Schedule – Part 2

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

Scheduling

No seminar next week as a number of people will be going to IPMI.

Talk Outline

1. A Review of last week’s lecture
2. Some additional C++ Constructs
   • Macros
   • Templates
3. Adding procedural code to VTK
4. CMAKE and project management

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, arguably, 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.
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 [] !!!

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.

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.

Parallel Hierarchy Branches

The vtkDataObject Branch
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:

```cpp
tvtkConeSource* cone = vtkConeSource::New();
```

This New() function calls the constructor of the cone class and returns the newly create 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

```cpp
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. 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.

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.

Talk Outline

1. A Review of last week’s lecture
2. Some additional C++ Constructs
   - Macros
   - Templates
3. Adding procedural code to VTK
4. CMAKE and project management

Macros

- Compilation is a multi-stage process
- The first step is called the "pre-processing stage"
- The pre-processor parses all directives beginning with "#" such as:
  ```
  #include
  #define
  #ifdef, #elsif #endif
  #define
  ```
- Most all of the above directives define shorthand for code to be either always or conditionally inserted at a specific point.

Macros II

- Macros are open to abuse and their use is minimized in C++ (unlike C) as the language has proper constructs for handling things previously implemented as macros.

```cpp
#define IMG_SIZE 20
char image[IMG_SIZE][IMG_SIZE];
```

- The preprocessor literally does a search/replace operation and all IMG_SIZE references get replaced by 20 before the code is passed on to the compiler proper.
- Macros tend to reduce program readability.
VTK and Macros

- VTK uses certain macros to help with the parsing of the C++ header files and the creation of wrapper code for TCL/Java/Python.

- The most common are:

  ```
  vtkSetMacro(member, type);
  vtkGetMacro(member, type);
  ```

  e.g. `vtkSetMacro(Length, float)` results in
  ```
  void SetLength(float f) { Length=f; }
  ```
  and `vtkGetMacro(Length, float)` results in
  ```
  float GetLength() { return Length; }
  ```

- `vtkSetObjectMacro(object, type)` and `vtkGetObjectMacro(object, type)` are used to access/modify pointer objects derived from `vtkObject`.

Templates

- Templates are an advanced programming feature which enable the use of generic programming. They enable the definition of ‘generic’ functions and classes.

- For the most part these are beyond the scope of this seminar.

- VTK uses templates in its implementation but NOT in its interface.

Templated Functions

- Consider a function that takes two arguments and returns the maximum of the two e.g.

  ```
  float maxval(float a, float b) {
  if (a>b) return a;
  return b;
  }
  ```

- For each type such as float, int, double etc. we need to write similar functions `maxval` by simply replacing “float” with the appropriate type. This is time-consuming and error-prone.

- Old style solution would be to use macros. These bypass proper compiler type-checking and can cause problems.

- C++ solution is to implement `maxval` as a templated function.

Templated Functions II

- The templated version of the function `maxval` takes the form:

  ```
  template <class T>
  T maxval(T a, T b) {
  if (a>b) return a;
  return b;
  }
  ```

- For any type that the operator `>` is defined `maxval` is a valid function e.g.
  - `maxval(3,4)` will return an integer 4
  - `maxval(3.0,4.0)` will return a float 4.0

- We can also have templated classes (e.g. the C++ Standard Template Library STL). Templated classes are heavily used in the Insight Toolkit (www.itk.org) which is closely related to VTK.

Templated Functions III

```
#include <stdio.h>

template <class T>
T maxval(T a, T b) {
  if (a>b) return a;
  return b;
}

int main(int argc, char** argv) {
  int a=maxval(2,7);
  float b=maxval(3.0,9.0);
  fprintf(stderr,"a=%d b=%.2f\n",a,b);
}
```

Talk Outline

1. A Review of last week’s lecture
2. Some additional C++ Constructs
   - Macros
   - Templates
3. Adding procedural code to VTK
4. CMAKE and project management
Adding procedural code to VTK

- Even in an object-oriented setup procedural code is still needed for simple functions such as e.g. Degrees to radians conversions.
- In C++ such code can be added as normal.
- To accommodate the limitations of the VTK parsers for scripting languages such as TCL all code that is to be accessible from a scripting language must be placed in a class derived directly or indirectly from `vtkObject`.
- As in Java, procedural code is then added as static members of a class (which may only have static members and is like a function container class).

Example: `vtkpxMath.h`

```c++
#ifndef __vtkpxMath_h
#define __vtkpxMath_h
#include "vtkObject.h"
class vtkpxMath : public vtkObject
{
public:
  static vtkpxMath *New();
  vtkTypeMacro(vtkpxMath,vtkObject);
  // Conversion Routines
  static float DegreesToRadians(float d);
  static float RadiansToDegrees(float r);
protected:
};
#endif
```

Example: `vtkpxMath.cpp`

