Murdoch
UNIVERSITY

Data Structures and Abstractions

Program Control in C++

Lecture 4/revision

Complete Topic 1

- You need to have completed the programming exercises and readings from topic 1.
- Completion of the previous lab exercises is particularly important.
- To be able to do the program in lab 2, you would also need to have completed the readings for topic 1.
  - Particularly chapter 3 “Input/Output” and the chapter on “Classes and Data Abstraction” from the textbook.

Control Statements

- Like almost all third generation languages, C++ has selection, iteration and call statements.
- Selection:
  - `if`
  - `switch`
- Iteration:
  - `for`
  - `while`
  - `do-while`
- Calls:
  - `functions`
  - `procedures`

Selection – IF

- if (condition)
- {   // always use these braces [1]
-           // Do something
- }
- else
- {   // always use these braces [1]
-           // Do something else
- }

Selection – Switch

- **The switch statement is used when there are choices based on the value of a single variable.**
- **The variable has to be an *ordinal* value: an integer or boolean.**

```
•     switch (variable)
•     {
•           case value1:
•               // do something
•               break; // do not miss this. If not required - comment
•           case value2:
•               // do something
•               break; // do not miss this. If not required - comment
•           default:
•               // do something
•     }
```

Iteration - WHILE

- While loops have the condition at the beginning of the loop.
- Therefore the loop must be *primed*, i.e. the condition must be set *before* the loop commences.
- Something inside the loop must change the condition so that it ends sometime!

```
// prime the loop

while (condition)
{
  // do something
  // loop body should have code which can alter
      // condition
}
```

Iteration – Do-While

- Do-while loops execute at least once, therefore the loop does need to be primed.
- Something inside the loop, however, must change the condition.
- They are rarely useful except for menus.

```
do
{
   // do something
} while (condition);
```

Iteration – For loop

- Very versatile: can combine traditional counting with while loop condition.
- Multiple parts in the initialise and increment sections are separated by commas. Where possible avoid these.

```
for (initialising code; condition; increment)
{
  // do something
}
```

Selection and Iteration Notes

- Using the best type of selection or iterator control makes it easier to read and debug code.
- Note that **break** statements are used in switch statements and *nowhere else!*
- **continue** and **goto** statements should be avoided [1]
- These statements make the code poorly structured and therefore harder to maintain and debug. [1]

Functions

- Functions are declared above main (called prototypes) and then defined below main.
- They return a value to the variable of a defined type.
- They can have parameters and can return values within the parameters.
- Advice that user named Function names start with a capital letter and use capitals for new words eg:
  FindMaximum ()
- The list of parameters used where the function is defined is called the *formal* parameter list.
- The list of parameters used where the function is called is called the *actual* parameter list.

```
// function.cpp [1]
// You need to have completed the programming exercises and readings from
// topic 1.

// Simple program showing function use
//
// Version
// 001 Nicola Ritter
//     Just a very simple function
//--------------------------------------------------------

#include <iostream>
using namespace std;

//--------------------------------------------------------
// Prototypes

int Sum (int num1, int num2);

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

```
•
•     int main ()
•     {
•           float num2 = 8;
•           float num1 = 22;

•           cout << num1 << " + " << num2 << " = " << Sum (num1,
                        num2)
•                                     << endl << endl;

•           return 0;
•     }

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

•     int Sum (int num1, int num2)
•     {
•           return num1 + num2;
•     }

•     //-----------------------------------------------------
```

Murdoch
UNIVERSITY

```
// function2.cpp
//
// Simple program showing function use with a variable parameter
//
// Version
// 001 Nicola Ritter
//     Just a very simple function
//-----------------------------------------------------
#include <iostream>
using namespace std;

//-----------------------------------------------------
// The '&' below means that the value can be changed within the
//   procedure or function

bool Divide (float num1, float num2, float &result);

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

```
int main ()
{
    float num1 = 0;
    float num2 = 0;
    float result = 0;

    cout << "Enter two numbers: ";
    cin >> num1 >> num2;

    if (Divide (num1, num2, result))
    {
        cout << num1 << " / " << num2 << " = " << result << endl;
    }
    else
    {
        cout << "You cannot divide by zero!" << endl;
    }

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

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

bool Divide (float num1, float num2, float &result)
{
    if (num2 == 0)
    {
        return false;
    }
    else
    {
        result = num1 / num2;
        return true;
    }
}

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

Procedures

- Procedures, in C++, are simply **void** functions:
-

```
void Sum (int num1, int num2, int &result)
{
   result = num1 + num2;
}
```

- When it is called:

```
Sum (num1, num2, result);
```

Style and Layout

- Use a comment line //--------------- to separate sections of a program.
- Follow the placement of braces as shown in the notes and the book.
- Indent the same amount each time: preferably four spaces.
- Ideally, you should be able to see the whole of a function on the screen: if you cannot then you need to split the function up.
- Avoid having more than two ‘return’ statements in any function.
- Avoid having two ‘return’ statements that are physically far apart in a function.
- Braces must be used for *every* block, even if only one line long. [1]

Readings

- Textbook (by Malik): Chapters 4, 5, and chapter on User Defined Functions.

Murdoch
UNIVERSITY

Data Structures and Abstractions

Pointers

Lecture 5

Error Types Reminder

- Don’t forget the different error types when coding.
- Debugging is an absolute necessity when coding and even more so when coding pointers, so discussing this briefly here is appropriate.
- There are five types of common errors:
  - *Syntax* errors are those that prevent your code from compiling because of incorrect use of language grammar.
  - Semantic errors are errors in meaning. For example, if you have an apple object and you try to add to an orange object. Much more common ones are when you try to assign a float value to an integer variable. These can also be called type errors.
  - When your code compiles and your program runs as planned, but the output is incorrect, you have a *logic* error. That is why you need a test plan! And why you actually run through it! You had to use the test tables in the revision exercise.
  - When your program crashes it is a *run-time* error. You need a test plan and testing.
  - Finally if the input data is incorrect in some way, then your output will be incorrect also: a GIGO error.

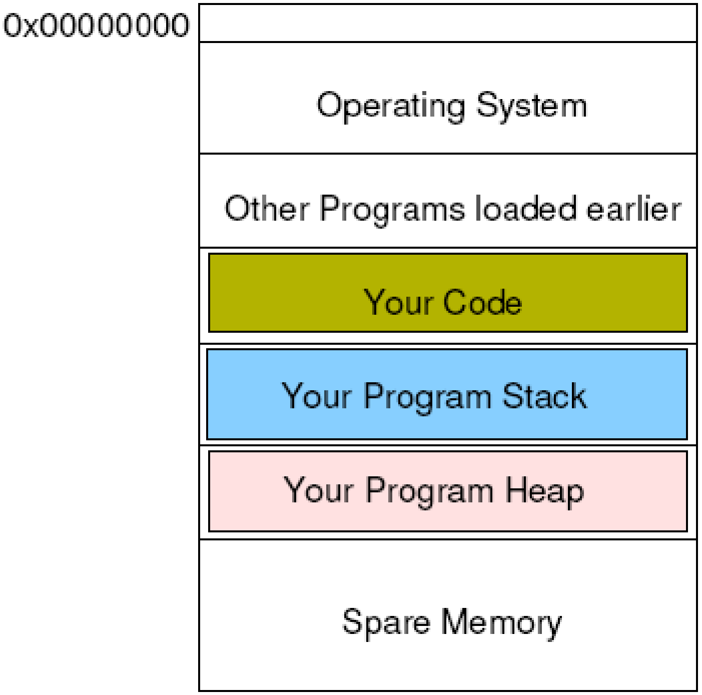
RAM

- When your program runs, the OS allocates RAM for its use. [1]

- This RAM is divided up into different sections for use in different ways by your program. The layout shown below can be different for different architectures and OS. Covered in your first year unit.

- The program stack stores variables, parameters etc. that are defined when you write your program.

- The program heap is used for memory locations that are defined as your program runs: *dynamically* allocated variables.



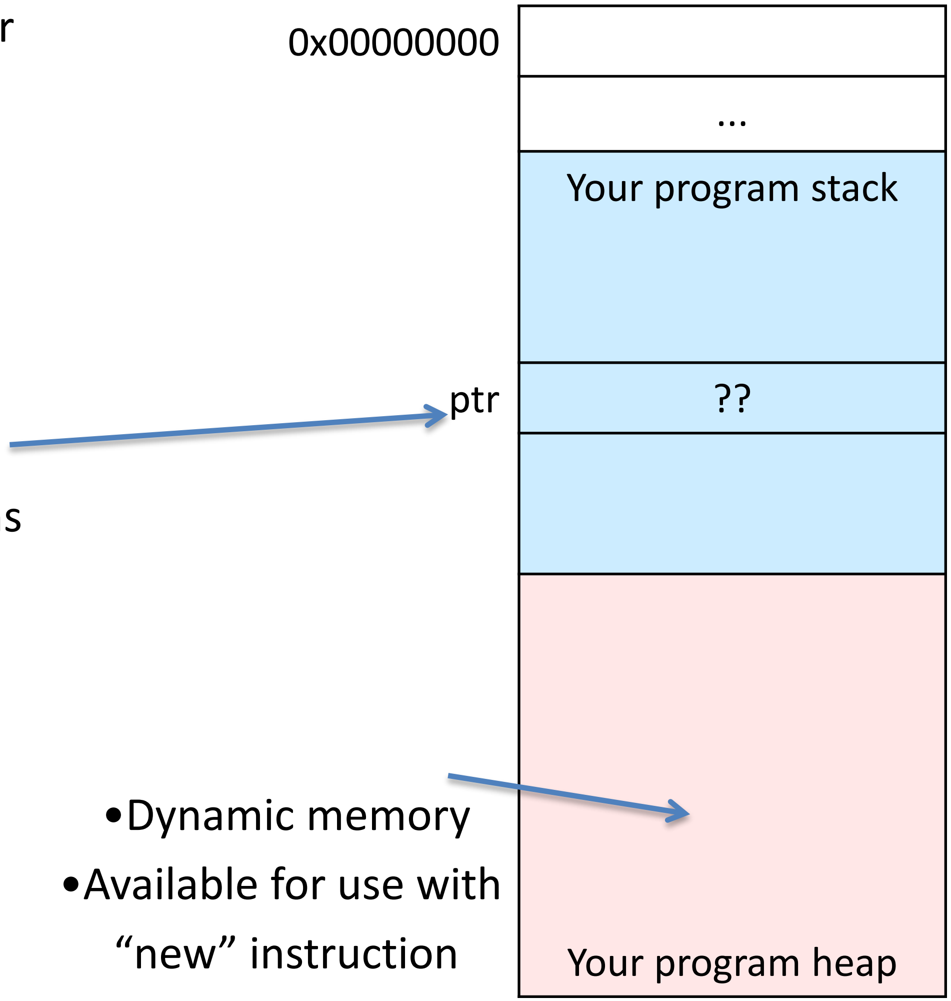
Pointers

- Pointers cause more heartache than is really necessary. [1]
- All they are is a variable that stores the address of another variable.
- Of course it is also possible to continue this ad-infinitum, which means that you can end up with an address, of an address, of an address...
- But in actually fact it is rare that you go beyond one, or two, levels of *dereference.*

Declaring Pointers

- This declares a pointer to an integer sized memory location.
- It does *not* allocate the memory location for you to store the actual data.
- Memory to store ptr itself but not the data. [1]
- Contents of ptr is whatever happens to be there by chance.

```
int f1()
{ // local vars on stack
        int *ptr;
}
```

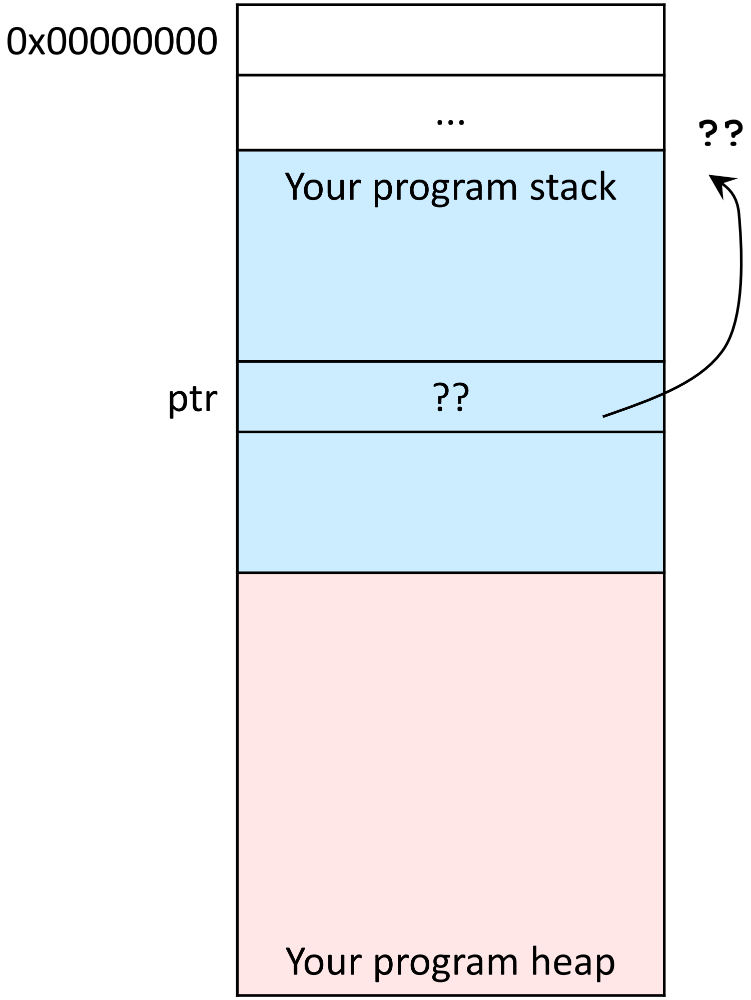


- Dynamic memory
- Available for use with "new" instruction

Declaring Pointers

- This means that trying to use the memory location contained in the pointer variable will crash the program or cause strange events.
- This is because the contents of ptr could be anything; i.e. it could be pointing *anywhere* in memory. [1]

```
int *ptr;
```



0x00000000

...

??

Your program stack

ptr

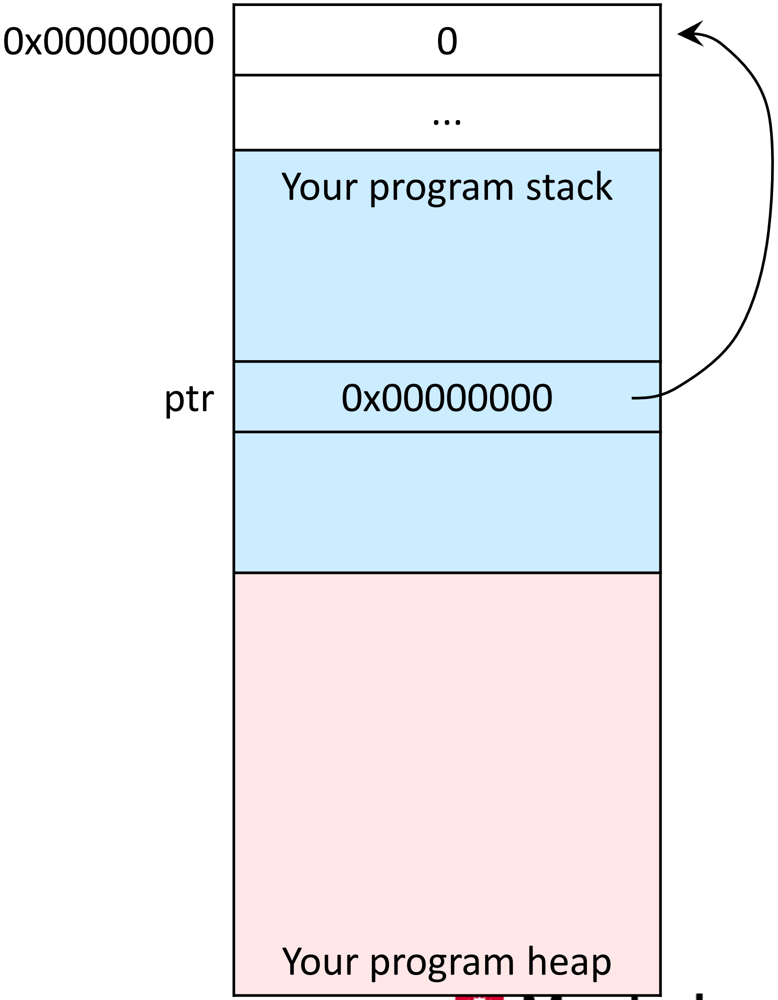
??

Your program heap

Declaring Pointers Safely

- To prevent accidental alteration of anything important, pointers should always be initialised to NULL.
- The zeroth memory location of RAM is kept empty for this reason. [1]

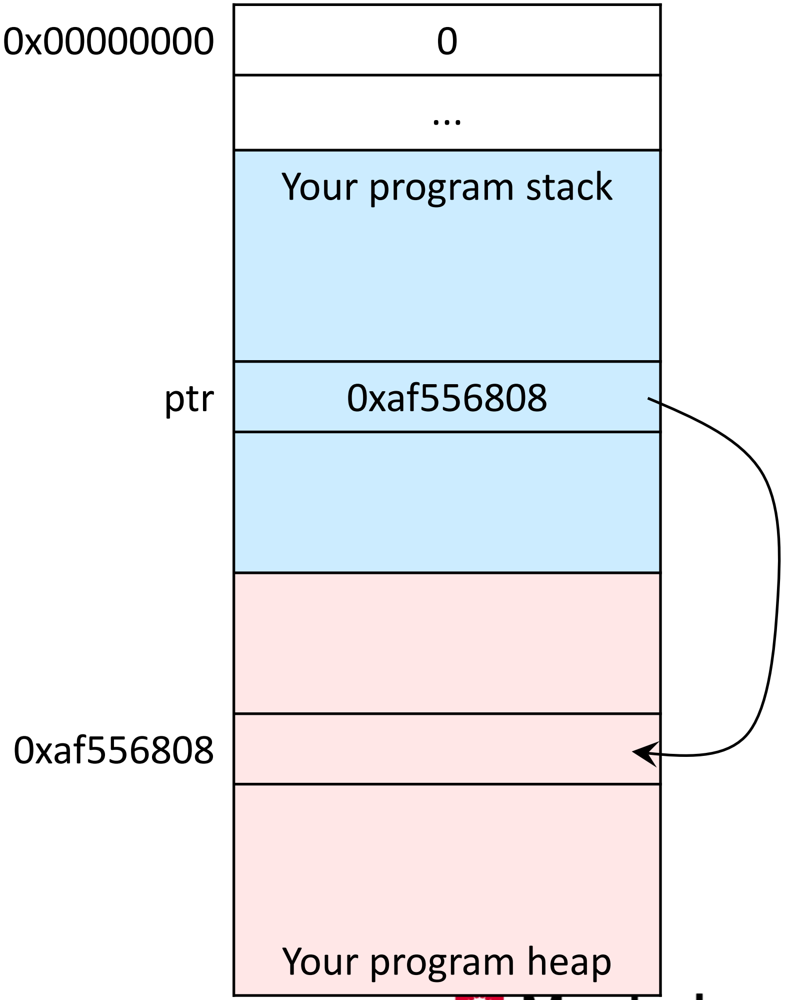
```
int *ptr = NULL;
```



Allocating Memory

- The allocation of memory is then done with the **new** keyword.
- This allocates a memory location on the heap of the given size (in this case an int).

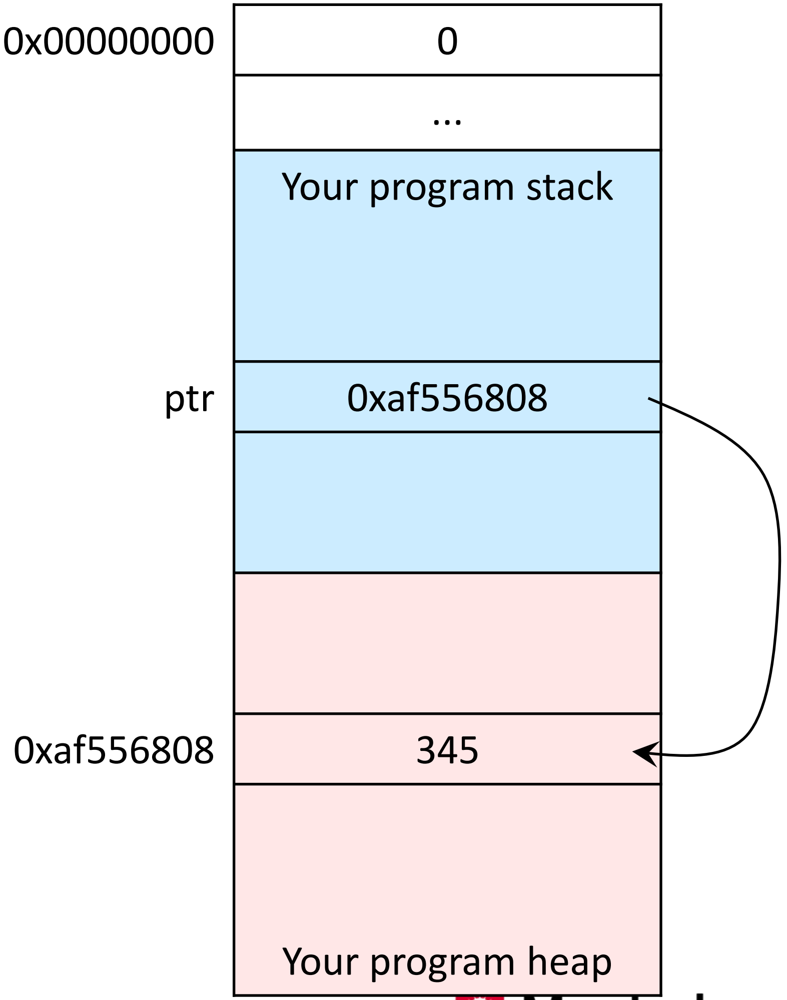
```
int *ptr = NULL;

ptr = new int; [1]
```



Using Pointers

- To place a value in the memory location, the * dereferencing operator is used. [1]

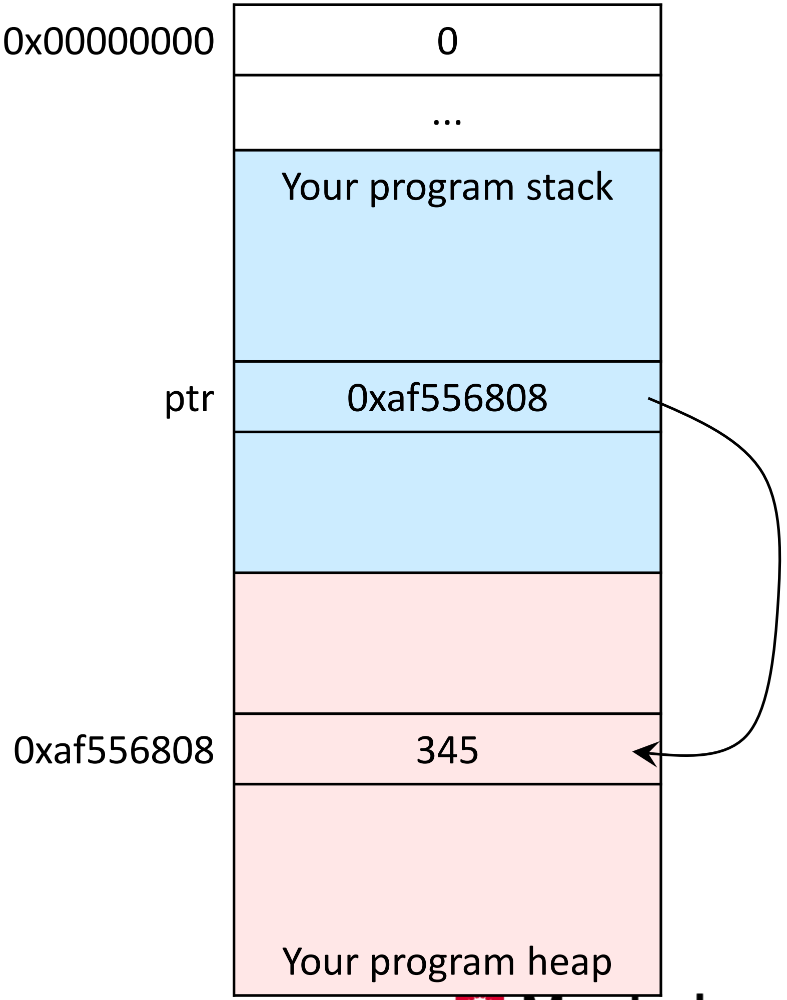
```
int *ptr = NULL;
ptr = new int;
*ptr = 345;
```



Using Pointers

- This is also used for accessing the location for output etc.

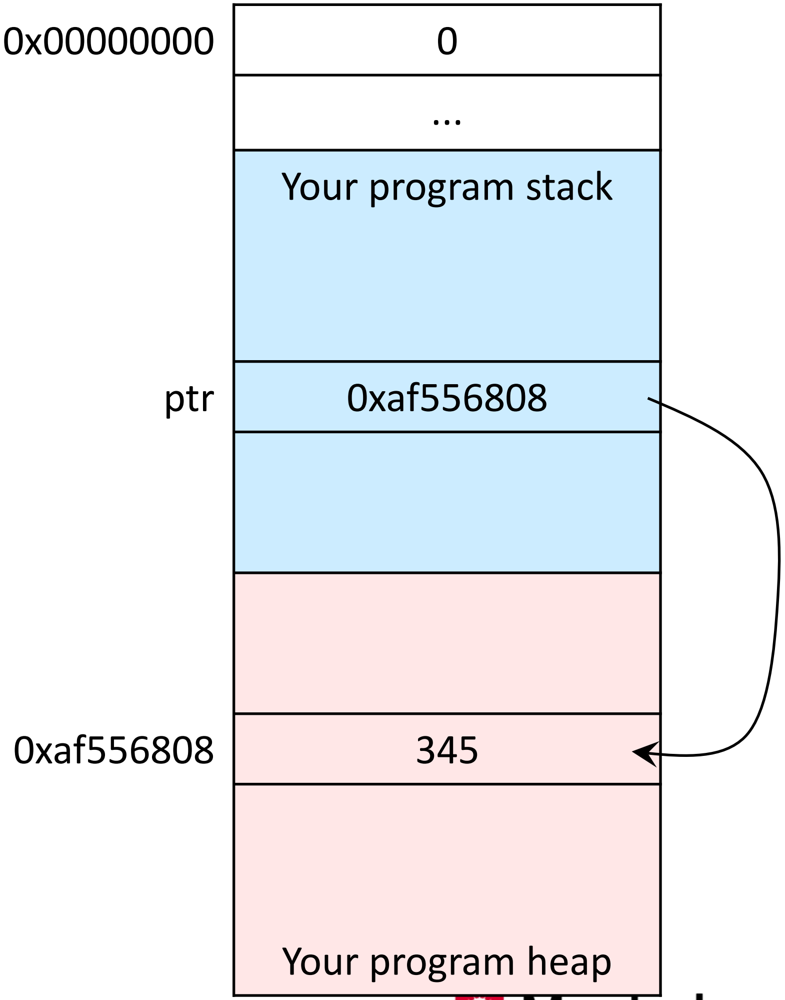
```
int *ptr = NULL;
ptr = new int;
*ptr = 345;
cout << *ptr << endl;
```



Where is the Pointer Pointing?

- You could also output the location of the memory being used, rather than the contents of the memory location.

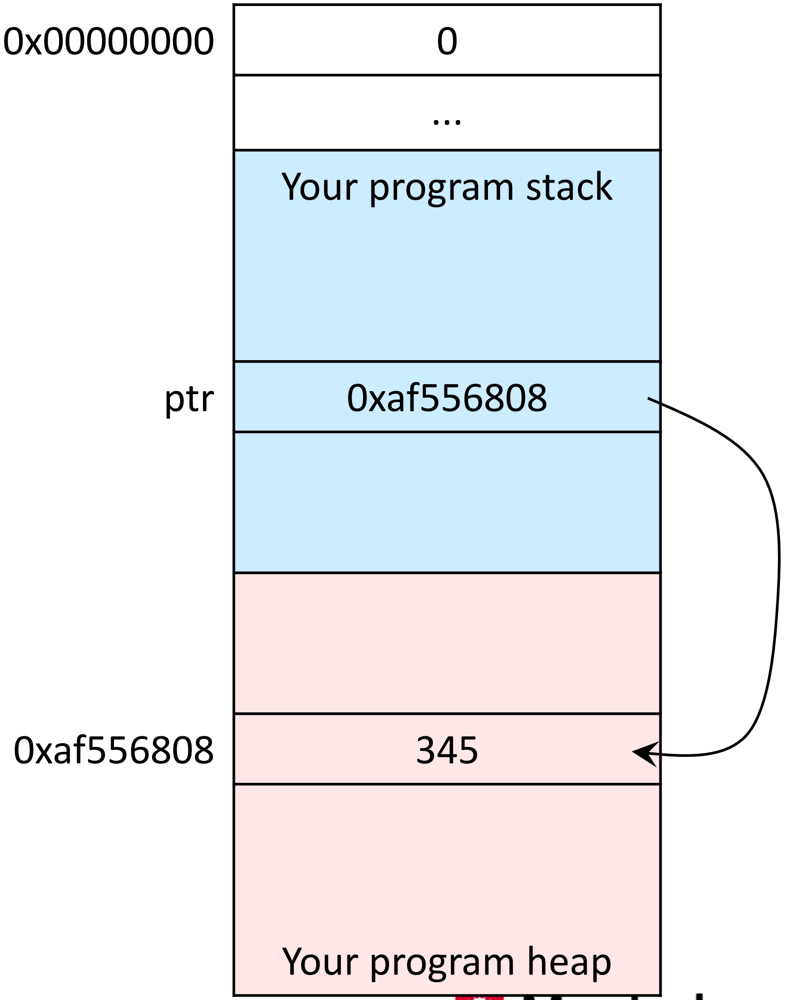
```
int *ptr = NULL;
ptr = new int;
*ptr = 345;
cout << ptr << endl;
```



Releasing Memory

- Every **new** must have a matching **delete**, so that memory is released back to the OS.

