Murdoch
UNIVERSITY

Data Structures and Abstractions

Minimal and Complete Class & Shallow vs Deep copy & Templating (Composite and Aggregation)

Lecture 5

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. [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;
}
```

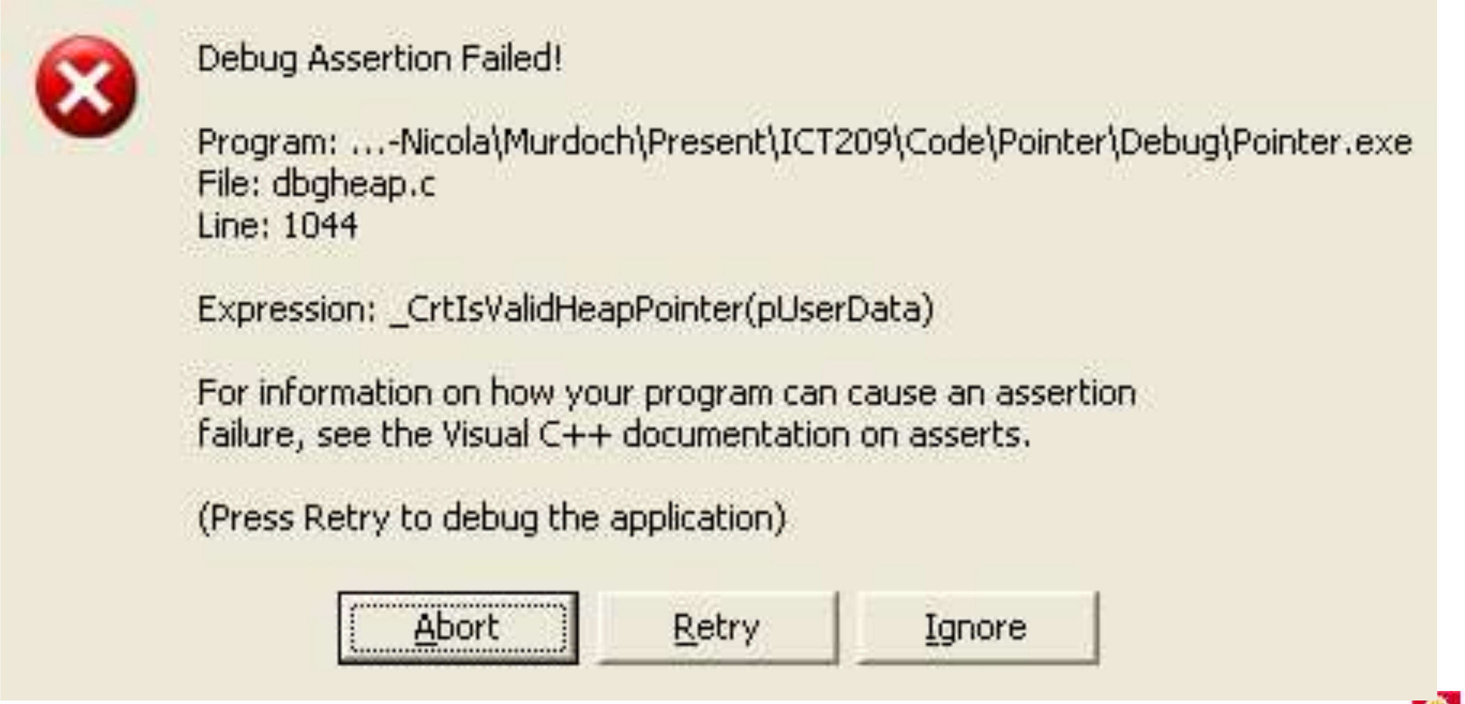
Output From Test Program

```
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:



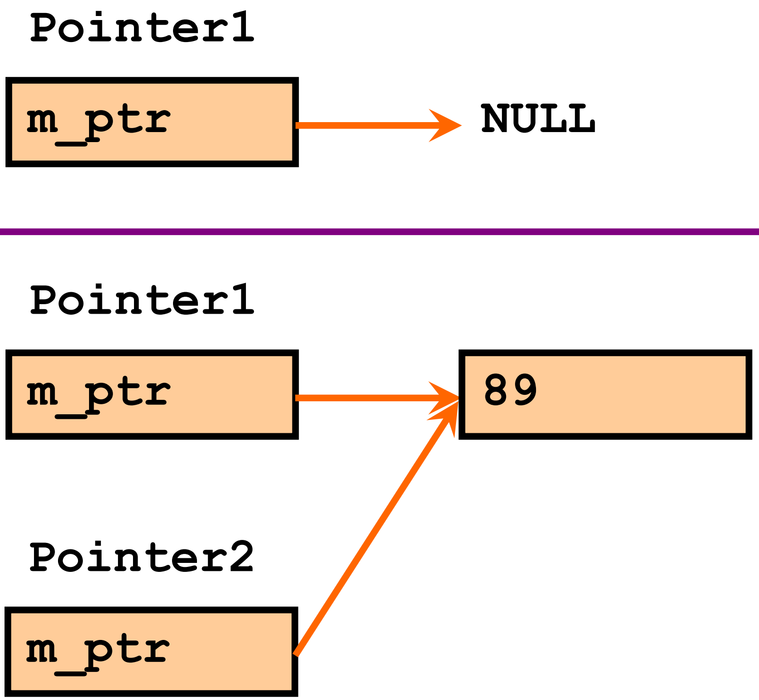
```
Debug Assertion Failed!

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;`

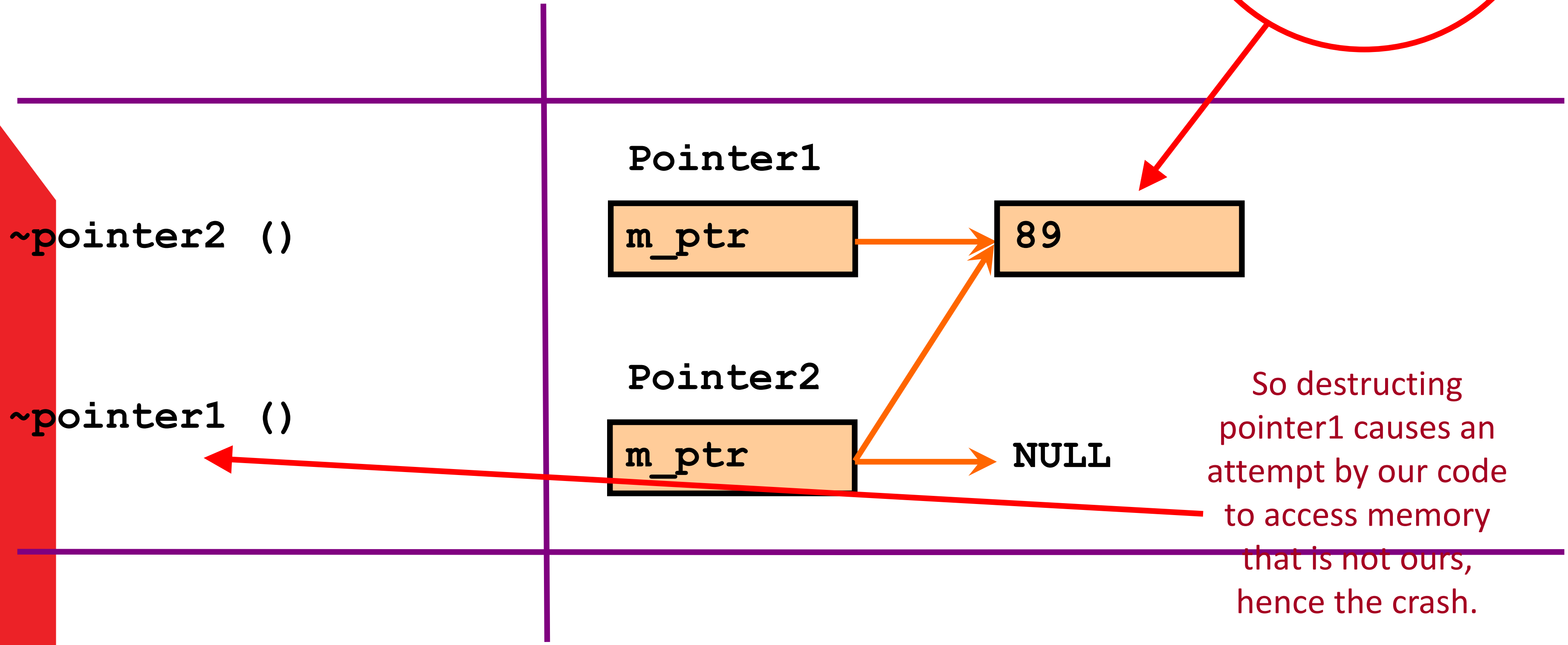
`pointer1.Set(89);`

`pointer2 = pointer1;`



END

Shallow Copy in Memory



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

`~pointer2 ()`

`~pointer1 ()`

Pointer1

`m_ptr`

`89`

Pointer2

`m_ptr`

`NULL`

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) {};
•	};
```