```c++
#include "vtkpxMath.h"
#include "vtkObjectFactory.h"
 vtkpxMath* vtkpxMath::New()
{
  // First try to create the object from the vtkObjectFactory
  vtkObject* ret = vtkObjectFactory::CreateInstance("vtkpxMath");
  if(ret)
    return (vtkpxMath*)ret;
  // If the factory was unable to create the object, then create it here.
  return new vtkpxMath;
}
float vtkpxMath::DegreesToRadians(float d) {
  return d*3.1415/180.0;
}
float vtkpxMath::RadiansToDegrees(float r) {
  return r*180.0/3.1415;
}
```

Talk Outline

1. A Review of last week’s lecture
2. Some additional C++ Constructs
   - Macros
   - Templates
3. Adding procedural code to VTK
4. CMAKE and project management

CMAKE

- This example is derived from the vtkMy example from the VTK 4.0 source tree
- Directory Structure
  - `Base_dir/lib` : contains common library code e.g. `vtkpxMath.h` `vtkpxMath.cpp`
  - `Base_dir/apps` : contains main programs
  - `Base_dir/build` : directory for compiler output
- In each of (Base_dir, Base_dir/lib, Base_dir/apps) there is a CMakeLists.txt file which defines the project for CMAKE.

CMAKE example (files)

```
~/tmp/example1_cpp> Is -R
./apps/  build/ lib/
```

```
../lib:
CMakeLists.txt vtkpxMath.cpp vtkpxMath.h
```

```
../apps:
CMakeLists.txt vtkMyEx1.cpp
```

```
../build:
[ empty at start of project ]
```

- CMakeLists.txt – configuration file for each directory
- vtkMy, cmake and hints contain additional info (hints can be empty!!)
CMAKE example (files)

```
~/tmp/example1_cpp>ls -R
.
  apps/  build/  lib/  tcl/  CMakeLists.txt  vtkMy.cmake  hints
./lib:
  CMakeLists.txt  vtkpxMath.cpp  vtkpxMath.h
./apps:
  CMakeLists.txt  vtkmyEx1.cpp
./tcl:
  vtkmyEx1.tcl
./build:
  [ empty at start of project ]
  • CMakeLists.txt – configuration file for each directory
  vtkMy.cmake and hints contain additional info (hints can be empty!)
```

CMakeLists.txt for lib

```
# Source files
#
# Here is where you can add the name of your local common classes.c
#
SOURCE_FILES (Common_SRCS
  vtkpxMath
  vtkmyNewClass)

ADD_LIBRARY (vtkmyCommon STATIC Common_SRCS)
#
# Specified Link Libraries
#
LINK_LIBRARIES ( vtkmyCommon
  vtkGraphics
  vtkCommon )
```

CMakeLists.txt for apps

```
#
# Add the executable
#
ADD_EXECUTABLE(vtkmyEx1 vtkmyEx1.cpp)
ADD_EXECUTABLE(myBigProgram myBigProgram.cpp)
#
# Specify Link Libraries
#
LINK_LIBRARIES ( vtkmyCommon
  vtkCommon )
```
The main program – vtkmyEx1.cpp

```c
#include <stdio.h>
#include "vtkpxMath.h"

int main(int argc,char** argv)
{
    fprintf(stderr,"In C++\n");
    fprintf(stderr," 180.0 degrees = %.2f radians\n",
            vtkpxMath::DegreesToRadians(180.0));
    fprintf(stderr," 2.0 radians = %.2f degrees\n",
            vtkpxMath::RadiansToDegrees(2.0));
    return 0;
}
```

A Simple TCL Script vtkmyEx1.tcl

```tcl
if { $tcl_platform(platform) == "windows" } {
    load ./vtkmyCommonTCL
} else {
    load ./libvtkmyCommonTCL.so
}
vtkpxMath m
puts stderr "In TCL"
puts stderr [ format "180.0 degrees = %.2f radians"
            [ m DegreesToRadians 180.0 ]]
puts stderr [ format "2.0 radians = %.2f degrees"
            [ m RadiansToDegrees 2.0 ]]
exit
```

Demos:

1. Running CMAKE
2. Compiling using VC++
3. Running executable
4. Running TCL Script

CMAKE (Windows)

CMAKE (UNIX)

Homework

- Download example1cpp.zip from class web-page.
- Run CMAKE on it and compile it.
  - (Warning Unix setup will only work on our machines with vtk in /usr/local/vtk4.
    Windows setup needs a working vtk 4.0-4.2 and cmake < 1.2. WE work with later cmake >1.2 in compatibility mode, see version 1.2.)
- Next seminar in two weeks.