Contents

1 Topic List

1.1 Chapter 13

1.1.1 Procedural and Object Oriented Programming

The distinction made between “vocabulary” and “important concepts” is generally somewhat arbitrary, but if there is a one or two word phrase that has important meaning, it will typically be placed under the Vocabulary section of each entry.

Vocabulary

A. **Procedural Programming**: Data is stored in a collection of variables and/or structures with a set of functions that perform operations on the data.

B. **Object-oriented Programming**: Data is stored in a single unit that contains both data and procedures (functions).

C. **Object**: The entity that contains the data and procedures.

D. **Attribute**: The data contained within an object.

E. **Member function**: The procedures an object performs.

F. **Encapsulation**: The combining of data and code into a single object.

G. **Data hiding**: An object’s ability to hide its data from code that is outside the object.

H. **Object Reusability**: The ability for components to be reused by other programs with limited modifications.

I. **Class**: Code that specifies the attributes and member functions that a particular object may have. It serves a similar purpose to a blueprint.

J. **Instance**: An object that has been created from a class. Each instance of a class is its own entity with its own attributes that store its data.

1.1.2 Introduction to Classes

Vocabulary

A. **Access Specifier**: Key words that specify what can access members of a class.

B. **Public Member Function**: These allow users to access the private member variables and perform other tasks with an object of a class.

C. **Prototype**: The declaration of something without actually defining its behavior.

D. **Scope Resolution Operator**: Two colons placed side by side (i.e. `::`) that identify something as a member of a class.

E. **Accessor**: A member function that gets a value from a member variable but does not modify it. They may also be referred to as getter functions.

F. **Mutator**: A member function that modifies a member variable. They may also be referred to as setter functions.

Important Concepts

1. In C++, the class is the construct primarily used to create objects.

2. The members of classes are private by default.

3. Private members *cannot* be accessed outside of the class. Public members *can* be accessed outside of the class.

4. Private and public members can be declared in any order; there are no rules governing the order in which they are placed.

5. The `'const'` keyword identifies a member function as an accessor. This keyword guarantees that a member function will not modify any data within the class.

Code Examples

E. `double Rectangle::getLength() const { return length; }`

F. `void Rectangle::setLength(double len) { length = len; }`

1.1.3 Defining an Instance of a Class

Vocabulary

A. **Instantiation**: The defining of a class object. This creates an instance of the class.

B. **Garbage**: The random values that may appear in a member variable that has not been initialized with a value.

C. **Stale Data**: Data that is no longer correct because the data that it is dependent on has been modified.

Important Concepts

1. Class objects are not created in memory until they are defined.
2. Class members are accessed with the dot operator.
3. Member variables are not automatically initialized to 0.
4. Using a mutator function on one instance will not modify the data of another instance of the class.
5. The results of calculations should be retrieved with accessor functions, rather than stored as an attribute, to prevent stale data.
6. Pointers can be defined to point to class objects.

7. If a pointer contains an object, you can access its members with the `'->'` operator. The statement `'a -> b'` is functionally equivalent to `'(*a).b'`.

8. Class object pointers can be used to dynamically allocate objects. In other words, you can use the `'new'` operator to dynamically allocate an instance of a class.

9. After a dynamically allocated object is freed with the `'delete'` operator, you should assign a value of `'NULL'` or `'nullptr'` to the object to prevent errors.

1.1.4 Why Have Private Members?

Important Concepts

1. Objects should protect their important data by making it private and providing a public interface to access that data.

2. The public interface for an object's data can protect it from invalid data.

1.1.5 Separating Class Specification from Implementation

Vocabulary

A. **Class Specification File**: The place where the declarations (but not definitions) of classes are stored. It is typically named after the class itself with a .h extension.

B. **Class Implementation File**: The place where member functions for a class are stored. It is typically named after the class itself with a .cpp extension.

C. **Include Guard**: A type of preprocessor directive that prevents the header file from being included more than once. There are three "keywords" associated with include guards: `#ifndef`, `#define`, and `#endif`. `'#ifndef'` determines whether a constant, typically something like `'SPIDER_H'`, where Spider is the name of the class, has been defined yet. If it hasn't been defined, the `'#define'` directive will define it. After everything has been defined, `'#endif'` will close the block.

Important Concepts

1. When header files are included, double-quotes are used instead of angle brackets.
2. The implementation files cannot be compiled into an executable, as they are not complete programs.