```
int *ptr = NULL;

ptr = new int;

...

delete ptr;
```



0x00000000 | 0

...

Your program stack

ptr | 0xaf556808

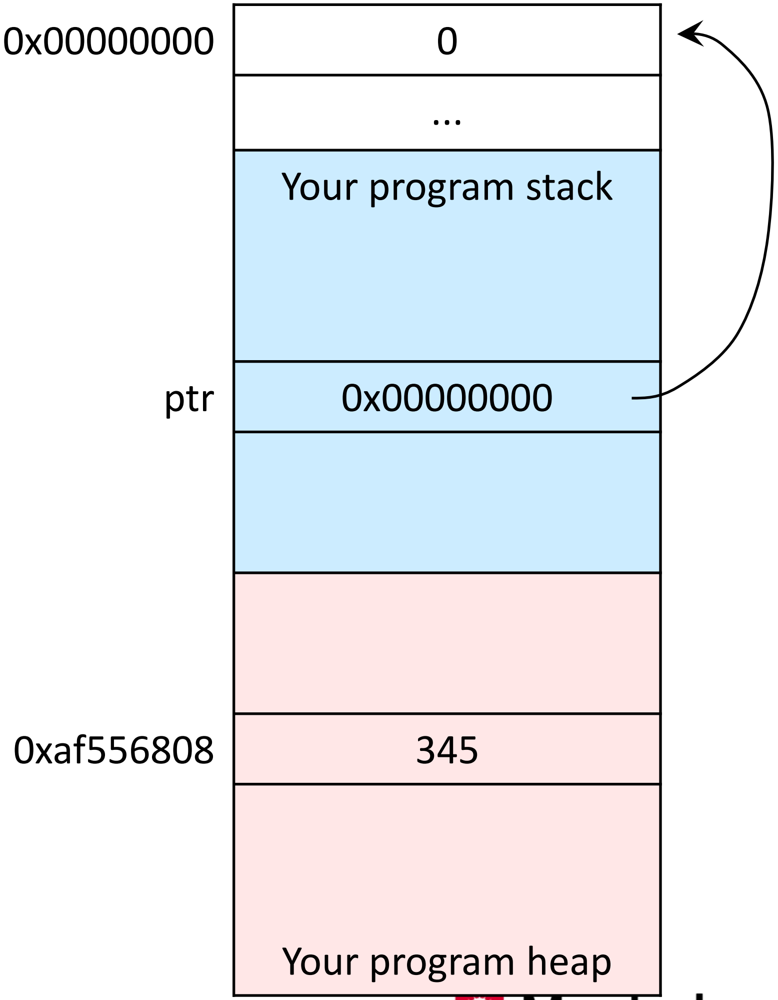
0xaf556808 | 345

Your program heap

Releasing Memory Safely

- Followed, of course, by reassigning the pointer to NULL, so that it does not point to memory over which it no longer should have control.

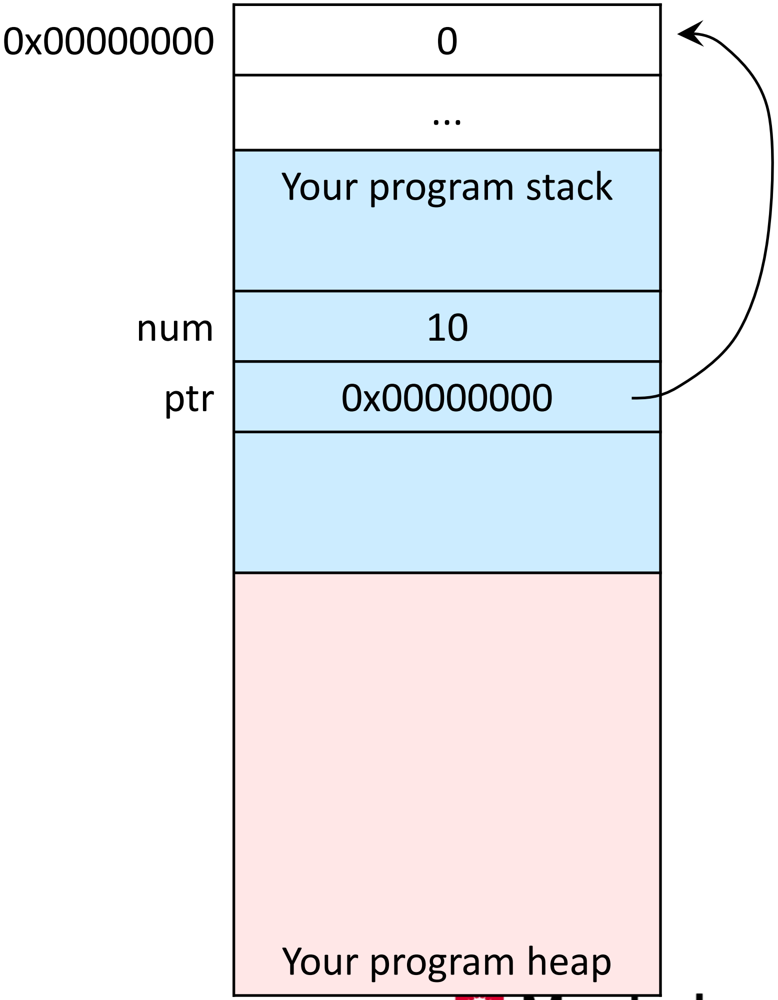
```
int *ptr = NULL;
ptr = new int;
...
delete ptr;
ptr = NULL; // to be safe
```



Pointing at Other Variables

- Pointers can also point at other variables:

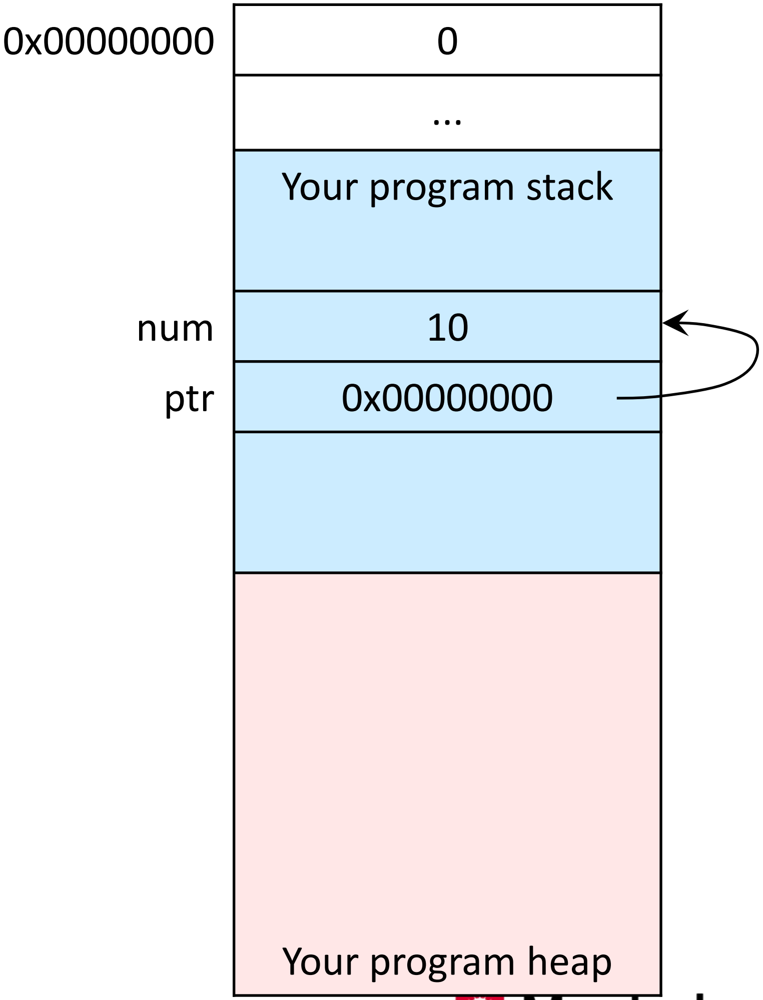
```
int *ptr = NULL;
int num = 10;
```



Pointing at Other Variables

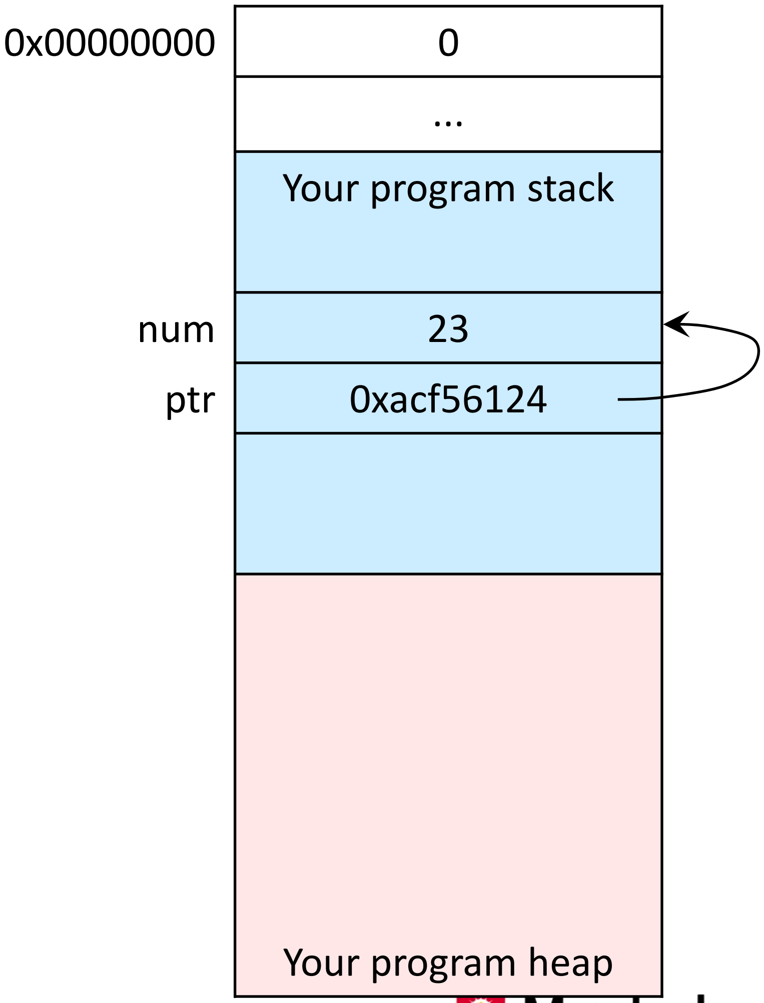
- Pointers can also point at other variables.

```
int *ptr = NULL;

int num = 10;

ptr = &num;
```



Pointing at Other Variables

- And change their value.



0x00000000

0

...

Your program stack

num

23

ptr

0xacf56124

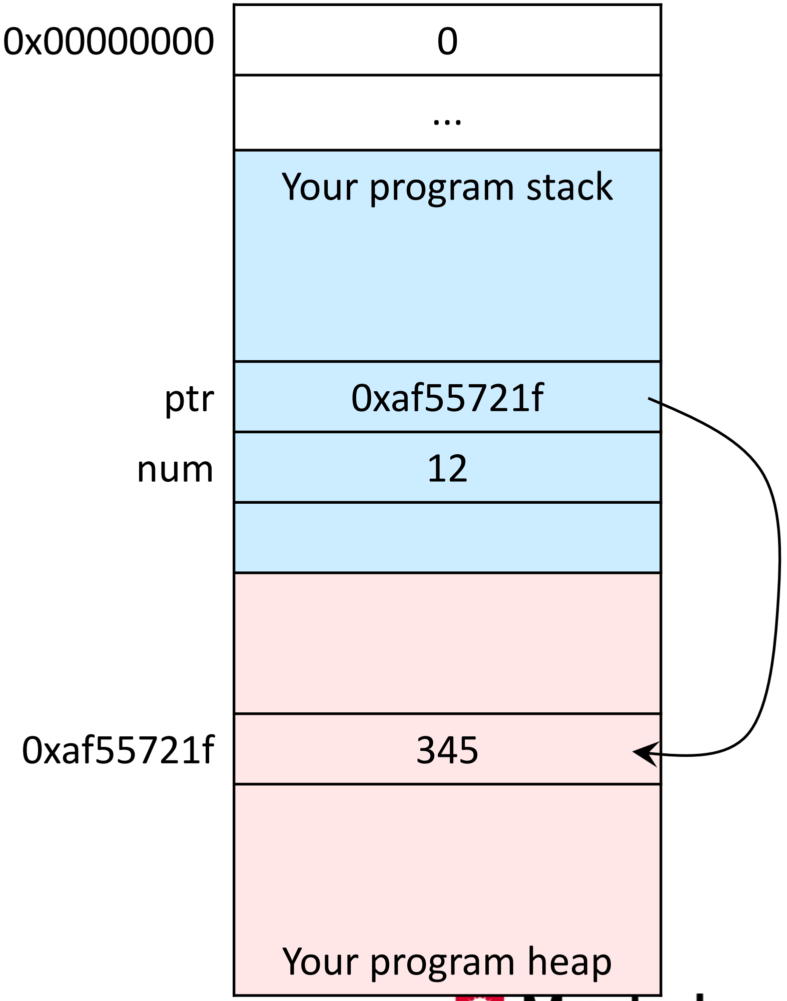
Your program heap

```
int *ptr = NULL;
int num = 10;
ptr = &num;
*ptr = 23;
```

Memory Leaks

- However care must be taken not to cause a memory leak.

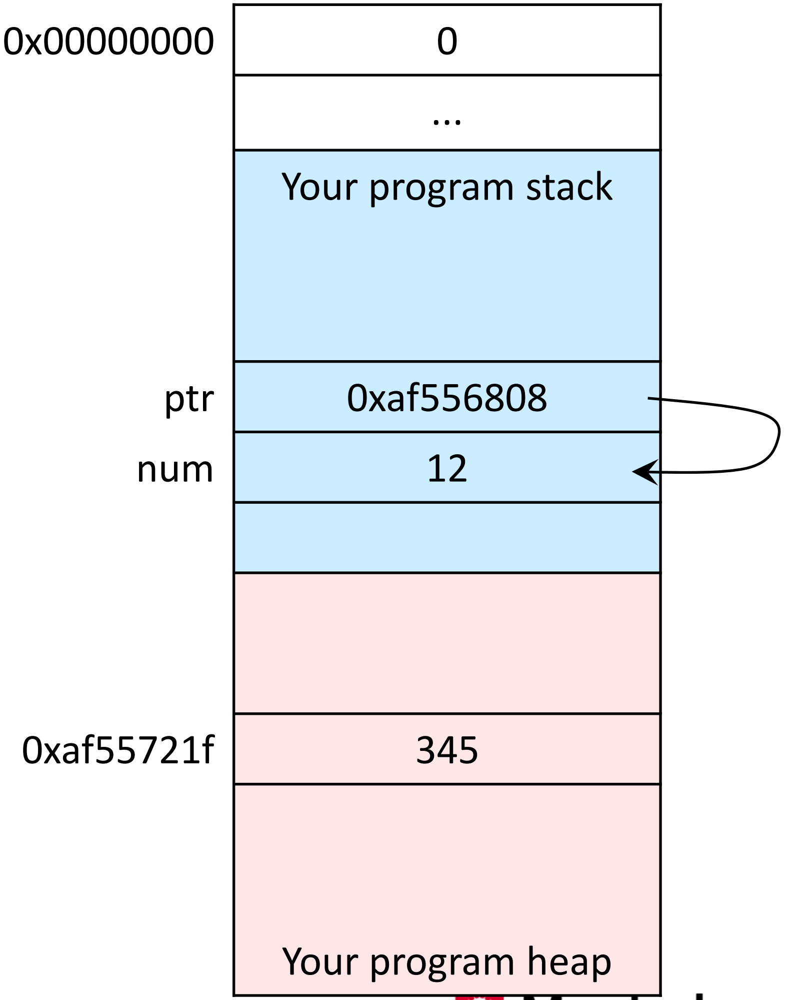
```
int num = 12;
int *ptr = new int;
*ptr = 345;
```



Memory Leaks

- However care must be taken not to cause a memory leak.

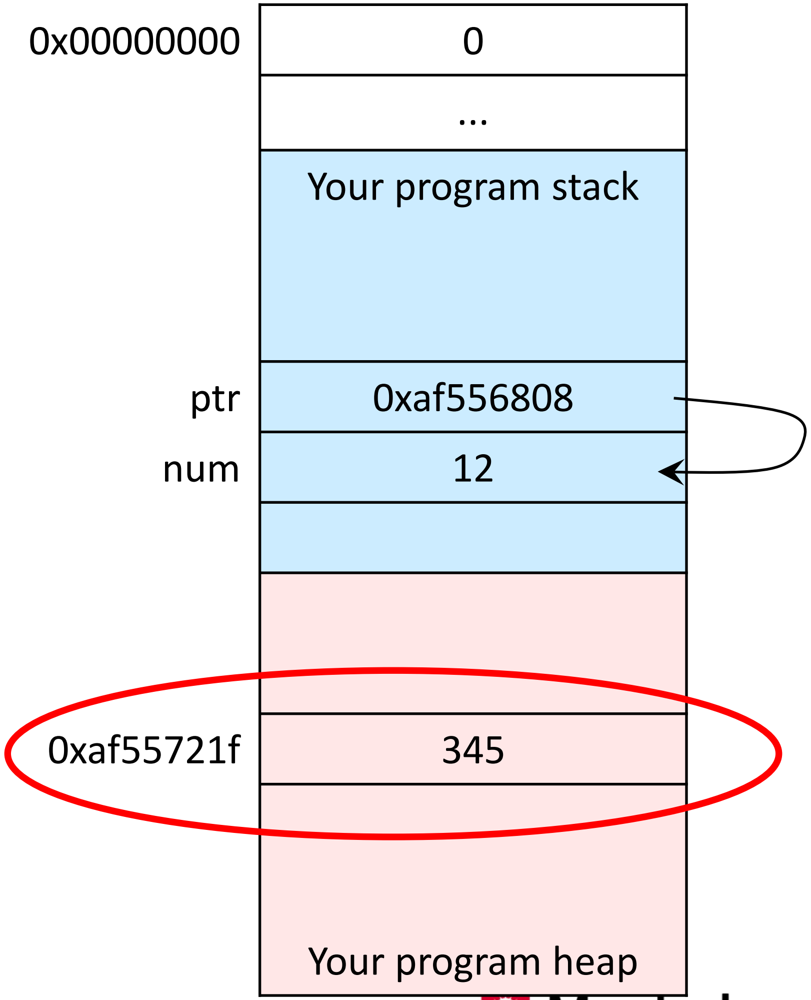
```
int num = 12;
int *ptr = new int;
*ptr = 345;
ptr = &num;
```



Memory Leaks

- The circled memory location is ‘lost’ until the program ends.

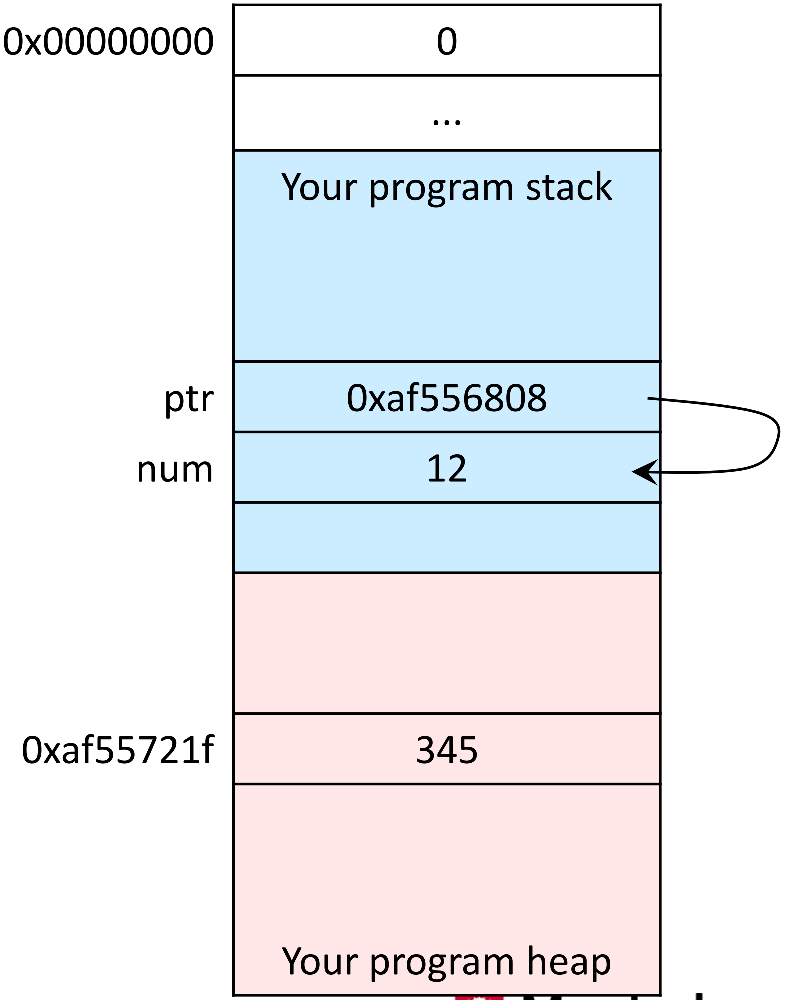
```
int num = 12;

int *ptr = new int;

*ptr = 345;

ptr = &num;
```



Avoiding Memory Leaks

- Of course, there should have been a delete between the last two lines of code.
- This releases the memory back to the OS again.

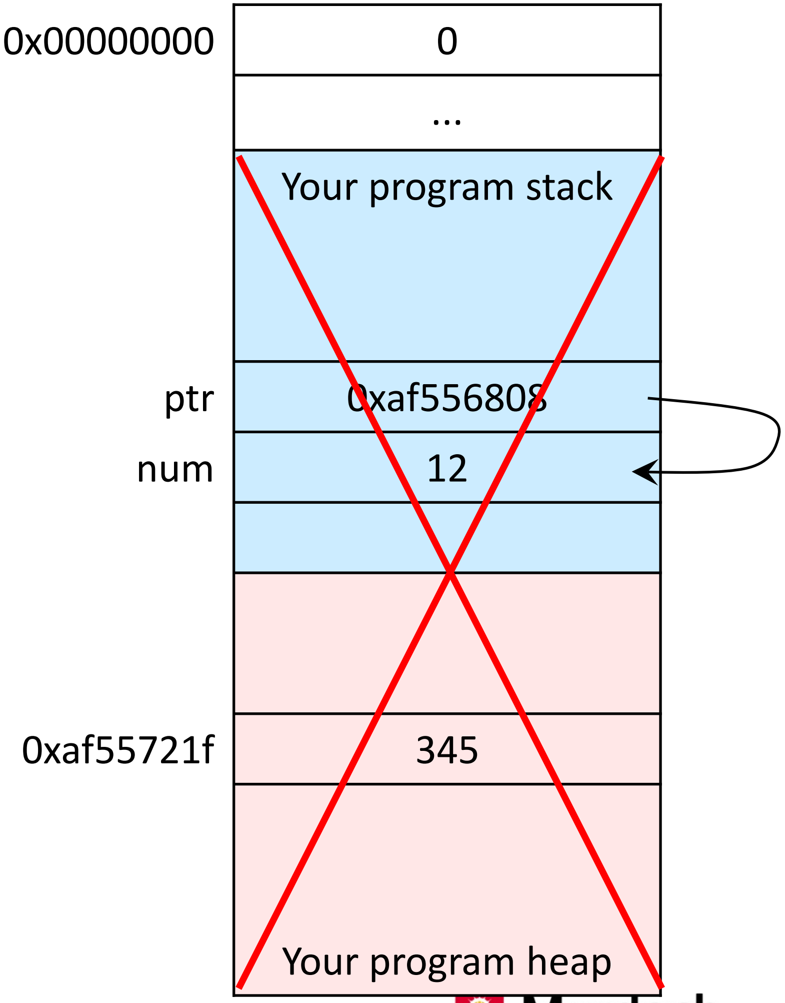
```
int num = 12;
int *ptr = new int;
*ptr = 345;
delete ptr;
ptr = &num;
```



Care with `delete`

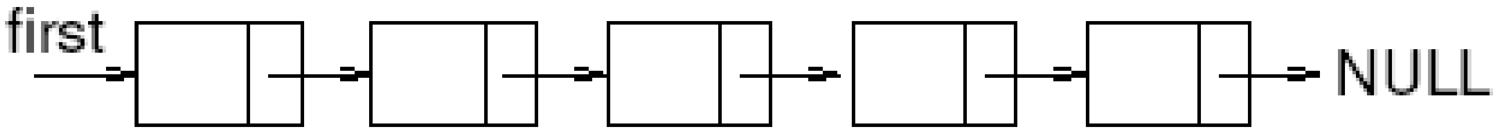
- But beware where you put it: trying to delete stack memory will cause trouble.

```
int num = 12;
int *ptr = new int;
*ptr = 345;
ptr = &num; //num not on heap
delete ptr; // oops
```



Uses of Pointers

- There are two main uses for pointers.
- The first is for array and string access.
- The second is where pointers are used to create lists or trees: data structures where the next piece of data can only be found by traversing a link from the last piece:



Malik: Chapter on Pointer - exercises

- 2. Given the declaration:

- int x;
- int *p;
- int *q;

- Mark the following statements as valid or invalid.
- If a statement is invalid, explain why:

- a. p = q;
- b. *p = 56;
- c. p = x;
- d. *p = *q;
- e. q = &x;
- f. *p = q;

- 3. What is the output of the following C++ code?

- int x;
- int y;
- int *p = &x;
- int *q = &y;

- *p = 35;
- *q = 98;
- *p = *q;

- cout « x « "   " « y « endl;
- cout « *p «   "   " « *q «   endl;

- 4. What is the output of the following C++ code?

- int x;
- int y;
- int *p = &x;
- int *q = &y;
- x = 35;
- y = 46;
- p = q;
- *p = 18;
- cout « x « "   " « y « endl;
- cout « *p «   "   " « *q « endl;

Readings

- Textbook (by Malik): Chapter on Pointers, Classes, .. etc. See subsections on Pointer Data Type and Pointer Variables; Address of operator; Dereferencing operator. A different edition may have a different chapter number and pages.
- 101 Coding Standards: **Rules 51 and 52**. It is one of the important references (see unit outline) we use in the unit. See 101 C++ Coding Standards **online** resource. [1]
- Watch the videos on pointers “Video Lecture on Pointers.htm” Function pointers are covered when we cover the tree data structure later on.
- Find out about “**RAII**”. Why is the RAII concept so important that you should not violate it?

Murdoch UNIVERSITY

Videos

- Pointers - Stanford University

  https://www.youtube.com/watch?v=H4MQXBF6FN4

- Bits and bytes; floating point representation - Stanford University

  https://www.youtube.com/watch?v=jTSvthW34GU

- How pointers get used; usage of void pointers

  https://www.youtube.com/watch?v=_eR4rxnM7Lc

Murdoch
UNIVERSITY

Data Structures and Abstractions

Parameters

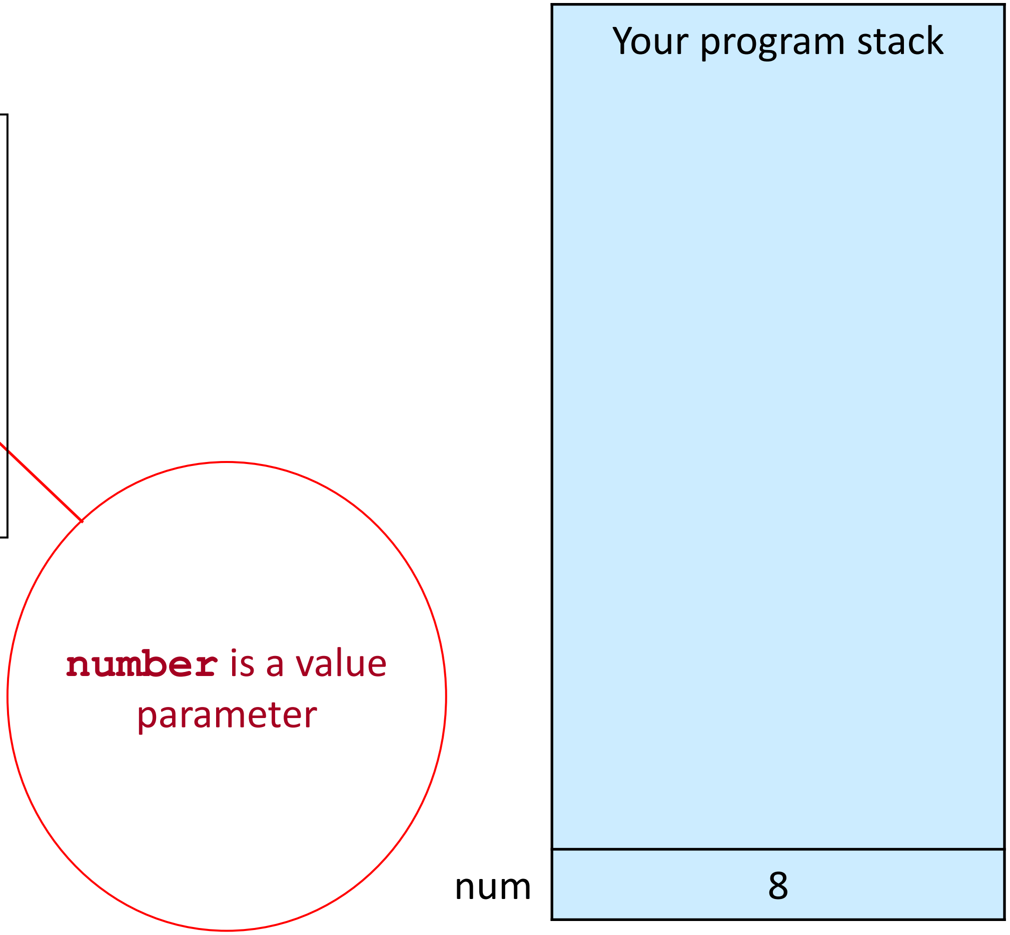
Lecture 6

Parameters

- Parameters can be passed in four main ways.
  - by value
  - by reference
  - by constant reference
  - by pointer

Value Parameters

```
void Func1 (int number);
...


int num = 8;
```



number is a value parameter

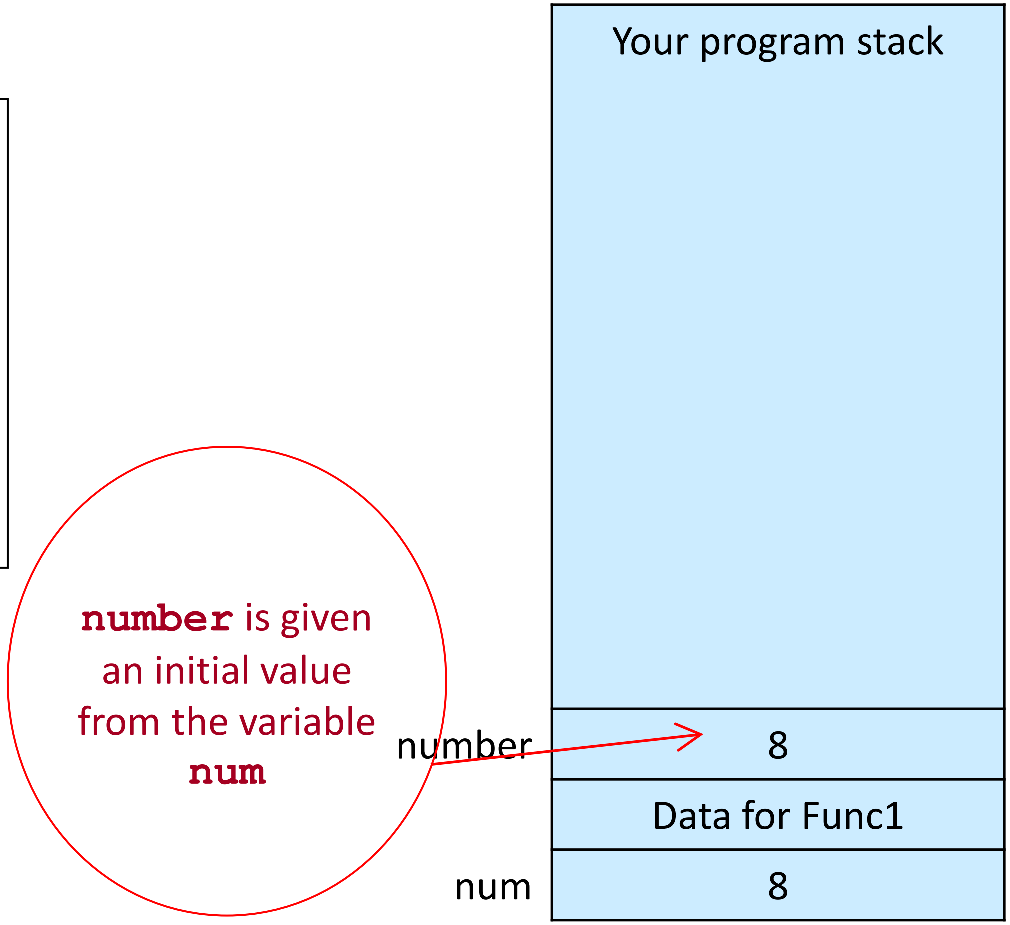
Your program stack

num

8

Value Parameters

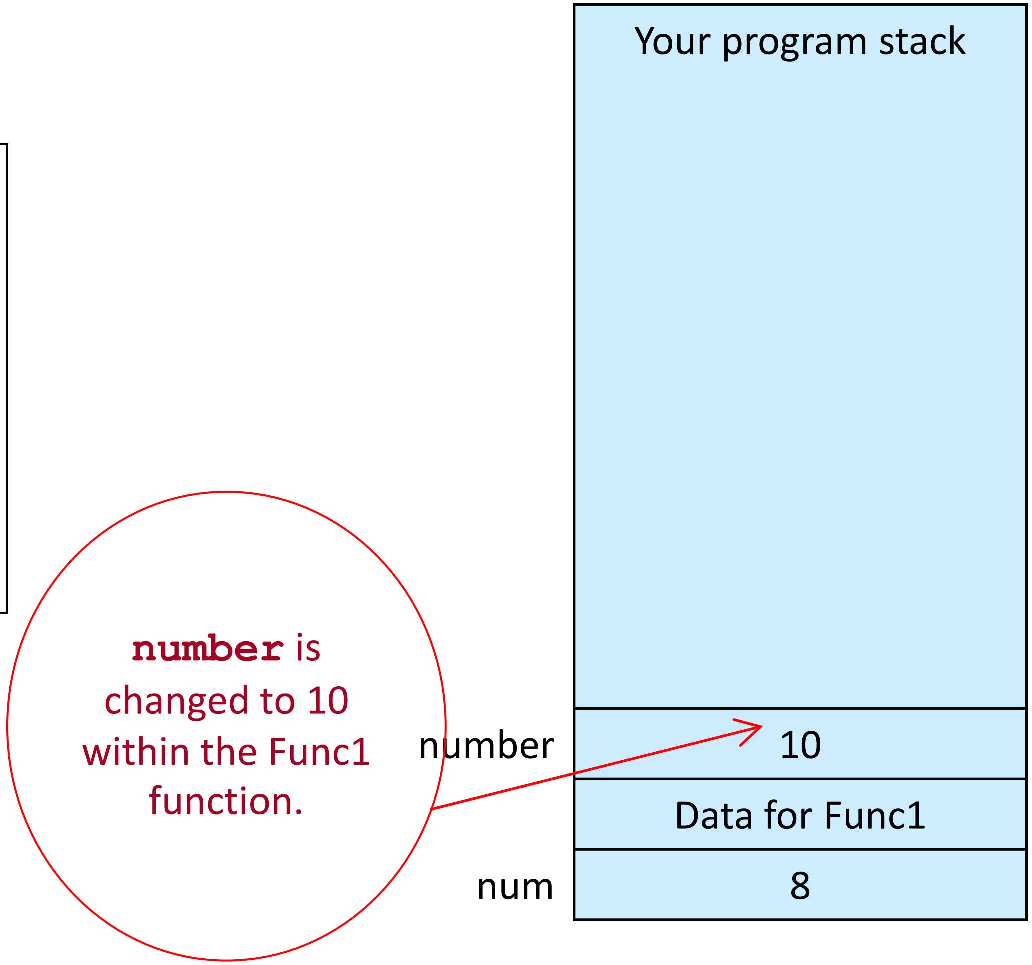
```
void Func1 (int number);
...


