Murdoch
UNIVERSITY

Data Structures and Abstractions

Unit Testing & UML & Data Dictionary & Big O Complexity & Vector & Deque

Lecture 4

Object Orientation in C++

- In C++ we set up three files for each class:
  - The header file (.h) stores the class interface and data definition.
  - The implementation file (.cpp) stores the class implementation.
  - The (unit) test file (.cpp) tests every method for all parameter bounds.

- *Rules*:
  - Each class should represent only ONE thing. (cohesion)
  - Every class *must* be tested in isolation in a test file.
  - The testing *must* occur before the class is used by any other class.
  - For every method in the class, you need to test all possible cases. This forms the class's Test Plan.

The Light Class

- The previous lecture notes looked at a light class.
- This class stored information about a light: its colour, radius and whether it was switched on.
- We therefore start by coding a header file (light.h) with this information.
- Remember that we code and test *incrementally.*
- Therefore we start the class with the bare minimum:
  - Constructor (initialiser).
  - Destructor (the opposite of the constructor). //?? Needed [1]
  - Output operator to **test** that data is what we expect. // for debugging only
  - Attribute declarations.

```
// Light.h
// Class representing a light
// Some methods shown, other methods you write.
// Version
// 01 - Nicola Ritter
// modified smr
//------------------------------------------------------------

#ifndef LIGHT_H
#define LIGHT_H

//------------------------------------------------------------

#include <iostream>
#include <string> // OO string

using namespace std;

//------------------------------------------------------------
```

Ensures that this file is only included in the compilation once.

```
class Light
{ // only some methods shown. You write the other methods needed.
   // convert normal comments to doxygen style comments
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();

        // provides a default output method, but be careful about friends
        // friends can be terrible, as they can mess up your privates
        friend ostream& operator << (ostream &ostr, const Light &light);// [1]

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//----------------------------------------------------------------

#endif
```

Class Name – capital first letter

public keyword: what follows is the interface.

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        // friends are no good in most situations
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-------------------------------------------------------------------

#endif
```

private keyword: what follows is hidden from outside the class.

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        // can do without friends
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//------------------------------------------------------------------

#endif
```

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        // why have friends?
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//------------------------------------------------------------------

#endif
```

all data must *always* be private or protected if you want sub-classes

Constructor: code is automatically run when an object is declared. [1]

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        // keep away from friends
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-----------------------------------------------------------------

#endif
```

Inline code can be used if a method has exactly one line only. [1]

```
class Light
{
public:
        Light () {Clear ();} // [1] information hiding?
        ~Light () {};

        void Clear ();
        // friends can be overrated [2]
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//------------------------------------------------------------------

#endif
```

Destructor: code is automatically run when an object goes out of scope.

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        //Even on facebook, friends can be no good
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//----------------------------------------------------------------

#endif
```

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        //Sun Tzu: Hold your friends close but your enemies closer
        //What does it say about having a “friend” so close that it is
        // inside the class.
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-----------------------------------------------------------------

#endif
```

But nothing needs to be done to a Light object when it destructs, so the method is empty. [1]

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        //Just because a language provides friendablity, doesn't mean
        // friends can be used without proper justification.
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-------------------------------------------------------------------

#endif
```

Every class needs a method that clears, resets, initialises or empties the object.

The Clear() function is more than one line, so it is defined in the source file (follows).

```
•	class Light
•	{
•	public:
•		Light () {Clear ();}
•		~Light () {};

•		void Clear ();
•
•	friend ostream& operator << (ostream &ostr, const Light &light);

•	private:
•		// Any string giving a colour is acceptable
•		string m_colour;
•		// In centimetres
•		float  m_radius;
•		bool   m_on;
•	};

•	//---------------------------------------------------------------

•	#endif
```

```
•	class Light
•	{
•	public:
•		Light () {Clear ();}
•		~Light () {};

•		void Clear ();
•		// No friends – enough said
•		friend ostream& operator << (ostream &ostr, const Light &light); [1]

•	private:
•		// Any string giving a colour is acceptable
•		string m_colour;
•		// In centimetres
•		float  m_radius;
•		bool   m_on;
•	};

•	//-----------------------------------------------------------------

•	#endif
```

‘friend’ operators and methods are those that link two different classes, in this case ostream and Light

```
class Light
{
public:
      Light () {Clear ();}
      ~Light () {};

      void Clear ();
      friend ostream& operator << (ostream &ostr, const Light &light);

private:
      // Any string giving a colour is acceptable
      string m_colour;
      // In centimetres
      float  m_radius;
      bool   m_on;
};

//-----------------------------------------------------------------

#endif
```

The return value of this operator is a reference to an output stream.

The operator we are *overloading* is the standard output operator. [1]

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-----------------------------------------------------------------

#endif
```

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on; // [1] don't start with _ in user classes
};

//---------------------------------------------------------------

#endif
```

All attributes are prefaced with m_

for 'member'

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-----------------------------------------------------------------

#endif
```

Don’t miss out the semicolon [1]

```
class Light
{
public:
        Light () {Clear ();}
        ~Light () {};

        void Clear ();
        friend ostream& operator << (ostream &ostr, const Light &light);

private:
        // Any string giving a colour is acceptable
        string m_colour;
        // In centimetres
        float  m_radius;
        bool   m_on;
};

//-----------------------------------------------------------------

#endif
```

Matches the #ifndef on previous slide

Murdoch UNIVERSITY

END

Light Definition

- There was one method (**`Clear()`**), and one operator (**<<**) that were more than one line long and hence were not be defined in the header class.
- Such methods and operators are defined in a source file with the same base name as the header file.

- // Light.cpp – implementation is separated from the interface/specification .h file
- //---

- #include "Light.h“

- //---

- void Light::Clear ()
- {
-     m_colour = "white";
-     m_radius = 0;
-     m_on = false;
- }

- //---

The header file is included as a local rather than system header file. [1]

```
// Light.cpp

void Light::Clear ()
{
    m_colour = "white";
    m_radius = 0;
    m_on = false;
}

//------------------------------------------------------
```

Light:: indicates the scope of the method. In other words it states that this method belongs to the Light class.

```
// Light.cpp
//------------------------------------------------------------


void Light::Clear ()
{
    m_colour = "white";
    m_radius = 0;
    m_on = false;
}

//------------------------------------------------------------
```

Each attribute (member variable) must be initialised.

END

```
ostream &operator << (ostream &ostr, const Light &light)
{
        ostr << light.m_radius << " cm "
            << light.m_colour << " light is ";
        if (light.m_on)
        {
            ostr << "on";
        }
        else
        {
            ostr << "off";
        }

        return ostr;
}

//------------------------------------------------------------------
```

Add in code to output information about the data. [1]

Testing

- *"Somehow at the Moore School (UPenn) and afterwards, one had always assumed there would be no particular difficulty in getting programs right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs."* – Maurice Wilkes, Computer Science pioneer and inventor of microprogramming.
- A lot has changed since Wilkes' keen observation as the amount of time and resources spent on testing has increased considerably; there now IT job roles called "Testers" and tools for automated testing.
- **You should not incorporate any subroutine, method, class or software component into the main build for the application unless incoming software is fully tested. There is a heavy price to be paid for ignoring this advice.**

Murdoch UNIVERSITY

The Test File

- So far we have a header file and a source file which together completely define a class.
- However we do not have a program because we have no main() function. [1]
- Nor do we have a test plan and this is very bad!! Test plans should exist when you have understood the specifications and before coding. After and while coding, you may want to add to the existing test plan.

Initial Test Plan

Test	Description (including why the test is needed)	Actual Test Data	Expected Output	Passed
1	Check that constructor initialises the data and check that output operator works.	NA – default constructor	0 cm, white light is off	
2	Check that setRadius works	2.5 is sent as parameter	*…fill in what is expected*	

Complete the class methods and then we can write a program that uses the class and follows the test plan. Do the testplan in a table or in a spreadsheet.

Murdoch UNIVERSITY

```
// LightTest.cpp This is the test program. To compile it needs only the
// .h file. To link, it will need the .cpp [1]
//---------------------------------------------------------------

#include "Light.h"

//---------------------------------------------------------------

int main()
{
        Light light;

        cout << "Light Test Program" << endl << endl;
        cout << "Test One" << endl;
        cout << light; // [2]

        cout << endl;
        return 0;
}
```

Runs the code in the constructor, i.e. the Clear () function.

Runs the code in the friend output operator

The destructor runs when the block in which the light was declared, ends.

LightTest.cpp

- LightTest.cpp is the test program for the light class. [1]
- When it runs it will take the tester through all the tests in the test plan.
- For each test there must be output letting the tester know which test is being run.
- When it is run, the appropriate ticks are placed in the test plan's 'passed' column.
- It is refactored regularly as the test plan grows.
- It is run in its entirety every time anything is changed in the class.
- Which means that you need to print a new copy of the test plan each time you make a change.
- It is your proof that the class you have written is without bugs.
- It allows you to say with confidence "this class is finished".

Readings [1].

- Go through the following In Absolute C++. Pages 284-293 very carefully. Via My Unit Readings (log in) . If the link is not working, contact the Murdoch University Library.
- Textbook, Chapter on Classes and Data Abstraction
- My Unit Readings: Testing and debugging (Chpt. 8). View Online at library site. You would struggle to finish the data structure unit/module if you can't write test harness and do unit tests for the software that you write.
- Chapter on Pointers, Classes, Virtual Functions, Abstract Classes and Lists, section on Shallow versus Deep Copy and Pointers; section on Classes and Pointers: Some Peculiarities.
- Chapter on Overloading and Templates, all the sections till (and including) section on Function Overloading.
- Reference book, C++ Coding standards: 101 Rules, Guidelines, and Best Practices, Herb Sutter https://ebookcentral-proquest-com.libproxy.murdoch.edu.au/lib/murdoch/detail.action?docID=5135988

Object Oriented Terminology (Revision)

Note:

It is object *oriented* not object orientated!!!

- A *class*
  - is a description of a data type;
  - it includes (encapsulates) both data, and algorithms that operate on the data;
  - data should always be protected from being changed by anything other than one of the class's own algorithms;
  - the data members are also known as attributes or properties;
  - the algorithms are called methods rather than functions.
- An *object*
  - is a particular instance (example) of a class.

Object Oriented Terminology (Revision)

- *Polymorphism, overloading, overriding*
  - In C++, refers to the use of the same name for more than one function or method;
  - In C++ polymorphism is used to specify abstract behaviour which may have different specific implementations. The version to invoke is determined at run-time. The abstract behaviour and specific behaviours are related by an inheritance hierarchy. This is just one example, typical for C++.
  - In C++ a child class can replace (override) a base class's method with its own version but this is not sufficient for polymorphism to occur.
  - C++ overloading is where the method or function name is the same but parameters can be different and the method or function to call is determined at compile time.
  - In C++ both methods and operators can be overloaded. Some operators cannot be overloaded. Check the appendix of the textbook for more information.