Code Examples

```cpp
// Circle.h
//  file
#ifndef CIRCLE_H
#define CIRCLE_H

class Circle {
  private:
    double radius;
  public:
    void setRadius(double);
    double getRadius() const;
};

#endif CIRCLE_H
```

```cpp
// Circle.cpp
// The implementation file

void Circle::setRadius(double r) {
    radius = r; }

double Circle::getRadius() const {
    return radius; }
```

1.1.6 Inline Member Functions

Vocabulary

A. **Inline Function**: A function that is defined inside the declaration of a class, rather than in an implementation file, for example. This eliminates the need to use the class name and scope resolution operator. Inline functions are generally faster than non-inline functions, but result in larger executable files.

B. **Inline Expansion**: When the compiler replaces a call to an inline function with the code of the function itself.

1.1.7 Constructors

Vocabulary

A. **Constructor**: A member function that is automatically called when an object is instantiated. All constructors have the same name as the class they are in. They can be used to initialize the member variables of the object and other setup operations.

B. **Member Initialization List**: An alternate method of initializing the variables of a class. In some situations, it can be faster than initializing the member variables inside of the constructor body.

C. **In-Place Member Initialization**: Allows you to initialize the variables of a class within its declaration. This is exclusive to C++ 11 and later.

D. **Default Constructor**: A constructor that takes no arguments. If no constructor is defined at all, a default constructor that does nothing will automatically be generated by the compiler.

Important Concepts

1. Constructors do not have return types because they cannot return any values.

2. Constructors may be automatically generated, but it is still best practice to define constructors for classes.

3. Creating a pointer to an object that is initialized with `'NULL'` or `'nullptr'`, the constructor does not execute because it does not actually create an object.

4. When an object is created with the `'new'` operator, its default constructor is automatically executed.

1.1.8 Passing Arguments to Constructors

Important Concepts

1. Constructors can have parameters and can accept arguments when objects are created.
2. Constructors can have default arguments. The value(s) can be listed in the parameter list of the declaration or in the header.
3. A constructor that has default arguments for all of its parameters will automatically become the default constructor for the class, and no arguments need to be called when creating an object with that constructor.
4. If every constructor in a class requires arguments, there will be no default constructor and arguments *must* be used when creating an object.

1.1.9 Destructors

Vocabulary

A. **Destructor**: Member functions with the same name as the class, preceded by a tilde character.

Important Concepts

1. Destructors are automatically called when an object is destroyed. A common use-case for destructors is to free memory that was dynamically allocated within a class object.

2. Destructors cannot accept arguments and cannot have a return type.

3. When you use the `'delete'` operator on a pointer to a dynamically allocated object, the destructor will automatically be called on that object.

1.1.10 Overloading Constructors

Vocabulary

A. **Constructor Delegation**: Calling different constructors within a constructor to perform certain steps in creating an object. This is exclusive to C++ and beyond.

Important Concepts

1. Just like any other functions, including member functions, constructors in C++ can be overloaded. This allows a class to have multiple constructors. As long as there is a different list of parameters, there can be multiple constructors.

2. A class may only have one default constructor and only one destructor. This extends to constructors with all default parameters: you cannot have both a constructor with no arguments and a constructor with all default parameters.

3. All member functions except the destructor can be overloaded.

1.1.11 Private Member Function

Vocabulary

A. **Private Member Function**: A member function that can only be used by functions within the class. Like private member variables, it cannot be called or accessed by functions outside of the class.

Important Concepts

1. Private member functions may only be called by other member functions in the class.

2. Member functions that could harm the data of an object if used improperly should be private.

1.1.12 Arrays of Objects

Important Concepts

1. You are able to create and work with arrays of class objects.

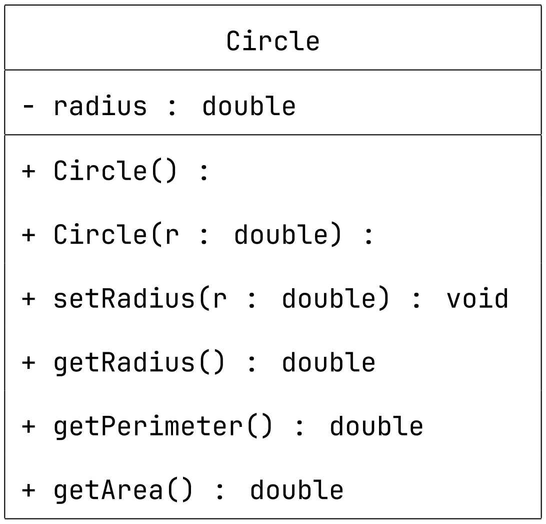
2. When no initializer list is used, the default constructor will be called for each object in the array.

3. In order to define an array of objects with non-default constructors, each object must be individually defined with an initializer list. If there are fewer initializers in the list than there are spaces in the array, the remaining objects will be created with a default constructor.

