VG101 Final Review - Part II

Zhang Yichi

UM-SJTU Joint Institute

August 6, 2018

Inheritance

When to use inheritance

- When you want to have a "subclass" of a class, which can do things the original class can do, but with some differences. (Example: SickCow from Cow)
- When you want to wrap several classes into a category. (Example: Rectangle, Circle, ... from Shape)

Access Specifier

```
1
    class Cow{
3     public: //Defines accessability for members
        ...
5     private:
        ...
7   };

9   class SickCow: public Cow{ //Defies type of inheritance
      ...
11  };
```

Visibility	Base	Derived	Other
Private	Yes	No	No
Protected	Yes	Yes	No
Public	Yes	Yes	Yes

- In a derived class, methods inherited from the base class can access those private members in the base class.
- The specifier friend can open access of protected/private members to some classes/functions
- One code quality requirement: Use private/protected to specify the member attributes in your class

Base \ Inheritance	Public	Private	Protected
Private	-	-	-
Protected	Protected	Private	Protected
Public	Public	Private	Protected

In general, you just need to use public inheritance.

Inherited or Contained

Getting clear about the relationship between classes is important when you are working on a big project

- If two classes are of the "is a" relationship (for example, SickCow is Cow), use inheritance.
- If two classes are of the "has a" relationship (for example, a Zoo has several Animals), make one class as a member of the other.

Hierachy Diagram

You should have the ability to "read" and "draw" hierachy diagrams.

- In hierachy diagrams, inheritance relationships are presented in the form of tree diagrams
- Containing relationship is written in single blocks.

Polymorphism

Essence/Purpose of Polymorphism

- The word means "many forms"
- Group objects of different classes into same array or other data structures.
- Same operations can be applied on objects of different classes, which can trigger their similar behaviour, but with their own styles.

Different measures are used to achieve polymorphism (Function overloading/overwriting, abstract base class, virtual functions, etc.)

Function Overloading

To make your methods inherited from base class behave differently in your derived class, you should use function overloading.

```
1
    class Cow{
3     public:
        void speak(){cout << "Moo.\n";}
5       ...
    };
7
    class SickCow: public Cow{
9     public:
        void speak(){cout << "Ahem..Moo.\n";}
11      ...
    };
13
```

Virtual Function

Why virtual functions

- Pointer to base class can be directed to an object of a derived class.
- But such a pointer cannot use methods overloaded by the derived class.
- Specify the original function in the base class as virtual can fix this.

```
class Cow{
  public:
    virtual void speak(){cout << "Moo.\n";}
    ...
};

```

Pure Virtual Function & Abstract Class

- To group different classes, a "top class" is used.
- In your top class you can define some "general methods"
- Of course these methods should be specified as "virtual" because they are surely to be overloaded, but in fact, they are of no use in the "top class".
- You don't need to define these methods and you can make it a pure virtual.

```
class Animal{
  public:
    virtual void speak()=0;
    ...
};
```

Note: A class containing a pure virtual method is an abstract class, which cannot be instantiated.

Multiple Inheritance & Virtual Class

A class can be derived from multiple classes.

```
1    class SickMadCow : public SickCow, public MadCow {...};
```

But in fact it will cause a Diamond Problem

Diamond Problem

If a class A are derived from two classes B,C, but B,C are both inherited from a base class D, the things in D will have two copies in A.

To solve diamond problem, you two methods are possible.

- Reconstruct the hierachy (recommended)
- When creating B and C, declare them as virtual

```
1   class Cow {...};
    class SickCow : public virtual Cow {...};
3   class MadCow : public virtual Cow {...};
    class SickMadCow : public SickCow, public MadCow {...};
5
```

External Libraries

To use a external library such as OpenGL, you should know the following points.

- Get clear where the library files are and write the #include command correctly.
- Know the namespace of the library correctly.
- Add extra options in compilation so that your compiler knows that an external library should be linked.

To plot graphs using OpenGL, the recommended way is to wrap everything into classes.

- Build an abstract shape class, which has several derived class which implements the drawing process of different shapes.
- For the specific figure you draw, let it contain several pointer to shape values, which can store the address of objects of different shape subclasses.
- If you have to plot a big scenario which contains different figures, you can do the same thing in a higher hierachy. Use a figure abstract class and make all specific figures the derived class of it.

