A Request

Do ask questions if you are not sure of what I am presenting. This is complicated material. Do not be embarrassed if you do not understand everything. I am covering at least a semester’s worth of material (14x2 = 28 hours) in about eight 1-hour lectures! Sitting there confused wastes both your time and mine!

Talk Outline

1. A Review of last week’s lecture
   - Memory Allocation Issues and Pointers
   - Object Oriented Programming Basics
   - Inheritance and Class Hierarchies
2. The VTK Object Hierarchy – a guided tour
3. Reference Counted Memory Allocation/Deallocation
4. Object Factories

Variables and Arrays

- C++ has basic data types such as short, int, float, char, double etc.
- The statements
  
  ```
  int a;
  float b[10];
  double c[5][5];
  ```

  define a single integer a, an one-dimensional array of floats b: b[0] .. b[9] and a two-dimensional array of doubles c[0][0] .. c[4][4] (All array indices start at 0.)

- Both a, b and c are implicitly allocated and will be deallocated when the function containing them exits. Their sizes are fixed at compile time.

Dynamic Memory Allocation

- Dynamic allocation allows the creation of arrays whose size is determined at runtime (e.g. loading an image whose size can vary).
- It is one of the key to writing memory efficient programs.
- It is, arguable, also the biggest source of problems and crashes in most C++ code. Most of the problems are caused by:
  1. Accessing/Deleting arrays/objects before they are allocated and initialized.
  2. Accessing/Deleting arrays/objects after they have been already deleted
  3. Neglecting to delete arrays/objects
Pointers

- Dynamic Memory Allocation reserves portion of memory for a specific data structure (array or object).
- The allocation process returns the physical memory address of the data structure, and in turn the physical memory address is stored in a pointer variable.
- The type of the pointer variable indicates the kind of object that is being stored.
- The type T* represents a pointer to memory holding objects of type T e.g. float* represents a pointer to a memory block holding an array of floats (arrays can have length 1 or greater!!!)

Allocating/De-allocating Pointers

- Memory allocation is performed by the `new` command e.g.
  ```cpp
  int* a = new int;
  ```
- `a` is NOT an integer. It is a pointer which contains a memory location. To access the value stored at the memory location pointed by `a` use the de-referencing operator `*` e.g.
  ```cpp
  *a = 1
  ```
  // This sets the value of the integer whose location is stored in `a` to 1
- When `a` is no longer needed the memory can be released using the `delete` command e.g.
  ```cpp
  delete a;
  ```

Allocating/De-allocating Arrays of Pointers

- As before allocation is performed by the `new` command e.g.
  ```cpp
  int* a = new int[10]; // Allocate an array of 10 integers
  ```
- `a` is a pointer which contains the memory location of the first element of the array. To access the values stored at the memory locations use:
  ```cpp
  a[0] = 1; a[3] = 2; etc.
  ```
- When `a` is no longer needed the memory can be released using the `delete []` command e.g.
  ```cpp
  delete [] a;
  ```
- Do not use the `delete` operator to delete arrays, it causes lots of problems use `delete []`!!!

Class Design

Classes consists of:

1. Data members -- just like C structures
2. Methods – special functions which are embedded in the class

Members and methods can be declared as public which makes them accessible from outside the class or protected/private which makes them inaccessible from outside the class methods.

Basic Class Methods

Typical Methods include:

1. One or more constructors to allocate memory and initialize the class instance. [Default Available]
2. ONE destructor to release memory and destroy the class instance. [Default Available]
3. Methods to access/modify data members.
4. Methods to perform additional processing.
5. Possibly methods to identify the class.
6. Operator Overloading methods to change default behaviour of certain operators such as = + - e.g. if we have a matrix class and `a,b,c` are of type matrix*
   ```cpp
   a = b + c
   ```
   will simply add the pointer addresses
   ```cpp
   += can be redefined to make a = b + c true matrix addition
   ```

A simple image class

// The interface
class pimage {
  public:
    // constructor - called by the new command
    pimage(int width, int height, int depth);
    // destructor - called by the delete command
    ~pimage();
    // access functions
    float getvoxel(int i, int j, int k);
    void setvoxel(int i, int j, int k, float v);
  protected:
    // in-accessible from outside the class
    int getindex(int i, int j, int k);
    float* voxel_array;
    int dimensions[3];
};
Using the pimage class

```cpp
pimage* animage = new pimage(100,100,16);
pimage->setvoxel(10,10,20,3.0);
float a=pimage->getvoxel(10,10,19);
delete pimage;
```

Notes:
1. The way that the pimage is implemented internally has no bearing on the user who can only modify voxels using the getvoxel/setvoxel methods. The implementation could be changed to a three-dimensional array instead with no impact on the interface.
2. Pimage is allocated/de-allocated just like an internal data type such as a float but
   * The constructor is called to create the object.
   * The destructor is called upon object deletion.