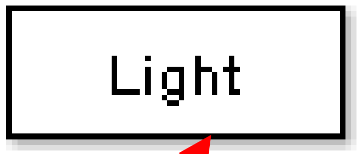
Example Class

- Consider a class simulating a light.

- It might have attributes of:
  - `int  colour`
  - `int  radius`
  - `bool switchedOn`

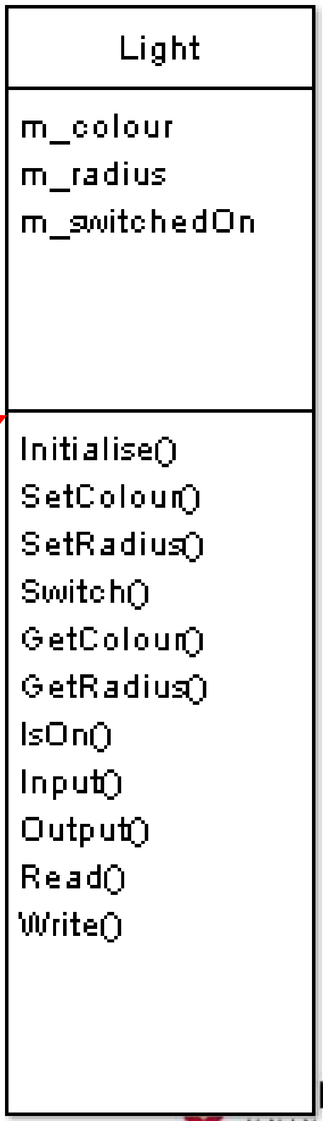
- There might then need to be methods that (not minimal) [1]:
  - initialised all objects of the light class;
  - set each of the attributes;
  - returned the current value of each of the attributes;
  - output the current value of each attribute to the screen;
  - allowed input of values from the keyboard;
  - saved the current values to file;
  - read the current values from file;
  - etc. .. and the “kitchen sink”

The Unified Modelling Language (UML)



High level UML Class Diagram: it shows class names and relationships only

Low level UML Class Diagram: it shows all the attributes and methods of the class



- While the low level diagram gives useful information, it results in *very* cluttered and hard to use diagrams. But if you are using a tool, then the low level diagram should be drawn and the more explicit descriptions can be given in a data dictionary. Don't forget to use -, + or #
- High level UML gives a good overview of your design. A data dictionary should also be provided.

END

The Data Dictionary [1] [2]

Name	**Type**	**Protectio n**	**Description**
Light			Simulates a light.
m_colour	integer	+	An integer light colour.
m_radius	float	-	The radius of the light.
m_switchedOn	boolean	+	True if the light is on.
Initialise()	procedur e	#	Sets the m_colour to white, m_radius to 0 and m_switchedOn to false.
SetColour(int colour)	boolean	+	If colour is positive it sets m_colour to colour and returns true. Otherwise it returns false.
SetRadius(float rad)	Boolean	Etc …	If radius is positive it sets m_radius to radius and returns true. Otherwise it returns false.
Switch()	procedur e		If the light is on, it switches it off. If the light is off it switches it on.
etc...			

Object Oriented Relationships (revision) [1]

- There are 4 object oriented relationships that we use in this unit. We have encountered them before.
  - Association
  - Composition
  - Inheritance
  - Aggregation

Murdoch UNIVERSITY

Association

- Refers to the properties of a class. [1]
  - Normally, a basic type is depicted as an **attribute** and something more substantial is depicted as an **association** in an UML diagram. The meaning is still the same.
- An association is a very low-level generic term meaning that two classes are “somehow related”.
- At the lowest level this relationship can also mean that one class “uses” another class somewhere in its algorithms. But see first point above.
- Depicted with a solid directed line.

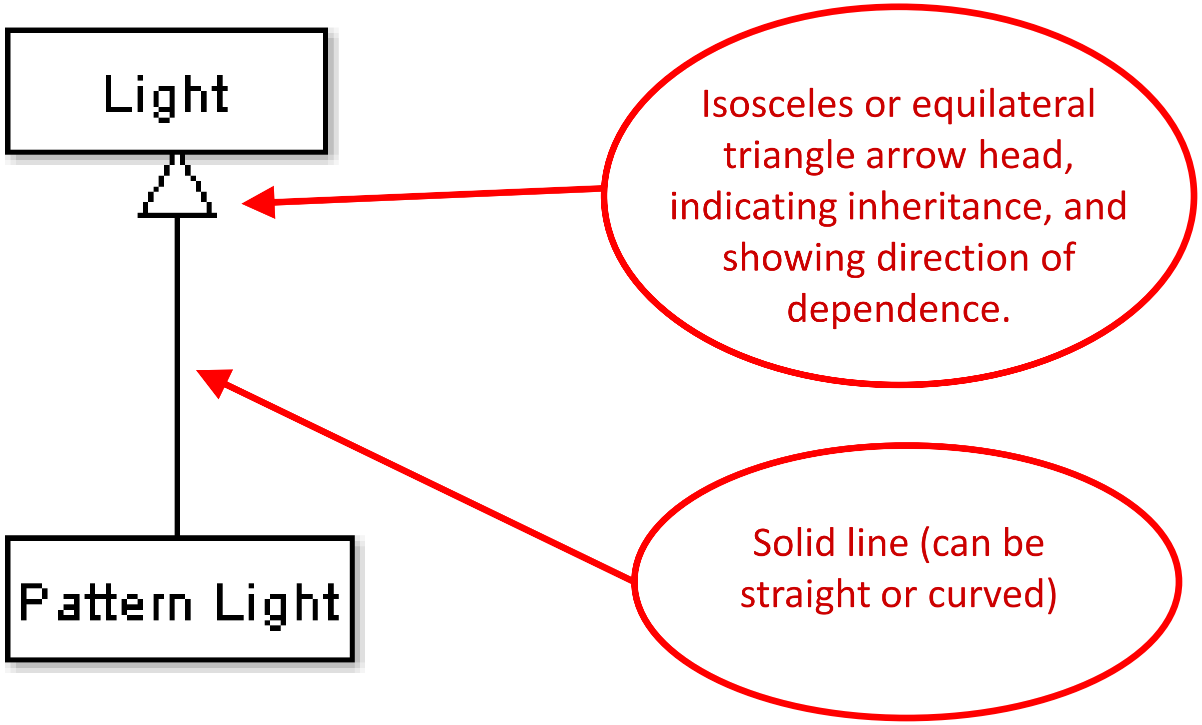
Inheritance

- Any class may *specialise* or *extend* another class but just because the language lets you do this does not mean you should be doing it without considering the abstraction that you are trying to model. [1]

- We can say that one class "*is a*" variety of another class.

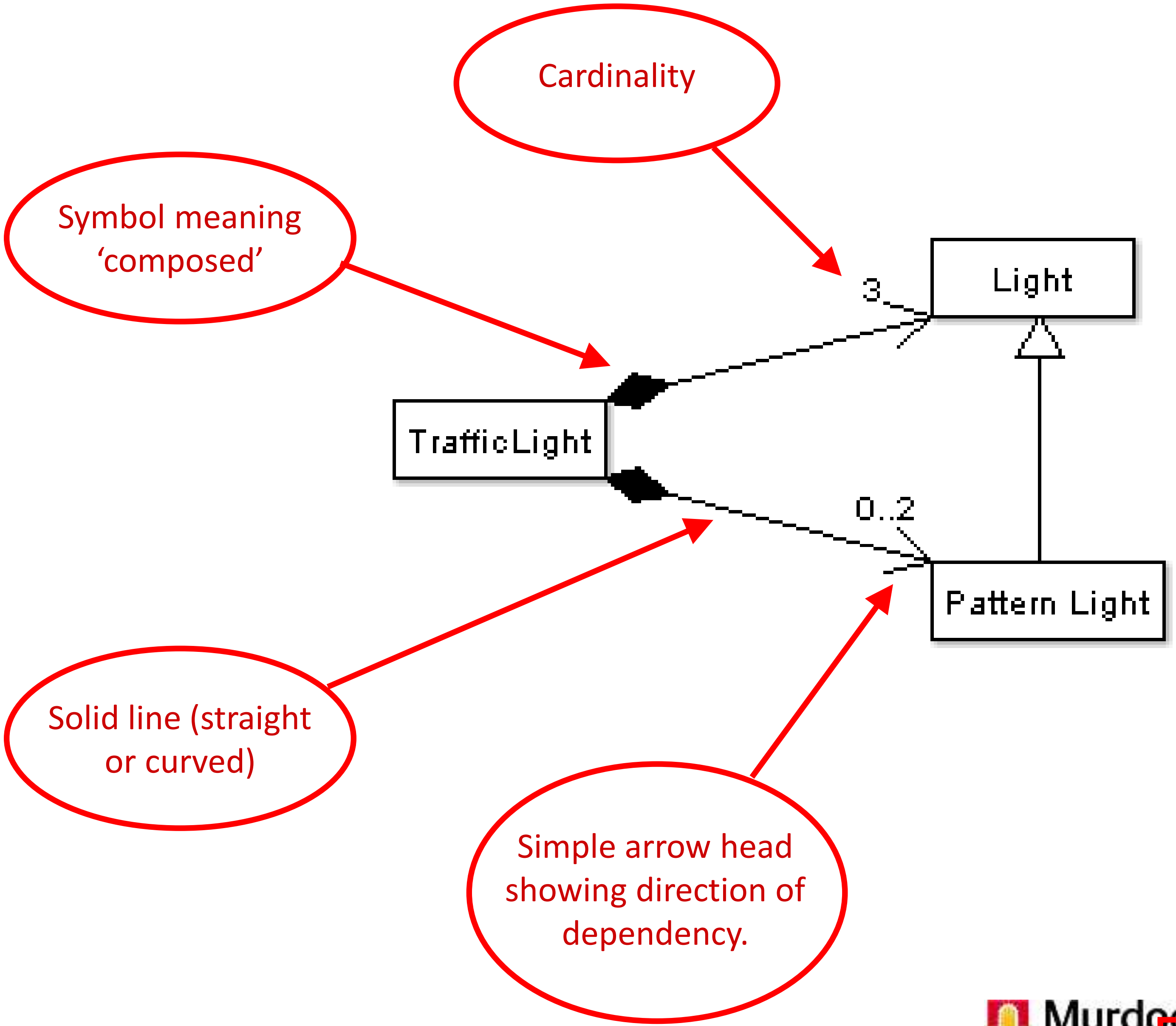
- We can *only* do this if it requires *all* of the data items *and* methods declared in the parent (base) class.