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

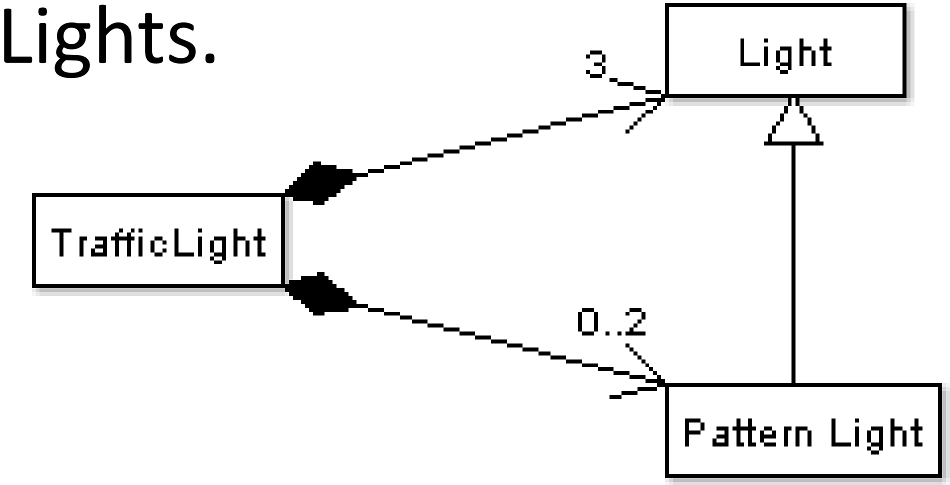
- Mutator/Set methods should set one piece of data only and should return a boolean to indicate success or failure.
- Call other methods rather than re-write code.
- Never do output in any method other than an output method. Data Structure classes do not have an output method. The use accessor/get methods.
- 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.

