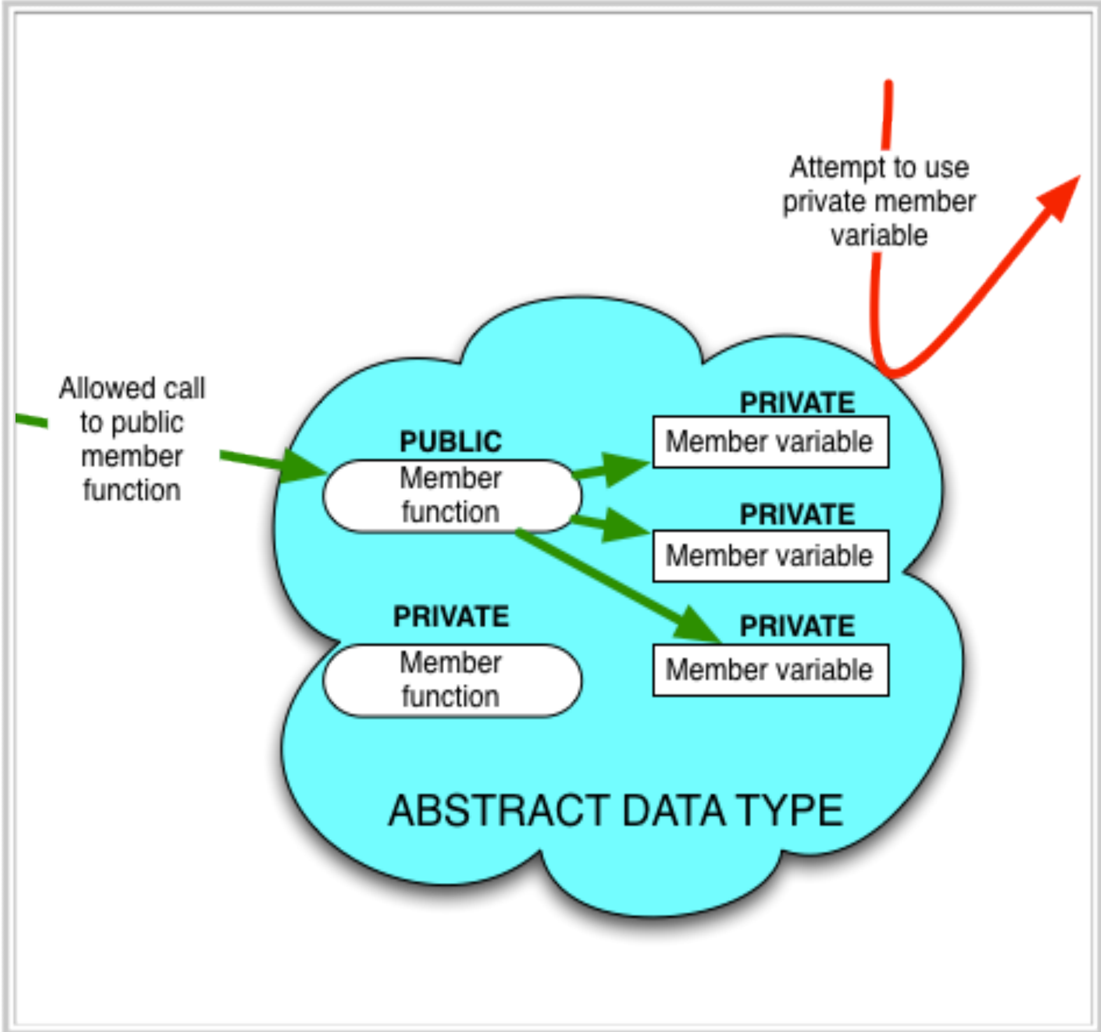
Player
- move : string - win_count : int - name : string
+ void set_name (string name) + void set_move(string move) + string get_move() + void update_win_count() + int get_win_count()

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

Abstract Data Types



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*.

More Complex Data Structures

- Another really good application of ADTs is in controlling access to more complex data structures.

- Vector? Vector provides bound checking on the at() function

  - What is the key difference between the `[]` of the array and the `at()` of the `Vector` class?

- How could you implement a `Vector`?

Example

- Define a circular queue
- We can do this in arrays, without any extra ADTs.
- What are the benefits and drawbacks of this approach?

The Circular Array

- Let' s design and build a circular array class together.
  - One view: as we keep inserting elements, we "wrap around" to the beginning of the array

- Technically, this is an example of modular arithmetic: no matter how big the number we' re dealing with is, we want the reference to fit into the range 0..(n-1)
  - How can we achieve this?

The Circular Array

- What's missing from a user's point of view?
- Does addElement have a problem?
- How can we fix it?
- Does the user of this ADT need to know about it?

```cpp
#include <iostream>
using namespace std;

class CircularArray{

private:
    int size;
    int* array;
    int arrayIndex;

public:
    CircularArray(int newSize);
    void addElement(int element);
    int getElementAt(int index);
    void printElements();
    ~CircularArray();
};

CircularArray::~CircularArray(){
    delete array;
}

CircularArray::CircularArray(int newSize){
    size = newSize;
    array = new int[size];
    arrayIndex = 0;

    for (int i = 0 ; i < size; i++){
        array[i] = 0;
    }
}

void CircularArray::addElement(int element){
    array[arrayIndex++] = element;
    if (arrayIndex >= size){
        arrayIndex = arrayIndex % size;
    }
}

int CircularArray::getElementAt(int index){
    return array[index % size];
}

void CircularArray::printElements(){
    for (int i = 0; i < size; i++){
        cout << array[i] << " ";
    }
    cout << "\n";
}
```

Test

- How can we test it?

- A few things you can think about:
  - Normal functionality
  - Boundary value
  - Special cases
  - Error handling

Summary

- In this lecture, we refined our ideas as to what an ADT is and how to make a class an ADT.
- This kind of design and implementation activity is crucial in developing good data structures to solve problems.