int num = 8;
Func1 (num);
```



number is given an initial value from the variable **num**

Value Parameters

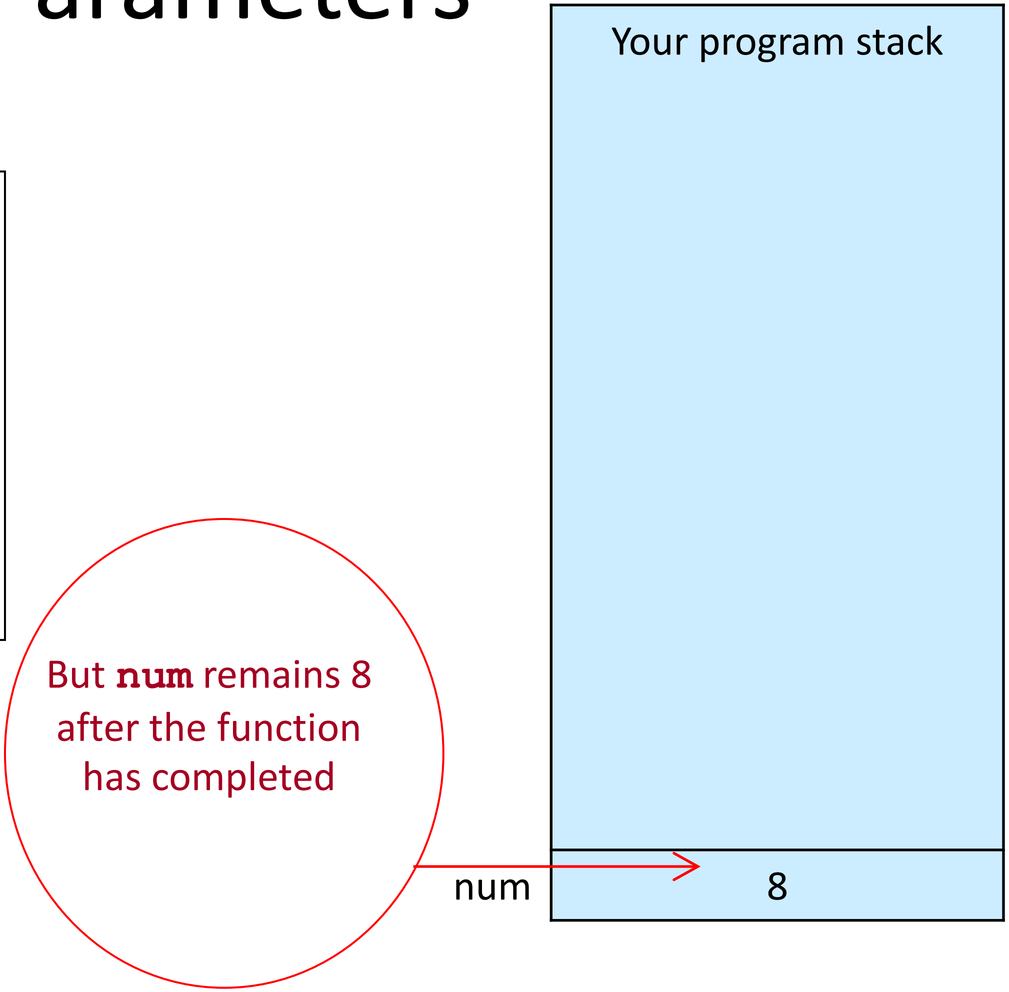
```
void Func1 (int number);
...


int num = 8;
Func1 (num);
```



number is changed to 10 within the Func1 function.

Value Parameters

```
void Func1 (int number);
...


int num = 8;
Func1 (num);
```

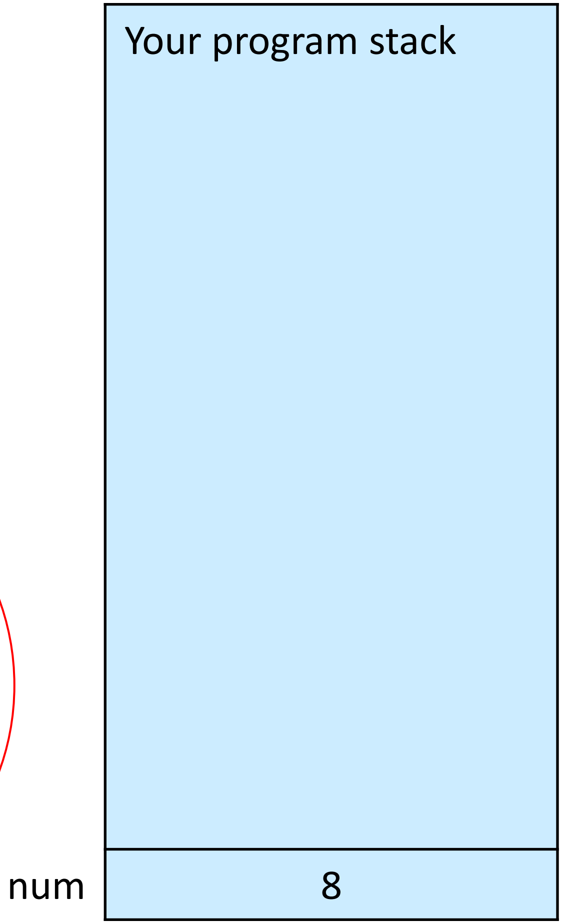


But **num** remains 8 after the function has completed

Reference Parameters

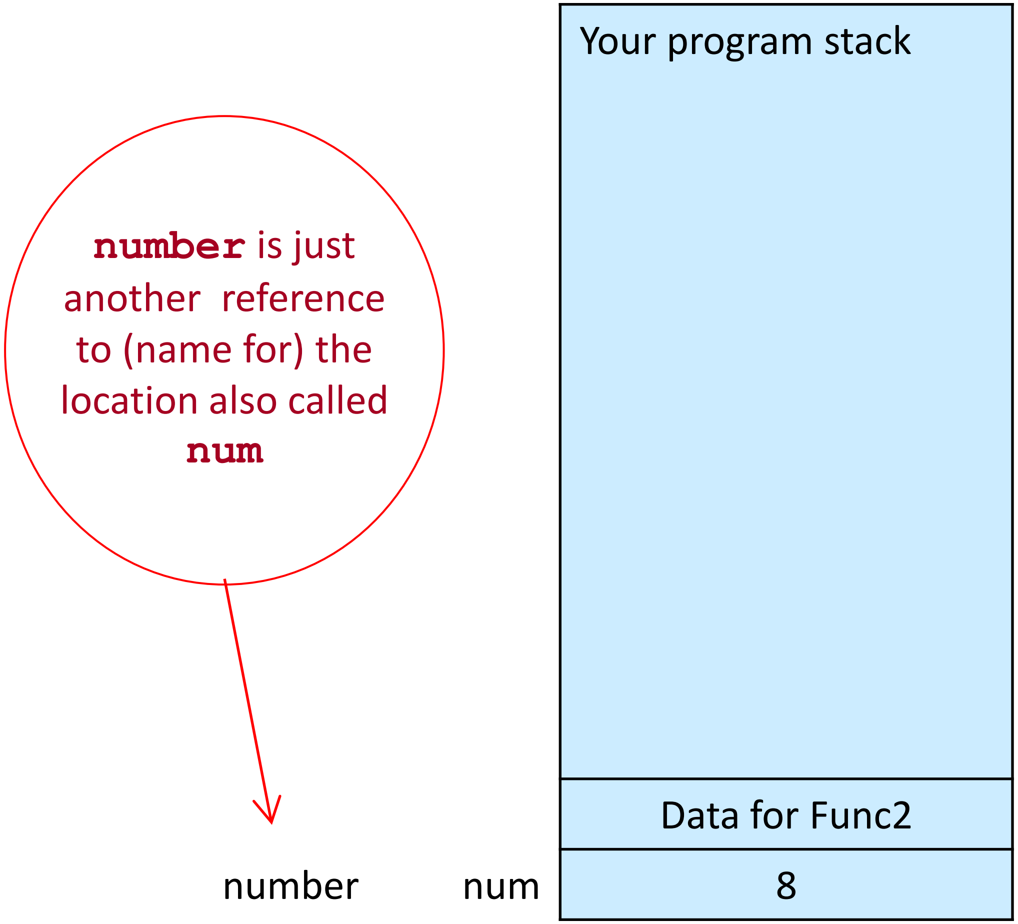
```
void Func2 (int &number); // [1]
...


int num = 8;
```

number is a reference parameter i.e. another name for something



Reference Parameters

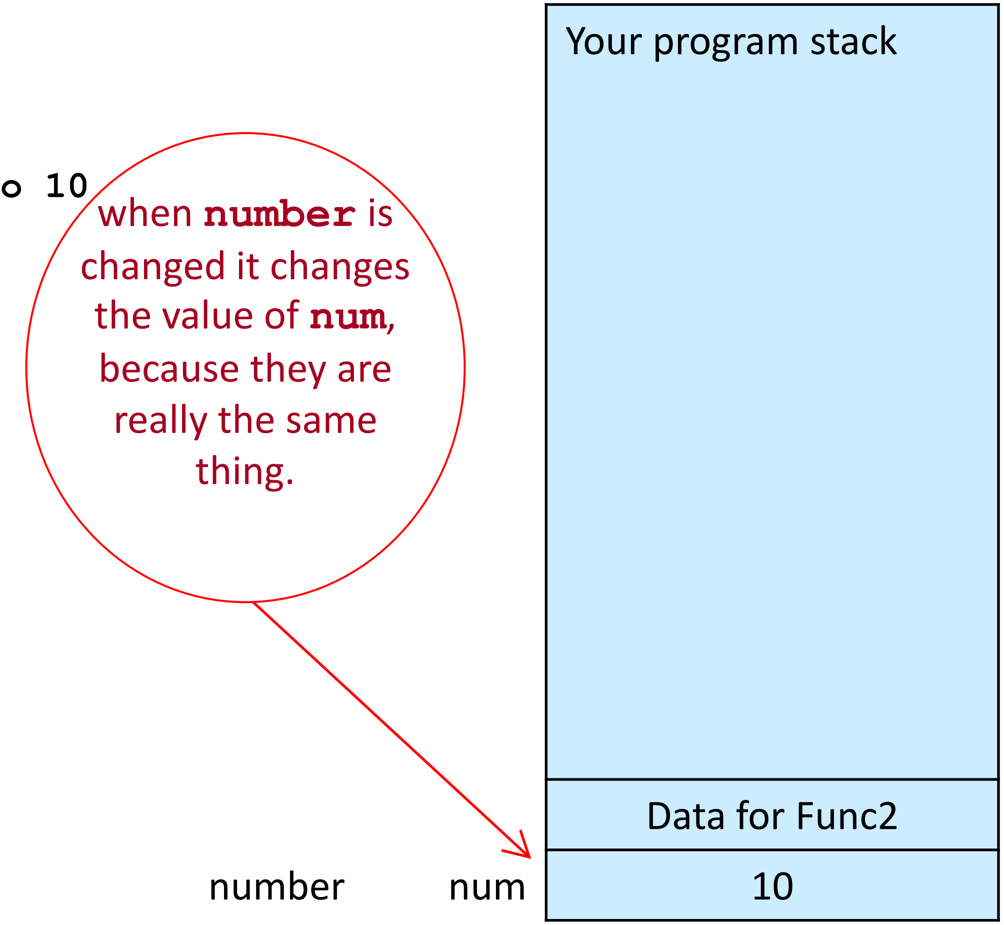
```
void Func2 (int &number);
...


int num = 8;
Func2 (num);
```



number is just another reference to (name for) the location also called **num**

Reference Parameters

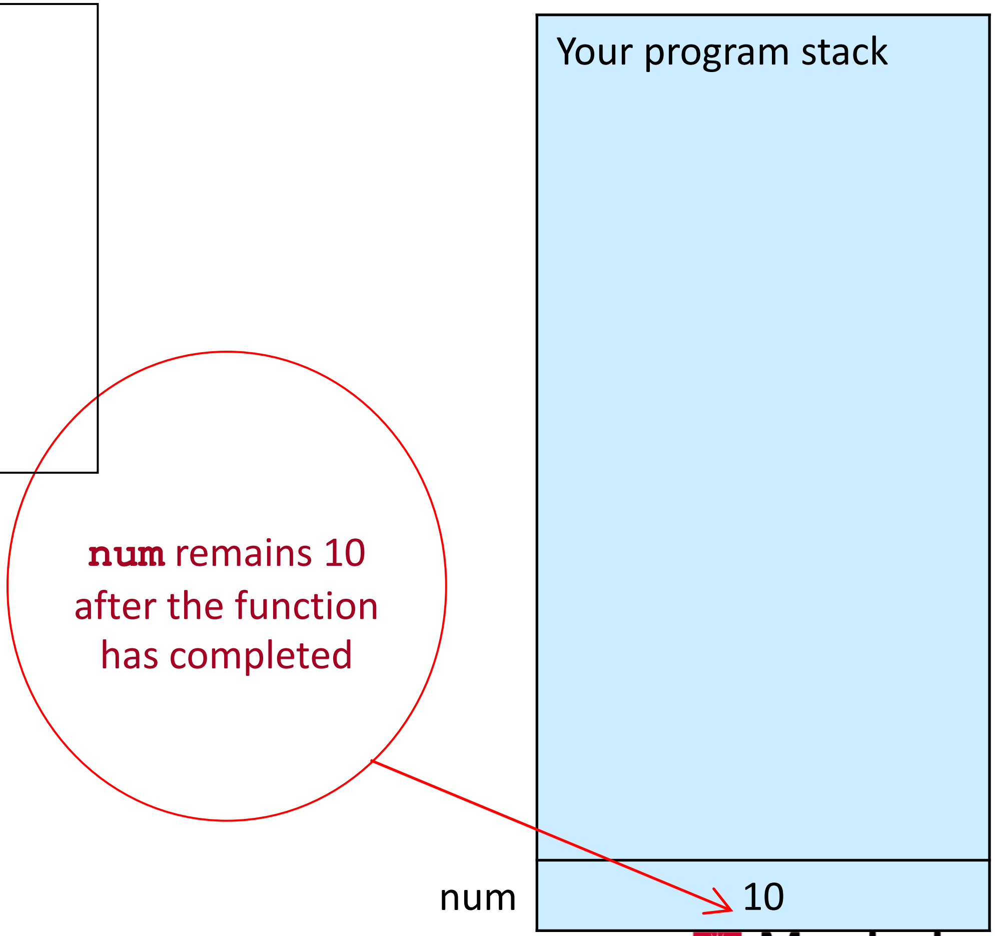
```
void Func2 (int &number){
 number = 10; //changes number to 10
}

//--------------------
int num = 8;
Func2 (num);
```



when **number** is changed it changes the value of **num**, because they are really the same thing.

Reference Parameters

```
void Func2 (int &number);
...


int num = 8;
Func2 (num);
```

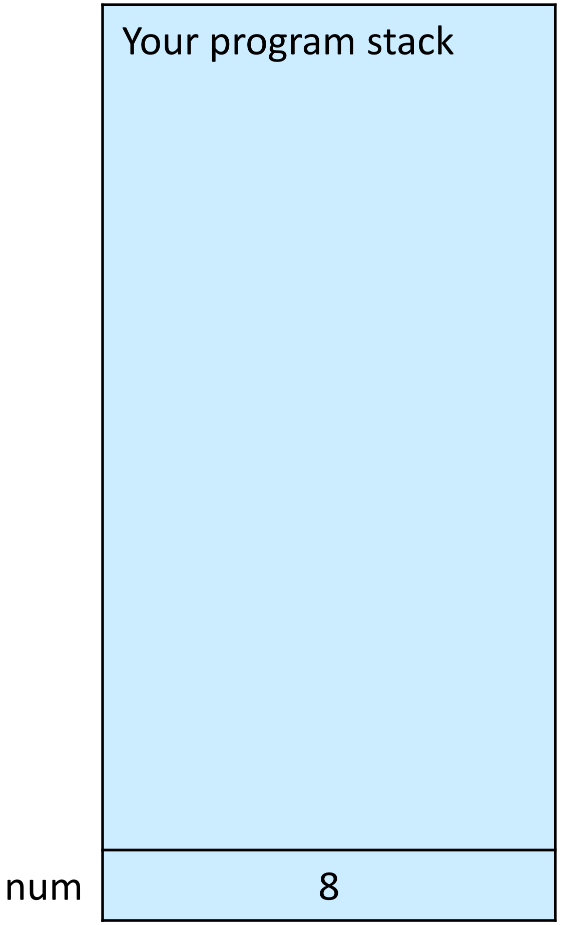


num remains 10 after the function has completed

Constant Reference Parameters

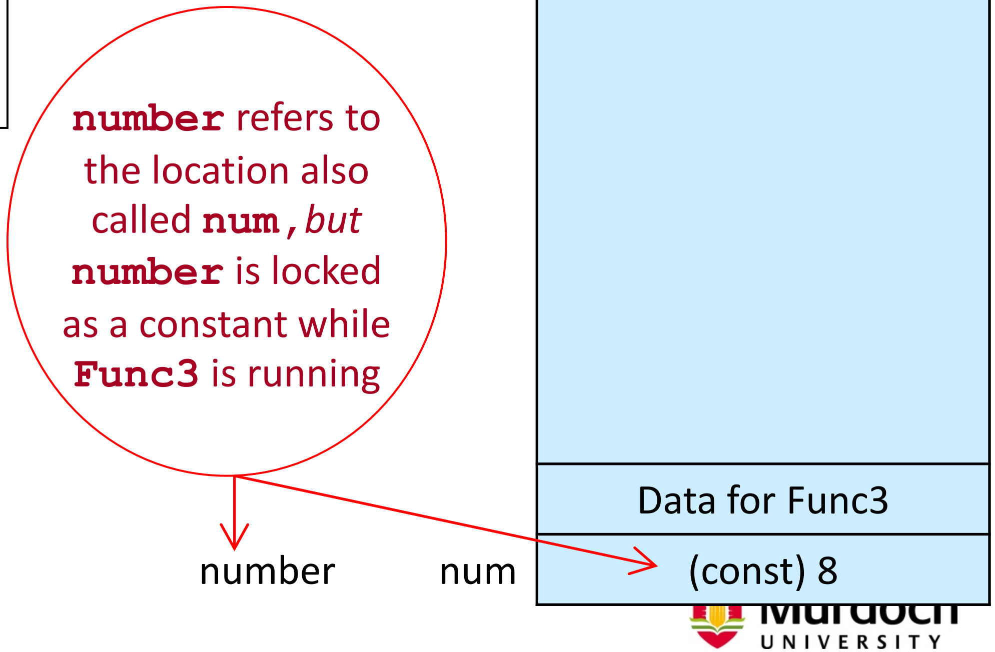
```
void Func3 (const int &number);
...


int num = 8;
```

number is a *constant* reference parameter



Constant Reference Parameters

```
void Func3 (const int &number);
...


int num = 8;
Func3 (num);
```



number refers to the location also called **num**, *but* **number** is locked as a constant while **Func3** is running

Your program stack

Data for Func3

number

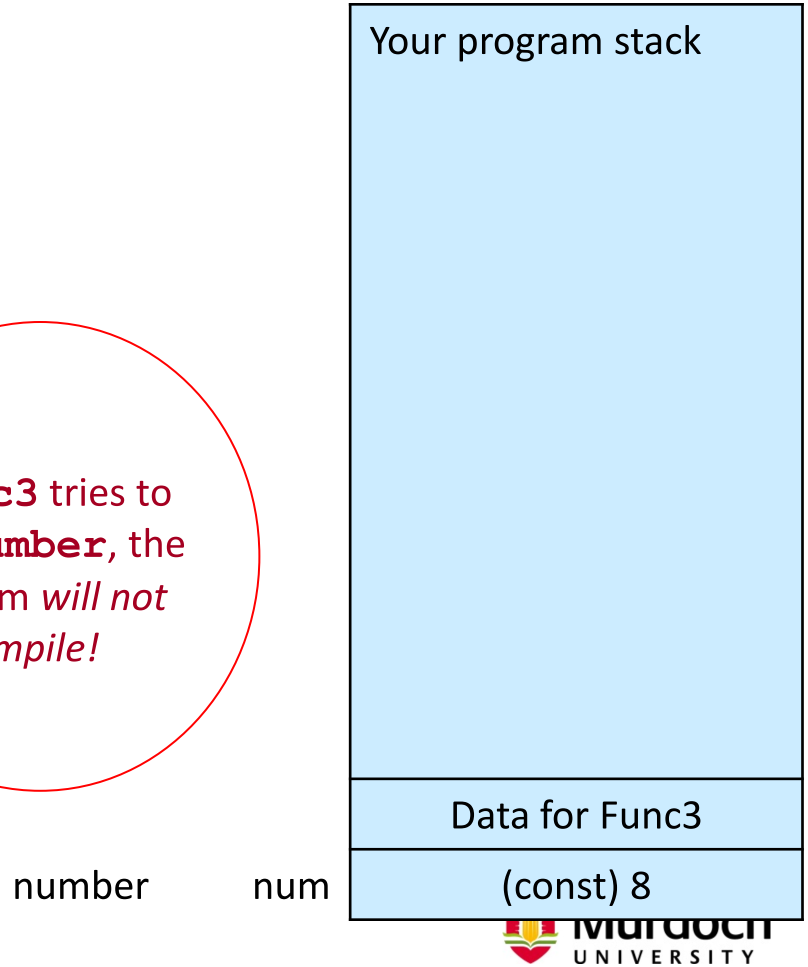
num

(const) 8

Constant Reference Parameters

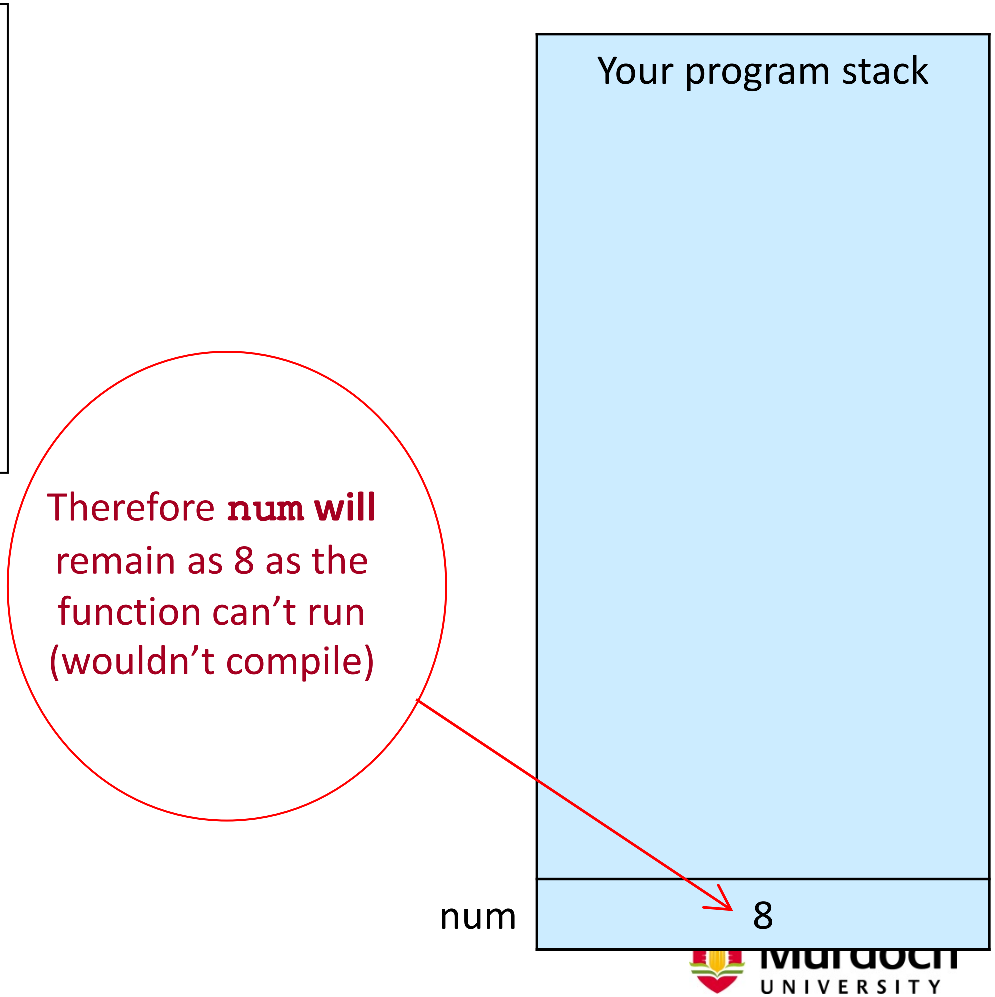
```
void Func3 (const int &number);
...// attempt to change number
...// will not compile - good

int num = 8;
Func3 (num);
```

If **Func3** tries to alter **number**, the program *will not compile!*



Constant Reference Parameters

```
void Func3 (const int &number);
...


int num = 8;
Func3 (num);
```

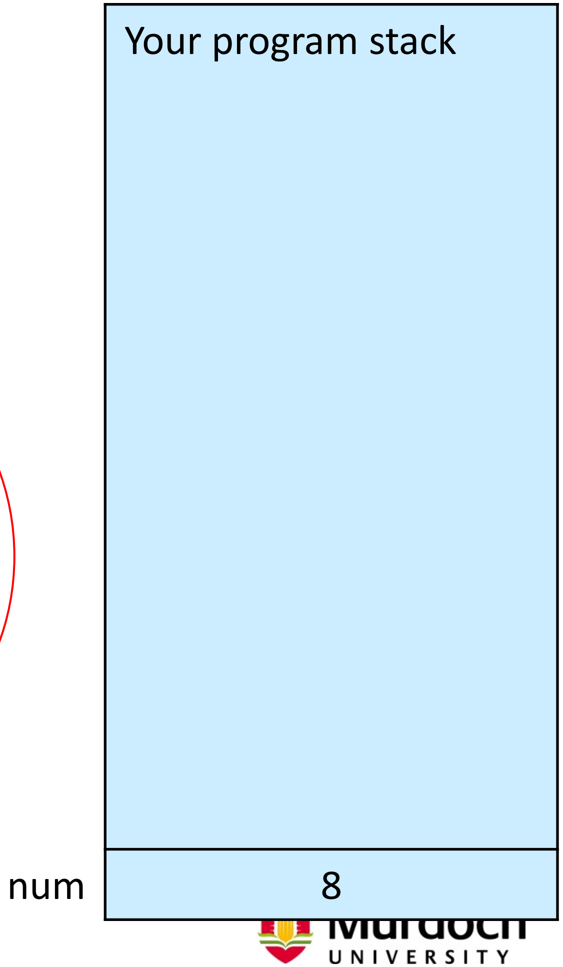


Therefore **num will** remain as 8 as the function can’t run (wouldn’t compile)

Pointer Parameters

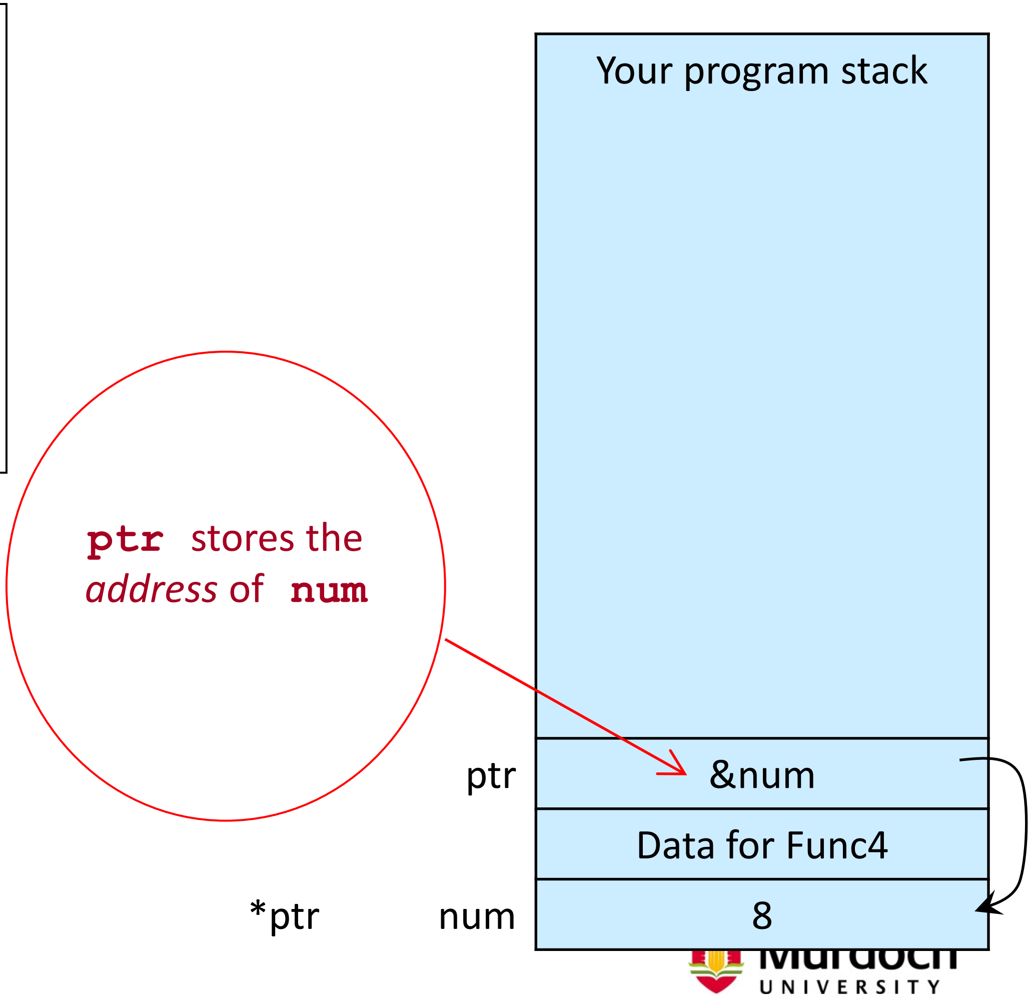
```
void Func4 (int *ptr);
...