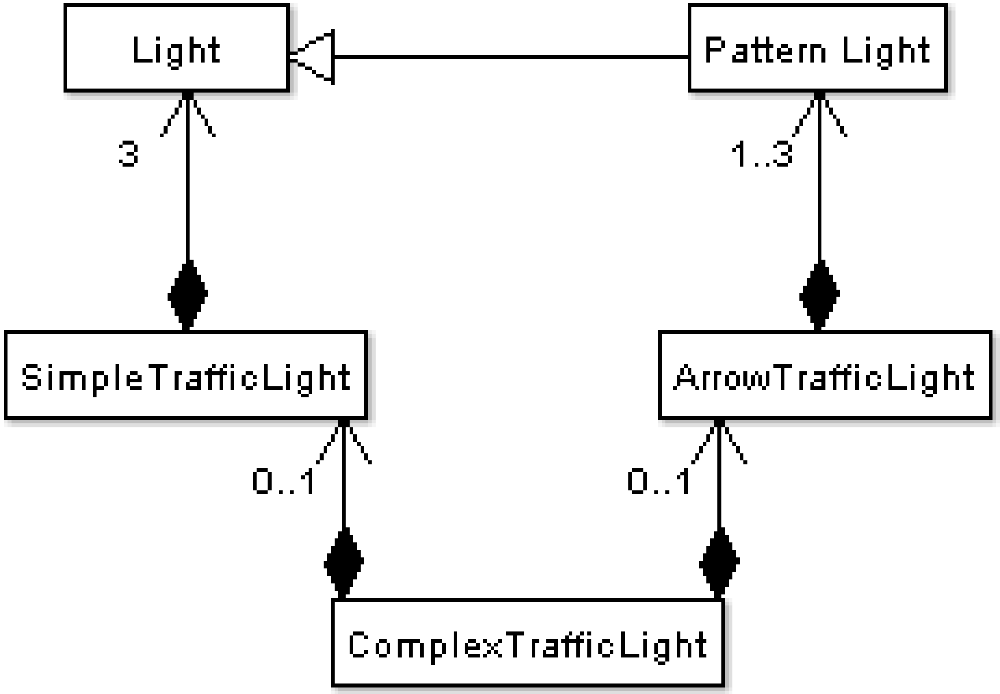
Composition

- 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.



