Lecture 13&14 (27/11)

- A derived class can **override** base class function.
- Overriding a function means changing the implementation of a **base** class function in a **derived** class.
- A derived class creates a function with the **same return type** and **signature** as a member function in the base class, but with a **new** implementation.
- When you make an object of the derived class, the **correct function** is called.
- When you override a function, its signature and return type **must** agree with the signature and return type of the function in the base class.
- The signature is: the **name** of the function, the **parameter** list, and the keyword **const**, if used.
- When you ***overload*** a method, you create more than one method with the *same name*, but with a **different signature**.
- When you ***override*** a method, you create a method in a derived class with the *same name* and the **same signature**.
- If Mammal has a method, move (), which is overloaded, and Dog overrides that method, the Dog method hides all the Mammal methods with that name.
- To call the base class member function explicitly using the base class's name. *"Fido**.Mammal::**move(6);"*
- C++ allows pointers to base classes to be assigned to derived class objects. Thus, you can write: *"Mammal* pMammal= new Dog;"*
- Virtual methods can be decided which method to call according to the user choices at **runtime**!
- When calling a Dog object, Mammal functions standard or virtual won't have effect, same as for the Mammal destructor.
- What if you forgot to make the Mammal destructor virtual?

- The Mammal destructor will be called, the Dog destructor won't be called!
- A pointer to Mammal can't call a function only created for class Dog.
- To solve this problem, cast the Mammal pointer to be a Dog pointer! "pDog->WagTail(); (**(Dog*)**pDog)->WagTail();"
- The standard, non-virtual, function existing in the base class will be called for objects of derived classes.
- A **v-table** is kept for each **type**, each object of that type keeps a virtual table pointer (called a ***vptr*** or ***v-pointer***), which points to that table.
- When the Dog constructor is called, and the Dog part of this object is added, the ***vptr*** is adjusted to point to the virtual function overrides (if any) in the Dog object.
- If any of your base class functions is virtual, then: the **destructor** should be virtual too. That's because you are willing to make use of polymorphism.
- When an object of a subclass is copied to an object of a superclass, the member variables/functions of the subclass are sliced off the object.

Lecture 15 (4/12)

- **Static** (or **Compile-time**) **binding** happens when determining which version of the (original/overridden) function will be called by the object **before** the program runs.
- **Dynamic** (or **Run-time**) **binding** happens during runtime where the version of the function to be called by the pointer is determined while the program is **running** (through the **v-ptr**) of each object. Used with **Polymorphism** and **Virtual** functions.
- If we didn't make multiple inheritance there are two solutions, polymorphism and casting down *"dynamic_cast<Class *> (xx);"*
- Multiple Inheritance: *class Cube : public Square, public Rectangle;*
- 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.
- What if base classes have member variables/functions with the same name? (1) redefines the multiply-defined function (2) use scope resolution operator ::
- The Shape class exists only to provide an **interface** for the classes derived from it; as such, it is an **abstract data type**, or **ADT**.
- A virtual function is made pure by initializing it with zero, as in virtual void Draw() = 0;
- Any class with **one or more** pure virtual functions is an abstract class, and it becomes illegal to instantiate.
- In fact, it is illegal to instantiate an object of any class that is an abstract class or any class that inherits from an abstract class and doesn't implement all the pure virtual functions, trying to do so causes a **compile-time error**.
- Putting a pure virtual function in your class signals two things to clients of your class: (1) Don't make an object of this class; derive

from it. (2) Be certain to override all the pure virtual functions your class inherits.

- Pure virtual functions in an abstract base class are never implemented, because no objects of that type are ever created, no reason exists to provide implementations.
- It can then be called by objects derived from the abstract class, perhaps to provide common functionality to all the overridden functions.

Lecture 16 (11/12)

- Function template: a **pattern** for a function that can work with many **data types**.
- When written, parameters are left for the **data types**.
- When called, **compiler** generates code for specific data types determined in function call.
- Better than Function Overloading: You need to create only **one version of the function**, **bug fixes** will be in **one place**, but **number of parameters** will be fixed.
- Function templates can be overloaded, each template must have a **unique parameter list**.
- All data types specified in **template prefix** must be used in **template definition**.
- Function calls must pass parameters for all data types specified in the template prefix.
- Like regular functions, function templates must be **defined** before being **called**.
- **No actual code** is generated until the function named in the template is called.
- A function template uses **no memory**.
- When passing a class object to a function template, ensure that all operators in the template are defined or overloaded in the class definition.
- Templates are often appropriate for multiple functions that perform the same task with different parameter data types.
- Develop function using usual data types first, then convert to a template: add template prefix then convert data type names in the function to a type parameter (i.e., a T type) in the template.
- Classes can also be represented by templates. When a class object is created, type information is supplied to define the type of data members of the class.

- Unlike functions, classes are instantiated by supplying the type name (int, double, string, etc.) at object definition.
- Pass type information to class template when defining objects:
  grade<int> testList[20];
  Grade<double> quizList[20];
- Use as ordinary objects once defined.
- Class can inherit from class templates:
- Must use type parameter T everywhere, base class name is used in derived class.

Lecture 17 (15/12)

- Exceptions also called **runtime errors**, indicate that something unexpected has occurred or been detected.
- Exception: object or value that signals a runtime error.
- Throw an exception: send a signal that an error has occurred.
- Catch/Handle an exception: process the exception; interpret the signal.
- **throw** – followed by an argument, is used to throw an exception.
- **try** – followed by a block {}, is used to invoke code that throws an exception.
- **catch** – followed by a block {}, is used to detect and process exceptions thrown in preceding try block. Takes a parameter that matches the type thrown, also called handler.
- A function that throws an exception is called from within a **try** block
- If the function throws an exception, the function terminates and the **try** block is immediately exited. A **catch** block to process the exception is searched for in the source code immediately following the **try** block.
- If a **catch** block is found that matches the exception thrown, it is executed. If no **catch** block that matches the exception is found, the program terminates.
- When an exception is thrown, lines of try block after the throw statement are not executed. When exception is caught, the code after catch block is executed. Catch blocks are generally written at the end through.
- The statements which may cause problems are put in try block. Also, the statements which should not be executed after a problem accursed, are put in try block. Note that once an exception is caught, the control goes to the next line after the catch block.

- If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.
- It is compiler error to put catch all block before any other catch. The catch(...) must be the last catch block.
- If no exception is thrown in the try block, the program jumps to the statement that immediately follows the try/catch construct.
- Recursive code and predefined functions such as **new** may throw exceptions.
- You can look at the catch blocks as being like overloaded functions, when the matching signature is found, that function is executed.
- An exception will not be caught, and the program will terminate, if it is **thrown from outside of a try block** or there is **no catch block that matches the data type of the thrown exception**.
- Once an exception is thrown, the program cannot return to throw point. The function executing **throw** terminates (does not return), other calling functions in **try** block terminate, resulting in unwinding the stack.
- If objects were created in the **try** block and an exception is thrown, they are destroyed.