int num = 8;
```

`ptr` is a pointer parameter, therefore within **`Func4, ptr`** is a pointer *not* an integer



Pointer Parameters

```
void Func4 (int *ptr);
...



int num = 8;
Func4 (&num);
```



ptr stores the *address* of **num**

Your program stack

ptr &num

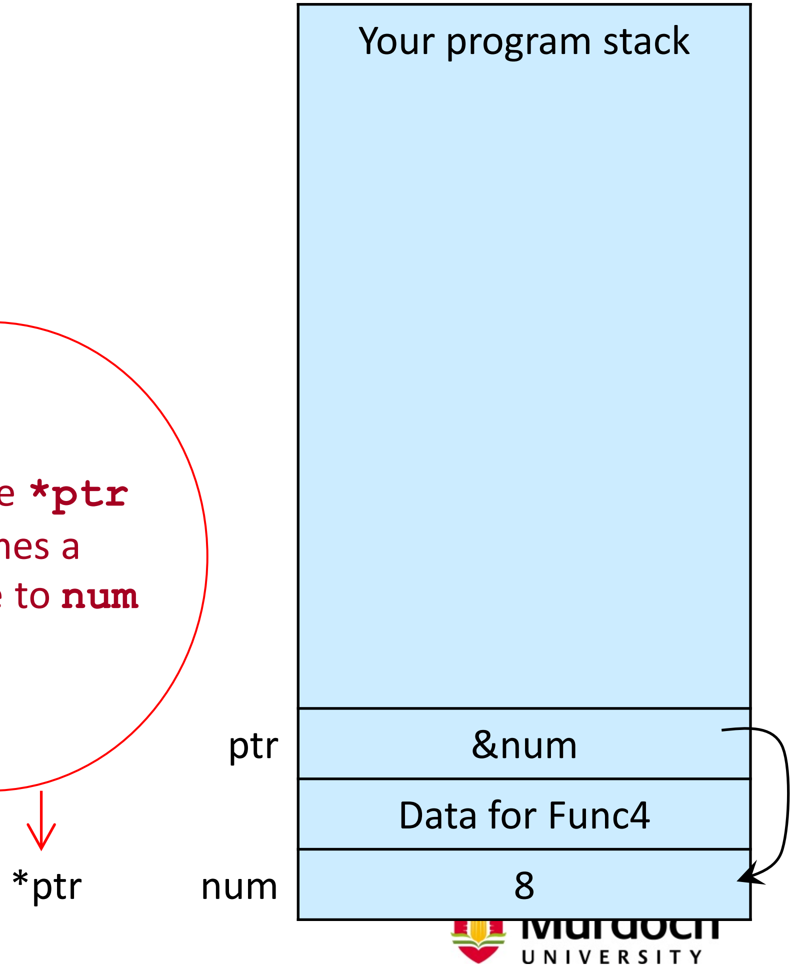
Data for Func4

*ptr num 8

Pointer Parameters

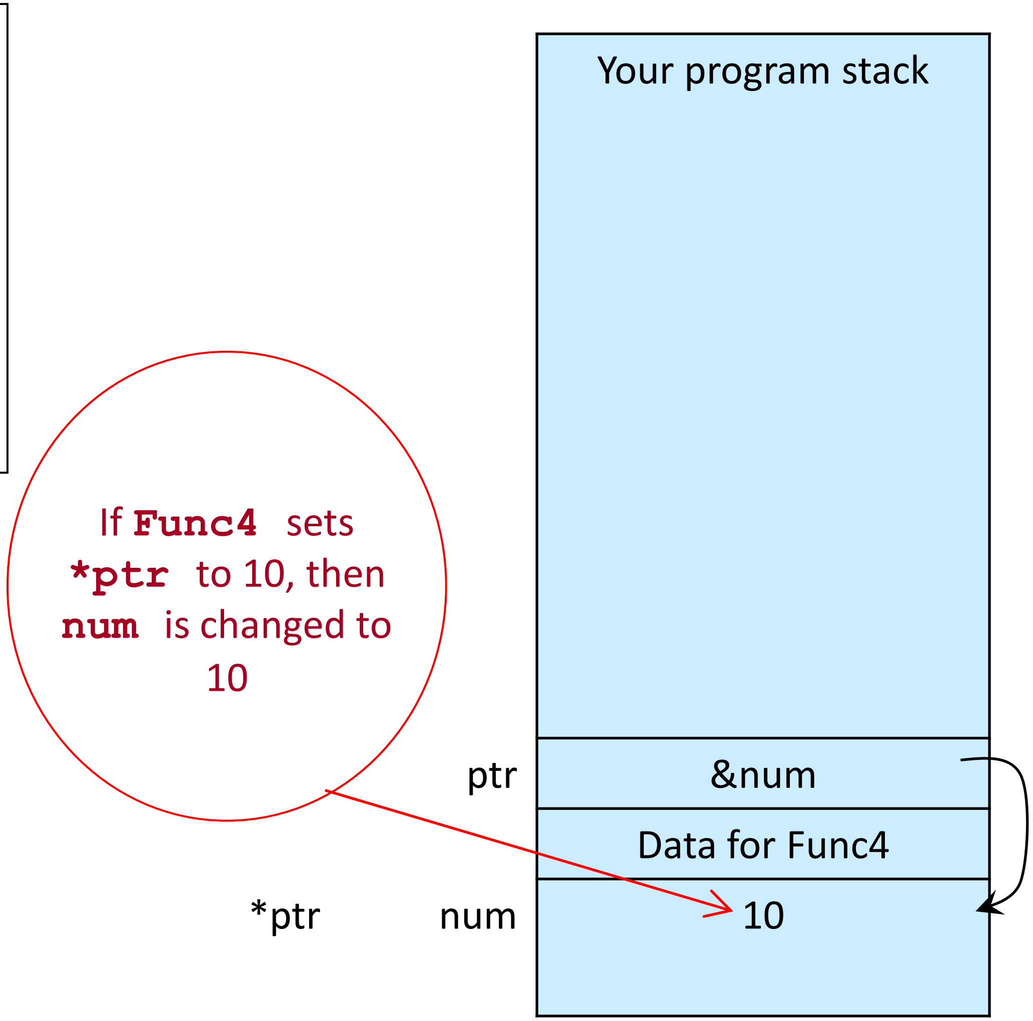
```
void Func4 (int *ptr);
...



int num = 8;
Func4 (&num);
```

Therefore ***ptr** becomes a reference to **num**



Pointer Parameters

```
void Func4 (int *ptr);
...


int num = 8;
Func4 (&num);
```



If **`Func4`** sets **`*ptr`** to 10, then **`num`** is changed to 10

Your program stack

ptr

&num

Data for Func4

*ptr

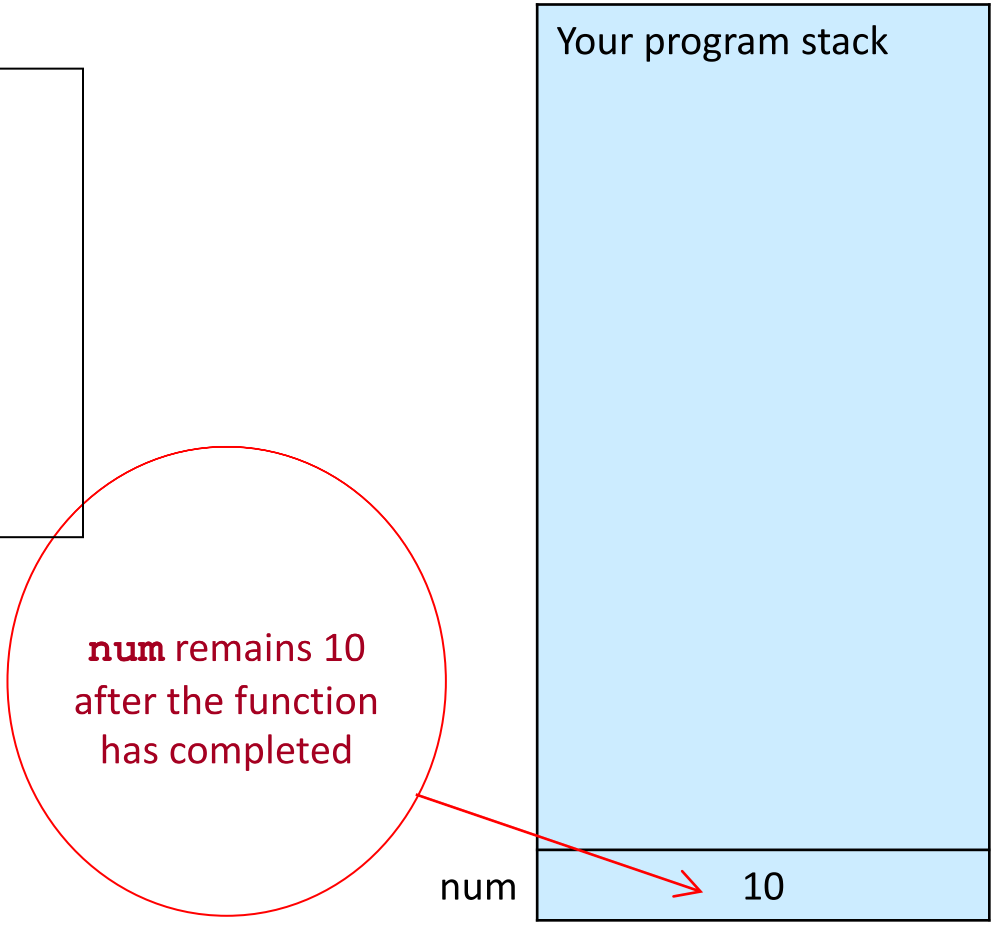
num

10

Pointer Parameters

```
void Func4 (int *ptr);
...


int num = 8;
Func4 (&num);
```

- For further work (not necessary for this unit), find out about smart pointers, auto_ptr, and Opaque pointer. [1]



num remains 10 after the function has completed

Pointers and References

- Find out about the following:
- It is possible to declare a pointer with no initial value? Is it possible to declare a reference which does not contain an initial value?
- A pointer variable can be changed to point to something else. Can this be done with a reference?
- A pointer can be set to contain the NULL (or **nullptr**) value. Can you make a reference NULL (or **nullptr**)? [1]
- Can you do pointer like arithmetic on references?

Murdoch UNIVERSITY

Readings

- Chapter(s) on User-Defined Functions, section on Value Returning Functions; section on Reference variables as parameters.

- If you are using another edition of the textbook, look up the chapter title and section number in the contents page.

Murdoch
UNIVERSITY

Data Structures and Abstractions

Arrays & Records

Lecture 7

- The revision exercise must be completed before starting on Topic 3.
  - A revision of arrays from the revision exercise is needed for this topic.

Arrays in C++

- Arrays in C++ are not a predefined type, they have to be defined by the programmer: [1]

```
const int SIZE = 10; [2]
...
typedef int ArrayType[SIZE];
...
ArrayType numbers;
```

- This declares an array of 10 integer point numbers with indexes from 0 to 9.
- Processing of arrays is the same in C++ as in C or Java, with [ ] giving access to elements.

Within Memory

- Within memory, the compiler has generated code to allocate 10 contiguous slots on the stack, each large enough to store one floating point number.

- The variable ‘**`numbers`**’, is in fact a “*pointer*” to the first of the 10 memory locations. But you cannot put a new address into it. What are the consequences if you were able to do this?

- This can be seen if you output the array variable without an index:

  ```
  cout << hex << numbers << endl
  ```

  would give output something like: **`0x23ab34`** [1]

  which is the address in RAM of the 0th location out of the 10 integers.

- However,

  ```
  cout << numbers[0] << endl;
  ```

  will output the contents of the 0th memory location.

```
// Arrays1.cpp
// Demonstrating array use
// Version: 01, Nicola Ritter
// Version: 02, SMR
//------------------------------------------------------

#include <iostream>

using namespace std;

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

const int SIZE = 10;
const int END = -1;

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

// Define a new type that is an array of SIZE integers
typedef int IntArray[SIZE];
```

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

// Returns the number of input values
//This is called a forward declaration
// Both the lines shown next are examples of forward declaration
int Input (IntArray &array);
void Output (int size, const IntArray &array);
// Why const?

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

int main()
{
        IntArray array;

        int size = Input (array);  // size returns how many
        Output (size, array);  // this is a procedure

        return 0;
}

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

```
int Input (IntArray &array)
{
        cout << "Enter up to " << SIZE
            << " positive integers, pressing <Enter> after each integer"
            << endl << "Use " << END << " to end input."
            << endl;

        int num;
        int index;
        cout << "Enter integer: ";
        cin >> num;
        for (index = 0; index < SIZE && num != END; index++)
        { // is the for loop the correct loop to use for this situation?
            array[index] = num;
            cout << "Enter integer: ";
            cin >> num;
        }

        return index;
}
```

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

void Output (int size, const IntArray &array)
{
     cout << endl << "Array at address " << array << " contains:"
      << endl;
     for (int index = 0; index < size; index++)
     {
        cout << index << ") " << array[index]
           << " at memory location " << &(array[index])
           << endl;
     }
     cout << endl;
}

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

Two Dimensional Arrays

- Arrays of more than one dimension are defined in a similar way:

```
float
twoDimArray[ROWS][COLS];
```

- Access is as per normal:

```
twoDimArray[row][col];
```

Dynamic Arrays

- Arrays can be declared dynamically as well:

  ```
  int *array = new int[size];
  ```

  declares an array of ‘size’ integers.

- As it is a dynamic declaration, the integers are on the *heap* rather than the stack. Don’t forget to delete them afterwards when you no longer need them.

- If there is not enough memory (new fails), array is set to NULL (or the preferred **nullptr**).

- However use of the array does not change unless it was set to NULL. What would happen if it was set to NULL?

Murdoch UNIVERSITY

```
// TwoDimArray.cpp
// Program designed to show the declaration and use of a dynamically
//   created two dimensional array
//
// Version
// 001 First Attempt, Nicola Ritter
// modified smr
//---------------------------------------------------------------

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

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

const int COL_WIDTH = 14;

//------------------Forward declarations---------------------------

void GetSizes (int &rows, int &cols);
float **CreateArray (int rows, int cols);
void Output (int rows, int cols, float **array);
void Initialise (int rows, int cols, float **array);
void Delete (int rows, float **&array);

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

```
int main ()
{
    // Declare a pointer to an array of pointers
    float **array = NULL;
    int rows, cols;

    GetSizes (rows, cols);

    // Get memory for the 2 dim array
    array = CreateArray (rows, cols);

    if (array != NULL)
    {
            // Set the table to contain 0s
            Initialise (rows, cols, array);

            // Output to check
            Output (rows, cols, array);

            Delete (rows, array); // this is a user defined delete, not C++ delete [1]
    }

    // put in an else part which says that memory for array could not be created.

    return 0;
}
```

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

void GetSizes (int &rows, int &cols)
{
    cout << "How many rows do you want? ";
    cin >> rows;
    cout << "How many columns do you want? ";
    cin >> cols;
    cout << endl;
}

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

```
float **CreateArray (int rows, int cols)
{
    // Get an array of pointers for the rows
    float **array = new float* [rows]; //draw this structure

    if (array != NULL) // how about an else
    {
        // Now get a row for each of these pointers
        for (int index = 0; index < rows; index++)
        {
            array[index] = new float[cols];
        }

        // Check that allocation occurred
        if (array[rows-1] == NULL)
        {
            Delete (rows, array);
        }
    }

    return array;
}
```

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

void Output (int rows, int cols, float **array)
{
          cout << "The array values are:" << endl;
          for (int row = 0; row < rows; row++)
          {
                    for (int col = 0; col < cols; col++)
                    {
                              cout << setw(COL_WIDTH) << array[row][col];
                    }
                    cout << endl;
          }
          cout << endl;
}

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

```
void Initialise (int rows, int cols, float **array)
{
     for (int row = 0; row < rows; row++)
     {
          for (int col = 0; col < cols; col++)
          {
            array[row][col] = 0;
          }
     }
}

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

```
void Delete (int rows, float**  &array)
{
        for (int row = 0; row < rows; row++)
        {
                if (array[row] != NULL)
                {
                    delete [] array[row]; // draw a diagram
                }
        }

        delete [] array;
        array = NULL;
}

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

Input Testing

- When testing a program it can become onerous to keep having to type in data.
- For this reason it is common to store test data in a file and run the program from the command line, redirecting data into the program:

```
aprogram.exe < data.txt
```

- The data stored in **`data.txt`** will be read into the program *as if it had been entered from the keyboard.* [1]
- Note that this is *not* the same as file input, it is a way to replicate keyboard typing, without having to do it again and again.
- Note that the output data can also be redirected to file:

```
aprogram.exe < data.txt > data.out
```

```
•       // redirect.cpp
•       // Testing redirected input from a file

•       #include <iostream>

•       using namespace std;

•       int main ()
•       {
•               int number;

•               do
•               {
•                               cin >> number;
•                               cout << number << endl;
•
•       [1]     } while (cin.good());

•               return 0;
•       }
```

Stops the loop when the redirected file reaches eof

Input File (data.txt)

```
2
45
67
8
912
```

User types in [2]

```
redirect < data.txt
```

Screen Output

```
2
45
67
8
912
```

Records

- Records (grouped data) are produced in C and hence C++ using the **`struct`** type:

```
typedef struct Person
{
 char firstName[SIZE];
 char surname [SIZE];
 int  age;
};
```

- A variable of this type is then declared in the same way as any other type:

```
Person person; [1]
```

- Access to parts of the person is done using the dot operator:

```
cout << person.firstName << " "
     << person.surname << " "
     << person.age << endl;
```

An Array of Records

- Similarly, we could declare an array of records as:

```
Person people[ARRAY_SIZE];
```

- And access an individual part of a single record:

```
people[index].firstname
```

Readings

- Textbook by Malik
  - Chapter on Arrays and Strings
  - Chapter on Records

Murdoch
UNIVERSITY

Data Structures and Abstractions

C Character Functions and Strings

Lecture 8

C Character Functions [1]

- There are many useful character functions available in C++.
- They are actually standard C functions.
- To use them use must include the header file **`<cctype>`**.
- Note that the original C header file was **`<ctype.h>:`** within C++ the ".h": is removed and a "c" added in front of the file name.
- Most of these functions query the classification of the character.
- Because they are C functions, not C++ functions, they return 0 for false and 1 for true.

Useful Character Functions

`isalpha(ch)`	Is it an alphabetic character?
`isascii(ch)`	Is it an ASCII character?
`isblank(ch)`	Is it a blank character?
`isdigit(ch)`	Is it a digit (0..9)?
`islower(ch)`	Is it a lowercase alphabetic character?
`isupper(ch)`	Is it an uppercase alphabetic character?
`ispunct(ch)`	Is it punctuation? In this sense punctuation is defined as any printable character that is not a space or an alphanumeric character.
`isspace(ch)`	Is it a space?
`isxdigit(ch)`	Is it a hexadecimal digit (a..f, A..F,0..9)?

Useful Character Functions cont.

There are two other useful functions:

toupper (ch)	If ‘ch’ is a lowercase character, return the uppercase character, or else return it unchanged.	**char ch = ‘a’;** **ch = toupper (ch);**
tolower (ch)	If ‘ch’ is an uppercase character, return the lowercase character, or else return it unchanged.	**char ch = ‘?’;** **ch = tolower (ch);**

C Strings

- C strings are simply character arrays, and therefore defined the same way as for other arrays.
- However, C strings must have a NULL character at their end, therefore if you want 20 characters, you must allow space for 21:

```
const int SIZE = 20;
...
typedef char StringType[SIZE+1];
...
StringType str;
```

- If you overflow your allocated space, strange things can happen. [1]

NULL

- The NULL character is the character with ASCII value 0:

```
char ch = 0;
```

- Or it can be designated as a character using a backslash:

```
char ch = ‘\0’;
```

- In fact many control and formatting characters have backslash equivalents, the most useful being:

newline	‘\n’
tab	‘\t’
quotes	‘\”’
backslash	‘\\’

Useful C String Functions

- C (and hence C++) has one of the most powerful sets of string manipulation functions available.

- C String functions require the `<cstring>` header file to be included.
- The string functions *do no overflow checking!*
- The require null terminated char arrays.

String Function List

`int strlen (str)`	Returns the number of characters from the start of the string to the NULL character.
`int strcmp(str1,str2)` [1]	Returns 0 if they are identical, a negative number if str1 is *lexographically* less than str2, and a positive number if str2 is greater than str1.
`int strncmp(str1,str2,n)`	Compares the first ‘n’ characters of each string. Returns 0 if they are identical, a negative number if str1 is *lexographically* less than str2, and a positive number if str2 is greater than str1.
`int strcpy(dest,src)` [1]	Copies src to dest, but does no bounds check.
`int strncpy(dest,src,n)`	Copies the first ‘n’ characters from src to dest, does no bounds check.

String Function List Continued

`int strcat (dest, src)`	Copies src to the end of dest, does no bounds check. [1]
`int strncat (dest, src, n)`	Copies the first ‘n’ characters of src to the end of dest, does no bounds check.
`char* strchr (str, ch)`	Searches str for the first occurrence of ch. Returns a *pointer* to ch, or NULL if the character does not occur.
`char* strrchr (str, ch)`	Searches str in for the last occurrence of ch. Returns a *pointer* to ch , or NULL if the character does not occur.

String Function List Continued

char* strstr (haystack, needle)	Searches haystack for needle. Returns a pointer to the position of needle within haystack, or NULL if the needle character string does not occur.
char *strrstr (haystack, needle)	Reverse search.
char* strtok(str, delim)	Returns a token string from str that is delimited by the string delim or NULL if not found. [1]
int atoi (str1)	Converts str1 to an integer and returns either the integer or 0 if str1 could not be converted to an integer.
float atof (str1)	Converts str1 to a floating point number and returns either the floating point number or 0 if str1 could not be converted to a float.
itoa (value, str1, 10)	Converts value to a base 10 string and puts it in str1. [2]

```
// strings1.cpp
// Simple program demonstrating the problems with overflow
// Version
// original by – Nicola Ritter
// modified smr

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

const int SIZE = 5;
typedef char StringType[SIZE+1]; // why + 1

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

Code

```
int main()
{
    stringType str1 = “abcde”;
    stringType str2;

    cout << "str1: " << str1 << endl;

    strcpy(str2, “0123456789”);

    cout << "str1: " << str1 << endl;

    strcpy(str1, “vwxyz");

    cout << "str1: " << str1 << endl;
    cout << "str2: " << str2 << endl;


    return 0;
}
```

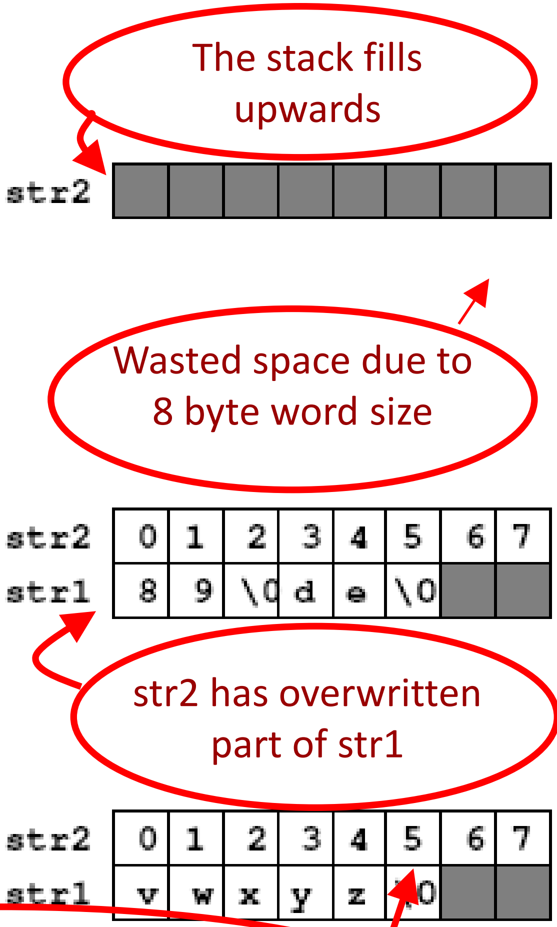
str1	a	b	c	d	e	\0		

Output

What output will we get?

```
str1: abcde


str1: 89


str1: vwxyz
str2: 01234567vwxyz
```

Memory



The stack fills upwards

str2								

Wasted space due to 8 byte word size

str2	0	1	2	3	4	5	6	7
str1	8	9	\0	d	e	\0		

str2 has overwritten part of str1

str2	0	1	2	3	4	5	6	7
str1	v	w	x	y	z	\0		

We have fixed up str1, but str2 still has no ‘\0’ at the end of it.

Pointer Access to Strings

```
// Output one character per line

void CharOutput (const StringType str)
{
    // Set a pointer at the beginning of the string
    //   stop when it reaches the NULL character
    //   the increment points the pointer at the next character
    for (char *ptr = str; *ptr != ‘\0’; ptr++) [1] //use nullptr
    {
      // Output the character at ptr
      cout << *ptr << endl;
    }
}
```

Using Functions that Return Pointers

```
// Count the number of times str2 appears in str1

int Count (const StringType &str1, const StringType &str2)
{
        int counter = 0;

        // Prime the while loop by looking for a first occurrence
        char *ptr = strstr(str1, str2);

        // While we have found an occurrence of str2 in str1
        while (ptr != NULL)
        {
                counter++;
                ptr++; // go to the next char in str1. why? See next line
                ptr = strstr (ptr, str2);
        }

        return counter;
}
```

Readings

- Textbook Chapter on Arrays and Strings, section on C-Strings
- Online reference to C string with examples
  - http://en.cppreference.com/w/cpp/string/byte
  - https://www.cplusplus.com/reference/cstring/
- Reminder:
  - It is essential to read the specified readings listed above. The lecture notes are not a substitute for the readings as the lecture notes do not cover everything you must know. See advice in unit guide.

Murdoch
UNIVERSITY

Data Structures and Abstractions

The STL, std string and more I/O

Lecture 9

What is the STL? [1]

- When C++ was written it was essentially a superset of C that allowed object oriented programming.
- At the start people either used the C-style arrays and strings (that we have already looked at and revised), or created their own classes.
- This was patently silly, so in the mid 1990s a standard set of templates was created for commonly used structures that are containers of data. [2]
- A container stores multiple objects of the same type. What changes between different types of container is the type of access, ordering and methods available.
- In this unit we will examine some of the most commonly used STL containers.

Problems with the STL

- The main problem with the STL is that it was clearly written by a committee.
- This means that:
  - It is ‘clunky’ – tries to please everyone.
  - Method names are often verbose, so everyone knows.
  - Use of methods may not intuitive all the time.
  - No bounds checking is done on arrays (vectors)!
  - The names of methods and classes is often non-mainstream. [1]
  - *Iterators* — which are generalisation of pointers — are used for a lot of the time, rather than indexes.
  - As it was written after C++ it might not always interface well with C++ and not all versions of C++ support it. [2]
  - But it is stable and usually well debugged – so can be used and usually very trusted.

The standard string class (C++)

- ANSI/ISO standard for C++ requires a string class to be a basic data type.
- Programmer should not have to worry about implementation details of the string class.
- It is part of the std namespace. [1]
- Is not part of C++ (as in reserved word); it is a programmer defined type available in the standard library.
- It has a wealth of good functions that make string handling much simpler.
- It has overloaded operators so that you can do such things as add to a string using the + operator.
- It makes reading in a whole line of text (including spaces) trivial.
- To use the standard string you must include the header file `<string>`.

The std string

- Unlike a C-style string, an std string cannot overflow.
- However no bounds checking is done on access.
- Do not assume that std string is NULL terminated. [1]
- Access to the string can be done as per any other string, using the [] operator.
- There are many good websites giving information on the various aspects of C++ including the std string.
- The following site is recommended. http://www.cppreference.com/cppstring/index.html

String Operator List

In each of the examples below, str1, str3 and str4 must be std strings, however str2 can be a c-style string. // std:string str1, str3, str4 [1]

`str1 = str2`	Copies str2 to str1

`str1 == str2`
`str1 < str2`
`str1 > str2`
	These give true or false depending on whether str1 is lexicographically equivalent to, less than or greater than str2.

`str1 = str3 + str4`	str1 becomes the concatenation of str3 and str4.
`str1 += str2`	str2 is appended to str1
`str1 += ch`	The character ch is appended to str1
`cout << str1`	str1 is output to screen
`cin >> str1`	The first word typed at the keyboard is stored in str1

Std::string methods

- Each method is overloaded multiple times.
- For example, to append to a string you could use:

`str1.append (str2)`	Appends str2 to the end of str1, where str2 can be a c-style OR std string.
`str1.append(str2, index, num)`	Append num characters from the part of str2 that starts at index.
`str1.append (str2, num)`	Append num characters from the start of str2, where str2 can be a c-style or std string.
`str1.append (num, ch)`	Add num repetitions of ch to the end of str1.
`str1.append (itr1, itr2)`	Append the string that starts at iterator itr1 and ends at iterator itr2.

List of Commonly Used Methods

- <various> below indicates that the method is overloaded and there are various possible parameters.
- For more information, go to http://www.cppreference.com/cppstring/index.html or your text book.

`str1.append (<various>)`	Append values str1.
`str1.c_str ()`	Returns a standard c-style string (char *). Required for file opening etc.
`str1.clear ()`	Empties the string.
`str1.compare (<various>)`	Compares two strings.
`str1.copy (<various>)`	Copies data into str1.
`str1.empty ()`	Returns true if the string is empty.
`str1.erase (<various>)`	Erases a part of the string.

Methods (continued)

str1.find (<various>)	Finds strings or characters within str1.
str1.rfind (<various>)	Finds strings/characters within str1, searching from the end.
str1.insert (<various>)	Inserts strings or characters into str1.
str1.length () str1.size ()	Return the length of the string (number of objects in it)
str1.push_back (ch)	Add the character to the end of the string.
str1.replace (<various>)	Replace strings/characters within str1.
str1.substr (index, num)	Returns num characters starting from index.
str1.swap (str2)	Swaps the two strings.

The getline Function

- A really useful non-member string function.
- It allows the input of sentences—strings with white spaces—into a string:

```
string str;

// Read in one word only
cin >> str;

// Read in characters until eoln
getline (cin, str); [1]
```

- It is also possible to read in only part of a line, by specifying a delimiter. For example:

```
// Read in characters from keyboard until a comma
getline (cin, str, ‘,’); [2]
// call the above as many times as needed to get each
// comma separated piece of data. See notes below
```

Using Iterators

- All of the STL uses iterators for access to data. The same is true for std string.
- For this reason, there are two functions that are standard for each container in the STL:

`.begin()`	Returns an iterator pointing at the first object in the container.
`.end()`	Returns an iterator pointing at the first memory location *past* the end of the container.

Simple string access program

Uses the normal index type access method

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

•       int main()
•       {
•               string str;

•               cout << "Enter a string: ";
•               getline (cin, str);

•               cout << endl
•             << "The characters in your string were: "
•             << endl;
•
•
        for (int index = 0; index < str.length(); index++)
        {
                cout << str[index] << endl;
        }
•               cout << endl;
•               return 0;
•       }
```

Simple string access program

Uses the iterator type access method

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

int main()
{
        string str;

        cout << "Enter a string: ";
        getline (cin, str);

        cout << endl
      << "The characters in your string were: "
      << endl;


for (string::iterator itr = str.begin();
     itr != str.end(); itr++)
{
        cout << *itr << endl;
}       cout << endl;
        return 0;
}
```

Testing

- Always test for:
  - empty strings;
  - empty files;
  - incorrect file names;
  - read-only files.

- Only assume data is correct if you have it stated in writing.

- Never assume the user won’t make a mistake.

- Always have a test plan and run through it ***every time*** you alter the program in any way.

Murdoch University

Text File I/O

- A name must be entered by the user.
- The file must be opened.
- A test must be done to check it has opened.
- Data is then read until EOF.

- File handling in C++ requires the `<fstream>` header file. See the lab exercises starting in topic 2.

