Programming for Image Processing/Analysis and Visualization using The Visualization Toolkit

Week 2: A quick introduction to intermediate C++

http://noodle.med.yale.edu/seminar/seminar.html

Xenios Papademetris
papad@noodle.med.yale.edu
BML 325, 5-7294

Schedule – Part 2

1. Review of Part 1 and Course Overview
2. C++ Pointers/Classes, Object Oriented Programming
3. Adding new VTK Commands/Cmake
4. Image-to-image filters/ surface to surface filters
5. Case Study I -- Iterative Closest Point surface matching
6. Case Study II – A Simple Segmentation Algorithm

Talk Outline

1. C++ a better C
2. Memory Allocation Issues and Pointers
3. Programming Styles: Procedural vs Object Oriented
4. Object Oriented Programming Basics
5. Inheritance and Class Hierarchies

C++ : A Better C

- C++ is almost completely a superset if C.
- It adds data abstraction and support for object oriented programming
- It does not force any particular programming style upon users (e.g. procedural vs OOP)
- It is constrained by the goal of supporting C-style programming (unlike languages with no legacy support such as Java).
- C++ compilers are now fairly mature (unlike 7-8 years ago)

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

Dynamic Memory Allocation II

- Modern garbage collection techniques can eliminate a lot of these problems (e.g. Java)
- Reference counted allocation/de-allocation techniques can be a big help (these are used by VTK)
- Really important to have a grasp of what pointers are and how they function.

Pointers

- Dynamic Memory Allocation reserves of 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 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.

```
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.

```
*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.

  delete a;

Allocating/De-allocating Arrays of Pointers

- As before allocation is performed by the `new` command e.g.

```
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:

```
a[0] = 1 ; a[3]=2; etc.
```

- When `a` is no longer needed the memory can be released using the `delete []` command e.g.

  delete [] a;

- Do not use the **`delete`** operator to delete arrays, it causes lots of problems use **`delete []`** !!!

Aside: Two-dimensional Arrays

- Two-dimensional arrays are created as an array of one-dimensional arrays: e.g. a 5x10 matrix could be stored as:

```
// allocate five rows of floats
float** matrix=new float*[5];
for { int i=0;i<=4;i++ } {
  matrix[i]=new float[10];
}
```

float** `a` is a pointer to an array of pointers of type float*.

- The array can be accessed in the usual way e.g. `matrix[0][2]=2.0`
- Deallocation is tricky and messy:

```
for { int i=0;i<=4;i++ } {
  delete [] matrix[i];
}
delete [] matrix;
```

Two-dimensional Arrays II

- Often two-dimensional arrays are simulated using one dimensional arrays e.g.

```
float* matrix=new float[5*10];
Matrix[10*row+column]=5;
```

- This is more cumbersome to use but avoids all issues with multi-dimensional pointers with respect to allocation/de-allocation.
- All VTK-arrays internally are stored as one-dimensional arrays.

Aside: Pointers and Arrays

- There are some further advanced/useless/dangerous operations with respect to manipulating arrays via pointers and pointer arithmetic
- It is beyond the scope of this lecture.
- Such techniques are extensively used where efficiency is critical (e.g. lots of internal VTK code uses pointer arithmetic)
- It is highly recommended that one avoids using these unless absolutely necessary.

Talk Outline

1. C++ a better C
2. Memory Allocation Issues and Pointers
3. Programming Styles: Procedural vs Object Oriented
4. Object Oriented Programming Basics
5. Inheritance and Class Hierarchies

Procedural Code

"Decide which procedures you want;
Use the best algorithms you can find"
(Stroustrup 1994)

- Still the most common programming paradigm
- Emphasis on the processing and the algorithms
- Key language features: passing arguments to functions and returning values from functions
- Offer ability to encapsulate algorithms and separate interface from implementation.

Interface vs Implementation

Interface – Utility.h

```
#include <math.h>
float deg2rad(float t);
```

Implementation – Utility.cpp

```
#inlude "utility.h"
float deg2rad(float t)
{
        return t*M_PI/180.0;
}
```

- We can later change the implementation of deg2rad without affecting any programs that call the function. The algorithm for degrees to radians conversion is encapsulated within the implementation of deg2rad

Data Abstraction and Object-Oriented Programming

- Encapsulation is taken one step further by defining new types of data objects (e.g. an image) complete with special operations (functions) that operate on these user defined types.
- The underlying data object is hidden from the programmer, and is only accessible via a selected set of operations visible to the outside.
- Programming design focuses on the creation of the appropriate objects and their interaction.

Abstract Data Types (Vermeir 2001)

An abstract data type consists of:

- A publicly accessible interface specifying which operations are available to inspect of manipulate a data object of that type.
- A hidden implementation that describes:
  - How the information associated with the object of the abstract data type is internally represented.
  - How the interface operations are actually implemented

- C++ implements Abstract Data Types as Objects (Classes)

Talk Outline

1. C++ a better C
2. Memory Allocation Issues and Pointers
3. Programming Styles: Procedural vs Object Oriented
4. Object Oriented Programming Basics
5. Inheritance and Class Hierarchies

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 identity 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**
   a=b+c will simply add the pointer addresses
   =,+ 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-accesible from outside the class
   int getindex(int i,int j,int k);
   float* voxel_array;
};
```

