Chapter 14: Readings

- Read the Sections
  - 14.1, 14.2, 14.3, 14.4, 14.5, 14.6, 14.7
- Do not read
  - 14.8, 14.9, 14.10

- Checkpoint Exercise
  - 14.1, 14.2, 14.3, 14.5, 14.6, 14.7, 14.8, 14.11, 14.12, 14.13, 14.14, 14.15, 14.16, 14.18, 14.19, 14.20, 14.21, 14.22, 14.23, 14.25, 14.26, 14.27, 14.28, 14.29,14.31,14.32, 14.33

Chapter 14: Questions

- Short Answer
  - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15, 16, 17, 18, 19, 21, 22, 23, 24, 25
- Fill-in-the-Blank
  - try all of them
- Algorithm Workbench
  - 39, 40, 41, 43
- True or False
  - try all of them
- Find the errors
  - try all of them
- Programming Challenges
  - 2, 3, 4, 7, 9

Chapter 15

- Sections
  - 15.1, 15.2, 15.3, 15.4, 15.5, 15.6, 15.7, 15.8
- Checkpoint exercise
  - 15.1, 15.2, 15.3, 15.4, 15.5, 15.6, 15.7, 15.8, 15.9, 15.10, 15.11, 15.12, 15.13, 15.14, 15.15, 15.16, 15.17, 15.18, 15.19
- Short answer
  - 1, 2, 4, 5, 6, 7, 8, 9, 10, 14, 15, 16, 17, 18
- Fill-in-the Blank
  - 20, 21, 22, 23, 24, 25, 26, 29, 30, 33
- Algorithm workbench
  - 36
- True/False
  - Try all of them
- Find the errors
  - Try all of them
- Programming challenges
  - 7, 8, 12, 13

Chapter 16

- Sections
  - 16.1, 16.2, 16.3, 16.4,
- Checkpoint exercise
  - 16.1, 16.2, 16.3, 16.4 (give an example), 16.6, 16.7 (consider int, long, unsigned, char, float, double), 16.8, 16.11,
- Short answer
  - 2, 3, 5, 7, 9
- Fill-in-the Blank
  - 11, 12, 14, 15,
- Algorithm workbench
  - 17, 20
- True/False
  - Try all of them
- Find the errors
  - Try all of them
- Programming challenges
  - 1, 3, 6, 7

Polymorphism: Definition

- C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Polymorphism: Multiple child class objects

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and "
            <<"I sound: "<<getSound();
    }
    virtual string getSound() const
    { return mammalSound;
    }
};
```

Polymorphism: Multiple child class objects

```
class Lion : public Mammal
{   string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const
    { return lionSound;
    }
    void display()
    {   cout<< "\nI am a Lion and "
            <<"I sound: "<<getSound();
    }
};
```

```
class Dog : public Mammal
{   string dogSound;
public:
    Dog() : Mammal()
    {   dogSound = "Baarrkkkk";
    }
    string getSound() const
    { return dogSound;
    }
    void display()
    {   cout<< "\nI am a Dog and "
            <<"I sound: "<<getSound();
    }
};
```

Polymorphism: Multiple child class objects

```
// using the parent object
int main()
{    Lion theKing;
     theKing.display();

     Dog thePet;
     thePet.display();

     // assigning child to the parent object
     Mammal * varMammal = &theKing;
     varMammal->display();
     // assigning another child
     varMammal = &thePet;
     varMammal->display();
}
```

```
/* Output of the program: */
     I am a Lion and I sound: Raawwrrrrr
     I am a Dog and I sound: Baarrkkkk
     I am a Mammal and I sound: Raawwrrrrr
     I am a Mammal and I sound: Baarrkkkk
```

Polymorphism: Requires References or Pointers

- Polymorphic behavior is only possible when a derived class object is used as a reference variable or a pointer.

Polymorphism:
Consider again this example

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and "
            <<"I sound: "<<getSound();
    }
    virtual string getSound() const
    { return mammalSound;
    }
};
```

```
class Lion : public Mammal
{   string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const
    { return lionSound;
    }
    void display()
    {   cout<< "\nI am a Lion and "
            <<"I sound: "<<getSound();
    }
};
```

Polymorphism: Requires References or Pointers

```
// using the parent object
int main()
{   Lion theKing;
    // assiging the derived to the usual parent object
    Mammal varMammal = theKing;
    varMammal.display();

    // assiging the derived to the parent object as reference
    Mammal &refMammal = theKing;
    refMammal.display();

    // assigning the derived to the parent as address
    Mammal * pointMammal = &theKing;
    pointMammal->display();
}
```

```
/* Output of the program */
I am a Mammal and I sound: Meeeoowww
I am a Mammal and I sound: Raawwrrrrr
I am a Mammal and I sound: Raawwrrrrr
```

Redefining vs. Overriding

- In C++, redefined functions are statically bound and overridden functions are dynamically bound.

- So, a virtual function is overridden, and a non-virtual function is redefined.