Extending pimage

```cpp
//The interface
class p2image : public pimage {
 public:
 // added functionality
 void fill(float a=0.0);
};

//The implementation
void p2image::fill(float a)
{
 int size=dimensions[0]*dimensions[1]*dimensions[2];
 for (int i=0; i<size; ++i)
     voxel_array[i]=a;
}
```

p2image is derived from pimage. It inherits all the functionality of pimage and adds the member function fill. While, on the surface, it appears that we could have added the method fill directly to pimage instead, this is not always the case especially if the source code for pimage is not available (i.e. pimage is part of a pre-compiled class library such as VTK)

Inheritance

- Previous trivial example showed how classes can be extended by deriving new classes from them
- Derived or child classes can also override standard behavior by redefining member functions.
- Can have abstract base classes which simply define an interface and leave the implementation to derived classes.
- The combination of multiple layers of derived classes leads to class hierarchies.

Talk Outline

1. A Review of last week’s lecture
   - Memory Allocation Issues and Pointers
   - Object Oriented Programming Basics
   - Inheritance and Class Hierarchies

2. The VTK Object Hierarchy – a guided tour

3. Reference Counted Memory Allocation/Deallocation

4. Object Factories

The VTK Hierarchy

- With the exception of some auxiliary classes all VTK classes derive from vtkObject
- vtkObject provides a common interface for all functionality that is common to the entire hierarchy:
  * Class Name and Class Type
  * Reference Counting for Allocation/Deallocation
  * Debugging flags
  * Print Functionality
  * Callback Hook Functionality (Observers)
  * Modification / Update Handling
- All the interface comes with a default implementation – no pure virtual functions!
Example Hierarchy Branch

vtkProcessObject

- Adds functionality for updating a GUI for an ongoing process
- Defines methods GetProgress / UpdateProgress etc that are called during the execution of complex and time-consuming operations
- Improves upon the callback handling interface

Example Hierarchy Branch

vtkSource

- Parent class of all sources and filters (a filter is simply a source with input(s))
- Defines GetOutput / GetOutputs
- Provides functionality for pipeline synchronization
- Key Methods:
  - vtkDataObject * GetOutput (int idx)
  - virtual void Execute ()
  - virtual void Execute Information ()

Example Hierarchy Branch

vtkImageSource

- Parent class of all sources and filters (a filter is simply a source with input(s)) which output images (vtkImageData)
- Overrides GetOutput / GetOutputs to output the specific data type vtkImageData as opposed to the more generic type vtkDataObject
- vtkSource
  - vtkDataObject * GetOutput (int idx)
- vtkImageData
  - vtkImageData * GetOutput ()
  - vtkImageData * GetOutput (int idx)
Example Hierarchy Branch

vtkImageGaussianSource

Provides specific functionality to create an image that is a sampled form of the Gaussian function:

Redefined from vtkObject
virtual const char * GetClassName ()
virtual int IsA (const char *type)
void PrintSelf (ostream &os, vtkIndent indent)
Size of Gaussian Image
void SetWholeExtent (int xMin, int xMax, int yMin, int yMax,
int zMin, int zMax)
Location of Center
virtual void SetCenter (float, float, float)
virtual void GetCenter (float &, float &, float &)
Function Specification
virtual void SetMaximum (float)
virtual void SetStandardDeviation (float)

Parallel Hierarchy Branches

vtkImageSinusoidSource

Provides specific functionality to create an image that is a sampled form of a sinusoid function

Redefined from vtkObject
virtual const char * GetClassName ()
virtual int IsA (const char *type)
void PrintSelf (ostream &os, vtkIndent indent)
Size of Gaussian Image
void SetWholeExtent (int xMin, int xMax, int yMin, int yMax,
int zMin, int zMax)
Specific Parameters for a sinusoid image
void SetDirection (float, float, float)
virtual void GetDirection (float &, float &, float &)
virtual float GetPeriod ()
virtual float SetAmplitude (float)
virtual float GetAmplitude ()

A Guided Tour of VTK

• The VTK class hierarchy is huge.
• We present some branches that are particularly pertinent.
• Full documentation can be found on-line (www.vtk.org) and in the VTK User’s Manual.