```
// text.cpp
// Text file I/O
// Reading a file of integers separated by spaces or new lines
//
// Version
// 01 - Nicola Ritter
// 02 – modified smr
//----------------------------------------------------------------

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

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

const int SIZE = 256;
typedef char strType[SIZE+1];        // name the type

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

```
•    int main ()
•    {
•        strType filename;  // C array of chars

•        // Get a file name from the user
•        cout << "Enter name of file to read (one word only): ";
•        cin >> filename;

•        // Declare the file variable and try to open it using that
•        //   file name
•        ifstream fstr (filename);
•               // Is the ready state 0 (no errors)?
•               if (fstr.rdstate() == 0) //[1]
•               {
•                   int number;

•                   // prime the while loop
•                   fstr >> number;
•                   while (!fstr.eof())
•                   {
•                       cout << number << endl;
•                       fstr >> number;
•                   }

•               }
•               else
•               {
•                   cerr << "Could not open file \"" << filename
•               << "\" for reading." << endl;
•               }

•               return 0;

•       }
```

Opening Files using std string

- string filename;
- cout << “Enter filename: ”;
- getline (cin, filename);

- ifstream fstr (filename.c_str());

- // The rest of the code is the
- // same as before

File names can now contain spaces

`c_str()`

gives the required access to the actual character string

`str.c_str()`

can be also be used with some of the standard c-style string functions

More About File I/O

- Opening a file for output is done in a similar way, except that it will be an output file stream:
  `ofstream fstr (filename);`
- To append to a file, you would open it with:
  `ofstream fstr (filename, ios::app);`
- When you are going to store more of one type of data in a file, it is necessary to store the number of each type. You don’t have to but this can cause problems in designing the algorithm. Algorithm would require asking the user or the value is hard coded.
- For example, when storing an array of 100 integers, you would write the 100 to the file first.
- When reading the file, you would then read the number 100 and therefore know that you had 100 integers to read.

Other Useful I/O Functions

- There are several generically useful I/O functions (as well as the formatting ones done previously) [1]:

`cin.good()`
`fstr.good()`
	Returns true if the stream (standard input or a file stream) has no errors.

`.flush()`	Flushes (empties) the stream (but not the actual file!).
`.peek()`	Peek at the next character without actually reading it.
`.ignore ()`	Ignore characters

Murdoch UNIVERSITY

Readings

- Chapter on Arrays and String, Section on C-Strings.
- Chapter on User-defined Simple Data Types, Namespaces, and the string type, section on string type.
- Chapter on Standard Template Library (STL)
- Website: http://www.cplusplus.com/faq/sequences/strings/split/
- Website: “Iterators”, https://www.geeksforgeeks.org/introduction-iterators-c/ - optional
- Video by Standford uni provided by academicearth. Links:
  - C/C++ Libraries explains string operations
  - C++ Console I/O

Murdoch
UNIVERSITY

Data Structures and Abstractions

Complexity The STL Vector [1] and Iterators

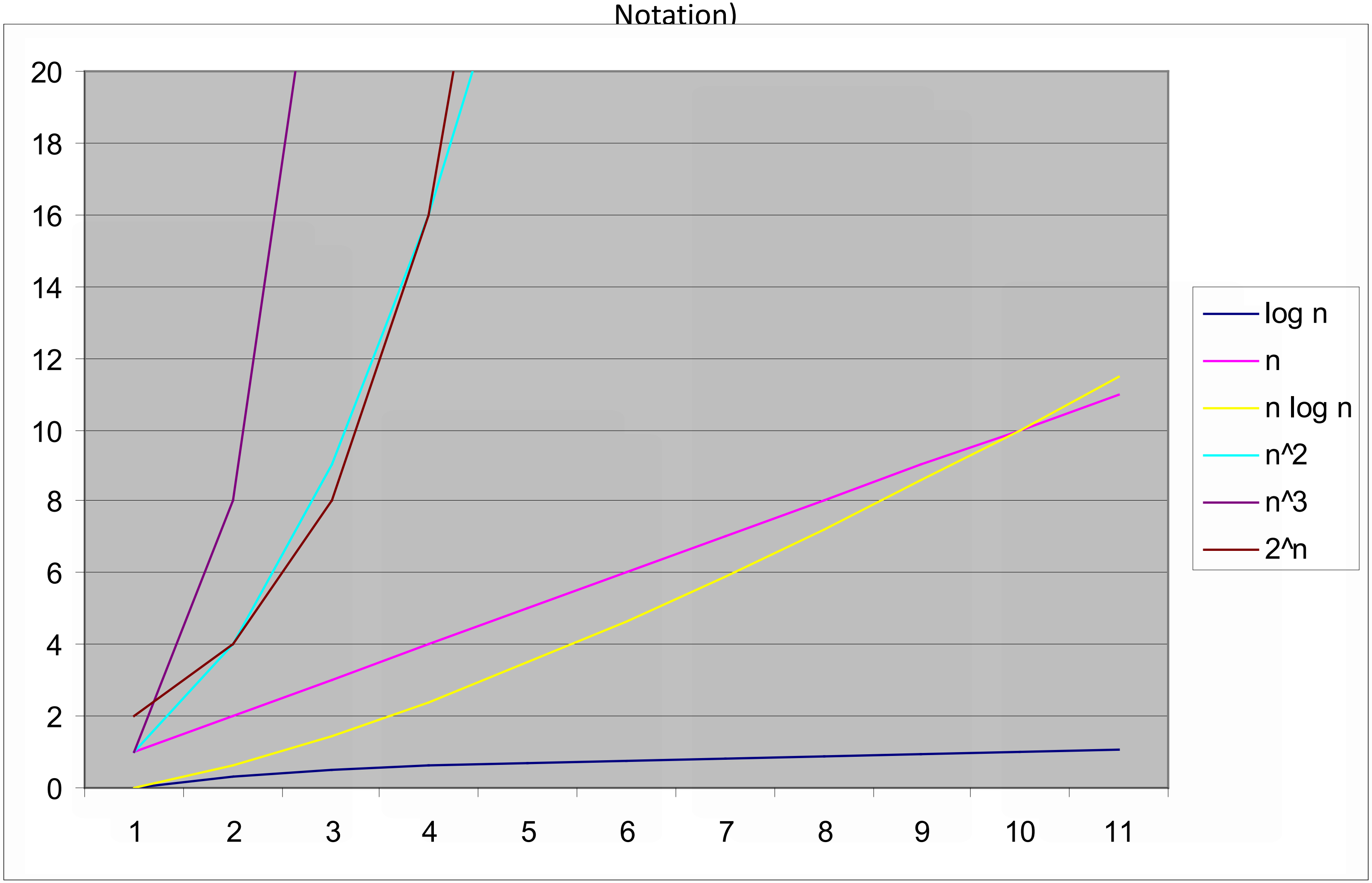
Lecture 10

Complexity

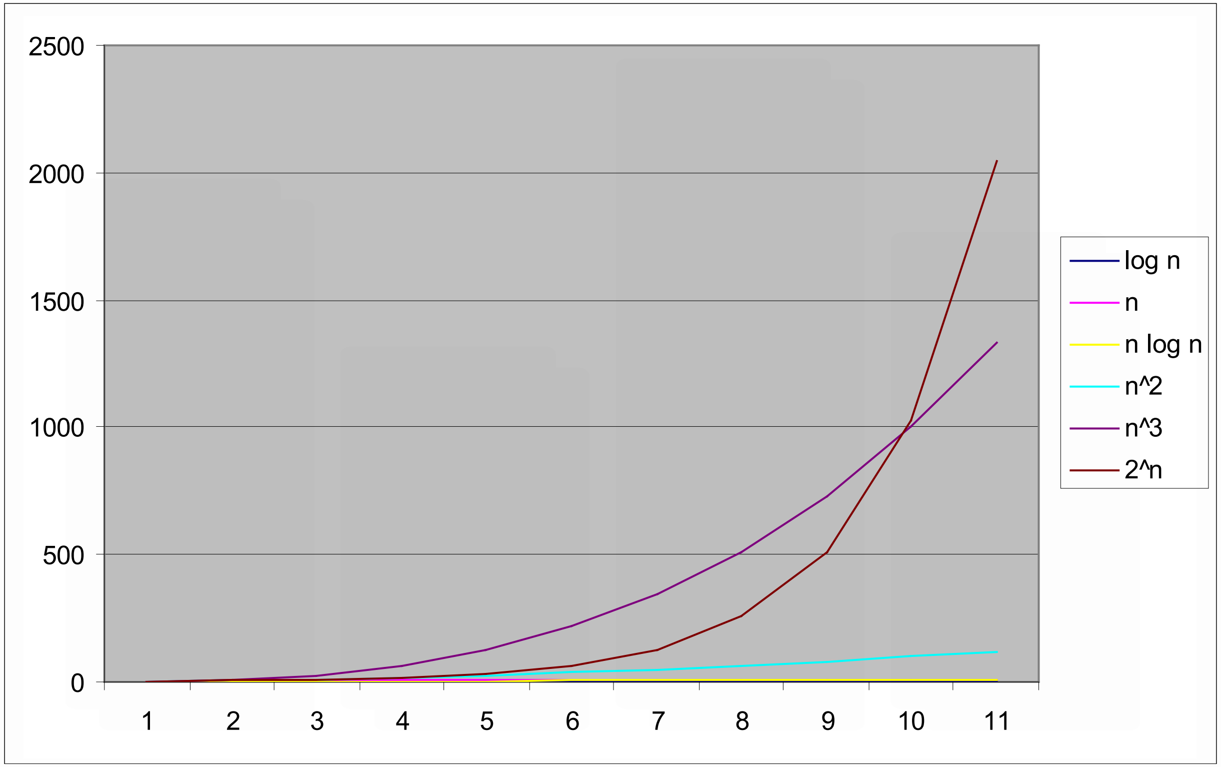
- The complexity of algorithms is measured in terms of how long they take to execute or how much resources they utilise. Our current interest is on the execution time of the algorithm.
- This in turn is often measured in terms of the number of operations that are required.
- Complexity is described in 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.

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}

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 [1] (**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.

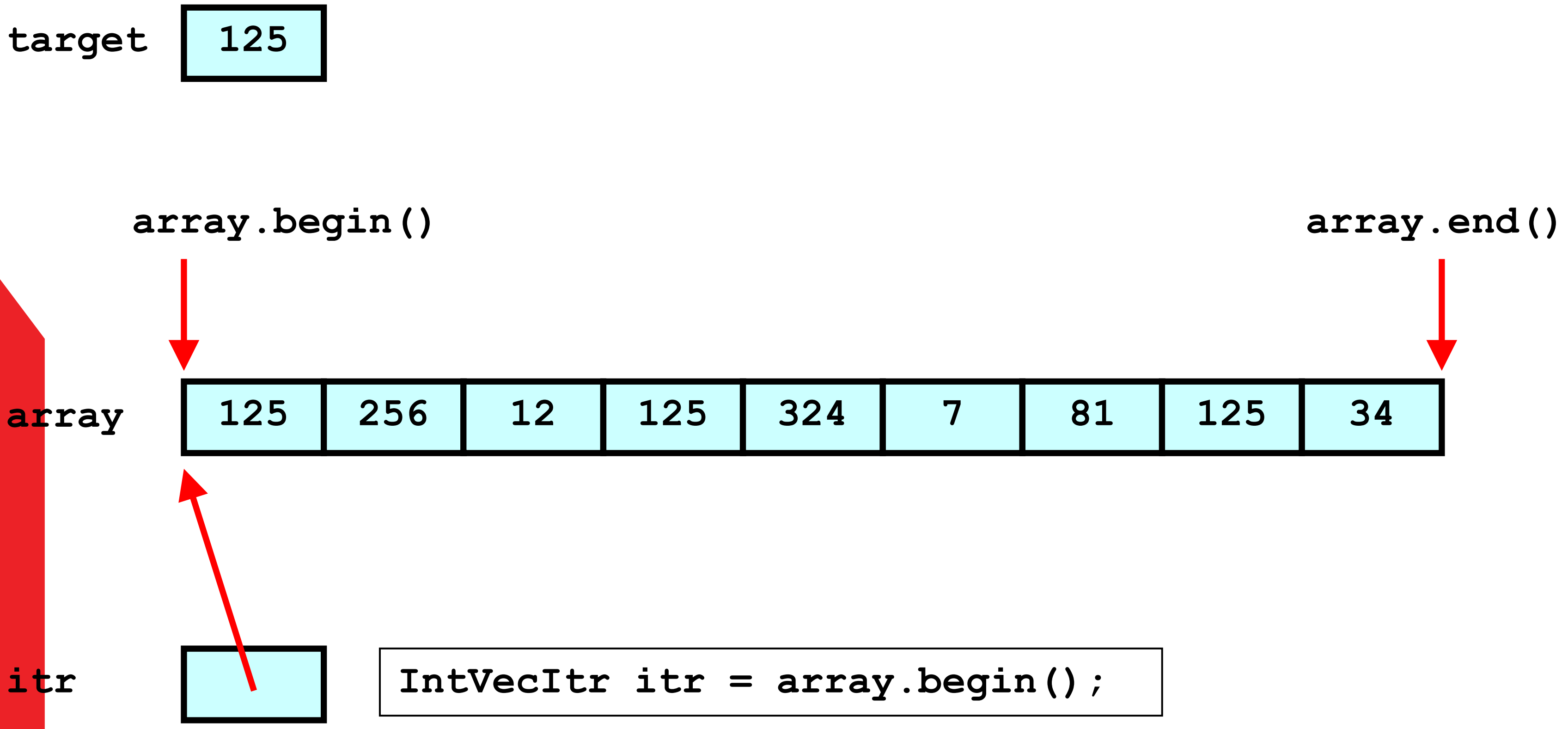
```
int target;
cout << “Enter number to be deleted: “;
cin >> target;

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



target 125

array.begin()

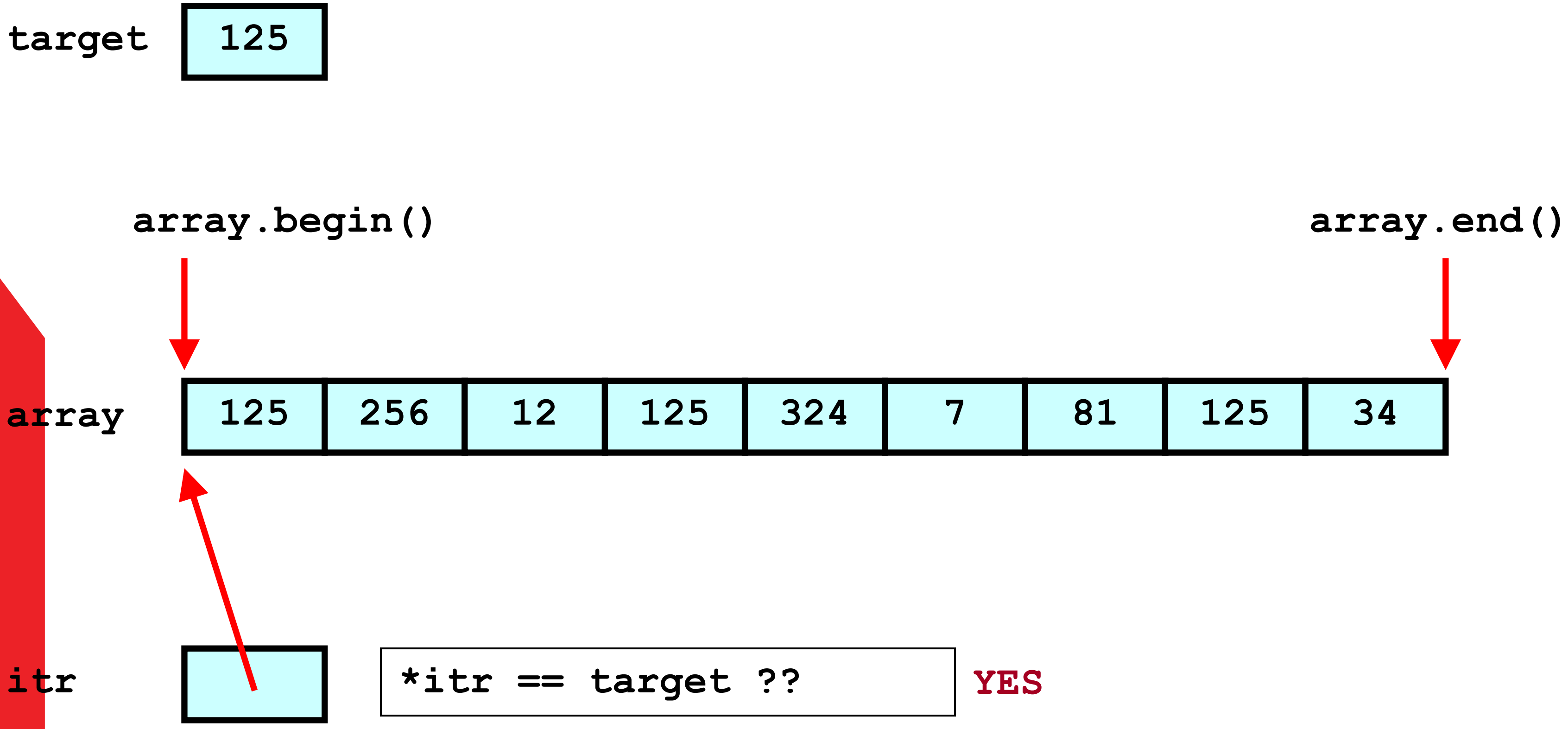
array.end()

array								
125	256	12	125	324	7	81	125	34

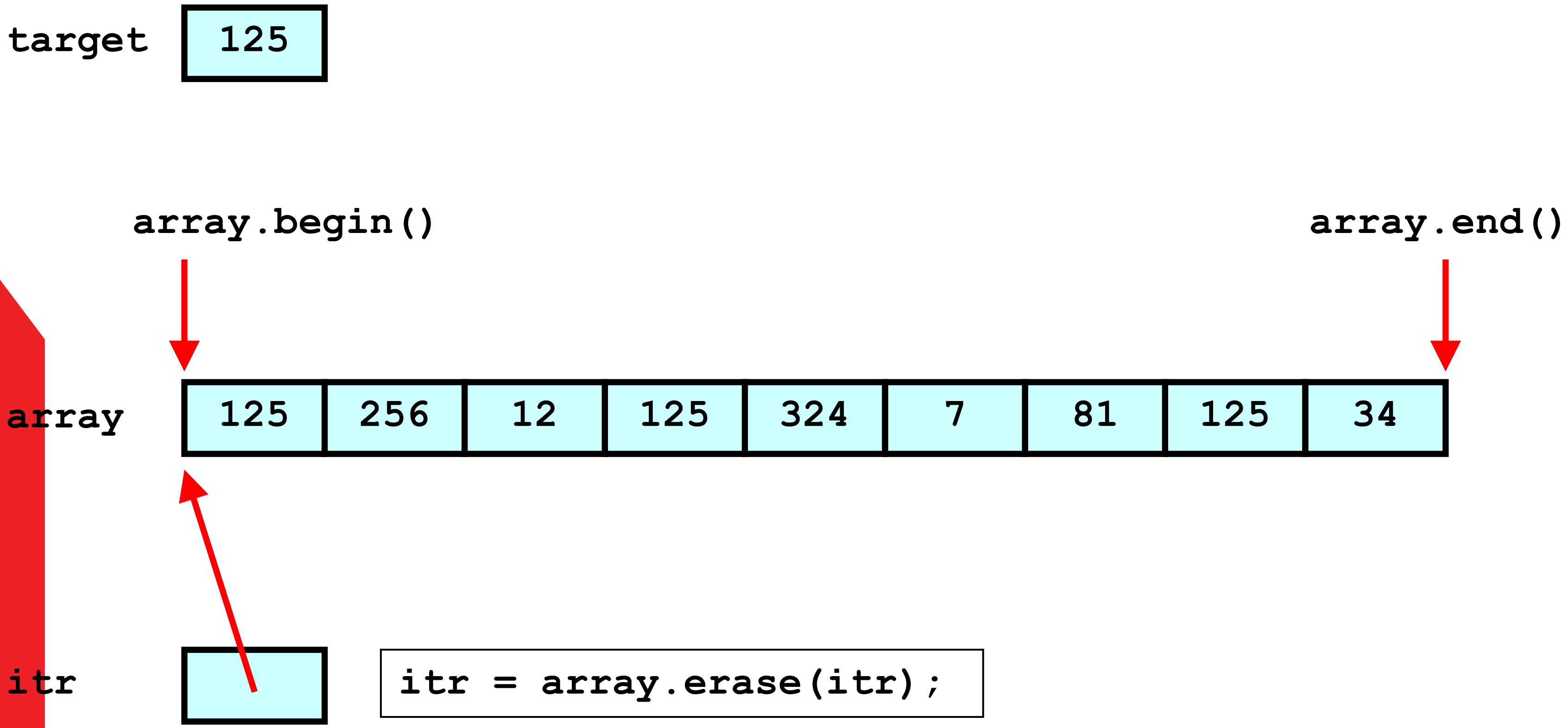
itr

```
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



target 125

array.begin()

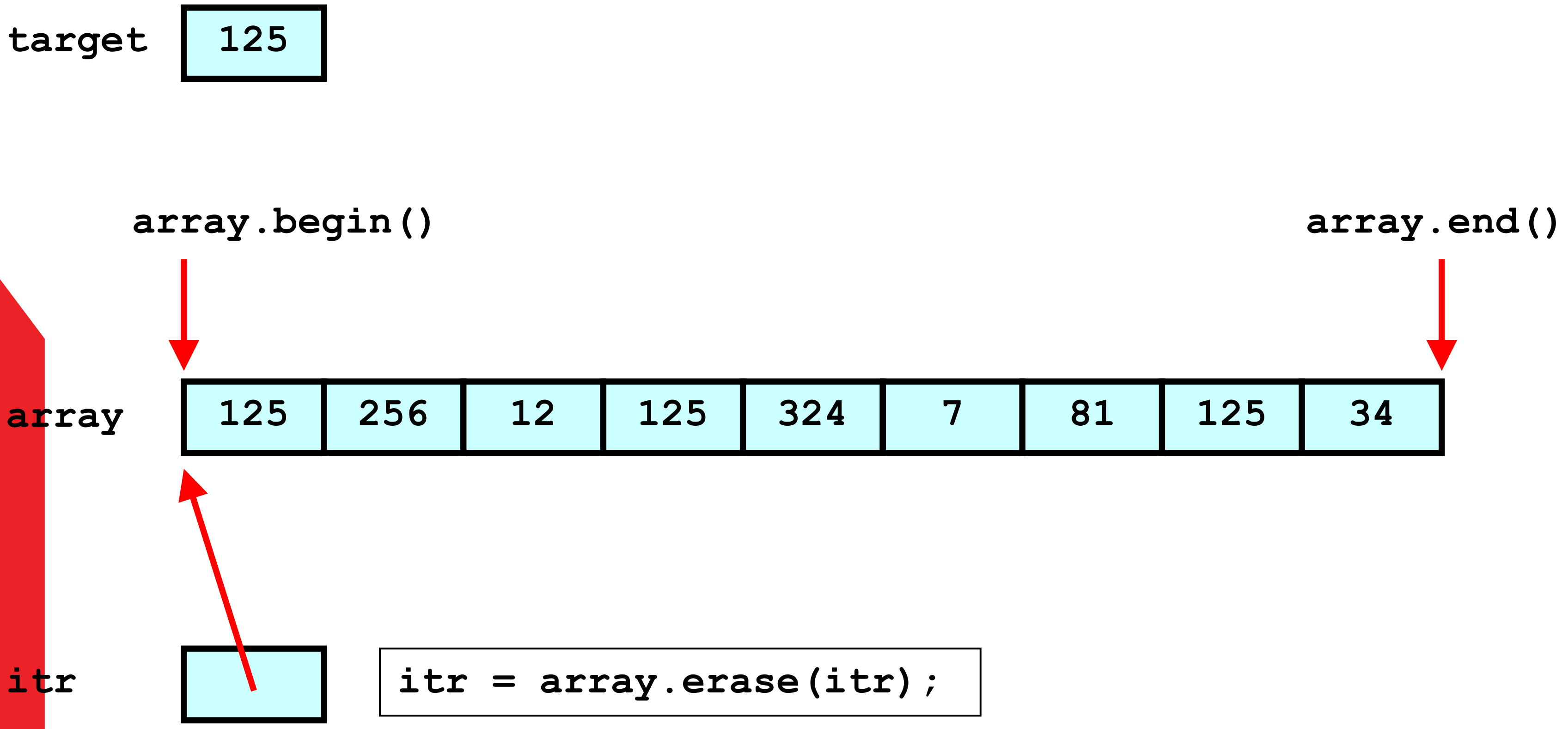
array.end()

array								
125	256	12	125	324	7	81	125	34

itr

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

Example



target 125

array.begin()

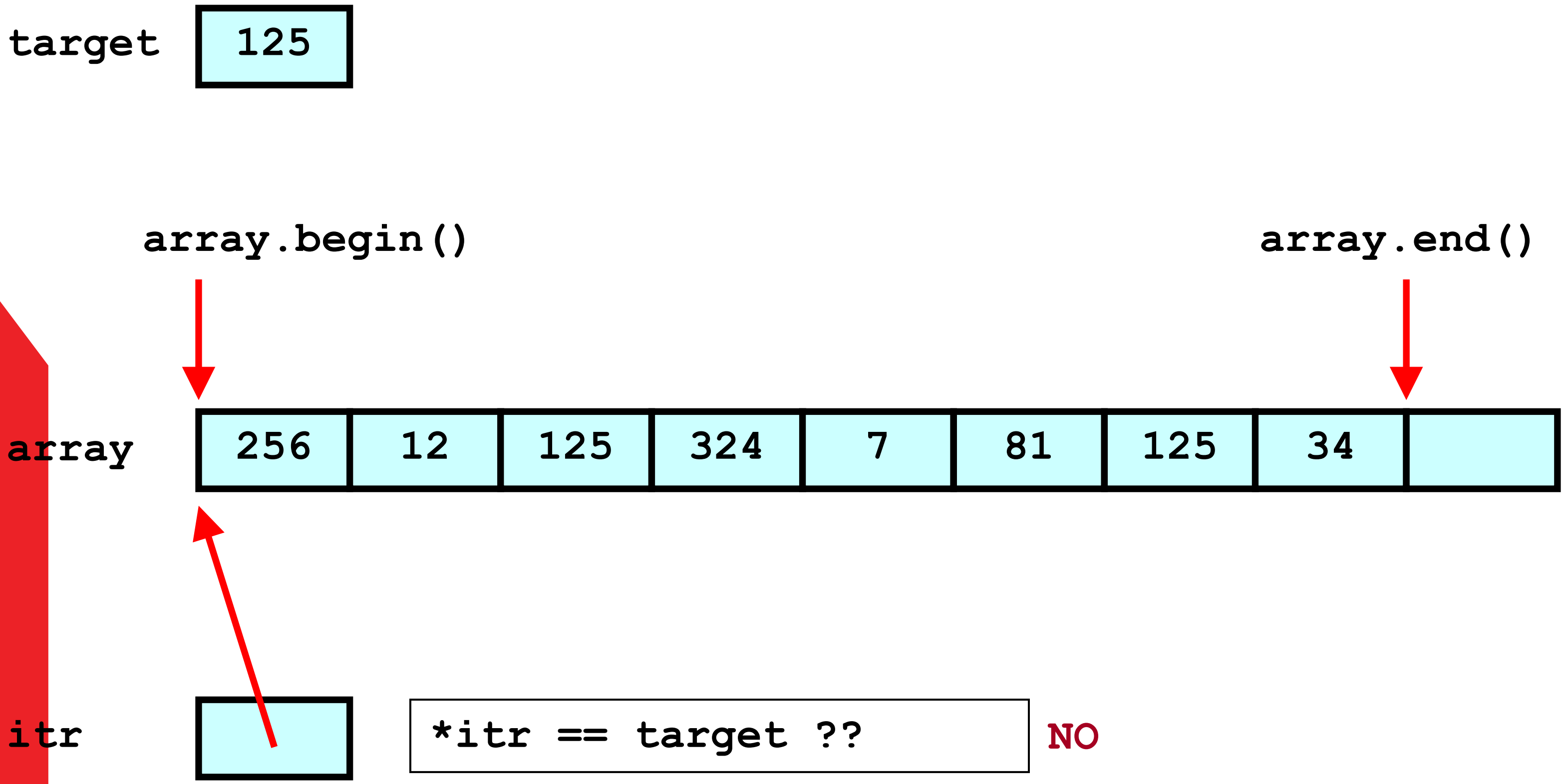
array.end()

array	125	256	12	125	324	7	81	125	34

itr

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

Example



target 125

array.begin()

array.end()

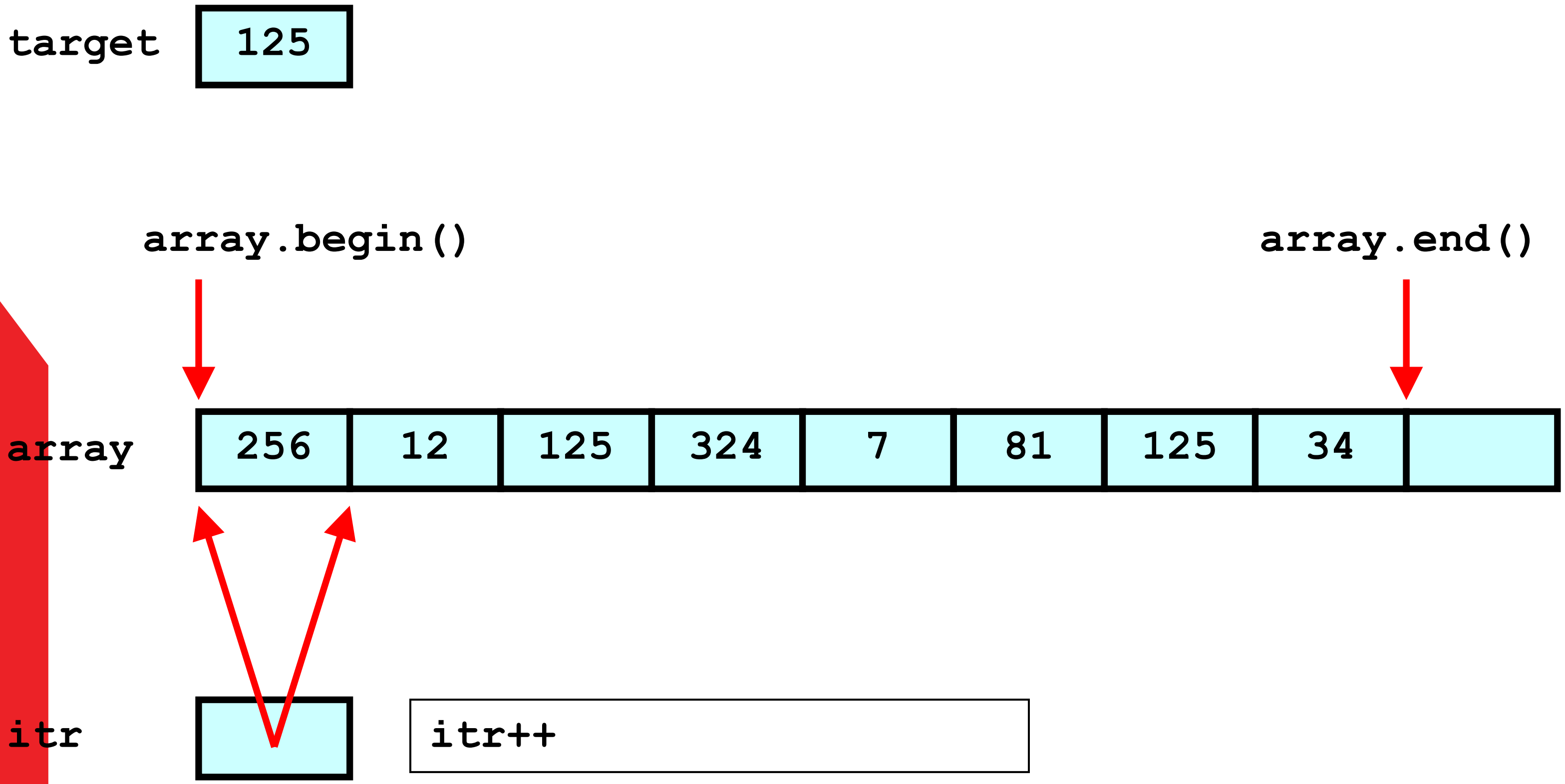
array	256	12	125	324	7	81	125	34	

itr

`*itr == target ??`

NO

Example



target 125

array.begin()