Inheritance


Light
Pattern Light
Isosceles or equilateral triangle arrow head, indicating inheritance, and showing direction of dependence.
Solid line (can be straight or curved)

Composition

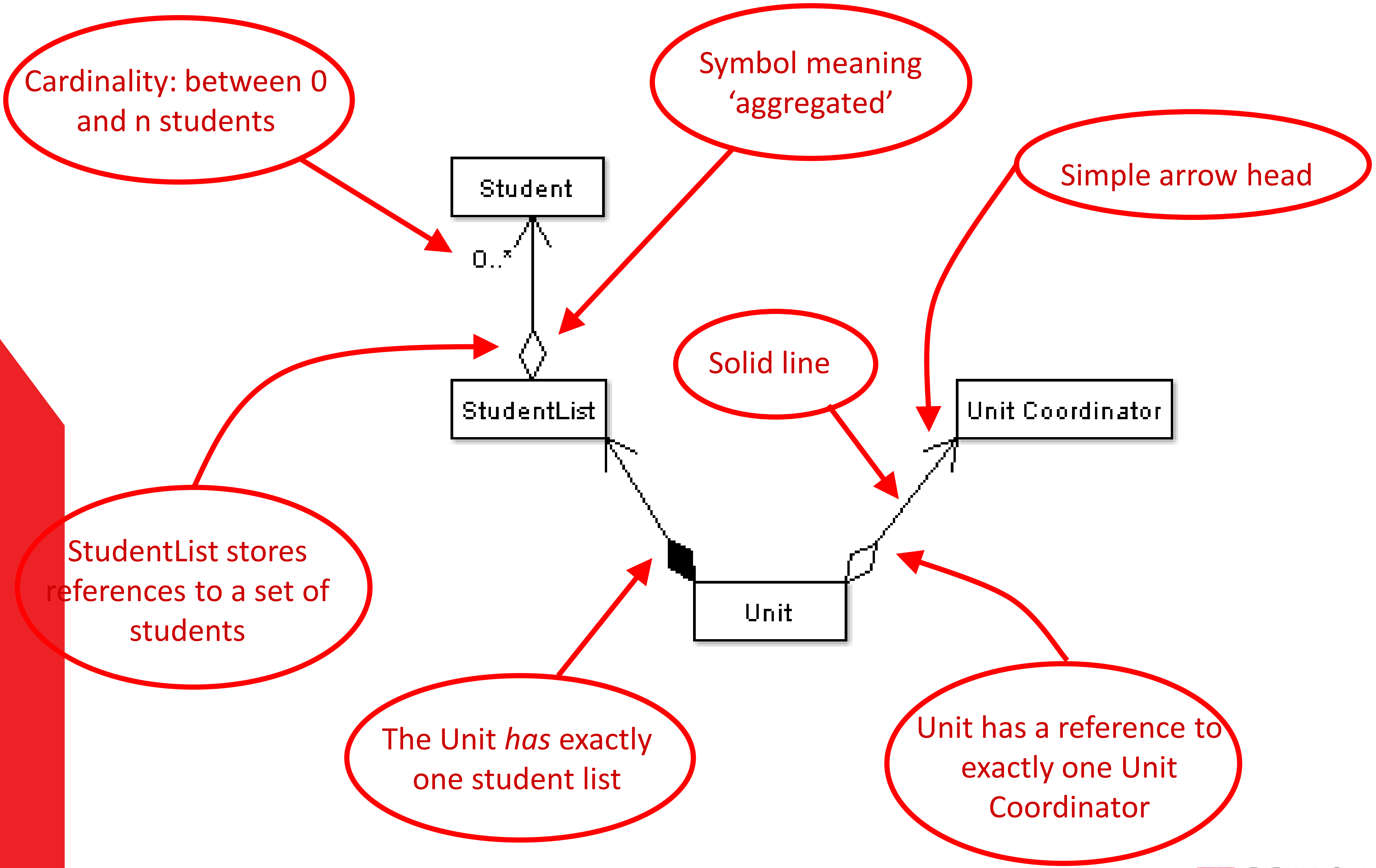
- Any class may be composed – in whole or part – of other classes.
- An instance must have only one owner object. There is no sharing with other owner objects.
- We can say that one class *"has a"* data item that is another class.
- For example, a class that emulates a simple traffic light would be composed of three lights, plus 0..2 lights that show arrows.


Cardinality
Symbol meaning 'composed'
Light
3
TrafficLight
0..2
Pattern Light
Solid line (straight or curved)
Simple arrow head showing direction of dependency.

END

Aggregation

- Some disagreement as to what this is. [1]
- We will use the following description.
  - Sometimes a class includes attributes that are pointers, references or keys to another class, but does not *control* the other class.
  - For example, a Unit class might contain students and lecturers, but if the unit is deleted, the lecturers and students are not.
  - This is called aggregation.
  - In words, one could say that one class *refers* to another class.


Cardinality: between 0 and n students
Symbol meaning 'aggregated'
Simple arrow head
Student
0..*
StudentList
Solid line
Unit Coordinator
StudentList stores references to a set of students
Unit
The Unit *has* exactly one student list
Unit has a reference to exactly one Unit Coordinator

Murdoch UNIVERSITY

The Object Oriented Incremental Method

- What classes are required?
  - Consider all the data that is to be operated on by the program and try to split it into ‘things’ e.g. person, student, lecturer, unit, etc.
  - As you choose classes, place them in a UML diagram (use StarUML – it is free). But you need to know how to draw these diagrams by hand as well.
  - Identify containers (groups) of data, for example students, lecturers, etc.: these will form classes themselves.

- Order the classes from most depended upon to least depended upon.

- Then for each class (from the one most depended upon), *before* coding the next one in the list:
  - Code ***and test*** its
    - Initialiser (constructor)
    - Output method [1]
  - Code ***and test*** its
    - Set method(s)
    - Get method(s)
  - Code ***and test*** all other methods ***one by one.***

Exercise

- A program is needed that will read in birth dates and output a person’s age. What OO data structures might be required?
- A program is needed to read in time in UTC (Coordinated Universal Time). The time for output or display is to be for a given location (e.g. Perth or Sydney). How would you design the time class (or classes) for each location?

Readings

- Textbook: Chapter on Classes and Data Abstractions. – Please read immediately.
- Reference book: *UML distilled: A Brief guide to the standard object modeling language* from My Unit Readings (chapters 1, 3 and 5) https://murdoch.rl.talis.com/index.html

Complexity

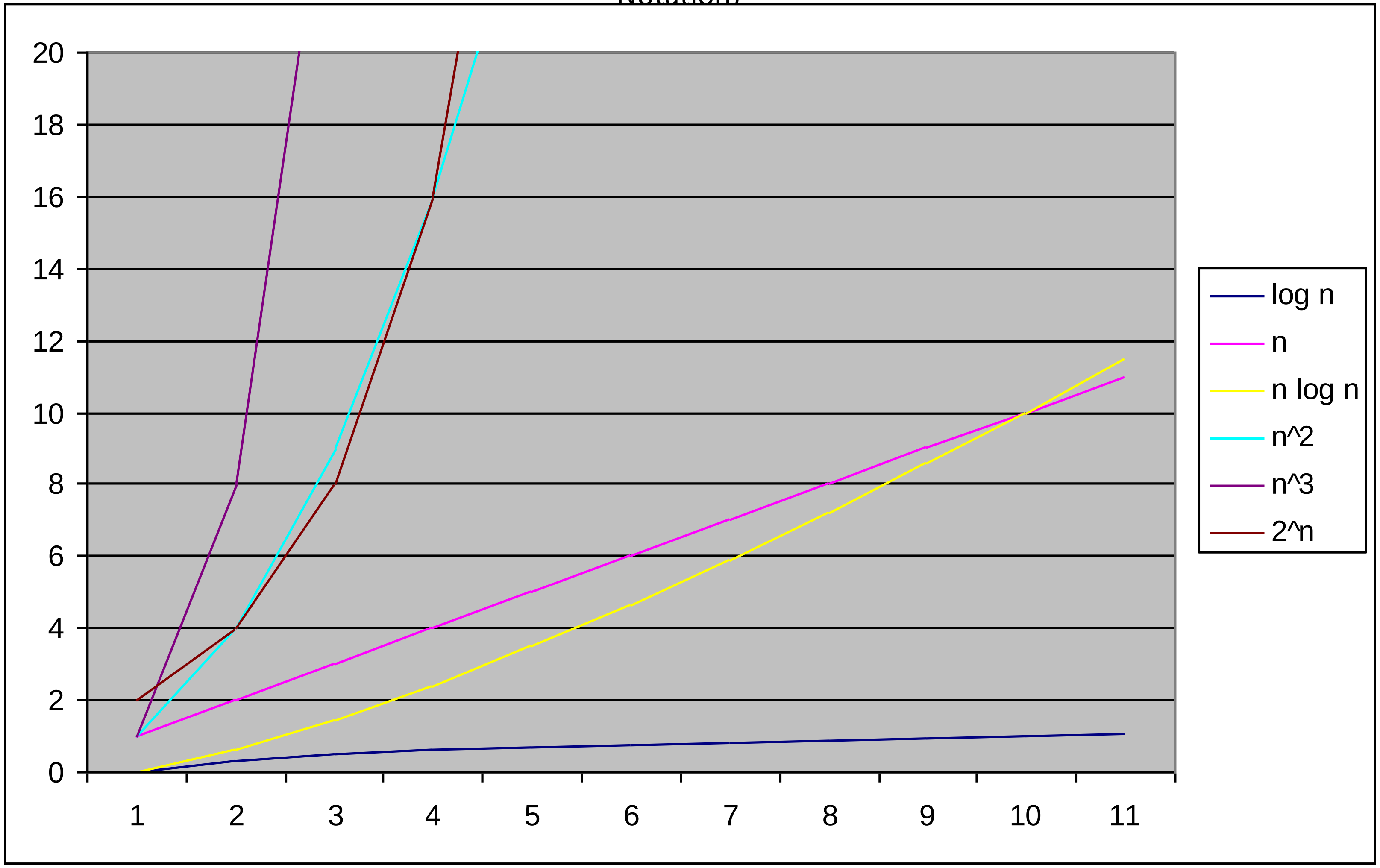
- The complexity of algorithms is measured in terms of how long they take to execute or how much resources they utilise. Our current interest is on the execution time of the algorithm.
- This in turn is often measured in terms of the number of operations that are required.
- Complexity is described in two ways: descriptively and using O notation. [1]
- O notation means ‘in the order of’. It is that value multiplied by some constant. The order represents the rate of growth. The constant is something that can depend on a particular computer and can vary from computer to computer.
  - So we don’t consider this constant and only look at the algorithm.
- For all algorithms, the measurement is relative to how many items are being processed.
- The number of items is designated as ‘n’ and Big-O is a measure of the upper bounds of the number of operations carried out by the algorithm as n grows large.