4. Objects in an array are accessed like anything else in an array.

1.1.13 Unified Modeling Language

1. **Unified Modeling Language (UML)**: A standard set of diagrams used to graphically depict object-oriented systems. There are three sections of a UML diagram: the top stores the class name, the middle stores the member variables, and the bottom contains the member functions. A `'-'` before a member name indicates that it is private, while a `'+'` indicates that it is public. The data types of functions and variables is indicate with a colon followed by the type. This also applies to function parameters. A full example is provided below and an explanation is given as well.



```
Circle
- radius : double
+ Circle() :
+ Circle(r : double) :
+ setRadius(r : double) : void
+ getRadius() : double
+ getPerimeter() : double
+ getArea() : double
```

1. The name of the class is `'Circle'`.
2. The class only has one attribute, `'radius'`. There is a minus beside it, so it is private. Finally, there is a colon followed by the `'double'` type, so we know that its type is `double`.
3. There are six member functions, all of which are public: two constructors, a mutator function, and three accessor functions. We know they are public because of the `'+'` signs before the name. We know that two of the functions are constructors because they have the same name as the class. We know which functions are accessors and mutators because of their names.
4. The accessor member functions all return doubles, as indicated by the colon followed by the `'double'` type keyword, the mutator member function returns `'void'`, and the constructors return nothing, as indicated by the lack of a type keyword.

1.2 Chapter 14

1.2.1 Instance and Static Members

Vocabulary

A. **Instance Variable**: These are variables that each instance of a class has, and only that class has it. They are entirely separate from the instance variables of any other class. If one instance's attribute of a certain name is modified, it will not affect the attribute of any other classes.

B. **Static Member Variable**: These are variables that all instances of a class share. They do not belong to any instance in particular, and may be accessed and modified by any instance of the class. There will only ever be one copy of a static member variable stored in memory at any given time. Unlike static member functions, static member variables can be accessed and modified by non-static functions by any instances of the class.

C. **Static Member Function**: These are functions that can only operate on static member variables, and not on instance variables.

Important Concepts

1. All instances of a class have access to the static variables of the class.

2. Static member variables may be stored and modified even if there are no instances of the class defined.

3. Static member functions may be called even if there are no instances of the class defined.

4. C++ automatically stores 0 in all uninitialized static member variables.

5. Static member variables are actually defined outside of the class declaration, and their lifetime is the lifetime of the program, not of any instances of the class.

6. Static member functions operate on static member variables before any instances of the class are ever defined.

7. There are two ways to call static member functions: If there are instances of the class defined, you can call them like normal. If there are no instances of the class defined, however, you must use the name of the class, followed by the scope resolution operator, and finally the name of the static member function.

1.2.2 Friends of Classes

Vocabulary

A. **Friend function**: Functions that are not members of a class, but still have access to the private members of the class. They may be stand-alone functions or they can also be members of other classes.

B. **Forward Declaration**: Informing the compiler that something will be declared later in the program. They are required when a piece of code must be processed before the code of whatever has been forward declared, such as when a friend function or class is being used.

Important Concepts

1. Both functions and entire classes can be declared friends of another class.

1.2.3 Memberwise Assignment

Important Concepts

1. Memberwise assignment occurs when an object is initialized with another object's data. Each member of one object is copied to its counterpart in the other object.

2. If memberwise assignment occurs on a pointer, there will be two pointers that point to the same location in memory. If the data that is pointed to is deleted by one object, the other object won't know and this will create an error.

1.2.4 Copy Constructors

Vocabulary

A. **Copy Constructor**: A special constructor that is automatically when an object is initialized with another object's data. It has the same form as other constructors, but uses a reference parameter of the same class type as itself. It is *only* called when an object is created, not just anywhere that assignment is occurring.

Important Concepts

1. You must use a reference object as the parameter in copy constructors.
2. Since copy constructors are used to copy data, constant reference parameters should be used, instead of regular reference parameters, to prevent inadvertent modification of the original object's data.
3. If reference parameters are not used, an infinite loop of object creations will occur, leading to a crash.
4. Memberwise assignment is performed by the default copy constructor.

1.2.5 Operator Overloading

Vocabulary

A. **`this` Pointer**: The `'this'` pointer is a special pointer that is automatically passed as a hidden argument to all non-static member functions, and always points to the instance of the class making the function call.

B. **Operator Functions**: Special member functions of a class that define how a particular operator works. They appear as the word `'operator'` followed by whatever operator is being defined, such as `'operator='`, which is the assignment operator.