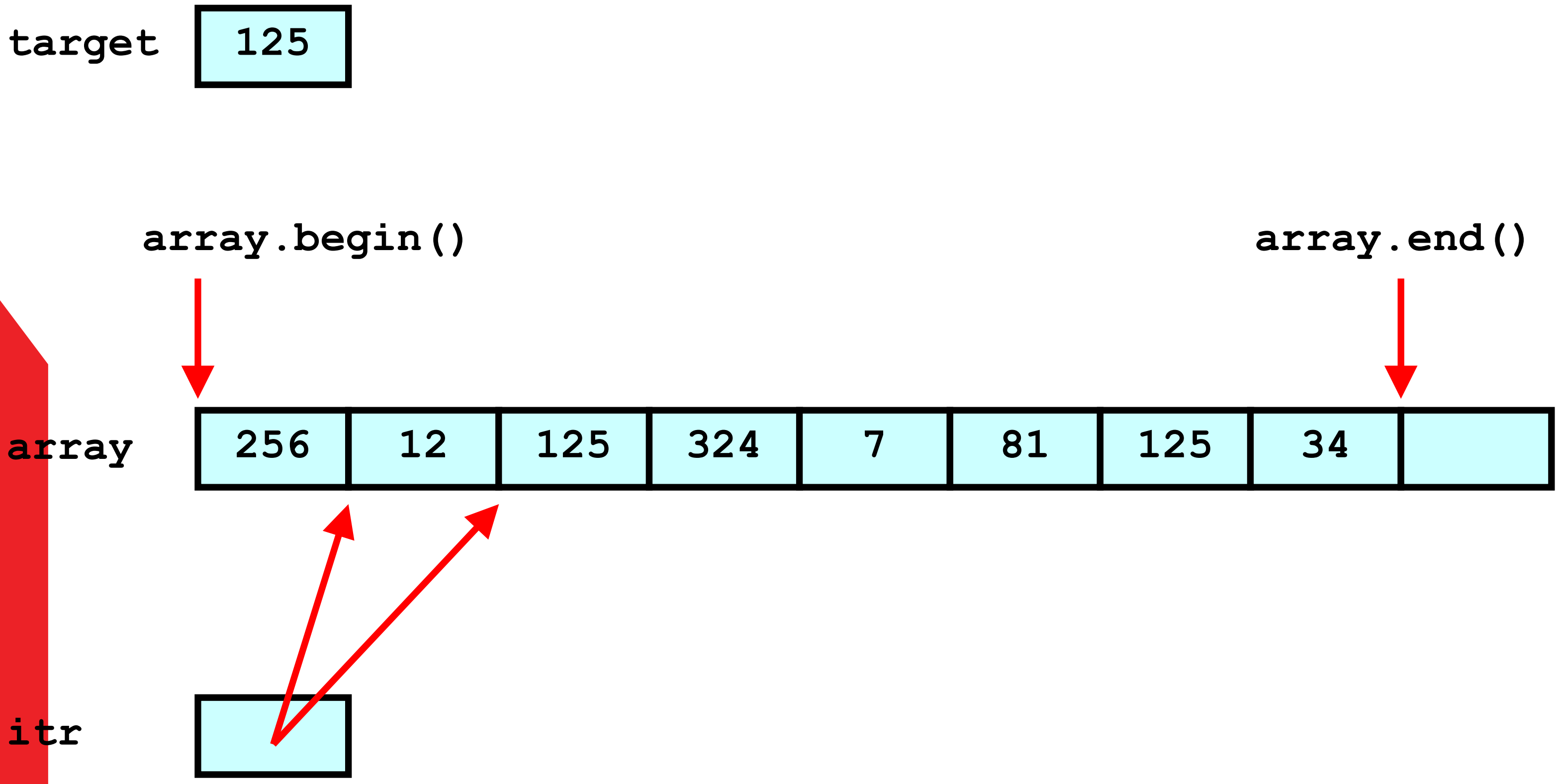
array.end()

array								
256	12	125	324	7	81	125	34	

itr

itr++

Example



target 125

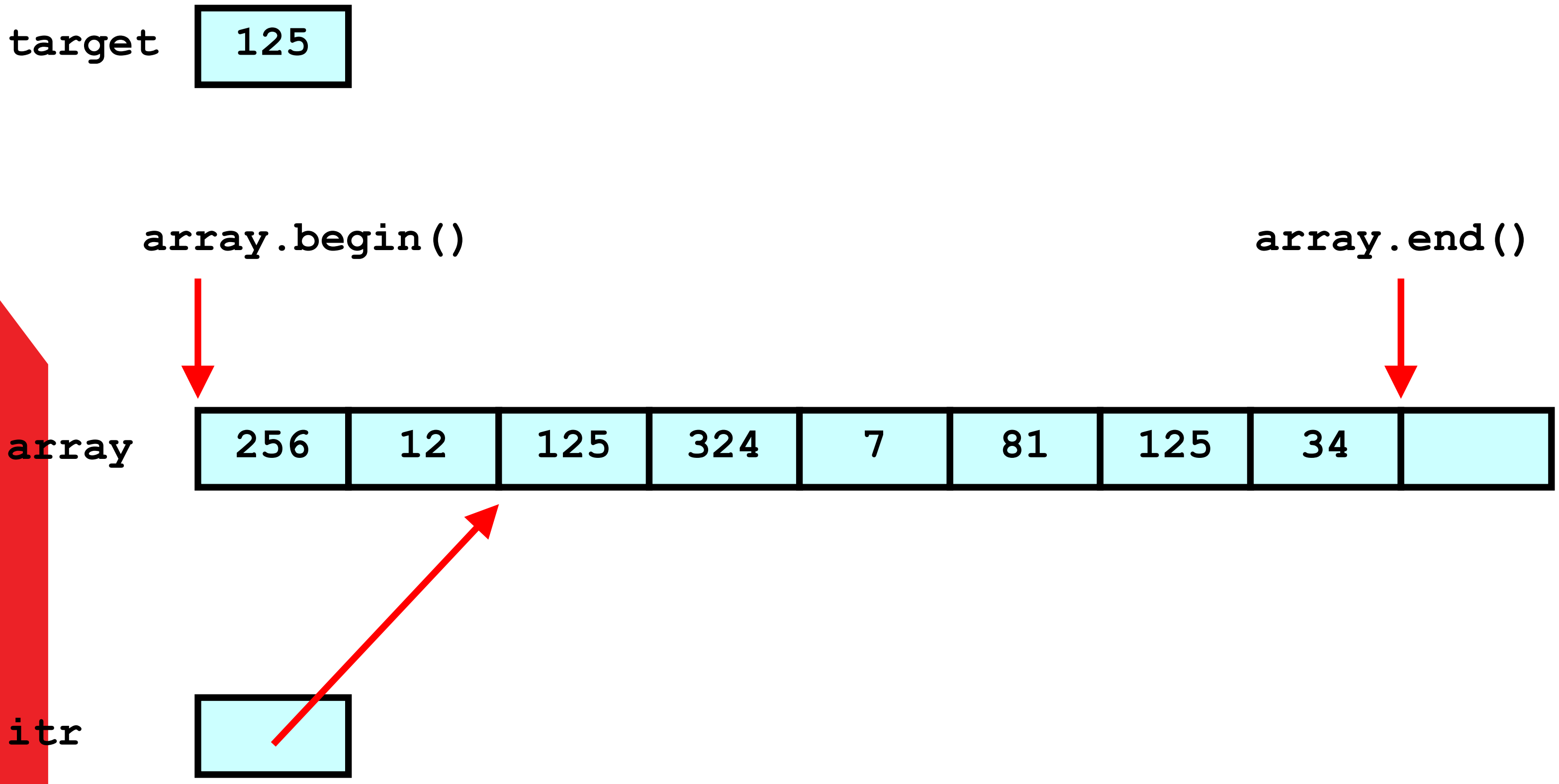
array.begin()

array.end()

array									
	256	12	125	324	7	81	125	34	

itr

Example



target 125

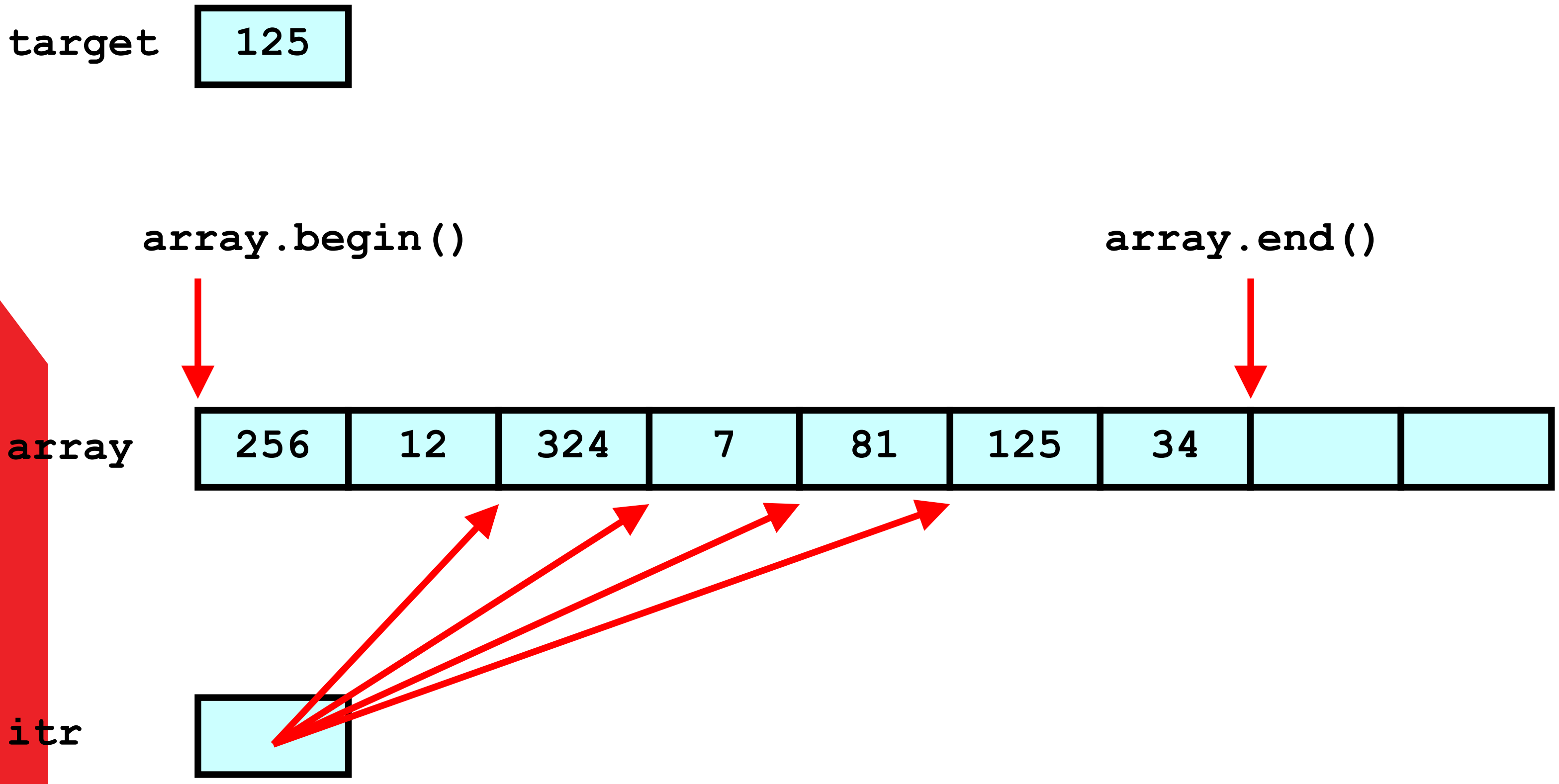
array.begin()

array.end()

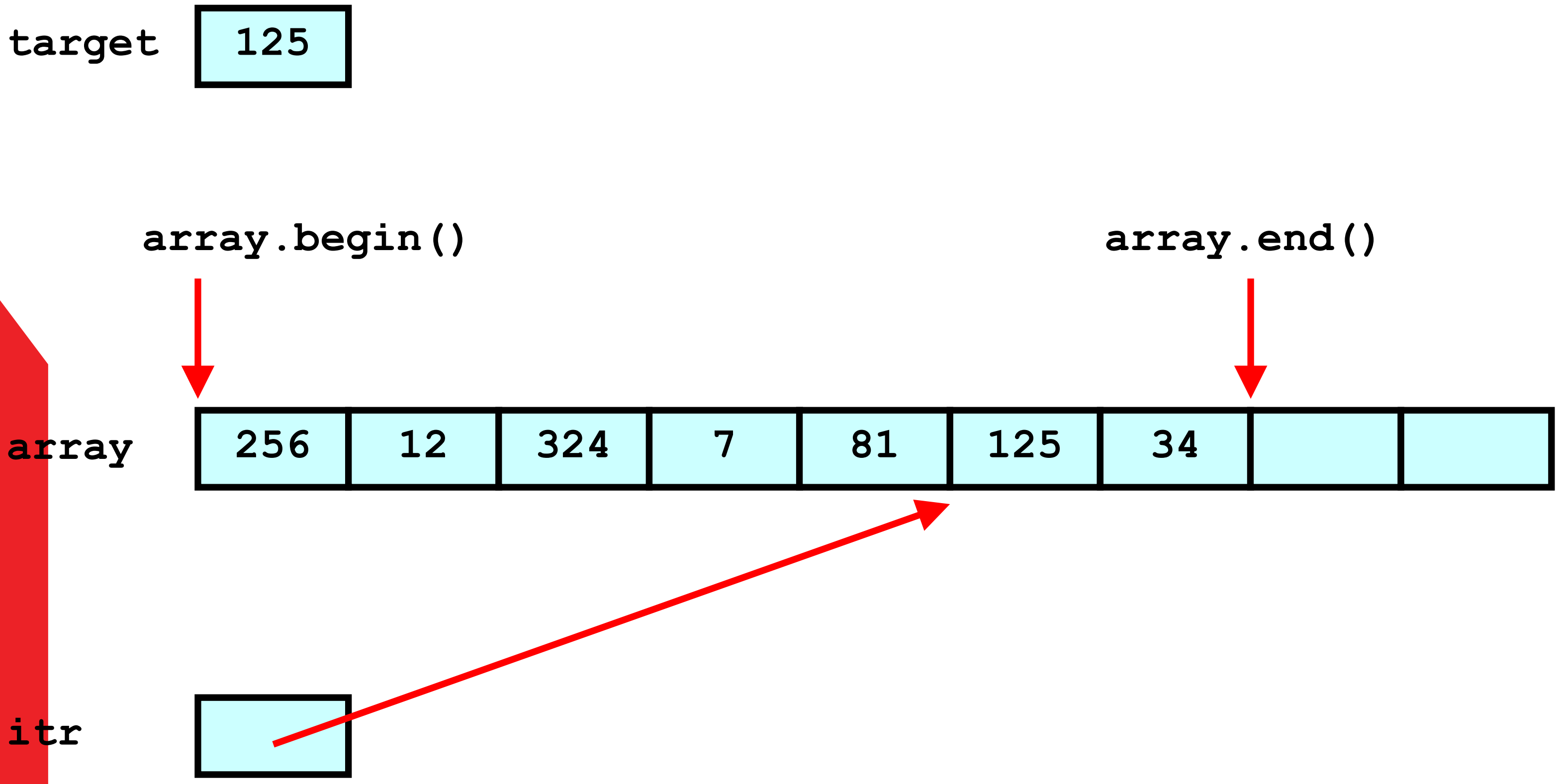
array	256	12	125	324	7	81	125	34	

itr

Example


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

Example



target 125

array.begin()

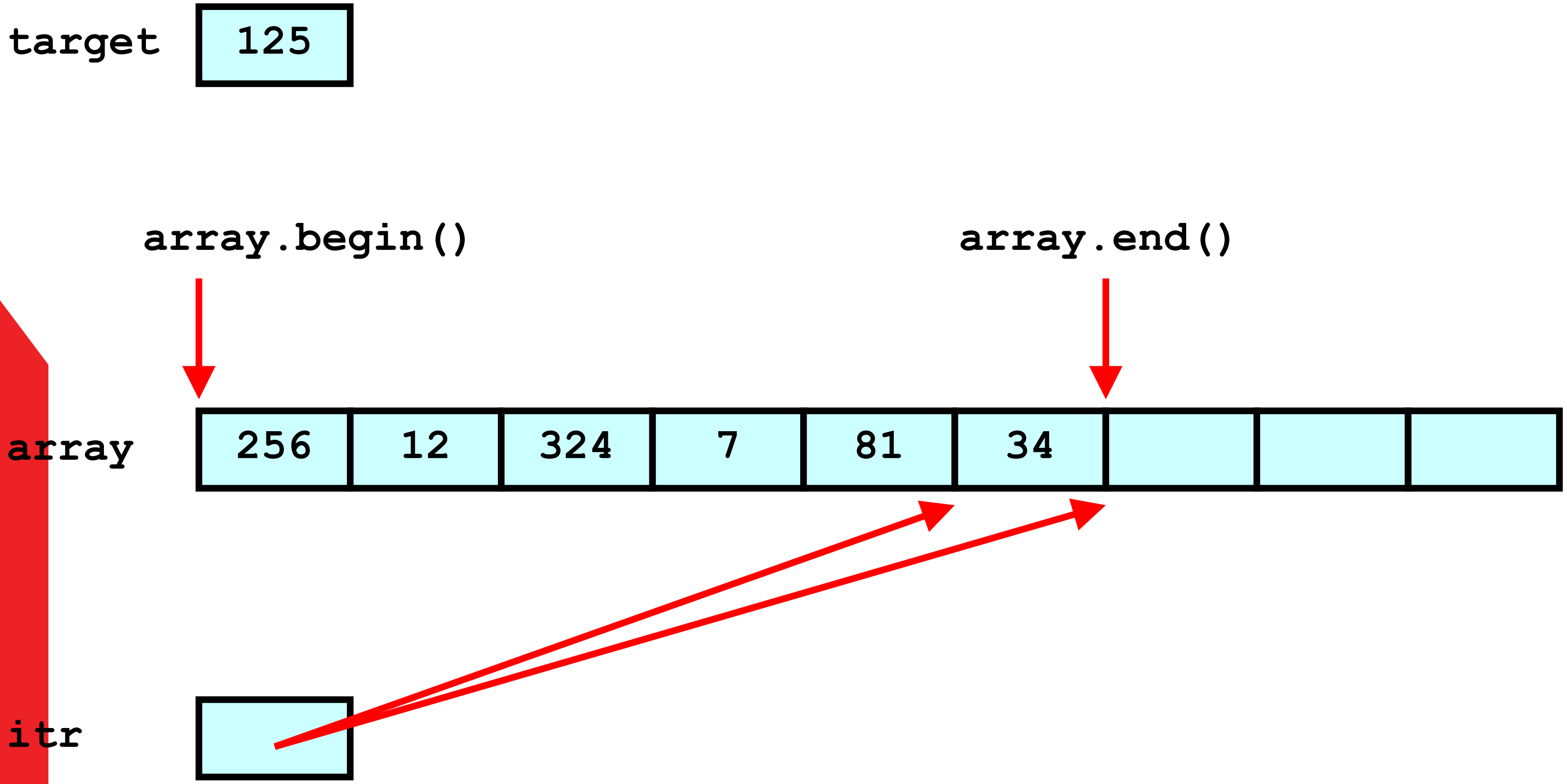
array.end()

array

256	12	324	7	81	125	34		

itr

Example



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

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]

Murdoch
UNIVERSITY

Data Structures and Abstractions

Object Oriented Design and Testing

Lecture 11

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.

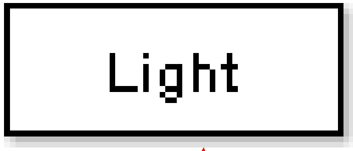
Example Class

- Consider a class simulating a light.

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

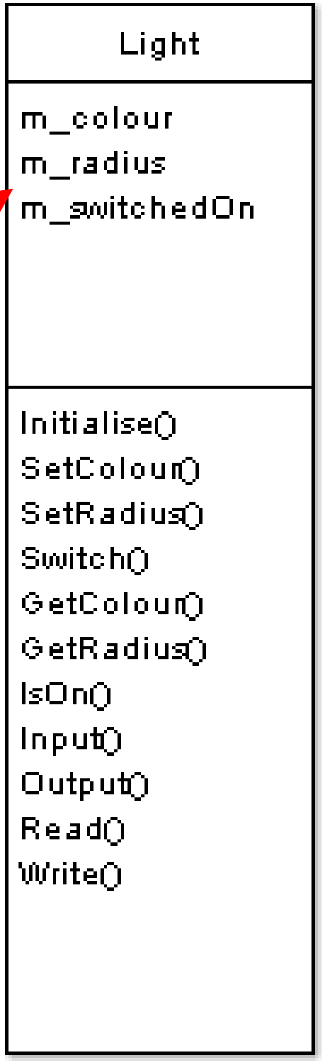
- There might then need to be methods that (not minimal) [1]:
  - initialised all objects of the light class;
  - set each of the attributes;
  - returned the current value of each of the attributes;
  - output the current value of each attribute to the screen;
  - allowed input of values from the keyboard;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

The Unified Modelling Language (UML) [1]



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.

The Data Dictionary [1] [2] [3]

Name	**Type**	**Protection**	**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()	procedure	#	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()	procedure		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

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.

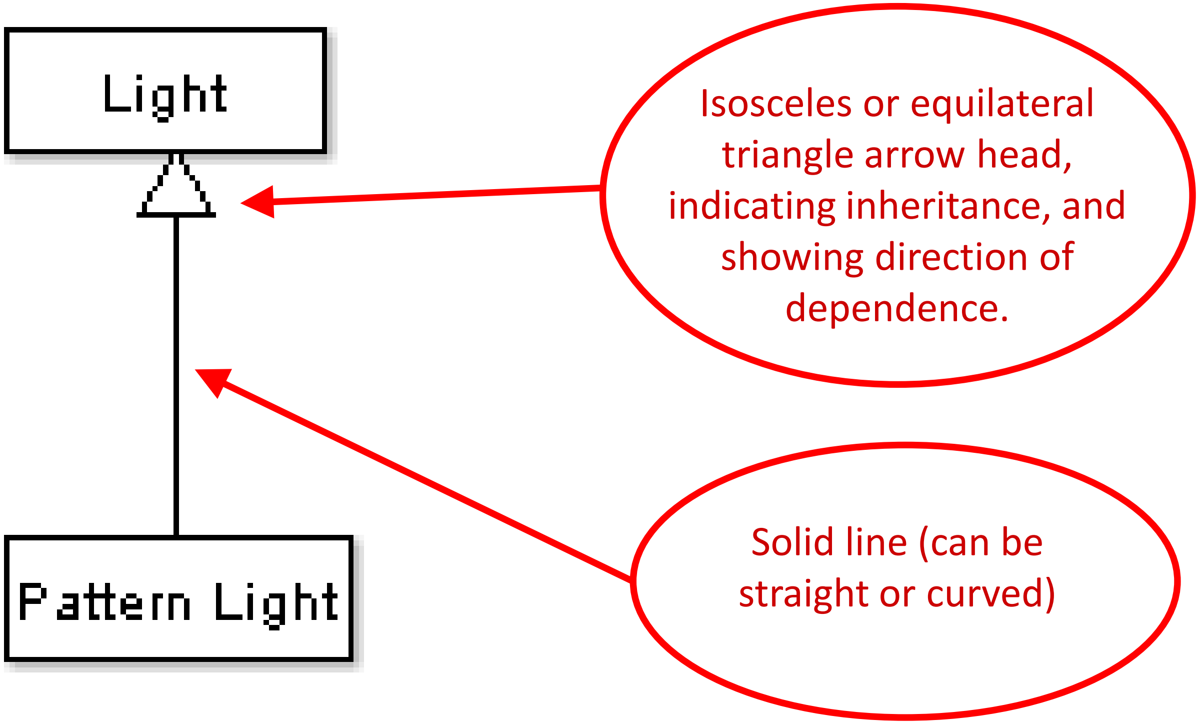
Inheritance

- 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



Isosceles or equilateral triangle arrow head, indicating inheritance, and showing direction of dependence.

Solid line (can be straight or curved)

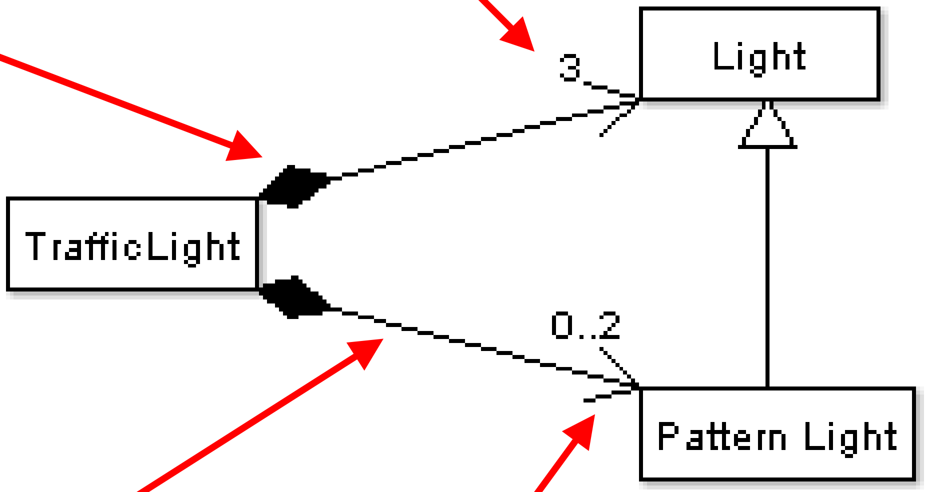
Composition

- 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.

- The arrow at the end of the composition line can be removed

Cardinality

Symbol meaning ‘composed’



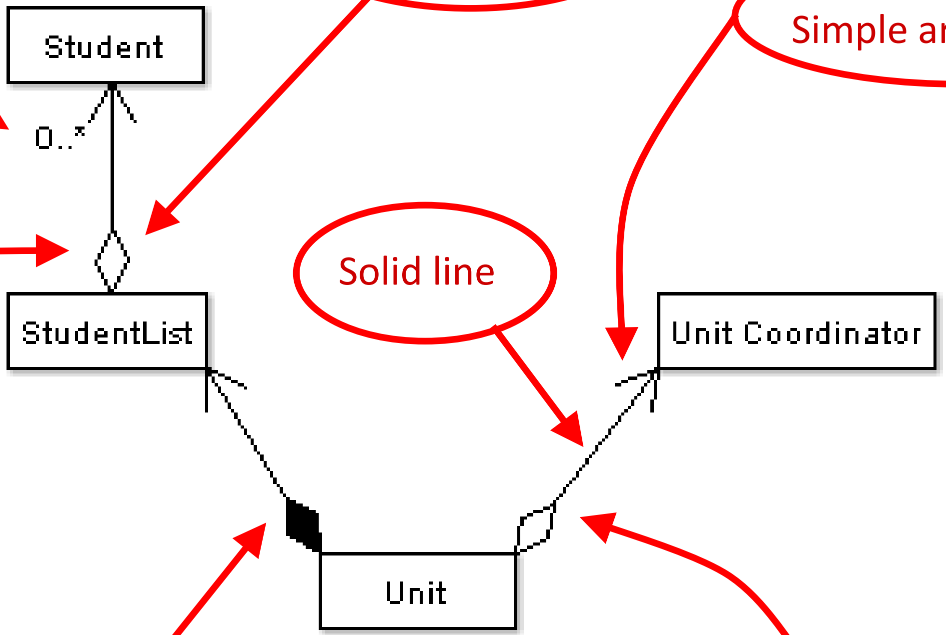
Solid line (straight or curved)

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 contained 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, but is a kind of “contained” relationship.



Cardinality: between 0 and n students

Symbol meaning ‘aggregated’

Simple arrow head

StudentList stores references to a set of students

Solid line

The Unit *has* exactly one student list **[1]**

Unit has a reference to exactly one Unit Coordinator

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?
- If your Date class objects are stored in an array, how would sort the array, assuming you had a sort algorithm like bubble sort?
- 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 now.
- Reference book: *UML distilled: A Brief guide to the standard object modeling language* from My Unit Readings (chapters 1, 3 and 5) ebook from https://murdoch.rl.talis.com/index.html
- Rules for Design Style, Coding Style in unit Reference book, C++ coding standards: 101 rules, guidelines, and best practices. Also look at rules 32 to 44. ebook https://go.exlibris.link/xwqTmlFk (online)

Murdoch
UNIVERSITY

Data Structures and Abstractions

Object Orientation, Classes and Unit Testing (revision)

Lecture 12

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. [1]
  - 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
```

Murdoch UNIVERSITY

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

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

        void Clear ();
        // can do without friends, ok for debugging purposes during development, but remove it
        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
```

Murdoch UNIVERSITY

```
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 ();}  [2]
        ~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
```

Murdoch UNIVERSITY

```
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. But can be used during debugging at development
        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.

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

‘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 ();
        // 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
```

```
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.

```
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 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; // [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

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 file. It is best not to have implementation in the class declaration. Separate the interface (declaration) from the implementation [1]

- Such methods and operators are defined in a source file (.cpp) with the same base name as the header (.h) 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.**

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.

```
// LightTest.cpp This is the unit 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.
- Rules for Design Style, Coding Style in unit Reference book, C++ coding standards: 101 rules, guidelines, and best practices. Also look at rules 32 to 44. Library ebook https://go.exlibris.link/xwqTmlFk

Murdoch
UNIVERSITY

Data Structures and Abstractions

Completing the Minimal and Complete Class

Lecture 13

Versioning and Backing Up

- Of course, as you add to the class you must:
  - **Backup the previous version before changing anything.**
  - Change the version number and describe what you are changing.
  - Test *everything* in the test plan again.

- The easiest way to store the backups is simply to have a directory called 'backup' and label the backups (presumably zip files) with the version number.
  For example: Light-01.zip, Light-02.zip etc.

- The zip file will contain the *entire* workspace, making reversion to a backup simple.

Versioning and Backing Up

- The previous backup and versioning method is **manually intensive**.
- There are automated tools available which help with this.
- You are encouraged to try out a tool like git http://git-scm.com/, Subversion http://subversion.tigris.org/ or Redmine http://www.redmine.org/. You might want to try a demo of Redmine at http://demo.redmine.org/.
- Subversion is server software and you as the user connect to it using a client which runs on your machine. One example is TortoiseSVN http://tortoisesvn.tigris.org/.
- **We recommend the following if you can' t make up your mind.**
  - **GithHub** https://education.github.com/ or
  - **Bitbucket** https://education.github.com/
- Make sure that the repositories are private.

Code & Test Plan

The additions are made in order:

- Set methods for each data member.
- Get methods for each data member.
- Overloaded assignment operator.
- Copy constructor.
- If **required** add:
  - overloaded input operator;
  - overloaded relational operators;
  - overloaded arithmetic operators;
  - file I/O methods;
  - other overloaded constructors;
  - processing methods

This list may not be the minimal set.

Should these be part of the class?

Think through this carefully.

I/O operators would not normally be part of the class – see previous topics. So where would you implement them? [1]

Change Plan

For each change or addition to a class, you must:

- Add the method descriptions to the header (.h) file.
- Add the code to the implementation (.cpp) file.
- Add tests to the test plan.
- Add tests to the test program (unit test program).
- Run all the tests every time and debug the code

Changes to the Header File

- // Light.h
- // Class representing a light
- //
- // Version
- // 01 - Nicola Ritter date ..
- // 02 – Nicola Ritter, date ...
- //   Added in Set methods
- // 03 – smr, date..
- // to convert all friend methods to non-friends,
- // non-members
- //---

It is very important to record what you changed!

Use Doxgen style comments in header (.h) files

```
// Needs doxgen comments
// separate the implementation – remove from inside the class
class Light
{
public:
        Light () {Clear ();}
        ~Light () {}; //[1]

        void Clear ();

        void SetColour (const string &colour) {m_colour = colour;}
        bool SetRadius (float radius);
        void Switch () {m_on = !m_on;}

        friend ostream& operator << (ostream &ostr, const Light &light); [2]

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

We are accepting any string as a colour, so there is no error state

Inline function, switches the light to the opposite of what it was

Changes to the Implementation (.cpp) File

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

bool Light::SetRadius(float radius)
{
        if (radius > 0)
        {
                m_radius = radius;
                return true;
        }
        else
        {
                return false;
        }
}

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

Changes to the Test Plan [1]

Test	Description	Actual test call/data	Expected Output	Passed
1	Check that constructor initialises the data and check that output operator works.	Light light	0 cm, white light is off	
2	Colour setting works.	light.SetColour (“red”)	0 cm, red light is off	
3	Setting a negative radius will fail.	light.SetRadius (-9.3)	Error message 0 cm, red light is off	
4	Setting with a positive radius will work.	light.SetRadius (9.3)	9.3 cm, red light is off	
5	Switching an off light on.	light.Switch ()	9.3 cm, red light is on	
6	Switching an on light off.	light.Switch ()	9.3 cm, red light is off	
7	Clearing a light that is on.	light.Switch() light.Clear ()	0cm, white light is off	

Changes to the Test File (Unit Test)

```
#include “Light.h”
using namespace std;   // expose everything – not good but it is convenient for now.

int main()
{
        Light light;
        cout << "Light Test Program" << endl << endl;

        cout << "Test One" << endl;
        cout << light << endl << endl;

        cout << "Test Two" << endl;
        light.SetColour("red");
        cout << light << endl << endl;

        cout << "Test Three" << endl;
        if (!light.SetRadius((float)-9.3))
        {
             cerr << "Radius must be greater than 0" << endl;
        }
        cout << light << endl << endl;
```

The (float) is called a ‘cast’. Without it, the compiler assumes 9.3 is a double rather than a float, and a warning is generated stating “truncation from 'const double' to 'float'” [1]

```
        cout << "Test Four" << endl;
        if (!light.SetRadius((float)9.3))
        {
            cerr << "Radius must be greater than 0" << endl;
        }
        cout << light << endl << endl;

        cout << "Test Five" << endl;
        light.Switch();
        cout << light << endl << endl;

        cout << "Test Six" << endl;
        light.Switch();
        cout << light << endl << endl;

        cout << "Test Seven" << endl;
        light.Switch();
        light.Clear();
        cout << light << endl << endl;

        cout << endl;

        return 0;
    }
```

This is about as long as a function should get.