Murdoch UNIVERSITY

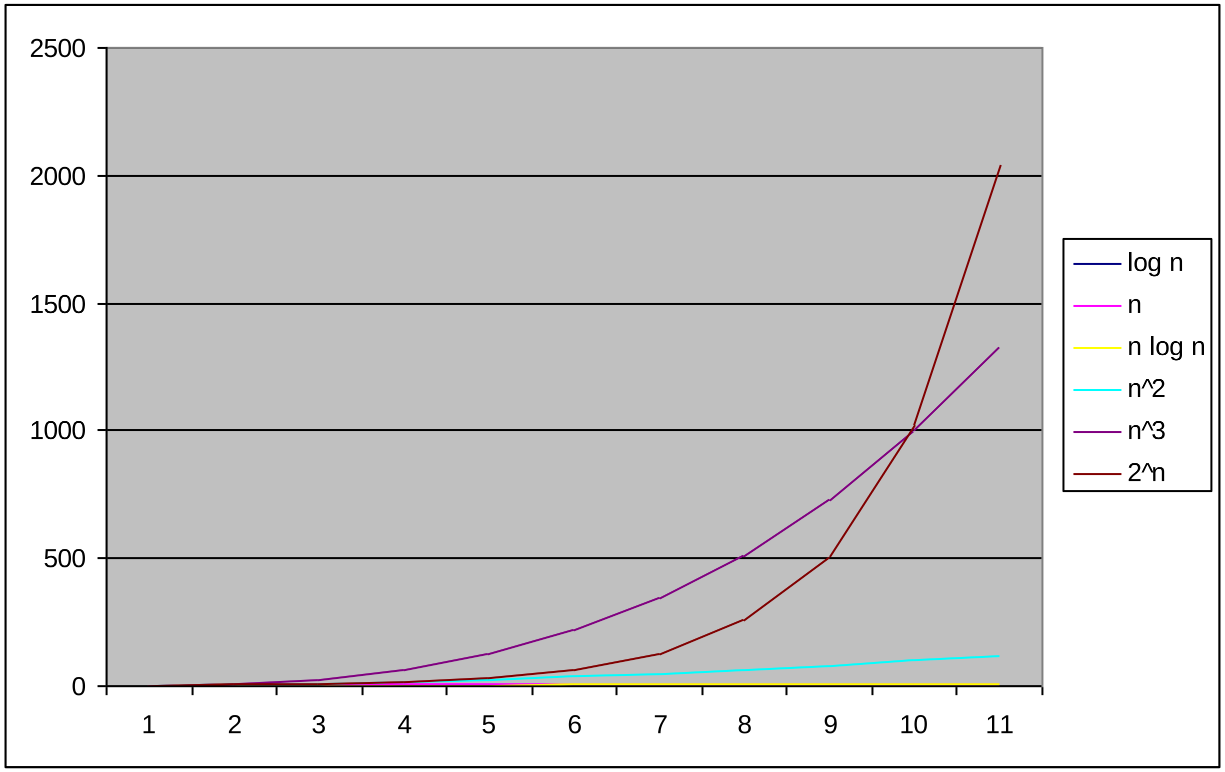
NAME	O notation	DESCRIPTION	Examples	n=1	n=10	n=20
constant time	k	It takes the same amount of time (k) no matter how many elements are being dealt with. If k is small this is the best case of all, as it will take the same time for 1 element as for 10,000.	t = k	k	k	k
logarithmic time	O(log n)	The time taken increases very slowly as n increases.	$t = \log_{10}(n)$	0	1	1.3
linear time	O(n)	The time taken increases in a straight line as n increases.	t = n	1	10	20
	O(n log n)	The time taken increases faster than n, but at a slower speed than the two below.	$t = n \log_{10}(n)$	0	10	26
polynomial time	$O(n^p)$	The time taken increases much faster than does n.	$t = n^2$ $t = n^3$	1 1	100 1000	400 8000
exponential time	$O(c^n)$	The time taken increases at a much, much higher rate than n. As n becomes very large, the time taken becomes almost infinite.	$t = 2^n$ $t = 10^n$	1 1	1024 10^{10}	1048576 10^{20}

Murdoch UNIVERSITY

Comparison at Scale 0-20 (textbook has a nicer diagram in the section on Big-O Notation)



Comparison at Scale 0-2500



The STL Vector (**can't** be used for assignment 1 – **not the same**)

- The STL vector is an array-like template class, but not an array.
- It can be instantiated as any type that you choose, including ones designed by you.
- It has lots of inbuilt methods, as well as having functions within the algorithm class that operate on it.
- Note that, like all STL classes, there is *no* bounds checking done.
- To use the vector template, you must include the <vector> header file.
- Accessing a particular element or adding an element to a vector takes *constant* time.
- Finding an element or inserting an element at a particular location within the vector takes *linear* time.

Using the STL vector

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

//-----------------------------------------------------------------

// Declare a new type that is a vector (array) of integers
typedef vector<int> IntVec;

// Declare an iterator for this type of structure
typedef IntVec::iterator IntVecItr;

// Declare a new type that is a vector of vectors of integers
//   i.e. create a two dimensional array
typedef vector<IntVec> IntTable;

//-----------------------------------------------------------------
```

```
IntVec array;
IntTable table;

// Seeding a random number generator
srand (time(NULL));

// Adding random data to a vector
for (int index1 = 0; index1 < SIZE; index1++)
{
        // Add a number between 0 and 100 to the end of the array
        array.push_back (rand() % 100);
}

// Outputting the single array:
for (int index3 = 0; index3 < SIZE; index3++)
{
        cout << array[index3] << endl;
}
```

```

        // Make a table with identical rows
        for (int index2 = 0; index2 < SIZE; index2++)
        {
            table.push_back (array);
        }

        // Outputting the table in columns
        for (int row = 0; row < SIZE; row++)
        {
            for (int col = 0; col < SIZE; col++)
            {
                cout << setw(5) << table[row][col] << " ";
            }
            cout << endl;
        }
```

Vector Methods

- Like strings, vectors have many methods.
- Again like strings, most of the methods have multiple overloads.
- A good listing of information can be found at http://www.cppreference.com/cppvector/index.html
- Unlike strings, there are only a few operators that apply to vectors.
- = and == are the two most useful operators that can be used with vectors.

- Only some examples shown – there are more.. [1]

`vec.clear ()`	Empties the vector.
`vec.empty ()`	Returns true if the vector is empty.
`vec.erase (<various>)`	Erases a part of the vector.
`vec.insert (<various>)`	Add data to the vector.
`vec.push_back (data)`	Add one piece of data to the end of the vector.
`vec.pop_back ()`	Delete the last item in the vector.
`vec.begin()`	Returns an iterator that points to the first item in the vector.
`vec.end()`	Returns an iterator that points to just after the last item in the vector.
`vec.size()`	Returns the size of the vector.
`vec.swap (vec2)`	Swaps the contents of the two vectors.

Vector Allocation

- When you push_back data into a vector, space needs to be allocated to the vector. This is done dynamically and the programmer doesn’t need to worry about creating storage. This is normal for STL containers.
- The space is not allocated one ‘slot’ at a time.
- Instead it is allocated as follows: [1]

```
IF size > allocation/2
 allocate (size+1) new slots
ELSE
 allocate 1 new slot
```

- This results in the vector always being less than half full, which must give some efficiency advantage.
- However it means that vectors are space wasters.
- Furthermore, if the vector shrinks later, this does *not* result in released memory: the memory allocated stays allocated until the vector goes out of scope.

Example of Iterator Use

- Lets suppose we have a vector containing integers.
- We decide, for some reason, that we want to remove all the elements that are equal to some particular number, entered by the user.
- This will require:
  - An iteration through the vector
  - Erasure of each target as we find it
- The easiest way to do this is using an iterator.

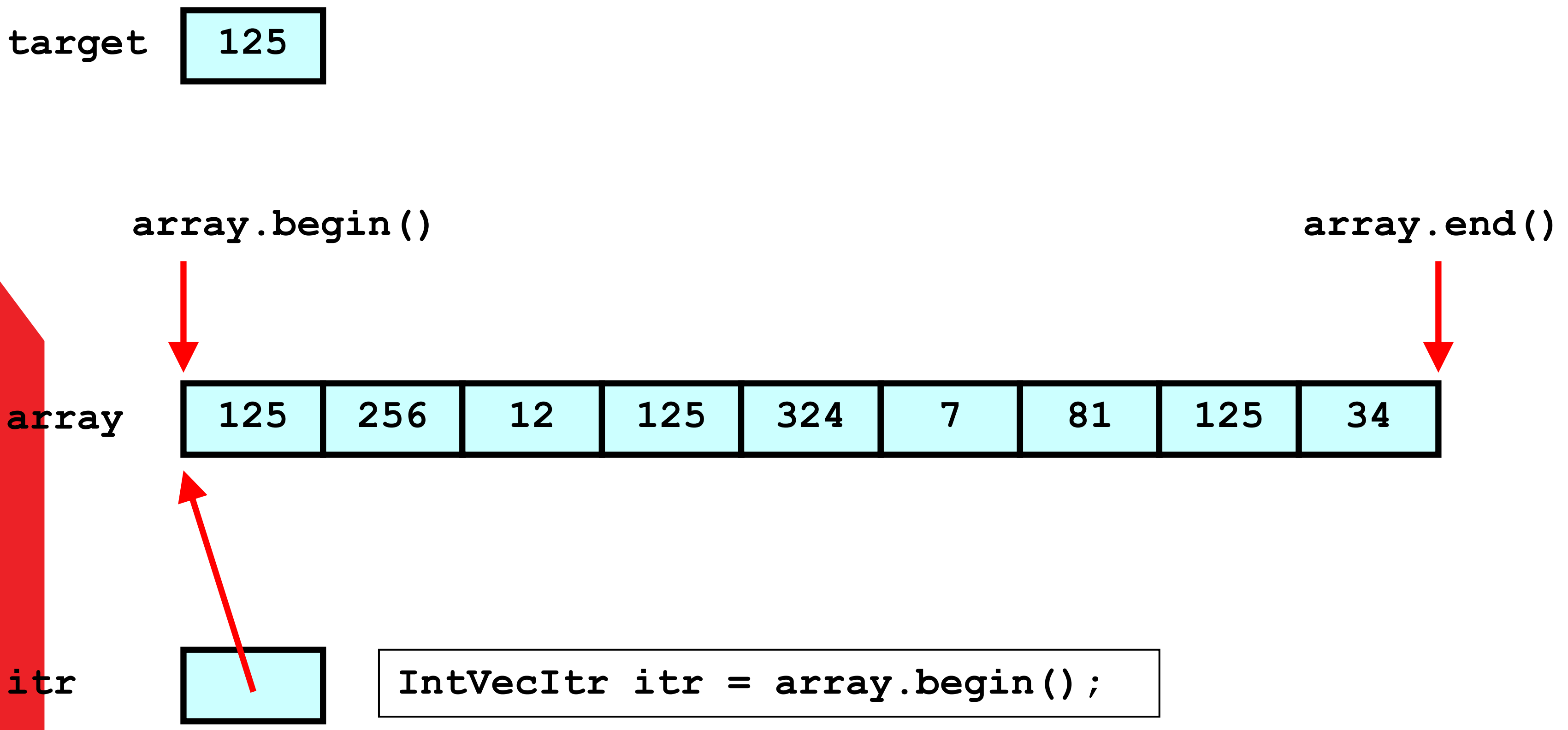
```
•         int target;
•         cout << "Enter number to be deleted: ";
•         cin >> target;