The vtkDataObject Branch
An aside: static members

- A static member or function of an object is a member that belongs to the class type and NOT to the specific instance.
- Consider the class circle below

```cpp
//INTERFACE
class circle {
  public:
    circle(float r);
    float getradius();
    static getnumberofcircles();
  protected:
    float radius;
    static int numberofcircles;
};
```

- The numberofcircles constructs belong to the circle class AS A WHOLE and NOT to any particular instance of it.

```cpp
//IMPLEMENTATION
int circle::numberofcircles = 0;
circle::circle(float r) {
  radius = r;
  numberofcircles = numberofcircles + 1;
}
circle::getradius() { return radius };
circle::getnumberofcircles() { return numberofcircles; }
```

Accessing the circle class

```cpp
int a = circle::getnumberofcircles(); // 0
circle* c1 = new circle(3.2);
int a1 = circle::getnumberofcircles(); // 1
float d1 = c1->getradius(); // 3.2
circle* c2 = new circle(11.1);
int a2 = circle::getnumberofcircles(); // 2
float d2 = c2->getradius(); // 11.1
```

- Static member functions can access protected functions of the class. They can be called even if no instance of the class exists.

VTK Constructors/Destructors

- In VTK classes constructors and destructors are protected!
- This results in preventing direct object allocation/deallocation, using the C++ new/delete commands!!!
- Instead object allocation/de-allocation occurs via the static member functions New() and Delete()
- This insertion of this level of indirection allows for the use of reference counted allocation/de-allocation techniques.
- In addition through the use of Object Factories, the static member New() may return an instance of a derived class instead that is more appropriate – see vtkRenderWindow example.
Reference Counted Memory Allocation/Deallocation

We create an object (allocate memory) using the static member New() as:

```c
vtkConeSource* cone = vtkConeSource::New();
```

This New() function calls the constructor of the cone class and returns the newly created object.

When the object `cone` is created it has a reference count of 1.

Every time an object is used its reference number is increased by 1. After execution of:

```c
Subd->SetInput(cone->GetOutput());
```

as the subd object increases the reference number of cone to ensure that it exists.

When subd is deleted or no longer needs cone it decreases its reference number by 1.

Calling the delete method e.g.

```c
cone->Delete();
```

also decreases the number of references by 1. When the number of references is equal to zero the memory is then released. Hence calling delete does not always result in immediately deleting the object.

Talk Outline

1. A Review of last week’s lecture
   - Memory Allocation Issues and Pointers
   - Object Oriented Programming Basics
   - Inheritance and Class Hierarchies
2. The VTK Object Hierarchy – a guided tour
3. Reference Counted Memory Allocation/Deallocation
4. Object Factories

Object Factories

1. Powerful mechanism of ensuring the appropriate class being created (often unknown to the user – i.e. behind the user’s back!!)
2. Can be used to create correct and efficient multi-platform code.
   - Consider the `vtkRenderWindow` class. The static member function `vtkRenderWindow* vtkRenderWindow::New()` is implemented such that it returns an instance of the appropriate derived class of `vtkRenderWindow` depending on the current platform via the use of lookup tables in the appropriate object factory. The user always manipulates the class as if it were a `vtkRenderWindow` and hence the user code is platform independent!!
3. Object Factories have other powerful features which are beyond the scope of this seminar!

The `vtkRenderWindow` Branch

Next Week

- First examples of actual VTK extensions.
- The use of CMAKE to organize projects and to ensure proper compilation of programs and libraries.
- Using TCL to test our VTK extensions written in C++.
Homework

• Please ensure that you have access to a computer with:
  – VTK version 4.0 or greater with Tk/Tk 8.3
  – Cmake version 1.2 or greater
  – A working C++ compiler (gcc or Visual C++)

• These exist on most of our Unix workstations at the lab both Linux and SGI's