Basic Functions in OpenGL

```
//Display and Timing Function Part
#include <GL/glut.h>
#include "home.h"
void TimeStep(int n) {
  glutTimerFunc(25, TimeStep, 0);
  glutPostRedisplay();
}
void glDraw() {
  double static width=1, height=1.5, owidth=.175;
  Home zh({0,-.25},width,height,owidth);
  zh.zoom(&width, &height, &owidth);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  zh.draw();
  glutSwapBuffers(); glFlush();
}

```

```
1   //Main Function Part
    int main (int argc, char *argv[]) {
3     glutInit(&argc, argv);
      // glutInitWindowSize(500, 500);
5     glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
      glutCreateWindow("Home sweet home");
7     glClearColor(1.0, 1.0, 1.0, 0.0);
      glClear(GL_COLOR_BUFFER_BIT);
9     glutDisplayFunc(glDraw);
      glutTimerFunc(25, TimeStep, 0);
11    glutMainLoop();
    }
13
```

```
//Shape Drawing Part
void Rectangle::draw() {
  glColor3f(r, g, b); glBegin(GL_QUADS);
  glVertex2f(p1.x, p1.y); glVertex2f(p2.x, p1.y);
  glVertex2f(p2.x, p2.y); glVertex2f(p1.x, p2.y);
  glEnd();
}

```

Note: In the final exam, there will be problems about OpenGL plotting, and the things above are what you can prepare in advance and use in the exam, together with the basic shape classes.

Compilation

The basic command in terminal to compile a C++ program (take Manuel's home program as an example) is :

```
sh $ g++ -o home main.cpp home.cpp figures.cpp -lglut -lGL
```

And the command to run the program is

```
sh $ ./home
```

But for more complex projects, the command will always be too long to type. In this situation, a technique called Makefile is useful.

First, it is necessary to introduce another strategy of compilation, which is turning the code in every .cpp file into machine code separately and then combine all the separated machine codes , together with the library codes.

To compile a single .cpp file(home.cpp for example) to a partial machine code, the following command is used.

```
sh $ g++ -c home.cpp
```

It will generate a file named home.o

Do the same thing for all three .cpp files and you will get three .o files. To combine the three and link the libraries, use the following command:

```
sh $ g++ -o home main.o home.o figures.o -lglut -lGL
```

And in fact a makefile is a file which integrates all commands above.

Template of a Makefile

```
CC = g++ # compiler
CFLAGS = -std=c++11 # compiler options
LIBS = -lglut -lGL # libraries to use
SRCS = main.cpp home.cpp figures.cpp
MAIN = home
OBJS = $(SRCS:.cpp=.o)
.PHONY: clean # target not corresponding to real files
all: $(MAIN) # target all constructs the home
  @echo Home successfully constructed
$(MAIN): $(OBJS)
  $(CC) $(CFLAGS) -o $(MAIN) $(OBJS) $(LIBS)
.cpp.o: # for each .cpp build a corresponding .o file
  $(CC) $(CFLAGS) -c $< -o $@
clean:
  $(RM) *.o *~ $(MAIN)
```

Using Templates

You can build classes/functions which can be applied to user-defined data types.

```
1   //In header file
    template<class TYPE>
3   class Complex {
      public:
5       Complex(){ R = I = (TYPE)0; }
        Complex(TYPE real, TYPE img) {R=real;I=img;}
7       void PrintComplex() {cout<<R<<'+'<<I<<"i\n";}
      private:
9       TYPE R, I;
    };
11  // In main.cpp
    int main () {
13    Complex<double> a(3.123,4.9876);
      a.PrintComplex();
15    return 0;
    }
17
```

STL

Standard template library (STL) contains some pre-defined template classes, especially some sequence containers.

- vector: Able for fast random access and adding/deleting ocjects at the end.
- deque: Able for fast random access and adding/deleting objects at the front/end.
- list: Able for fast access only for the head/tail elements, but convenient to add/delete
- iterator: An object used to traverse and browse elements in sequence containers.

There are also some container adaptors, which are used for some special cases.

- queue: first-in-first-out (FIFO)
- stack: last-in-first-out (LIFO)
- priority_queue: The largest element will always be placed at the top. The objects should be comparable(you can offer your own comparison function)

Some functions/algorithms such as find(), count(), unique(), sort(), reverse(), remove(), random_shuffle(), max and min.

Note:

STL containers can be used for high efficiency and it is allowed to use in Part B. Feel free to use if you want to.

Tips about Final

- Read through all questions.
- Manage your correctly.
- Use available materials.
- Don't be hurried, complete the questions step by step.

Thank you !