•         IntVecItr itr = array.begin();
•         while (itr != array.end())
•         {
•                 if (*itr == target)
•                 {
•                         itr = array.erase(itr); [1] [2]
•                 }
•                 else
•                 {
•                         itr++;
•                 }
•         }
```

Defined earlier

erase returns an iterator to the next item

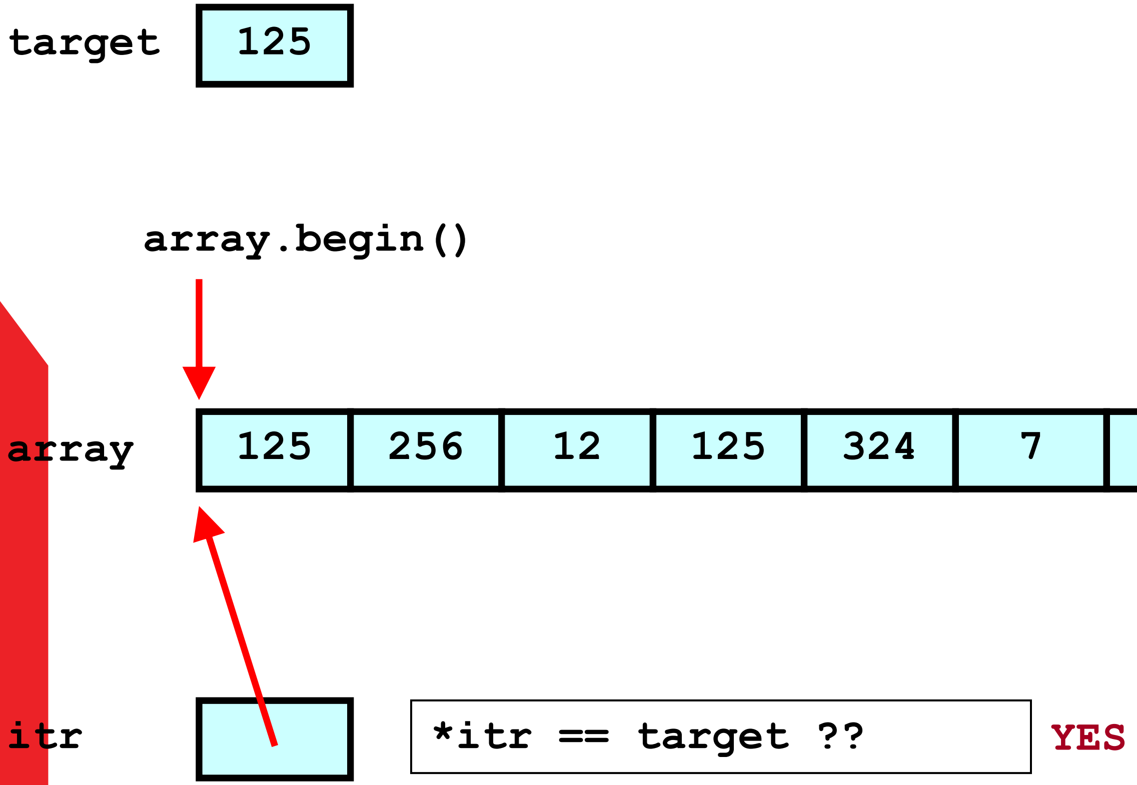
so we only need to increment itr if we did not delete an item

Example

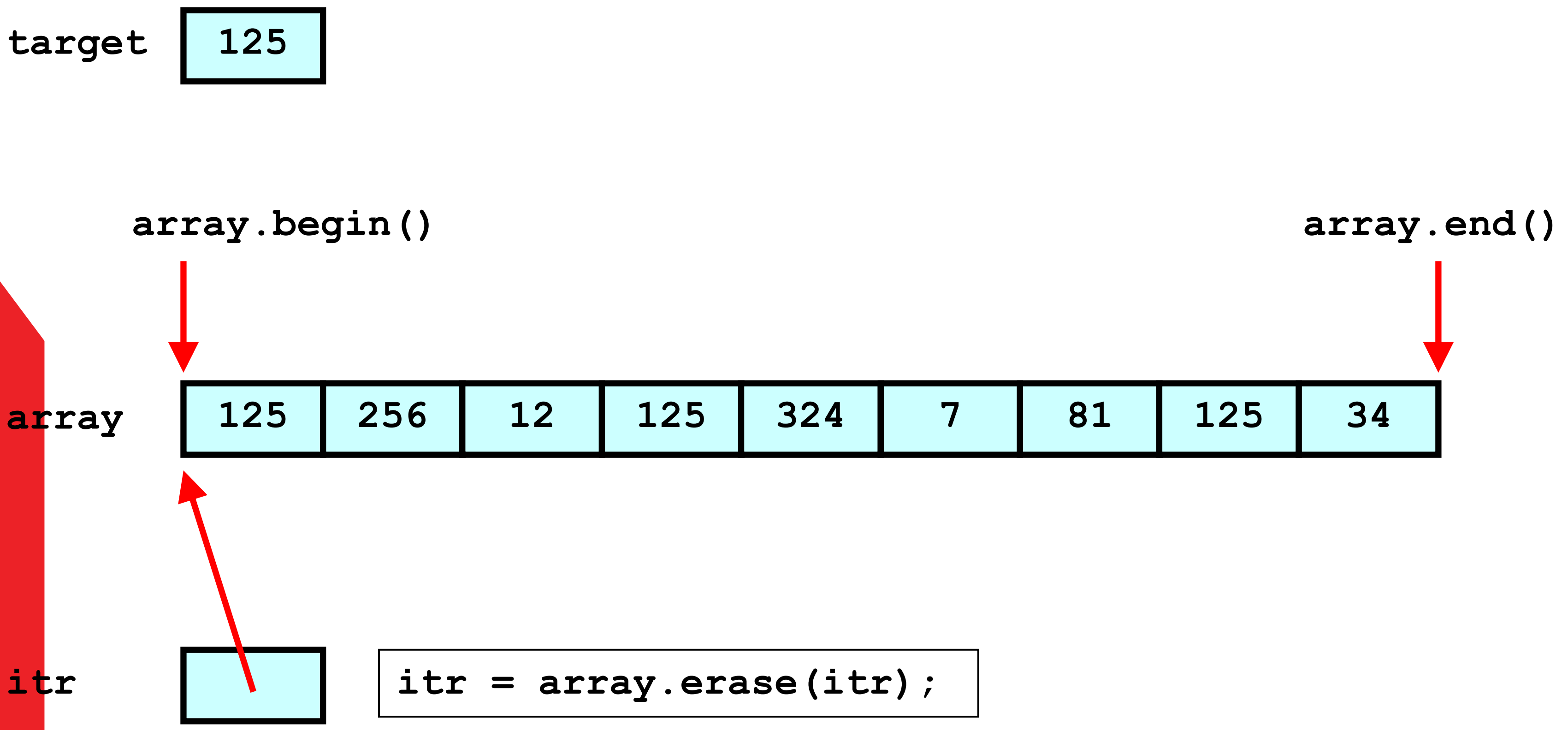