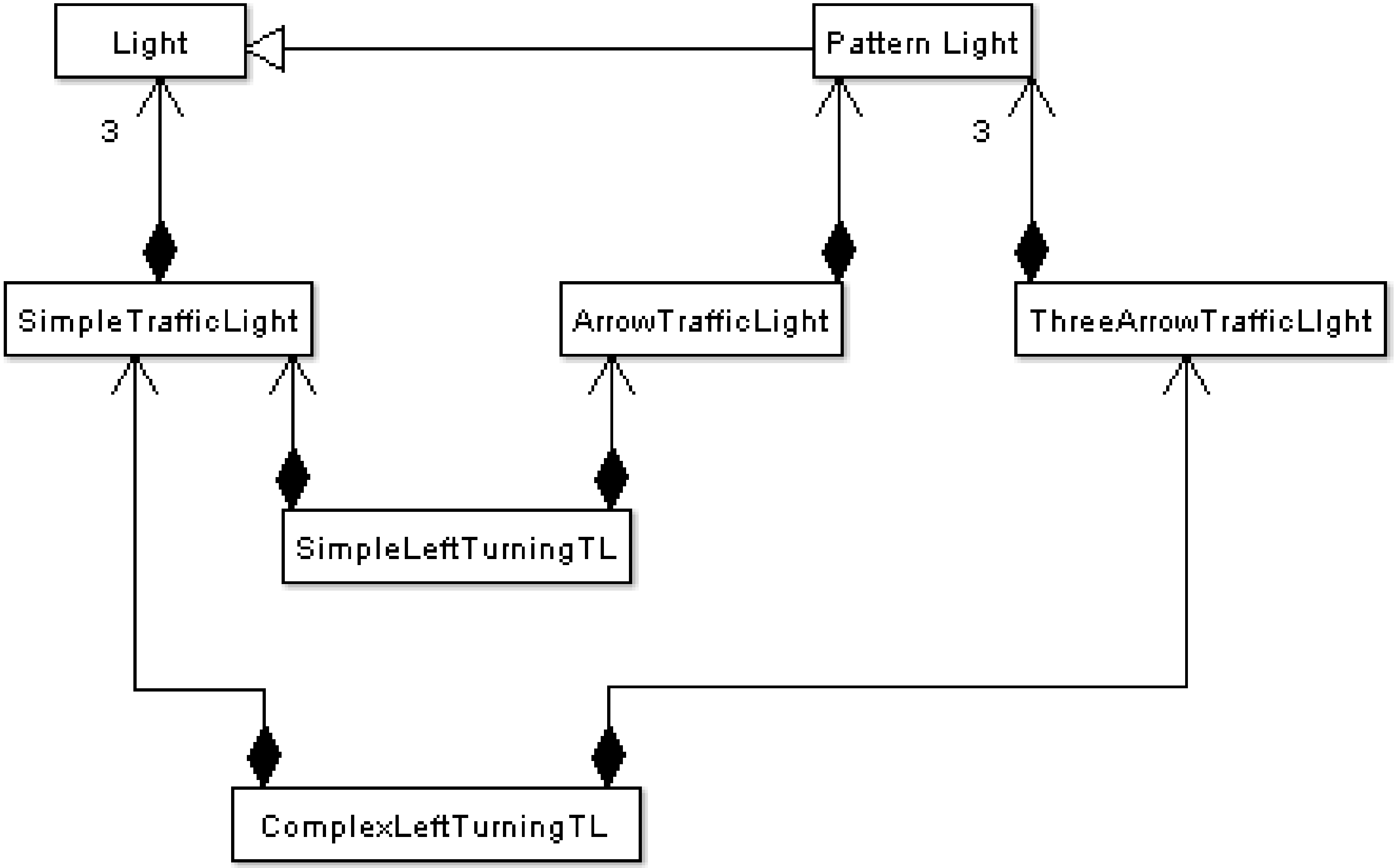
Design Change

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



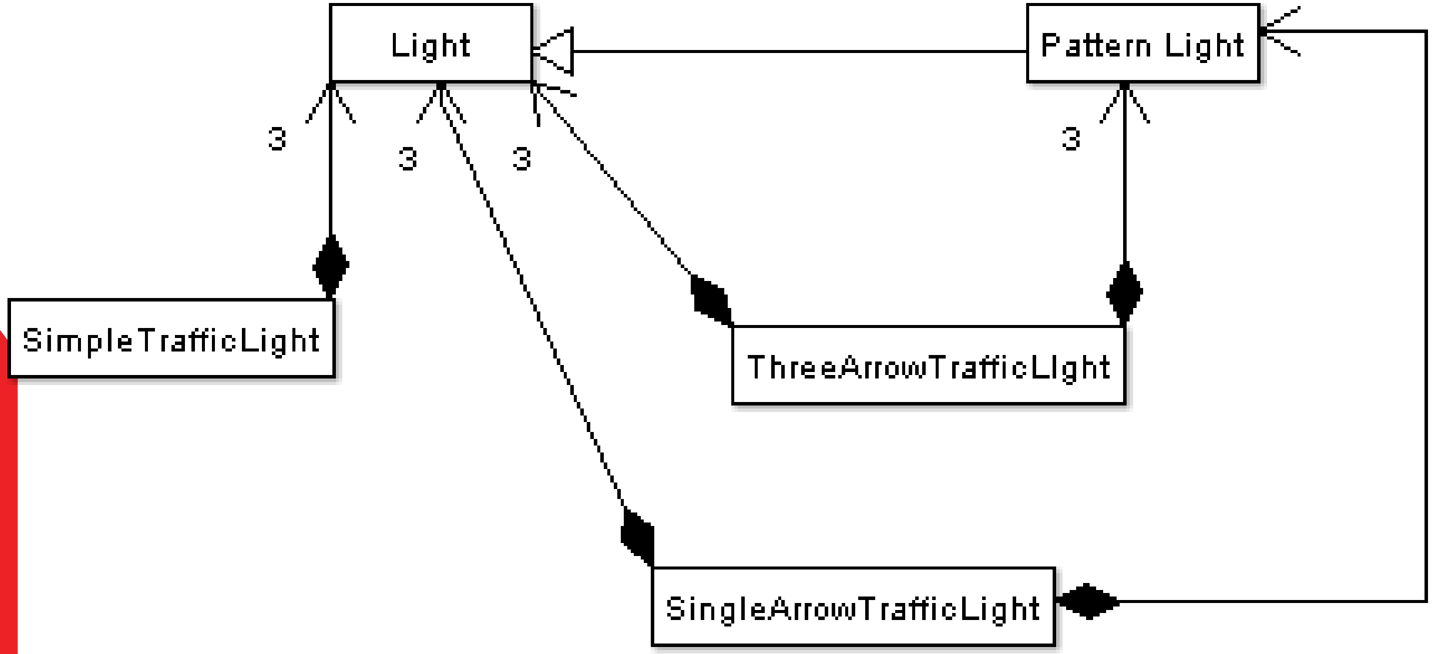
Another Design Change

- 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:


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
```

```
// 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

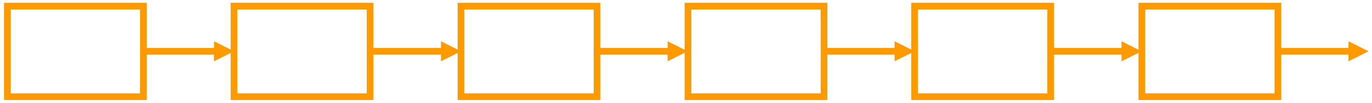
- Aggregation is used when a class has an attribute that is the object of another class, but it does not have control over the construction and destruction of that object.
- A common use of aggregation occurs in Windows programming, where many objects may want to refer to the current window, however 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

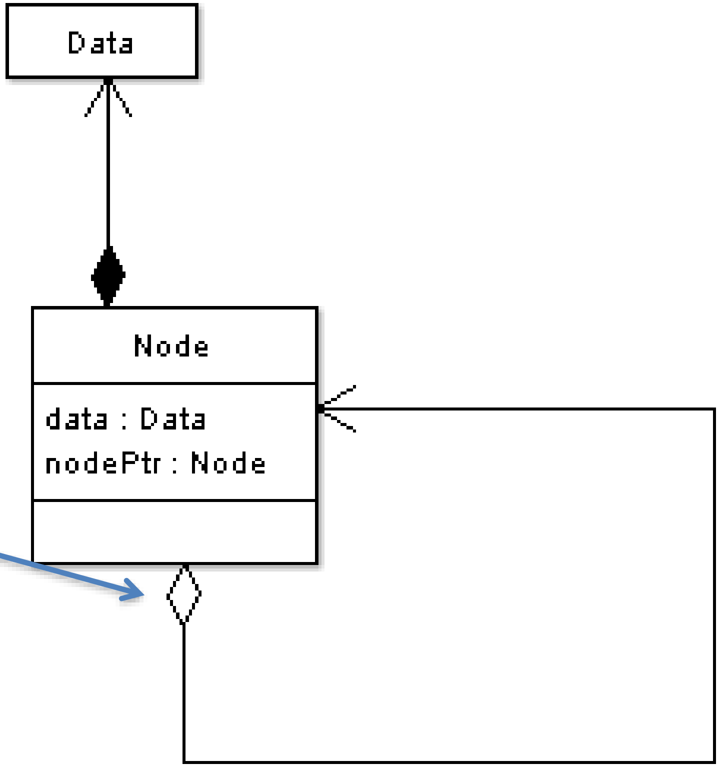
- A good example of aggregation is what occurs in a linked list.
- A linked list is exactly what it sounds like: each piece of data is combined with a link (pointer) to the next piece of data. This combination is called a *node*.
- In that situation the node class is composed of the data, but aggregates 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

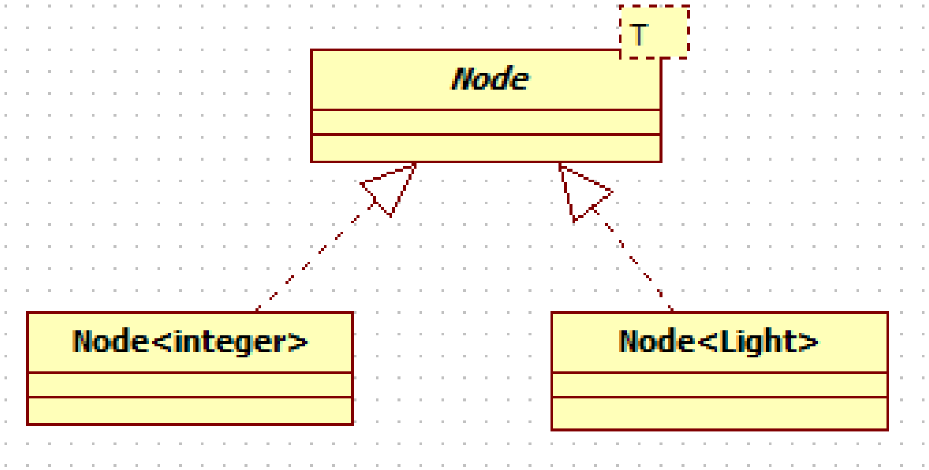
- Within our program, we then instantiate a node in the same way as with the STL:

```
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.

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);
    ```

Inheritance

- **Inheritance tends to get overused and badly used.**

- **However there are some times 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.

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.