If more tests get added, then main() must become a function that calls other functions that do the actual tests.

Each test can be in its own function. Makes things a lot neater.

Output From LightTest

- Light Test Program

- Test One
- 0 cm, white light is off

- Test Two
- 0 cm, red light is off

- Test Three
- Radius must be greater than 0
- 0 cm, red light is off

```
Test Four
9.3 cm, red light is off


Test Five
9.3 cm, red light is on


Test Six
9.3 cm, red light is off


Test Seven
0 cm, white light is off
```

Refactored LightTest.cpp

```
// LightTest.cpp
// modularised unit test - preferred way so that each
// test number matches the test plan number.
// Approach for unit testing classes for assignment

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

#include "Light.h"

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

void Test1 (); //print after construction
void Test2 (); // set colour
void Test3 ();
void Test4 ();
void Test5 ();
void Test6 ();
void Test7 ();
```

It is also a good idea to comment each call to indicate what the test is doing. Get this comment from the test plan table.

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

int main()
{
    Light light;

    cout << "Light Test Program" << endl << endl;

    Test1 (); //print after construction
    Test2 (); // set colour
    Test3 ();
    Test4 ();
    Test5 ();
    Test6 ();
    Test7 ();

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

It is also a good idea to comment each call to indicate what the test is doing. Copy/Paste from testplan table

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

void Test1 () // comment each test here too  - copy/paste from testplan table
{
        Light light;

        cout << "Test 1" << endl; // make it more descriptive
        cout << light << endl << endl;
}

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

void Test2 () // set colour
{
        Light light;

        cout << "Test 2" << endl;
        light.Set("red");
        cout << light << endl << endl;
}
```

- etc...

Get Methods [1]

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

•         void Clear ();

•         void SetColour (const string &colour) {m_colour = colour;}
•         bool SetRadius (float radius);
•         void Switch ();

•         void  GetColour (string &colour) const {colour = m_colour;}
•         float GetRadius () const {return m_radius;}
•         bool  IsOn () const {return m_on;}

•         etc.
```

Get Methods

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

    void Clear ();

    void SetColour (const string &colour) {m_colour = colour;}
    bool SetRadius (float radius);
    void Switch ();

    void  GetColour (string &colour) const {colour = m_colour;}
    float GetRadius () const {return m_radius;}
    bool  IsOn () const {return m_on;}

    etc.
```

Objects are returned parameter-wise.

Get Methods

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

    void Clear ();

    void SetColour (const string &colour) {m_colour = colour;}
    bool SetRadius (float radius);
    void Switch ();

    void  GetColour (string &colour) const {colour = m_colour;}
    float GetRadius () const {return m_radius;}
    bool  IsOn () const {return m_on;}

    etc.
```

Methods that should not change the data are declared as const. This ensures that they *cannot* change the data.

Shallow versus Deep Copy

- The assignment operator, copy constructor and destructor must always be overloaded (written) for a class that has pointer data (data on the heap). [1]
    - If *any* of these 3 are needed, *all 3* are needed.
- To be safe, always write them but keep them empty for non-pointer data.
- This is because if you do not do so, the compiler will provide default versions for you.
- Such default versions may **not** do what you actually want them to do for pointer data. If there is no pointer data member, then the default versions are just fine – but see above concerning safety.
- For example, if you have a pointer in a class, the default versions would copy the value of the *pointer* itself, rather than make a copy of what the pointer is pointing to!
- The copying of a pointer instead of that to which it is pointing, is called a shallow copy. It results in one data being pointed to by more than one pointer.
- The copying of the contents of the memory to which it is pointing is called a deep copy.

Simple Pointer Class

- class Pointer // simple illustration only – not complete to demonstrate what happens if care is not exercised in //design when the advice that is provided earlier is not followed.
- {
- public:
-   Pointer () {m_ptr = NULL;} // nullptr is preferred instead of NULL
-   ~Pointer () {Clear ();}
-   void Clear ();
-   // Returns false if there is no memory available
-   bool Set (int number);
    - // friend shouldn't be here, but it is convenient for now to do convenient output. Convert it non-friend and non-member as an exc.
    - // a get method would be needed to make the conversion work.
-   friend ostream& operator << (ostream &ostr, const Pointer &pointer);
- private:
-   int *m_ptr; // it is an integer pointer. [1] [2]
- **// Has pointer data, so copy constructor and assignment operator is also needed along with the destructor**
- };

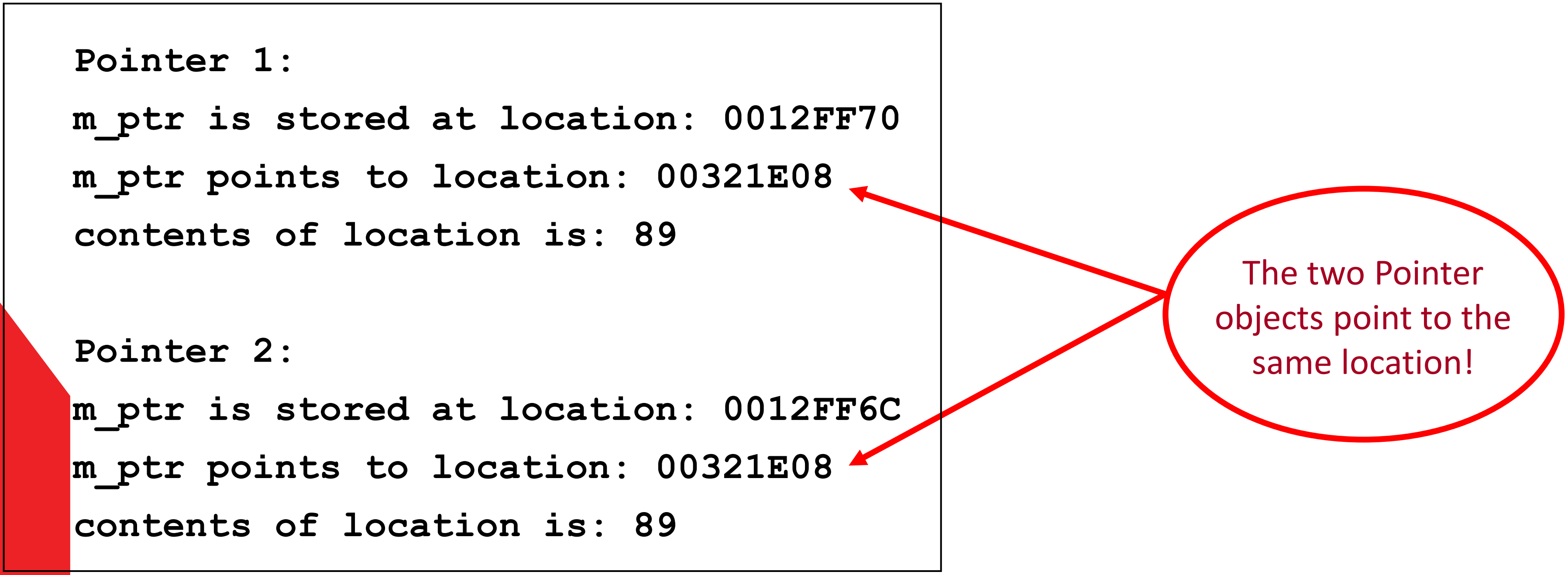
```
//---------------------------------------------------------------------

void Pointer::Clear ()
{
    if (m_ptr != NULL)
    {
        delete m_ptr;
    }

    m_ptr = NULL;

}
```

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

bool Pointer::Set (int number)
{
        if (m_ptr == NULL)
        {
             m_ptr = new int; // “new” creates the memory space (heap) to store the number value
        }

        if (m_ptr == NULL)  // no more heap memory available
        {
             return false;
        }
        else
        {
             *m_ptr = number; // copy the number value in the newly created heap memory

             return true;
        }
}
```

```
//-------------------------------------------------------------------------
// is declared friend, so direct access to private data member
// for debugging purposes only

ostream& operator << (ostream &ostr, const Pointer &pointer)
{
        ostr << "m_ptr is stored at location: " << &(pointer.m_ptr)
             << endl;
        ostr << "m_ptr points to location: " << pointer.m_ptr << endl;
        ostr << "contents of location is: " << *pointer.m_ptr << endl;

        return ostr;
}

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

```
int main()
{
     Pointer ptr1;
     Pointer ptr2;

     ptr1.Set (89);
     ptr2 = ptr1;

     cout << "Pointer 1:" << endl;
     cout << ptr1;
     cout << endl;

     cout << "Pointer 2:" << endl;
     cout << ptr2;
     cout << endl;

     return 0;
}
```

Output From Test Program

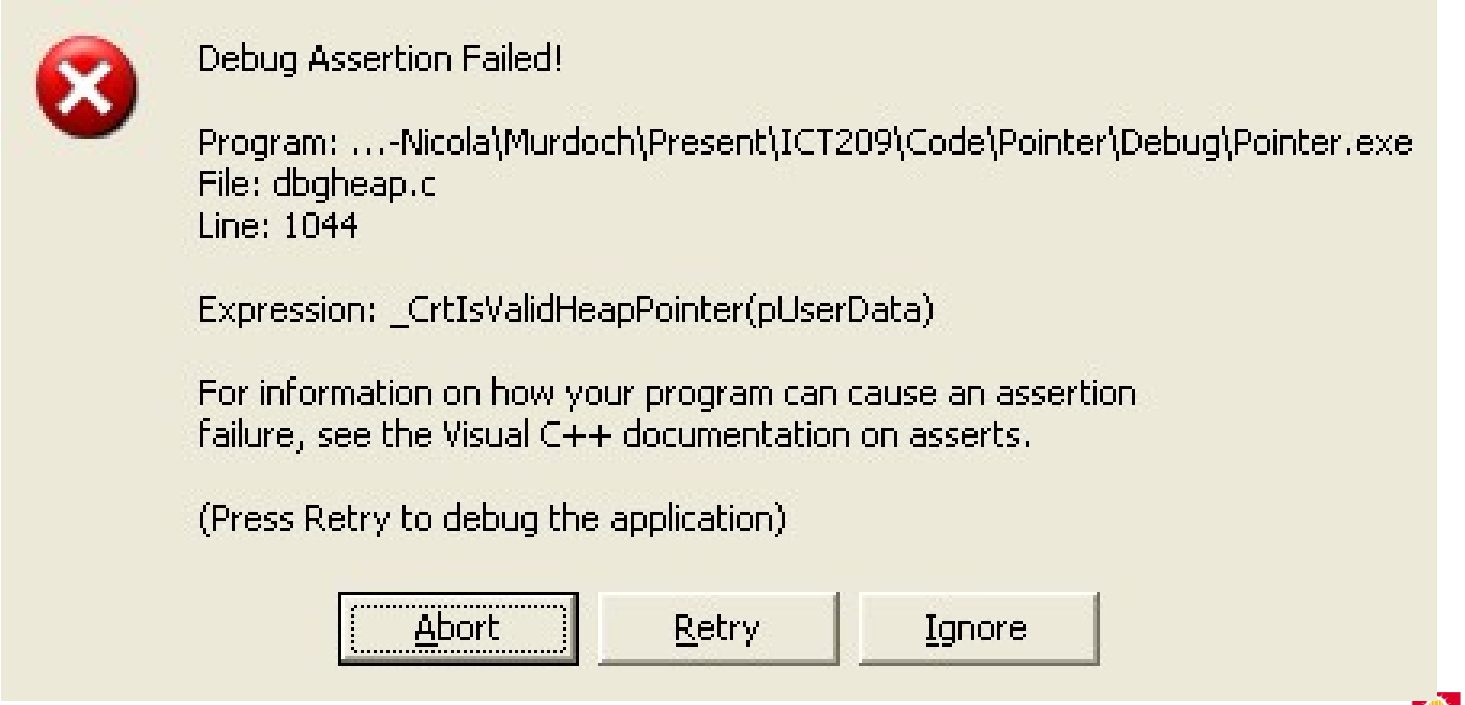


```
Pointer 1:
m_ptr is stored at location: 0012FF70
m_ptr points to location: 00321E08
contents of location is: 89

Pointer 2:
m_ptr is stored at location: 0012FF6C
m_ptr points to location: 00321E08
contents of location is: 89
```

The two Pointer objects point to the same location!

The Destructor

- Destructors are actioned in the opposite order to the construction of objects.
- Therefore in the test program, pointer2 destructs, followed by pointer1.
  - But in this case, it does not matter as the problem exists either way.
- When pointer2 destructs, it releases the memory to which it points.
- Unfortunately, when pointer1 destructs it tries to do the same thing, so we get:



Debug Assertion Failed!

Program: ...-Nicola\Murdoch\Present\ICT209\Code\Pointer\Debug\Pointer.exe
File: dbgheap.c
Line: 1044

Expression: _CrtIsValidHeapPointer(pUserData)

For information on how your program can cause an assertion
failure, see the Visual C++ documentation on asserts.

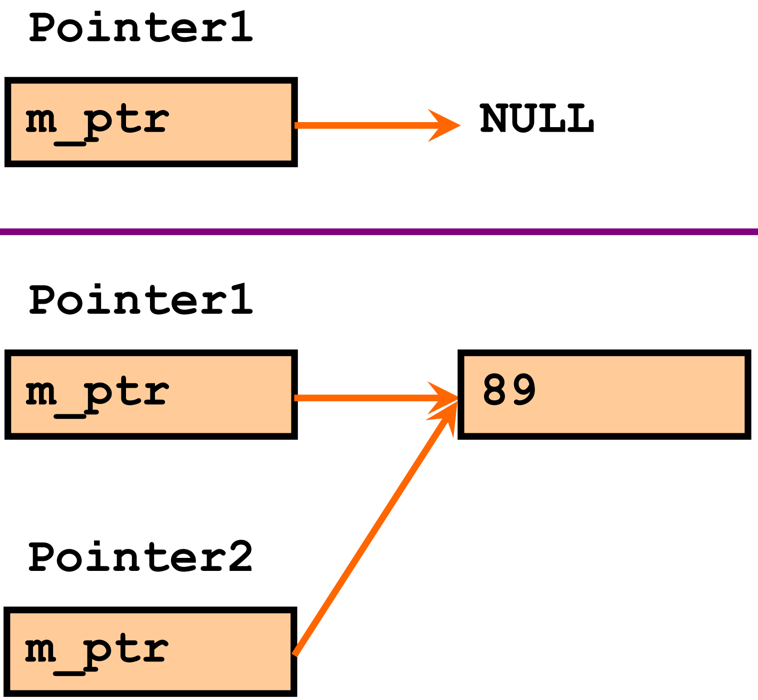
(Press Retry to debug the application)

Shallow Copy in Memory

```
Pointer pointer1;
```

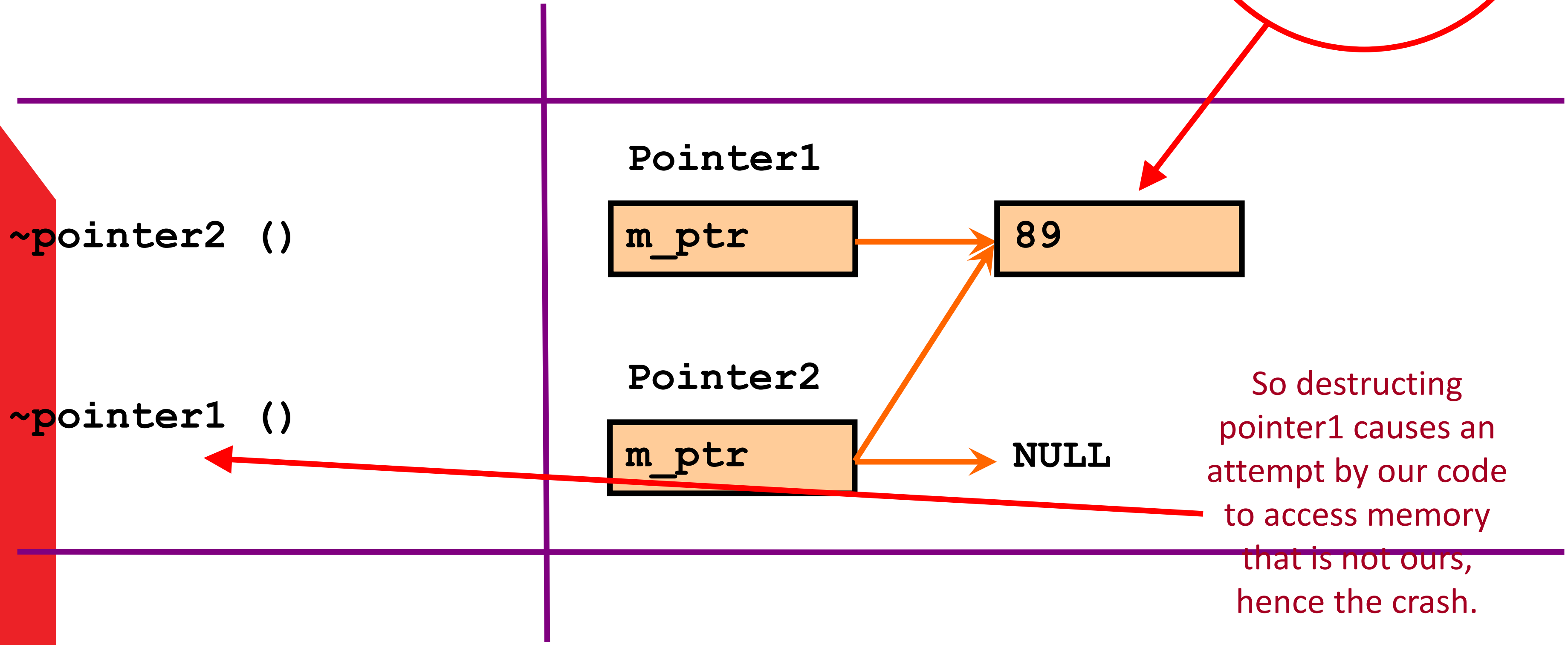
```
pointer1.Set(89);
```

```
pointer2 = pointer1;
```



END

Shallow Copy in Memory



The memory location is reclaimed by the OS, and no longer available to our program.

So destructing pointer1 causes an attempt by our code to access memory that is not ours, hence the crash.

Preventing a Shallow Copy

- You can ensure a deep copy by
  - Writing a copy method;
    - Calling it from the assignment operator;
    - Calling it from a copy constructor.

- OR
  - Privatising the copy constructor;
  - Privatising the assignment operator;
  - Thereby preventing the compiler from creating default versions.

Ensuring a Deep Copy

```
class Pointer
{
public:
        Pointer () {m_ptr = NULL;} //prefer nullptr

        Pointer (const Pointer &initialiser);

        ~Pointer () {Clear ();} // destructor prevents memory leaks

        void Clear ();

        bool Copy  (const Pointer &rhs); // [1] should be private or protected

        // Returns false if there is no memory available
        bool Set (int number);

         // Get method would be needed when converting to non-friend, non-member

        friend ostream& operator << (ostream &ostr, const Pointer &pointer); // covert to non-friend, non member

        Pointer& operator = (const Pointer &rhs);

private:
        int *m_ptr;
};
```

Copy constructor

Copy method not the copy constructor [1]

Overloaded assignment operator

Copy Constructor

- //---
- Pointer::Pointer (const Pointer &initialiser)
- {
-     m_ptr = NULL; // Set method needs this, constructor sets to null
-     Copy (initialiser);
- }

Copy () is then called, but note that we cannot return a value from a constructor, so if Copy () fails, we have no way of knowing in code.

Copy Method

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

bool Pointer::Copy (const Pointer &rhs)
{
    if (rhs.m_ptr != NULL) // what happens if you don’t check?
    {
        return Set (*(rhs.m_ptr)); // with rhs int data value
    }
    else
    {
        return false;
    }
}
```

Overloaded Assignment Operator

- //---
- Pointer& Pointer::operator = (const Pointer &rhs)
- {
-     Copy (rhs);
-     return *this;
- }

Like the constructor, it simply uses the Copy () method.

It is also similarly dangerous!

‘this’ is a pointer to the object itself.

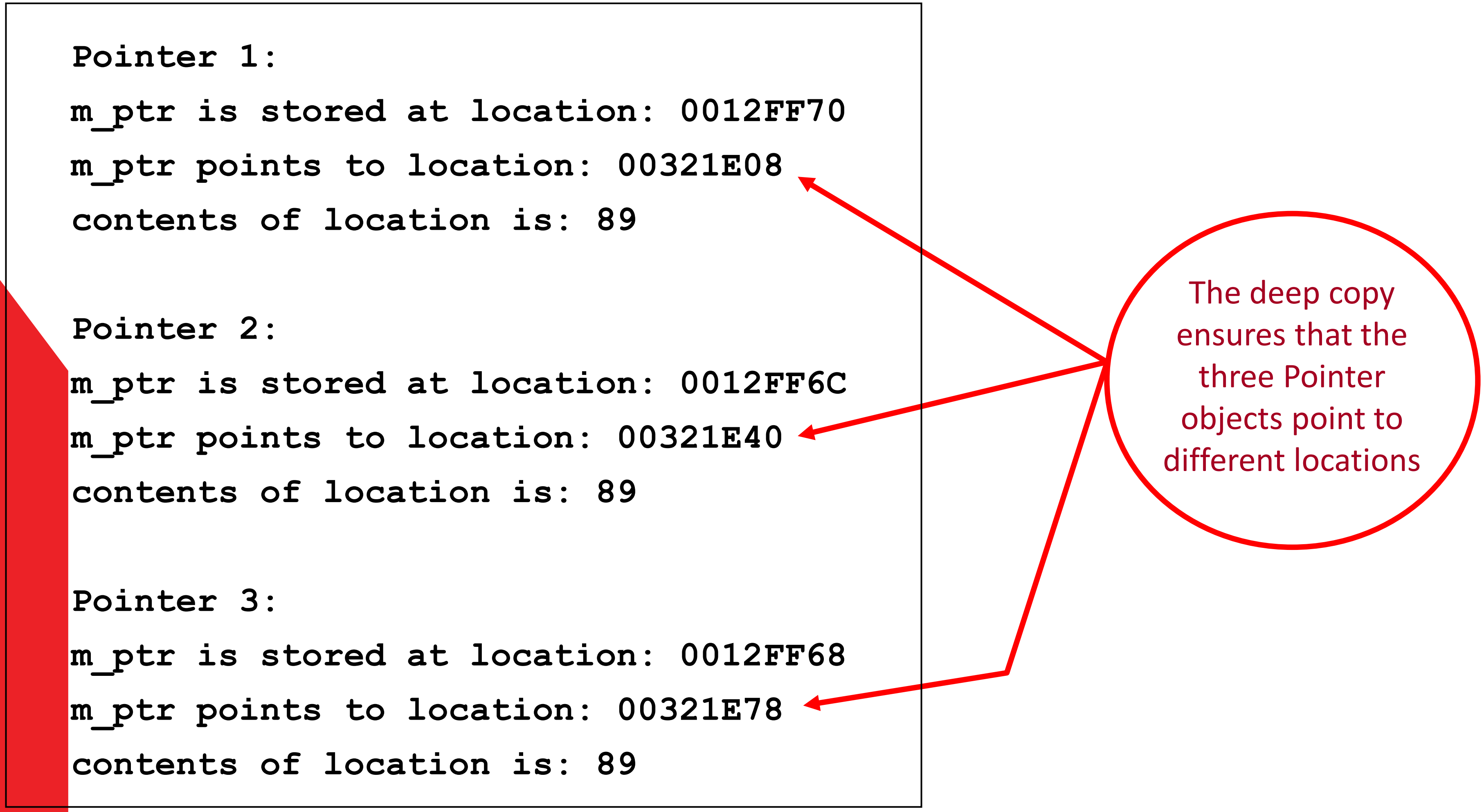
Therefore ‘*this’ is the contents of the object.

Returning it fulfills the requirements of the assignment operator.

```
int main()
{
        Pointer ptr1;
        Pointer ptr2;

        ptr1.Set (89);
        ptr2 = ptr1;

        Pointer ptr3 (ptr1);

        cout << "Pointer 1:" << endl;
        cout << ptr1;
        cout << endl;

        cout << "Pointer 2:" << endl;
        cout << ptr2;
        cout << endl;

        cout << "Pointer 3:" << endl;
        cout << ptr3;
        cout << endl;

        return 0;
}
```

These two statements now lead to deep copies of ptr1

Output From Test Program



```
Pointer 1:
m_ptr is stored at location: 0012FF70
m_ptr points to location: 00321E08
contents of location is: 89

Pointer 2:
m_ptr is stored at location: 0012FF6C
m_ptr points to location: 00321E40
contents of location is: 89

Pointer 3:
m_ptr is stored at location: 0012FF68
m_ptr points to location: 00321E78
contents of location is: 89
```

The deep copy ensures that the three Pointer objects point to different locations

Preventing Default Versions [1]

```
•       class Pointer
•       {
•       public:
•               Pointer () {m_ptr = NULL;}
•               ~Pointer () {Clear ();}

•               void Clear ();
•               bool Copy  (const Pointer &rhs) {return Set (*rhs.m_ptr);}

•               // Returns false if there is no memory available
•               bool Set (int number);

•               friend ostream& operator << (ostream &ostr, const Pointer &pointer);

•       private:
•               int *m_ptr;

•               Pointer& operator = (const Pointer &rhs) {return *this;}
•               Pointer (const Pointer &initialiser) {}; //=delete [2]
•       };
```

Privatised declarations prevent outside code using neither of the assignment operator nor the copy constructor. [2]

Preventing Default Versions

```
class Pointer
{
public:
        Pointer () {m_ptr = NULL;}
        ~Pointer () {Clear ();}

        void Clear ();
        bool Copy  (const Pointer &rhs) {return Set (*rhs.m_ptr);}

        // Returns false if there is no memory available
        bool Set (int number);

             / / convert to non-friend, non-member. Get method would be needed.
        friend ostream& operator << (ostream &ostr, const Pointer &pointer);

private:
        int *m_ptr;

        Pointer& operator = (const Pointer &rhs) {return *this;}  [1]
        Pointer (const Pointer &initialiser);
};
```

The test program will now be prevented from compiling.

Some Final Points

- Mutator/Set methods should set one piece of data only and should return a boolean to indicate success or failure.
- Call other methods rather than re-write code.
- Never do output in any method other than an output method. Data Structure classes do not have an output method. The use accessor/get methods.
- A data class should not do input from file/keyboard or output to screen/file.
- Make every class you write *minimal*: only include those methods that you know you need. See earlier notes about what constitutes minimal.

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Textbook: 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.

Murdoch
UNIVERSITY

Data Structures and Abstractions

Parameters and Inheritance

Lecture 14

Time Wasting Code

```
// Pointless program that does nothing!
int main()
{
    for (int index = 0; index < 10; index++)
    {
        Light light; // constructor used
        light = InputLight ();
    }

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

```
Light InputLight ()
{
    Light light; // constructor used

    float radius;
    cout << "Enter radius of light in centimeters: ";
    cin >> radius;
    light.SetRadius(radius);

    string colour;
    cout << "Enter colour of light: ";
    cin >> colour;
    light.SetColour (colour);

    return light; // copy constructor used
}
```

How Many Constructions?

Index

0

```
5: Light light;
```

Construction Count

1

How Many Constructions?

Index

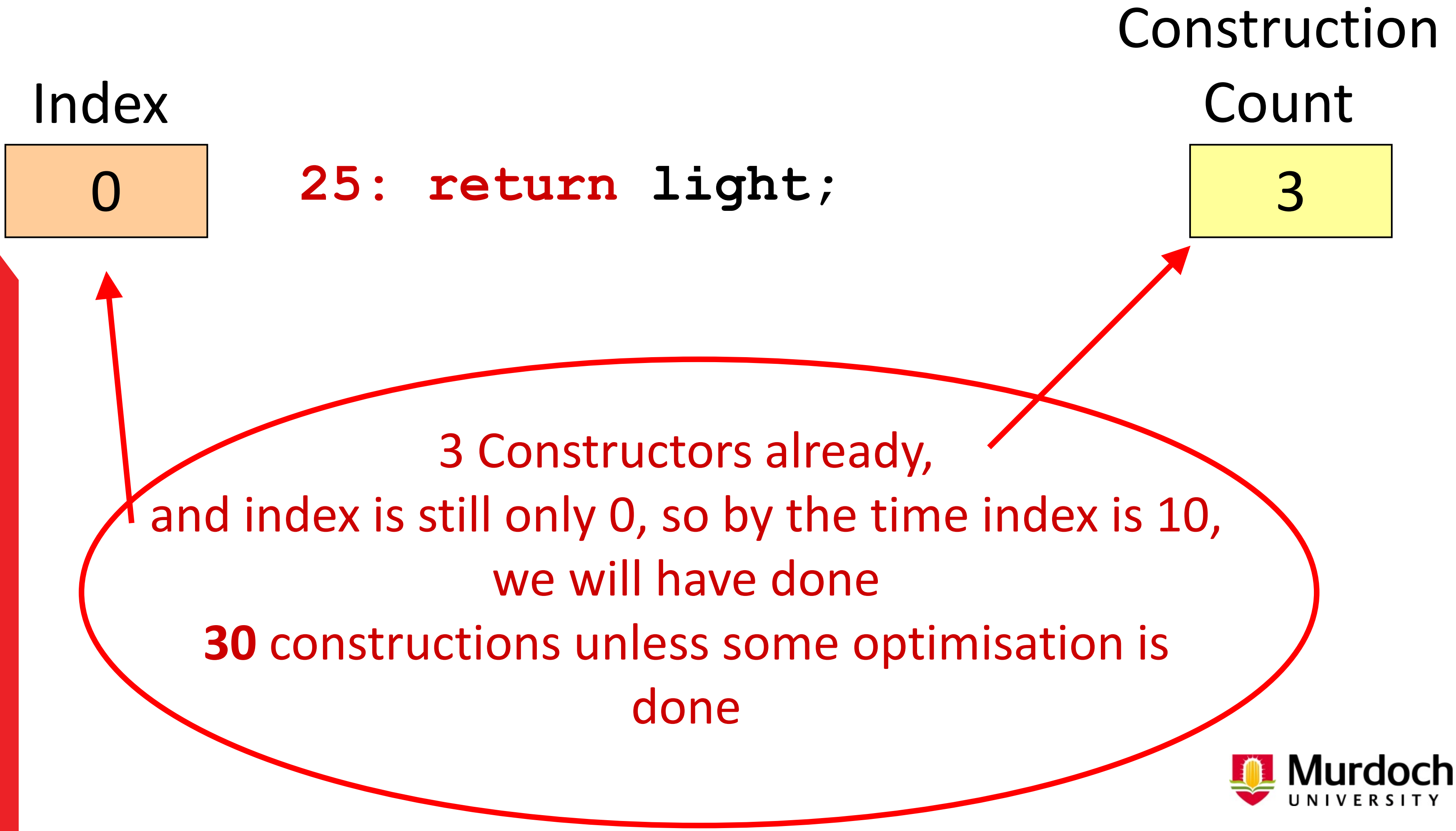
0

```
14: Light light
```

Construction Count

2

How Many Constructions?