```
IntVecItr itr = array.begin();
```

Example

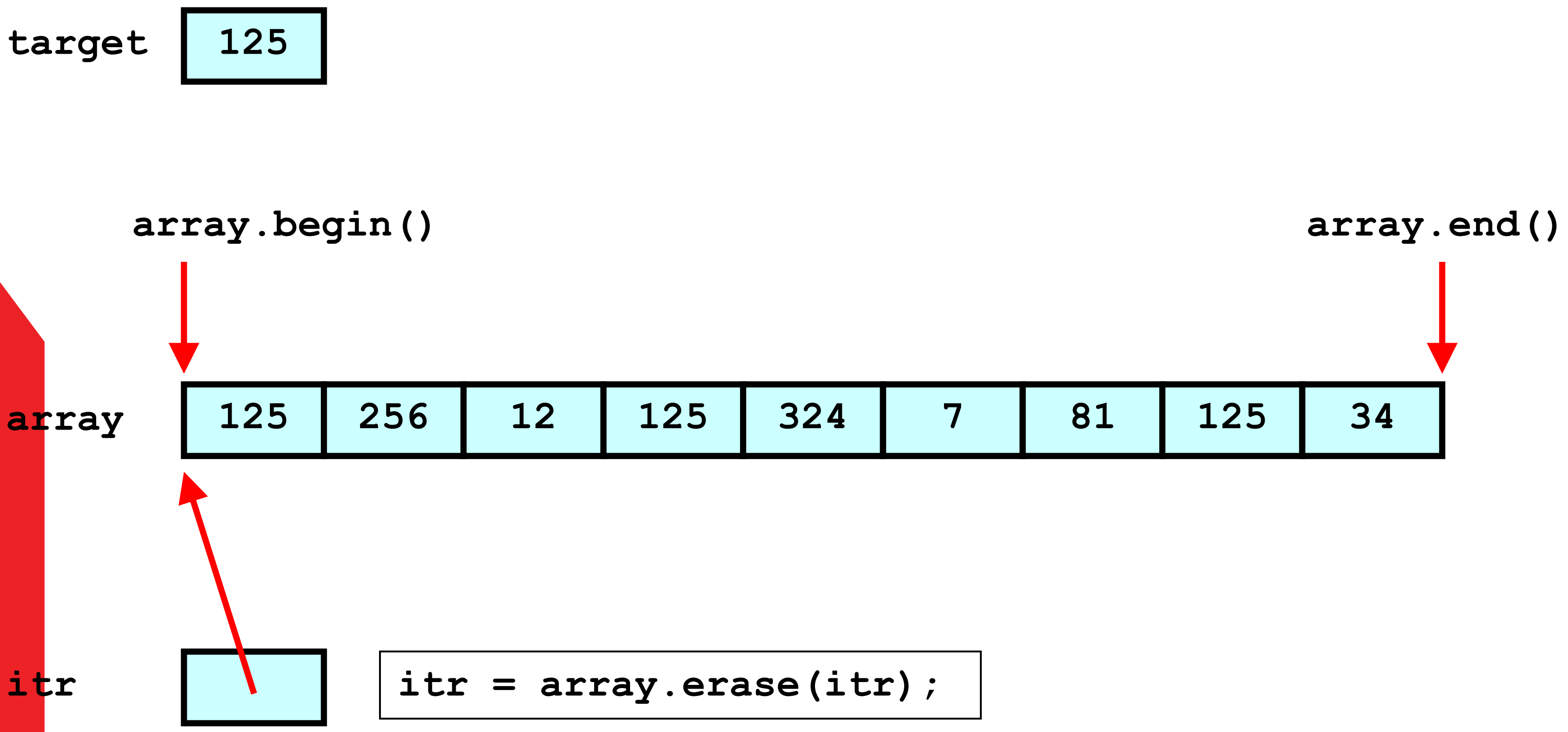

target
125
array.begin()
array.end()
array
125
256
12
125
324
7
81
125
34
itr
*itr == target ??
YES

Example



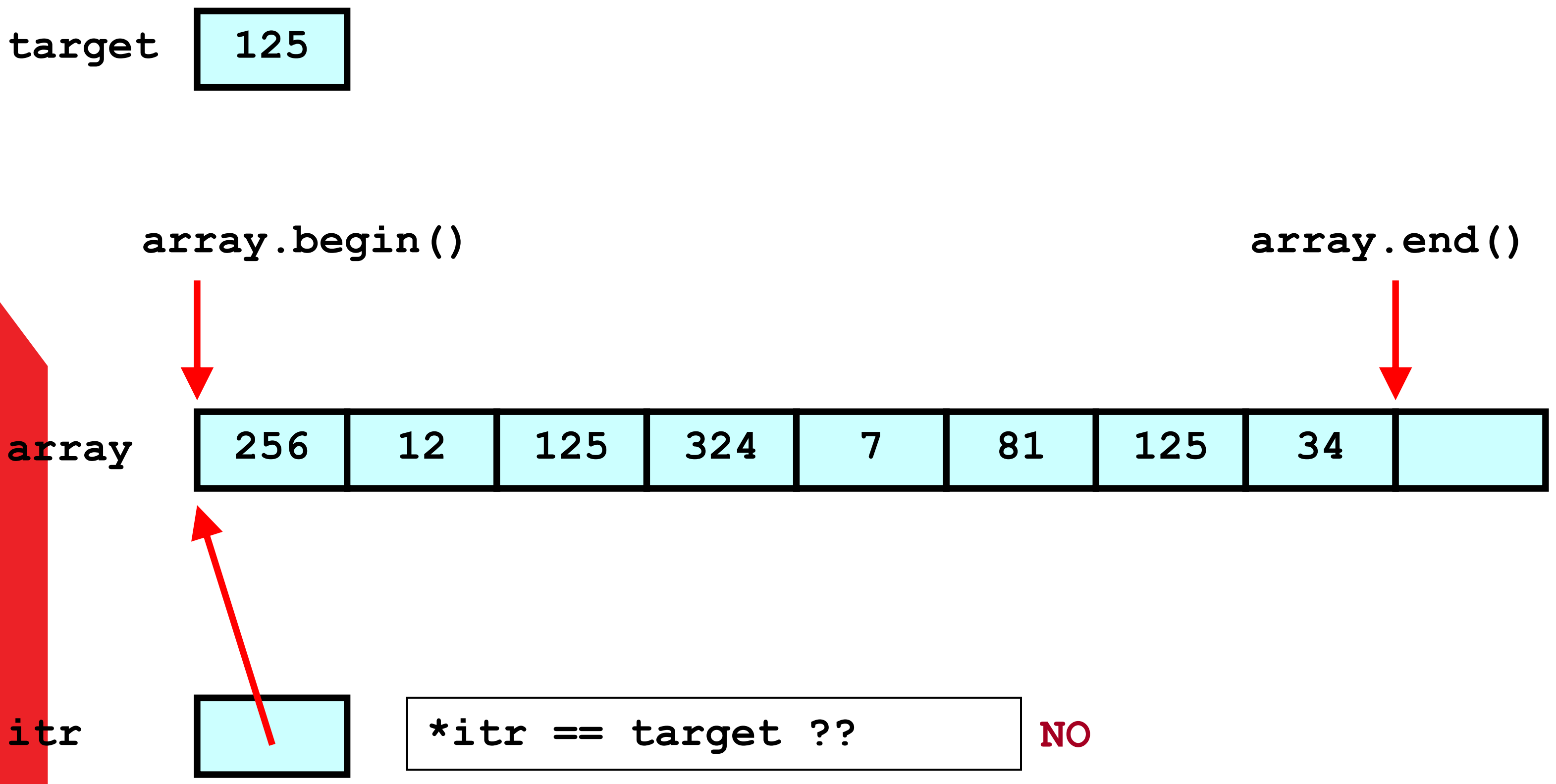
```
itr = array.erase(itr);
```

Example



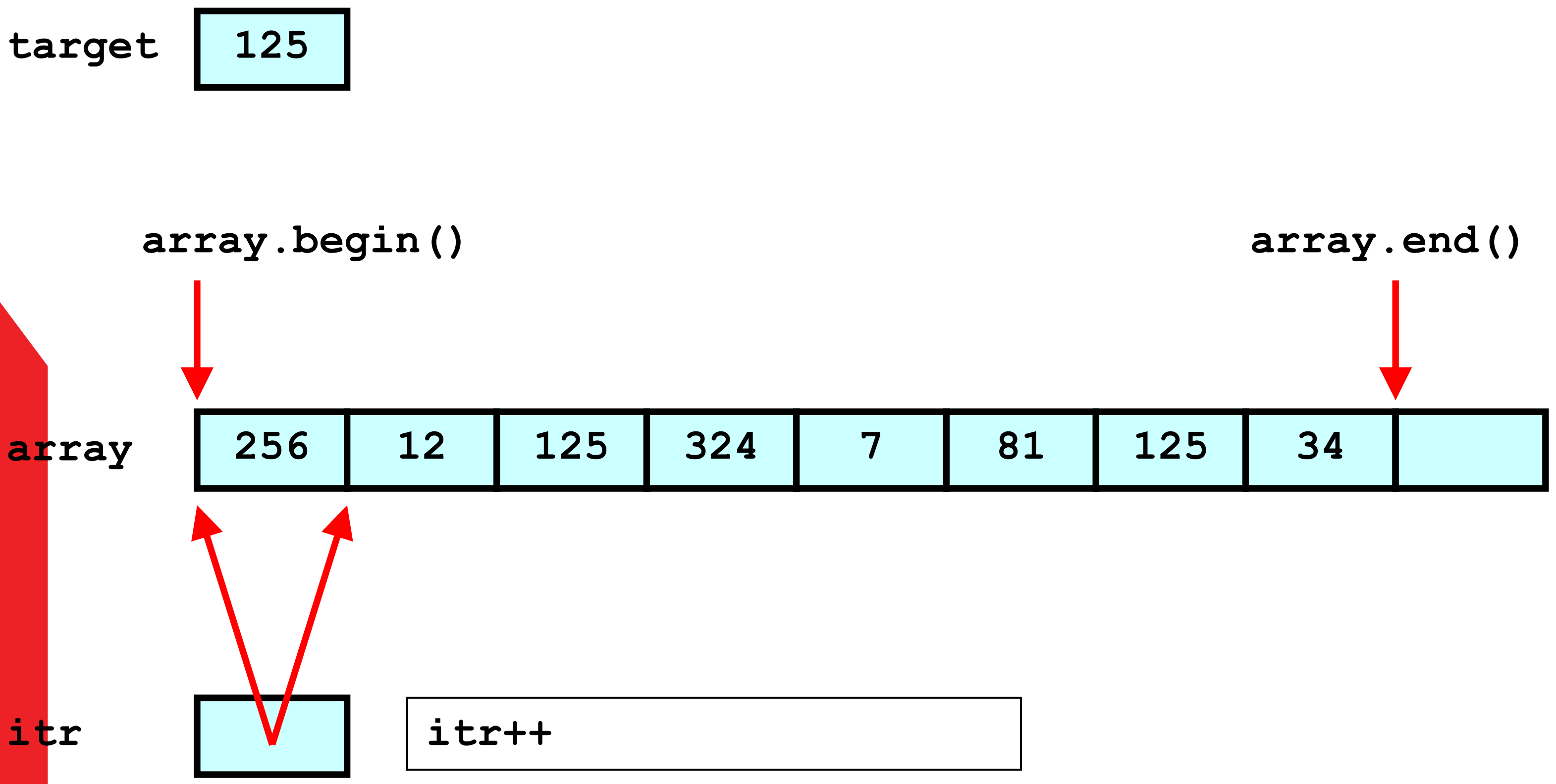
```
itr = array.erase(itr);
```

Example



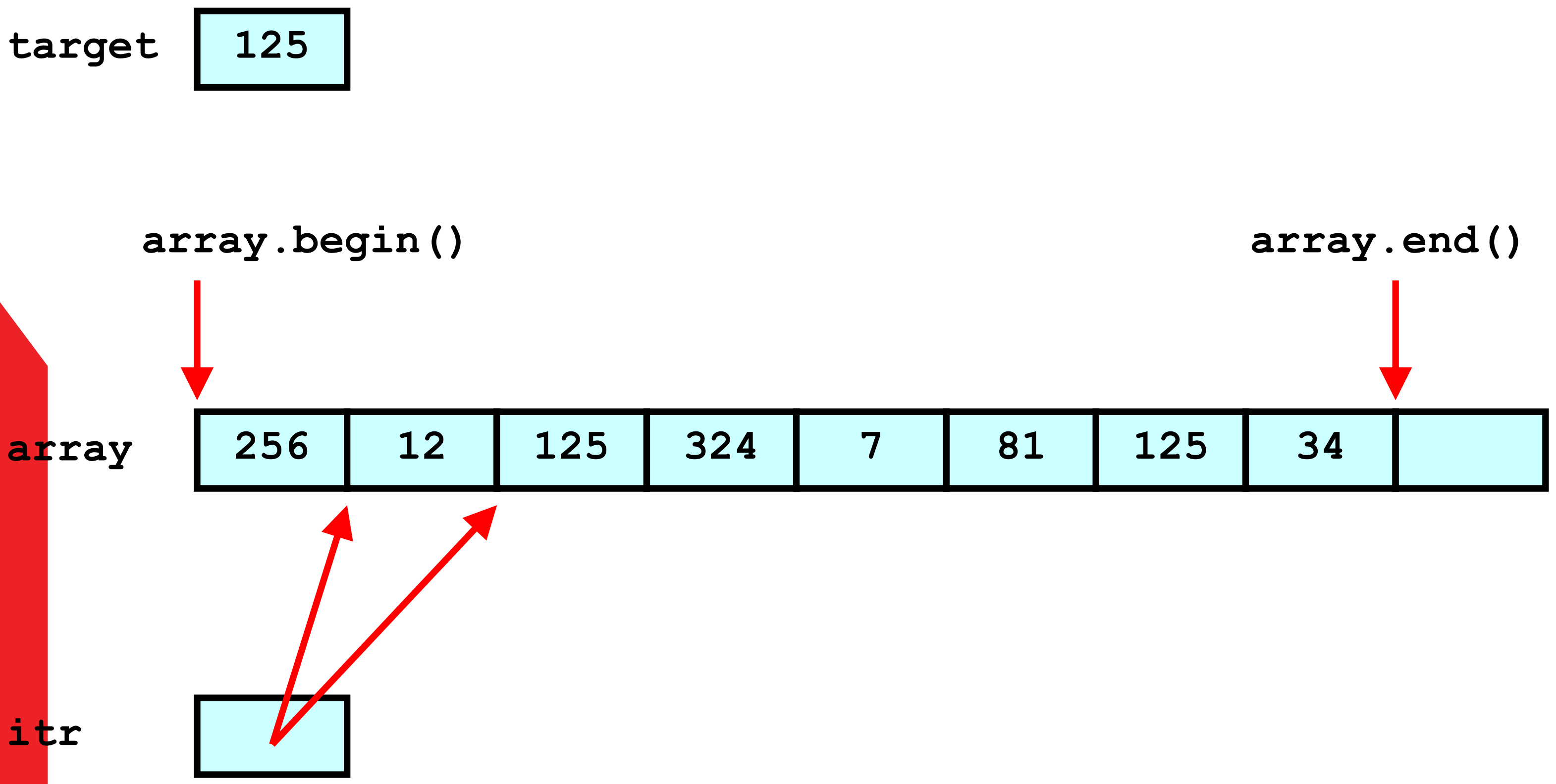
target
125
array.begin()
array.end()
array
256
12
125
324
7
81
125
34
itr
*itr == target ??
NO

Example

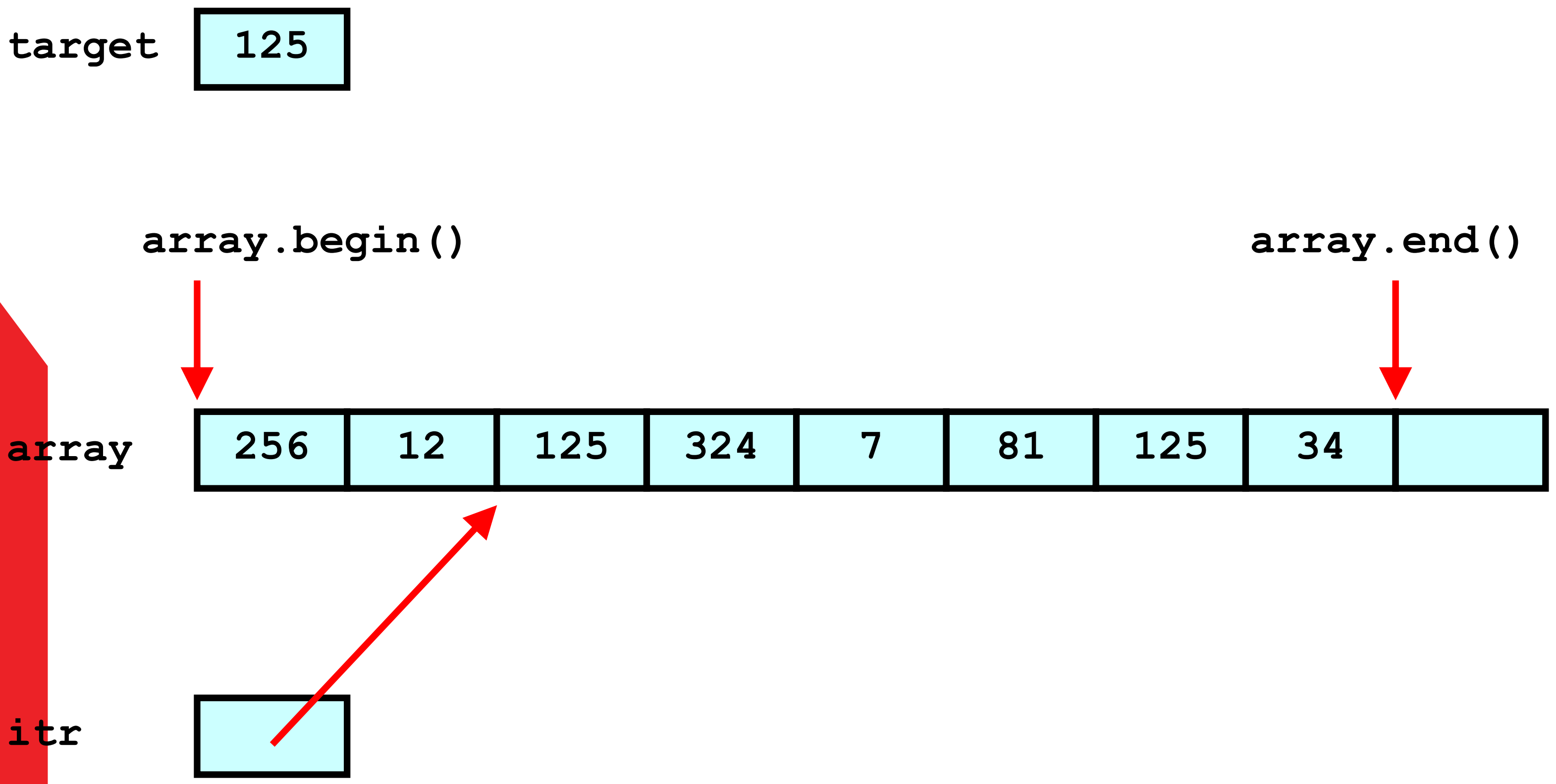


target
125
array.begin()
array.end()
array
256
12
125
324
7
81
125
34
itr
itr++

Example

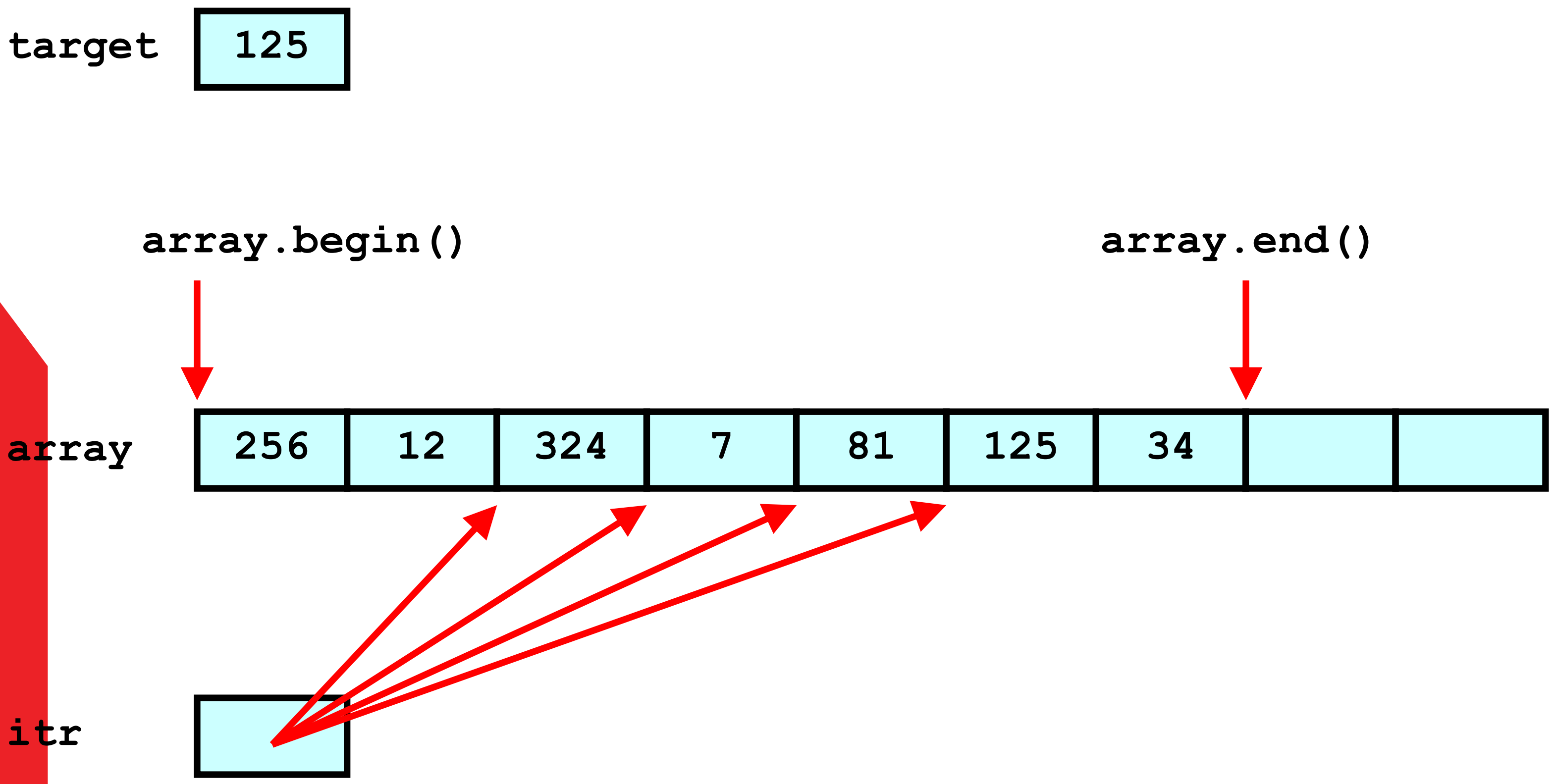

target
125
array.begin()
array.end()
array
256
12
125
324
7