Using the pimage class

```
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.

The Implementation I

```
// constructor
pimage::pimage(int width,int height,int depth)
{
   dimensions[0]=width;
   dimensions[1]=height;
   dimensions[2]=depth;
   int tsize=dimensions[0]*dimensions[1]*dimensions[2]
   voxel_array=new float[tsize];
   for (int i=0;i<tsize;i++)
       voxel_array[i]=0.0;
}

pimage::~pimage()
{
   delete [] voxel_array;
}
```

The Implementation II

```
// access functions
float pimage::getvoxel(int i,int j,int k)
{
  int index=getindex(i,j,k);
  return voxel_array[index];
}

void pimage::setvoxel(int i,int j,int k,float v)
{
  int index=getindex(i,j,k);
  voxel_array[index]=v;
}

// internal function getindex()
int pimage::getindex(int i,int j,int k)
{
  return k*dimensions[0]*dimensions[1]+j*dimensions[0]+i;
}
```

Extending pimage

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

//The implementation
p2image::void fill(float a)
{
 for (int i=0;i<tsize;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)

Talk Outline

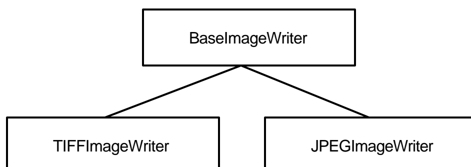
1. C++ a better C
2. Memory Allocation Issues and Pointers
3. Programming Styles: Procedural vs Object Oriented
4. Object Oriented Programming Basics
5. Inheritance and Class Hierarchies

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.

A Proper Class Hierarchy example – saving images to a file

- This is loosely based on vtkImageWriter.



BaseImageWriter

This is an abstract parent class and it provides functionality common to the specialized classes TIFFImageWriter and JPEGImageWriter

```
// Sample Interface
class BaseImageWriter {
       public:
       // Constructor and Destructor
       BaseImageWriter();
       virtual ~BaseImageWriter();

       // Methods
       virtual void SetInput(vtkImageData* input)l;
       virtual void SetFilename(char* filename);
       virtual void Write()=0; // No Implementation for Write

       protected: // Internal Stuff
       char* Filename;
       vtkImageData* Input;
};
```

TIFFImageWriter and JPEGImageWriter

These are derived from BaseImageWriter and provide concrete implementations of the Write() function e.g.

```
class TIFFImageWriter : public BaseImageWriter {
       public:
       virtual void Write();
};

void TIFFImageWriter::Write() {
       // code to save Input image to Filename as TIFF
}

class JPEGImageWriter : public BaseImageWriter {
       public:
       virtual void Write();
};

void JPEGImageWriter::Write() {
       // code to save Input image to Filename as JPEG
}
```

Adding more functionality to JPEGImageWriter

We could also override the SetFilename method to ensure a .jpeg extension e.g.

```
class JPEGImageWriter : public BaseImageWriter {
       public:
       virtual void SetFilename(char* filename);
       virtual void Write();
       virtual void SetJPEGCompression(float c);

       protected:
       float compression;
};

void JPEGImageWriter::SetJPEGCompression(float c) {
   compression=c;
}
void JPEGImageWriter::SetFilename(char* filename) {
       // Check if filename ends in .jpeg
       // else remove extension and add .jpeg extension
       // Copy to Filename
}
```

Whereas TIFFImageWriter accepts the default SetFilename functionality of BaseImageWriter, JPEGImageWriter modifies it to achieve a specific objective.

Type Issues -- Polymorphism

•Consider the function:

```
void SaveImage(vtkImageData* image,
               BaseImageWriter* writer,
               char* filename)
{
   writer->SetFilename(filename);
   writer->SetInput(image);
   writer->Write();
}
```

The writer argument is specified as of type BaseImageWriter. This results in a pointer of either TIFFImageWriter or JPEGImageWriter being able to be passed to the the SaveImage function. Any methods not defined in the interface of BaseImageWriter (e.g. SetJPEGCompressionFactor) are not available from writer as the SaveImage function does not know that they exist!!!

Some Additional Comments

- Parent classes define the common interface for derived classes
- Parent classes can also define default implementations of some of the methods of the interface.
- Methods which are meant to be overridden by derived classes are marked as virtual.
- Methods for which the parent only provides an interface and not an implementation are pure virtual methods – they must be overridden by the derived class.
- Classes containing pure virtual methods cannot be instantiated. Only derived members of this classes can be instantiated!

"Homework"

- Get a good C++ book and read about Object – Oriented stuff.
- In particular make sure you undestand
  - Inheritance
  - Virtual functions
  - Private/public/protected designations
  - Pointer allocation/de-allocation
- Next week we will look at the structure of VTK – which is based on the Object Oriented Design Paradigm.