Index

0

```
25: return light;
```

Construction Count

3

3 Constructors already,
and index is still only 0, so by the time index is 10,
we will have done
30 constructions unless some optimisation is
done

Good Code (refactored)

```
// Pointless program that does nothing!
int main()
{
    Light light;
    for (int index = 0; index < 10; index++)
    {
        InputLight (light);
    }

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

The constructor is now outside the loop.

The data is now returned as a parameter rather than function-wise.

```
void InputLight (Light &light)
{
    float radius;
    cout << "Enter radius of light in centimeters: ";
    cin >> radius;
    light.SetRadius(radius);

    string colour;
    cout << "Enter colour of light: ";
    cin >> colou
    light.SetCo
}
```

The data is now returned as a parameter rather than function-wise.

Therefore we do not need a local variable.

The 30 constructions is now reduced to only **one**!

Summary

- Construction of an object takes time.
- Therefore the more constructions, the slower the code.
- When returning an object by value function-wise, there is a hidden construction (copy construction) as the data is transferred back to the calling function.
- This extra construction time cost is exacerbated if it is placed within a loop with a local variable.
- Objects passed-by-value into a function also cause an extra construction – copy constructor used.

Rules

- It is important to follow the correct rules for parameter passing and function returning.

- The rules are designed to make your code as efficient and bug-resistant as possible.

- Failure to stick to these rules may cost you marks in assignments. You would also have spent time trying to fix poor code, so it is not worth ignoring the rules.

- *Rule: [1]* Simple types (int, float etc) are passed by either value or non const reference or returned function-wise:
  - Nothing is to be returned:

    ```
    void DoSomething (float num);
    ```

  - Change expected to the variable to be returned:

    ```
    void DoSomething (float &num);
    ```

  - Something is expected to be returned:

    ```
    float DoSomething (float num);
    ```

- *Rule:* Objects are always passed by reference
  - No change expected to light:

    ```
    void DoSomething(const Light &light);
    ```

  - Change expected to light:

    ```
    void DoSomething(Light &light);
    ```

Murdoch
UNIVERSITY

Inheritance

- Inheritance tends to get overused and badly used.

- However, there are occasions when it is both correct and useful.

- Inheritance is correct to use when:
  1. **the derived class (sub-class, child) “is a” parent class (super-class),**
  2. **the derived class requires all the data declared in the parent,**
  3. **the derived class uses every method defined in the parent.**

Examples

- The PatternLight described in one of the earlier lectures is a good example of correct use of inheritance:
  - PatternLight *is a* Light
  - PatternLight requires **`m_colour`**, **`m_radius`** and **`m_on`**
  - PatternLight will use all of Light's methods.

- A poor example would be Square inheriting from Rectangle:
  - Square *is a* Rectangle
  - BUT, Square does *not* need **`m_width`**, which would have been defined in Rectangle.

Protected Data

- For the Light class, we made all data private.
- Private data is protected from absolutely everything, including derived classes.
- Therefore when you derive a class, you need to alter the parent class so that the data is *protected* rather than private if you want derived classes to access parent data.
- Protected data is protected from view by the outside world, but available to derived classes.
- One can make all data protected as a rule, and then never have to go back and make changes. [1] But this is terrible. Only classes that are meant to be derived from should have "protected" specified.

Constructors and Destructors

- When you construct a class that is derived from another class, the default constructor of the parent class is automatically run before the constructor of the derived class. [1]
- However the destructor of the parent class is not automatically run.
- To ensure that it *is* automatically run, you need to add the '**virtual**' keyword in front of it (parent destructor) in the header file:

```
virtual ~Light ();
```

- Destructors are run in *reverse* order: the child class and then the parent class.

Murdoch UNIVERSITY

Virtual Methods

- If a method in the parent class is to be over-ridden in the child class, then it too is declared as virtual: [1]

```
virtual void DoSomething (); // parents
```

- If the child class wishes to access the parent class' version, then it uses the scope resolution operator:

```
void Child::DoSomething ()
{
  parent::DoSomething();// call parent's
  version.
}
```

Required Changes to the Light Class

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

    virtual void Clear ();
    //...

protected:
    // Any string is acceptable, we shall assume it is a colour
    string m_colour;
    // In centimetres
    float  m_radius;
    bool   m_on;
};
```

The destructor becomes 'virtual'

The Clear() operator becomes virtual

Data becomes 'protected' rather than private.

```
// A light that shines through a cutout giving it a particular shape
//
// Version
// 01 - Nicola Ritter
//
//---------------------------------------------------------------

#ifndef PATTERN_LIGHT
#define PATTERN_LIGHT

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

#include "../Light/Light.h"

//---------------------------------------------------------------
// Available shapes
//---------------------------------------------------------------

const int NO_SHAPE = 0;
const int LEFT_ARROW = 1;
const int RIGHT_ARROW = 2;
const int MAX_SHAPE = 2;
```

The Light header file must be included.

You can list as many shapes as you want, but make sure that MAX_SHAPE is changed to match the highest number

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

class PatternLight : public Light [1]
{
public:
        PatternLight () {Clear();}
        PatternLight (const PatternLight &plight);
        virtual ~PatternLight () {};

        void Clear ();

        bool SetShape (int shape);
        int  Get () const {return m_shape;}

        friend ostream& operator << (ostream &ostr, const PatternLight &light); [2]
        PatternLight & operator = (const PatternLight &plight);

private:
        int m_shape;

};

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

#endif
```

The Clear method overrides those in the Light class

Set and Get methods are only required for this class' attributes.

```
void PatternLight::Clear ()
{
    Light::Clear();
    m_shape = NO_SHAPE;
}
```

PatternLight calls the Clear() from the Light parent, before initialising its own attributes.

```
bool PatternLight::SetShape (int shape)
{
    if (shape >= NO_SHAPE && shape <= MAX_SHAPE)
    {
        m_shape = shape;
        return true;
    }
    else
    {
        return false;
    }
}
```

```
ostream& operator << (ostream &ostr, const PatternLight &light)
{
        ostr << static_cast<Light>(light);

        if (light.m_on && light.m_shape != NO_SHAPE)
        {
                ostr << ", showing ";
                switch (light.m_shape)
                {
                        case LEFT_ARROW:
                          ostr << "left arrow";
                          break;
                        case RIGHT_ARROW:
                          ostr << "right arrow";
                          break;
                }
        }

        return ostr;
}
```

The static_cast tells the compiler to redefine light as a Light instead of a PatternLight.

This line of code, therefore, runs the output code in the parent class.

Therefore all this method has to do is output information based on this class' attributes

Make sure you *only* use a static_cast when there is an inherit relationship between the two.

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Chapter on Inheritance and Composition.

Murdoch
UNIVERSITY

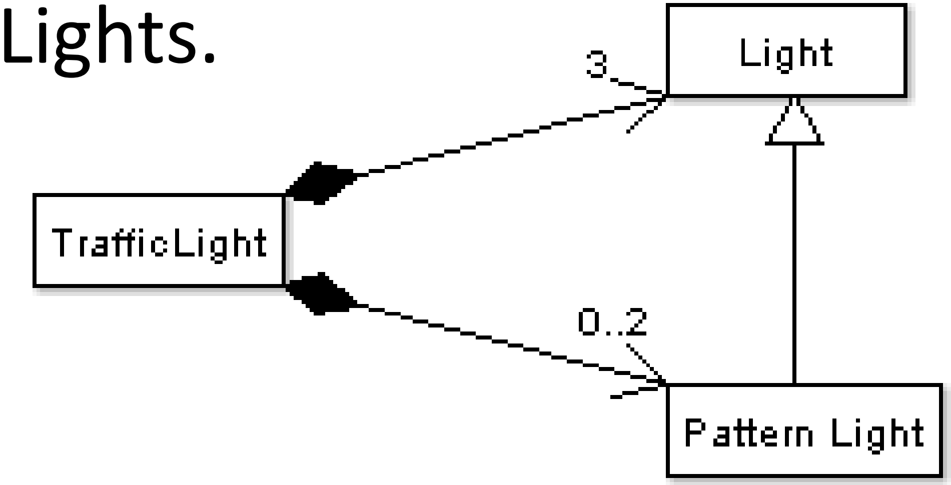
Data Structures and Abstractions

Composition, Aggregation & Templates

Lecture 15

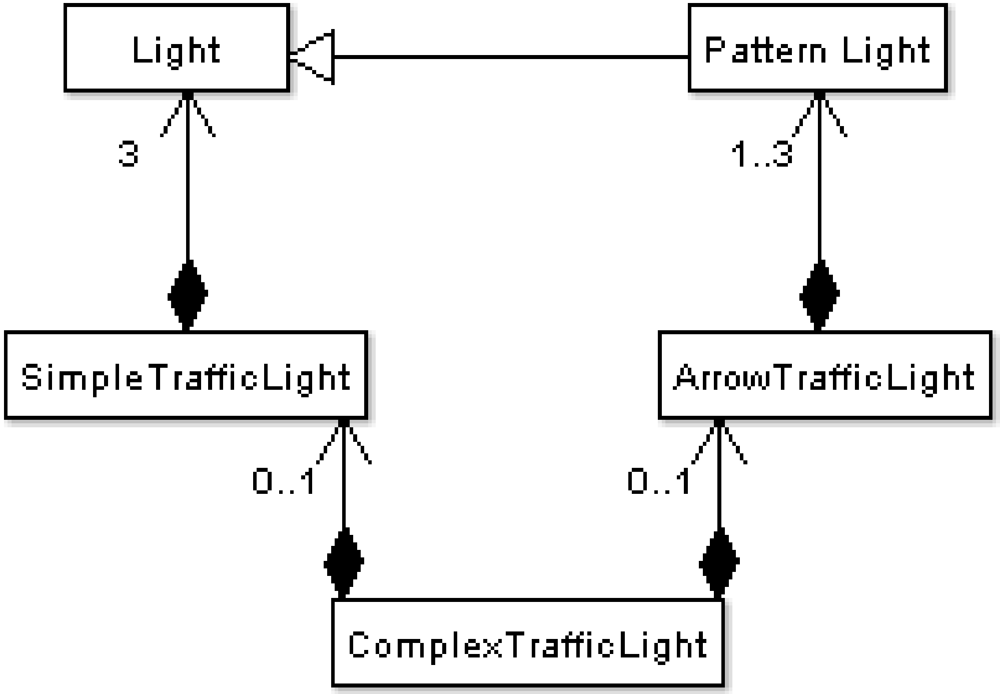
Composition

- Composition is where one class has a data member that is an object of another class.
- The example given in Lecture 11, was TrafficLight, which had three Lights and 0..2 PatternLights.



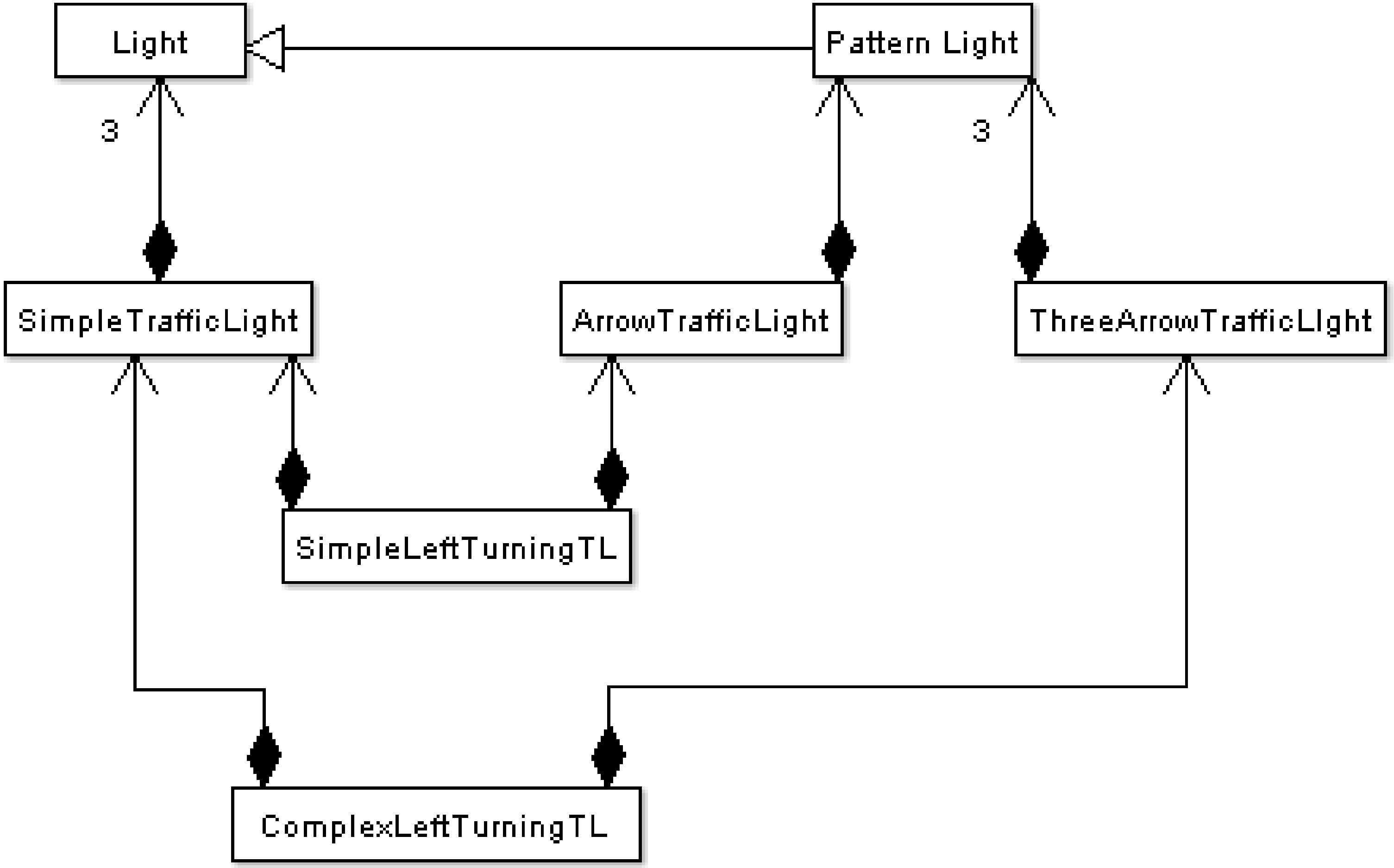
Design Change

- Lets say that after thinking about the problem a design change was needed as shown below.



Another Design Change

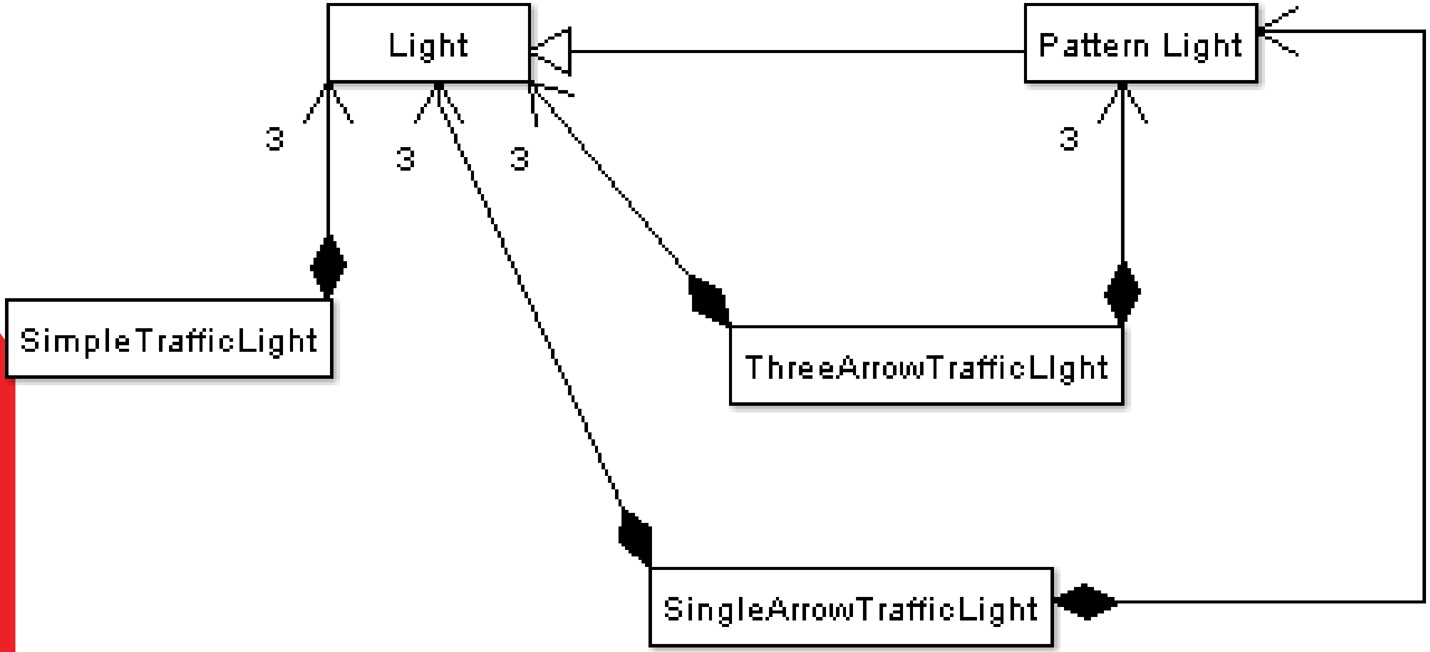
- And then when working on the algorithm, it was realised that the behaviour changed depending on the number of lights in a traffic light.
- This would have meant that within the code there would be lots of `if` statements to do with how many of each type of light.
- This was a sure sign that the design was still incorrect, so another change was made to the design.


Light
Pattern Light
3
3
SimpleTrafficLight
ArrowTrafficLight
ThreeArrowTrafficLIght
SimpleLeftTurningTL
ComplexLeftTurningTL

Yet Another Design Change

- However, after a while I hit problems again.
- The composition seemed forced and the whole thing very complicated: a sure sign it was still incorrect.
- So I went out for a drive to look at traffic lights in action.
- I quickly realised that a complex traffic light was not composed of a simple traffic light and a three arrow traffic light, it was actually composed of three single and three pattern lights.
- So my ‘final’ design was:

Murdoch University


Light
Pattern Light
3
3
3
3
SimpleTrafficLight
ThreeArrowTrafficLight
SingleArrowTrafficLight

Design Changes Continued

- Of course there would be lots more types of traffic light than just these. [1]
- And if I was really coding the *whole* thing, I might well decide that my original design (or something else entirely) was more correct.
- Which is the wonderful advantage of software engineering over conventional engineering: design need not be static but the most important lesson is when starting on a problem solving task, find out what is the real problem!!!
  - Don't try to just imagine what the problem is going to be.
  - Do some "leg work", talk to end users, as changes in design (even in software engineering) has costs associated with it.
- Remember:
  - Code incrementally.
  - If everything is becoming too complicated: you have almost certainly stuffed up the design.
  - Don't be afraid to change your design. Implementing a wrong design will make the solution useless.
  - Don't be afraid to question the design of others.
  - Refactor incrementally.
  - Test everything after every change.

Finite State Machines

- The traffic light classes are examples of Finite State Machines.
- An FSM has a limited set of states that are visited one after the other.
- It is not possible to have two states at once: you cannot have both a red and green light showing in a normal FSM. There are fuzzy FSMs but these are outside the scope of this unit.
- FSM are used in simulation and modelling of many industrial and mechanical processes. Even the compiler you are using to compile your code uses state machines.
- A single **`Change()`** method replaces all the **`Set()`** methods.
- A single **`StateIs()`** method replaces all **`Get()`** and output methods.
- The term **`Initialise()`** is used rather than **`Clear()`**, as it is only usually called at the start.

```
// Constants.h
// Required by several classes
//------------------------------------------------------------

#ifndef CONSTANTS
#define CONSTANTS

// Traffic light colours
const int RED_LIGHT = 0; [1]
const int ORANGE_LIGHT = 1;
const int GREEN_LIGHT = 2;

// FSM error state
const int ERROR_STATE = -1;

// Size of a traffic light
const float TL_RADIUS = 12.5; // cm

#endif
```

Constants that are required by multiple classes are put in a header file of their own.

```
// SimpleTrafficLight.h
// Comments and includes etc up here as per normal – use doxygen comments instead
// friend operator used for debugging. Design does not require it.
//--------------------------------------------------------------

const int STL_NUMBER = 3;

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

class SimpleTrafficLight
{
public:

        SimpleTrafficLight () {Initialise();}
        ~SimpleTrafficLight () {};

        // Initialise the class
        void Initialise ();

        // Change the light to the next state
        bool Change ();

        // Output the state – for debugging/demo purposes only.
        friend ostream& operator << (ostream &ostr, const SimpleTrafficLight &light);
private:
    int m_state;
    vector<Light> m_lights; [1]

    // Clear all old data
    void Clear ();

    // Set the light sizes and colours
    void InitialiseLights ();
};

#endif
```

```cpp
// SimpleTrafficLight.cpp
// Comments and includes as per normal
//-----------------------------------------------------------

void SimpleTrafficLight::Initialise ()
{
	Clear ();

	// Add the correct number of lights to the vector
	Light light;
	for (int index = 0; index < STL_NUMBER; index++)
	{
		m_lights.push_back (light); // think about this. Is it the same light?
	}

	InitialiseLights ();
}
```

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

void SimpleTrafficLight::Clear ()
{
    m_lights.clear();
    m_state = ERROR_STATE;
}

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

```
void SimpleTrafficLight::InitialiseLights()
{
        // Set the radii of the lights
        for (int index = 0; index < STL_NUMBER; index++)
        {
             m_lights[index].Set(TL_RADIUS);
        }

        // Set the colours
        m_lights[RED_LIGHT].Set("red");
        m_lights[ORANGE_LIGHT].Set("orange");
        m_lights[GREEN_LIGHT].Set("green");

        // Switch the red light on
        m_lights[RED_LIGHT].Switch();

        // Set the state
        m_state = RED_LIGHT;
}
```

```
bool SimpleTrafficLight::Change()
{
    switch (m_state)
    {
        case RED_LIGHT:
            m_lights[RED_LIGHT].Switch();
            m_lights[GREEN_LIGHT].Switch();
            m_state = GREEN_LIGHT;
            break;
        case GREEN_LIGHT:
            m_lights[GREEN_LIGHT].Switch();
            m_lights[ORANGE_LIGHT].Switch();
            m_state = ORANGE_LIGHT;
            break;
        case ORANGE_LIGHT:
            m_lights[ORANGE_LIGHT].Switch();
            m_lights[RED_LIGHT].Switch();
            m_state = RED_LIGHT;
            break;
    }

    return (m_state != ERROR_STATE);
}
```

```
// As an exercise convert this to non-friend, non-member operator
ostream& operator << (ostream &ostr, const SimpleTrafficLight &light)
{ [1]
        for (int index = 0; index < STL_NUMBER; index++)
        {
                string colour;
                light.m_lights[index].Get(colour);
                if (index == light.m_state)
                {
                   ostr << "O " << colour << endl;
                }
                else
                {
                   ostr << "o" << endl;
                }
        }

        return ostr; [2]
}
```

Simple Unit Test Program

```
int main() // not a complete unit test until you have code to test each method.
{
        SimpleTrafficLight light;

        cout << "Each time you press <Enter> the lights will change,"
                 << "use <Ctrl>-C to end the program." << endl;

        while (true)
        {
                cout << light << endl;
                light.Change();
                getchar(); // clunky!!
        }

        cout << endl;

        return 0;
}
```

Aggregation

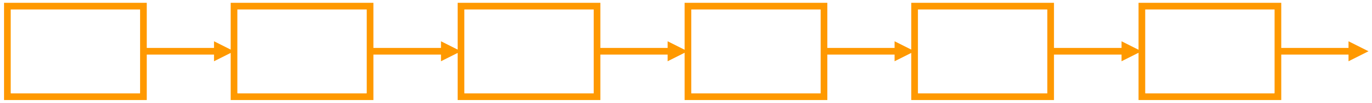
- Aggregation is used when a class has an attribute that is the object of another class, but it does not have control over the construction and destruction of that object.
- A common use of aggregation occurs in Windows programming, where many objects may want to refer to the current window, however none of them have the power to delete (close/destruct) the window.
- Similarly, a Unit class would be associated with a lecturer and students, but would not control them, so a Unit offering would have an aggregation of a Unit, lecturers and students, rather than a composition of them. Unit offering would be the class that contains all of these aggregations.
- Aggregation necessitates using an attribute that is either an index, reference or pointer to the aggregated object.
- Does aggregation actually exist at all, or is the class actually composed of a *pointer* or *reference?*
- In this unit we will not worry about the why's or wherefore's we will simply use and talk about aggregation as described above.

Linked Lists

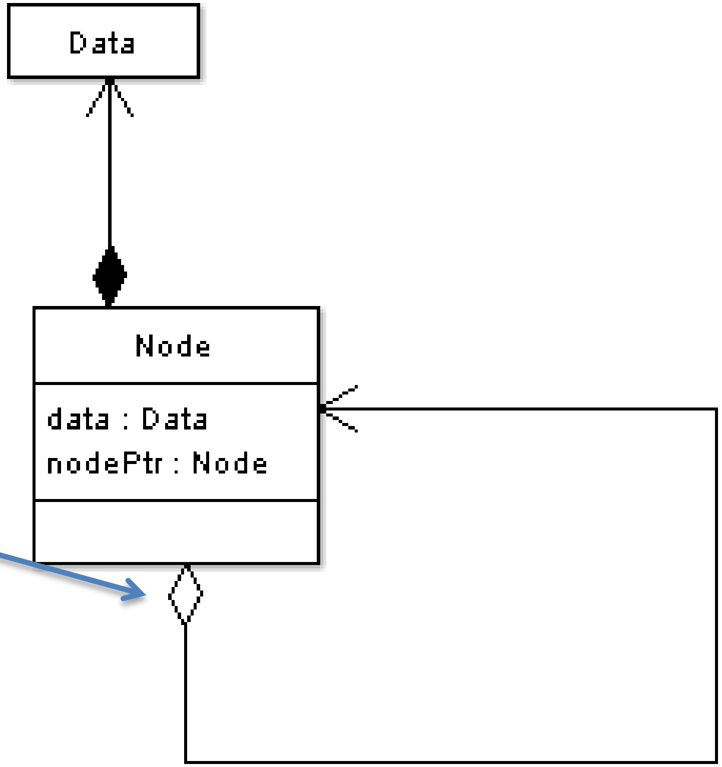
- A good example of aggregation is what occurs in a linked list.
- A linked list is exactly what it sounds like: each piece of data is combined with a link (pointer) to the next piece of data. This combination is called a *node*.
- In that situation the node class is composed of the data, but aggregates the next node.
- This is because if we delete/remove a node, it does not automatically delete the following node.
- We will cover lists again in a later lecture. For this topic, you need to know the basics.

Linked List Diagrams

Abstract View



UML Diagram



When you have this notation, **is Node responsible for deleting itself?**

- In C++, a simplified node class that stores a single integer piece of data, might look like this:

```
•    class Node [1]  // class or struct – think carefully
•    {
•      public:
•           Node () {m_next = NULL;} // or nullptr. Always initialise to null
•           ~Node () {} // [2]

•           Node *GetNext ();
•           void SetNext (Node *next) {m_next = next;}

•           int GetData () {return m_data;}
•           void SetData (int data) {m_data = data;}

•      private:
•           int m_data; // data is strongly associated with Node – see UML
•           Node *m_next;  // aggregation. Would the destructor delete this?
•    };
```

Templates <>

- The Node class is almost identical no matter what type of data is stored within it.
- Therefore, rather than re-write it every time we want to store a different data type, we use a *template.*
- Templates are *descriptions* of types which have to be instantiated with a particular type at run time.
- We have already used them when using the STL.
- A template node class would look like this:

```
template <class DataType> [1]
class Node
{
public:
      Node () {m_next = NULL;}
      ~Node () {}

      Node *GetNext ();
      void SetNext (Node *next) {m_next = next;}

      void Data (DataType &data) {data = m_data;}
      void SetData (const DataType &data) {m_data = data;}

private:
      DataType m_data; // same idea as before about relationship
      Node *m_next;
};
```

We tell the compiler it is a template. The type is now a parameter.

We use DataType to replace ‘int’ everywhere

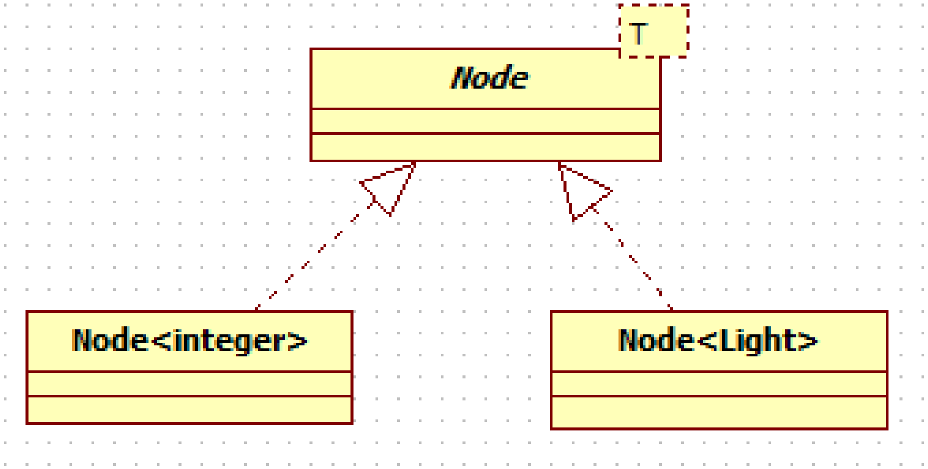
- Within our program, we then instantiate a node in the same way as with the STL:

```
typedef Node<int> IntNodeType; //IntNode is a type
IntNodeType intNode;
```

- IntNodeType is a type; in this case an integer Node class.
- The template Node can't contain anything as T is not bound to a type. Once bound to a type, it is ***realised***, and can be used in an application.
- There are limitations to templates:
  - They can only be used where all the methods and attributes are the same for every class.
  - They require all the code to go in the header file or else the code file has to be included! See textbook chapter on, Overloading and Templates for further discussion.
  - They can make the code *very* difficult to debug.
- They should be avoided except for very simple classes with only a few, clearly defined methods and attributes – generic classes.

In UML

- The box (with T) should have a dashed border. Arrows are diamond shaped. Lines are dashed. (Realisation [1] created in *StarUml* tool)



- The Realised types Node<integer> and Node<Light> can contain data.
- The template Node cannot contain data.

Exercise

- How would you the ***Law of Demeter [1]*** be applied when applied to objects that are composed of other objects (composition or aggregation)?
- Would there be situations where it would make sense to violate this law?

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Chapter on User-Defined simple data types, Namespaces and the string Type.
- Chapter on Inheritance and Composition.
  - You should go through the Programming example: Grade Report in the chapter on Inheritance and Composition.
- Entire chapter on Pointers, Classes, Virtual Functions, Abstract classes, and Lists.
- Chapter on Overloading and Templates.