81
125
34
itr

Example

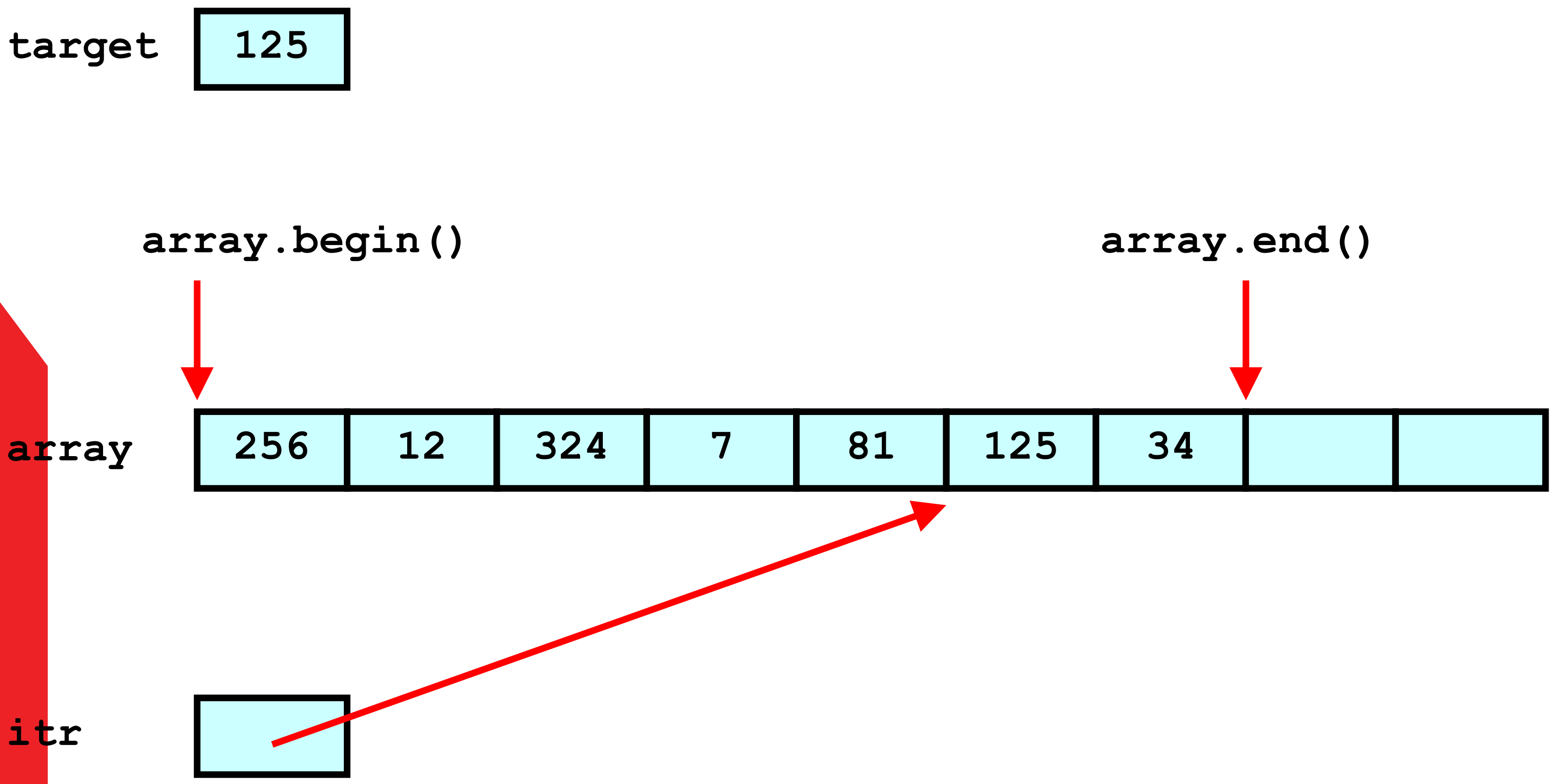


target
125
array.begin()
array.end()
array
256
12
125
324
7
81
125
34
itr

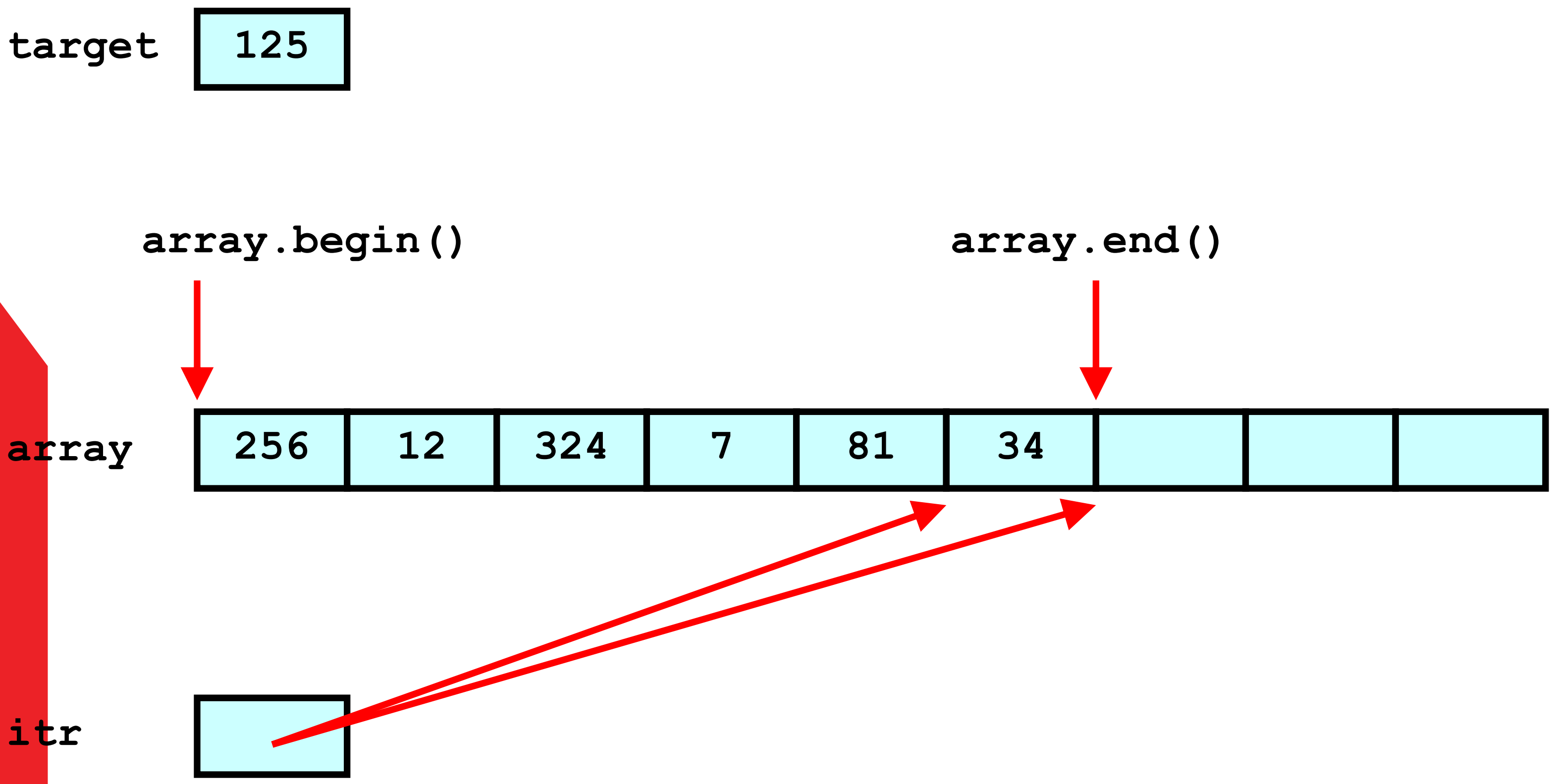
Example


target
125
array.begin()
array.end()
array
256
12
324
7
81
125
34
itr

Example


target
125
array.begin()
array.end()
array
256
12
324
7
81
125
34
itr

Example



- What is the Big-O value for erase on vectors?

END

The Algorithm Class

- You may have noticed that there were no ‘find’ or ‘sort’ methods for the vector.
- That is because they are part of the STL algorithm class instead.
- To use this class, you must include the <algorithm> header file.
- The following code does the same repetitive delete as the previous code, but uses the ‘find’ algorithm.