C++ 11's override and final Key Words

- The override key word tells the compiler that the function is supposed to override a function in the base class.
  - The keyword is used in the derived classes
  - only virtual member functions can be marked 'override'

- When a member function is declared with the final key word, it cannot be overridden in a derived class.
  - The keyword is used in the base classes
  - only virtual member functions can be marked 'final'

- Study the overrideKeyword.cpp and the finalKeyword.cpp programs

C++ 11's override Keyword: only for the virtual functions

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and I sound: "
            <<getSound();
    }
    virtual string getSound() const
    { return mammalSound; }
};
```

```
class Lion : public Mammal
{
    string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const override
    { return lionSound; }

    // error: no such method exist
    // to override in the base class
    string getSound(int pitch) const
    { return lionSound; }
};
```

C++ 11's override Keyword: to prevent error

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and I sound: "
            <<getSound();
    }
    virtual string getSound() const
    { return mammalSound; }
};
```

```
class Lion : public Mammal
{
    string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    string getSound() const override
    { return lionSound; }

    // error: no such method exist
    // to override in the base class
    string getSound(int pitch) const
    { return lionSound; }
};
```

C++ 11's final Keyword

```
class Mammal
{   string mammalSound;
public:
    Mammal()
    {   mammalSound = "Meeeoowww";
    }
    void display()
    {   cout<< "\nI am a Mammal and I sound: "
            <<getSound();
    }
    virtual string getSound() const final
    { return mammalSound; }
};
```

```
class Lion : public Mammal
{
    string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
    // Error: a final function can not
    // be overriden in the derived class
    string getSound() const
    { return lionSound; }
};
```

15.7

Abstract Base Classes and Pure Virtual Functions

Abstract Base Classes and Pure Virtual Functions

- Pure virtual function: a virtual member function that must be overridden in a derived class
  - without overriding the function, the we won't be able to create an object of the class

- Abstract base class contains at least one pure virtual function:
  - `virtual void functionName() = 0;`
  - The = 0 indicates a pure virtual function
  - There will be no function definition

Abstract Base Classes: No objects allowed

```
class Mammal
{    string mammalSound;
     public:
     Mammal()
     {    mammalSound = "Meeeoowww";
     }
     void display()
     {    cout<< "\nI am a Mammal and"
               <<"I sound: "<<getSound();
     }
     virtual string getSound()   = 0;
};
```

```
// Attempting to create object of
// the base class
int main()
{    Mammal * theKing = new Mammal();
     theKing->display();
}

// Output: can not create
// an object of Mammal class
```

Deriving Abstract Base Classes: Must implement pure virtual function

```
class Lion : public Mammal
{
    string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }
};
```

```
// Attempting to create object of
// the derived class
int main()
{   Lion * theKing = new Lion();
    theKing->display();
}

// Output: can not create
//  object of the Lion class
```

Deriving Abstract Base Classes: Must implement pure virtual function

```
class Lion : public Mammal
{
    string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }

    // be overriden in the derived class
    string getSound()
    { return lionSound; }
};
```

```
// Attempting to create object of
// the derived class
int main()
{   Lion * theKing = new Lion();
    theKing->display();
}

// Output: will work as the
// getSound function has been
// implemented
```

Object of Abstract Base Classes: Can be assigned the child objects

```
class Lion : public Mammal
{
    string lionSound;
public:
    Lion() : Mammal()
    {   lionSound = "Raawwrrrrr";
    }

    // be overriden in the derived class
    string getSound()
    { return lionSound; }
};
```

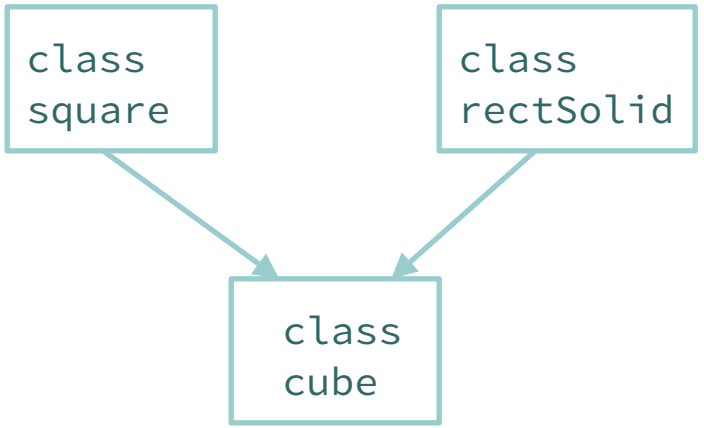
```
// Purpose of the object of the
// abstract class
int main()
{   Mammal * myMammal = new Lion();
    myMammal->display();
}

// Output: base class object
// of an abstract class can be
// assigned the object of the
// derived classes
```

15.8

Multiple Inheritance

Multiple Inheritance

- A derived class can have more than one base class
- Each base class can have its own access specification in derived class's definition:
  - class cube : public square, public rectSolid;



Multiple Inheritance

- Arguments can be passed to both base classes' constructors:
  - `cube::cube(int side) : square(side), rectSolid(side, side, side);`
- Base class constructors are called in order given in class declaration, not in order used in class constructor

Multiple Inheritance

- Problem: what if base classes have member variables/functions with the same name?
- Solutions:
  - Derived class redefines the multiply-defined function
  - Derived class invokes member function in a particular base class using scope resolution operator ::
- Compiler errors occur if derived class uses a base class function without one of these solutions