```
•           int target;
•           cout << "Enter target to delete: ";
•           cin >> target;

•           IntVecItr itr = find (array.begin(), array.end(), target);
•           while (itr != array.end())
•           {
•                       array.erase(itr);
•                       itr = find (itr, array.end(), target);
•           }
```

- Note how much tighter (shorter) the code is now.

Available Algorithms

- The algorithm class contains over forty algorithms.
- Please see the following links which have examples of use:

  http://www.cppreference.com/cppalgorithm/index.html

  http://www.cplusplus.com/reference/algorithm/
- The use of them assumes various operators are available for the data with which they are instantiated.
- This is no problem for a vector of integers or floats etc, as these already have all arithmetic and logical operators defined.
- Later on when we code classes, we will have to *overload* these operators if we wish to use the algorithm class's algorithms.

Iterators Again

- If you actually have the index of something you want to delete or erase, you can 'add' index to the `.begin()` iterator to get the correct thing to delete (or insert).

- For example:
```
array.erase (array.begin() +
10);
```

The STL deque class

- Pronounced as “deck”
- The STL deque (double ended queue) class is almost identical to the vector class, except that it has as extra:
  - `push_front(const DataType &data)`
  - `pop_front ()`
- As well as the ones that work at the back.
- deque can grow dynamically at either end.
- Both of these functions work in constant time.

deque

- Study the deque examples in the textbook in the chapter on “Standard Template Library”

Exercise

Using the STL vector as your data structure: [1]

- **Write a simple program that will read in numbers from a file and output the mean and median to screen.**
- **Modify the program to output the standard deviation.**

Readings

- Textbook: Chapter on Searching and Sorting algorithms, section on Asymptotic Notation: Big-O Notation. If this is not found in your edition of the textbook please see the reference book "Introduction to Algorithms" chapter on Growth of Functions, section on O-notation.
- For a more comprehensive coverage of algorithms and their efficiency, see the reference book, Introduction to Algorithms. Chapters 1 to 3.
- Textbook Chapter on STL:
  - Sections related to the sequence container **vector** and **iterators** related to the sequence container vector.
  - The sequence container **deque** is also found in the above chapter.
  - **Skip** sections on ostream iterator and copy function.
- Website: "C++ Reference", http://www.cplusplus.com/reference/ [1]
- Website: CPP reference, http://www.cppreference.com/wiki/ [2]