C. **Self-assignment**: A type of assignment where an object is assigned to itself. Under certain circumstances, such as when assignment deletes dynamically allocated data, this can lead to major problems.

D. **Dummy Parameter**: A nameless, typically temporary parameter.

Important Concepts

1. Operator overloading allows you to redefine how operators work when used with class objects.

2. It is commonplace to name the parameter of the operator function being used `'right'`, as it falls on the right side of the operator when it is called.

3. Operator functions can also be used like other member functions. An example usage of this is as follows: `objectOne.operator=(objectTwo);`. This statement is identical in function to `'objectOne = objectTwo'`.

4. The assignment operator should have checks to prevent self-assignment. This is as simple as the following statement: `if (this != &right)`. As long as the two objects are not equal, then whatever is contained within the if-statement will execute.

5. In order to return the object that is pointed to by the `'this'` pointer, you must dereference it, then return it as normal: `return *this;`.

6. Because there is no separate operator for them, in order to overload a postfix operator, you must use a dummy parameter and a temporary local variable. The dummy parameter is like any other function parameter, except it is only of type `'int'`. The temporary local variable will be an object of the class that is created for the sole purpose of modifying its data and returning it. This is in contrast to the prefix operator, where no temporary local variable is needed and no parameter is passed to the operator function.

7. The only difference between overloading a relational operator and any other binary operator is that relational operators should always return a boolean value, rather than an object of the class.

8. The stream insertion `'<<'` and stream extraction `'>>'` operators can also be overloaded. The insertion operator must return the type `'ostream'` and must have a reference parameter of type `'ostream'` and a reference parameter of the object's class. The extraction operator is very similar, but uses `'istream'` in place of `'ostream'`.

9. The subscript operator (`'[]'`) can also be overloaded like any other operator. The primary purpose of overloading this operator is to create array-like data structures, such as an array that performs bounds checking (unlike the regular array in C++). It can only have one parameter, and that parameter is what is placed inside the brackets when calling the subscript operator. It is common to name this parameter `'sub'`, as it is the "subscript" value.

1.2.6 Object Conversion

Important Concepts

1. Special operator functions can be written that will convert a class object to any other type. To overload these operators, you simply use `'operator'` followed by the type to convert to, such as `double` or `int`.

1.2.7 Aggregation

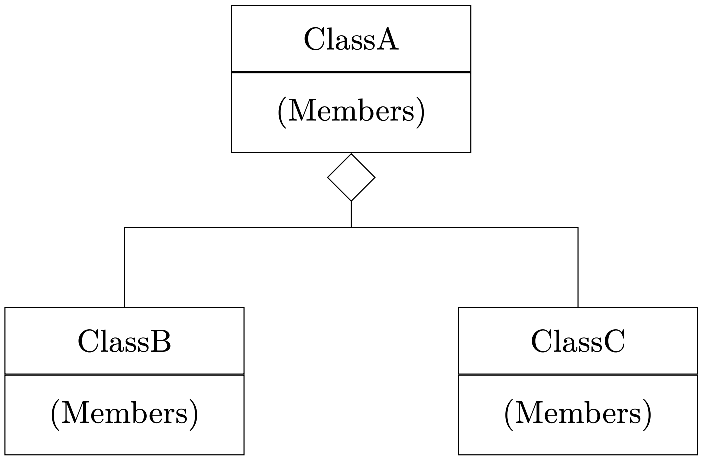
Vocabulary

A. **Aggregation**: When a class contains an instance of another class.

B. **Aggregate Class**: The class that contains instances of other classes.

Important Concepts

1. Aggregate classes can be use member initialization lists to call the constructors for each of its member objects.
2. Aggregation can be depicted in UML diagrams by a diamond shape connecting one or more separate UML diagrams.



1.2.8 Rvalue References and Move Semantics

Vocabulary

A. **Lvalue**: Values that persist beyond the statement that created them *and* have names that make them accessible to other statements in the program.

B. **Rvalue**: Values that are temporary and cannot be accessed beyond the statement that created them.

C. **Lvalue Reference**: This is what we generally consider to be a "reference". In C++ 11 and beyond, these are referred to as lvalue references, instead of just 'references'.

D. **Rvalue Reference**: These are a certain type of reference that can only refer to temporary objects that are about to be destroyed and would otherwise have no name. They are used almost identically to regular lvalue references, except with a double ampersand instead of a single ampersand (&& versus &). They cannot refer to data that is an referable by an lvalue.

E. **Move Assignment Operator**: An operator that allows you to avoid unnecessary operators by directly transferring the data stored by one object to another, rather than copying the data then deleting the original data.

F. **Move Constructor**: A special type of constructor that performs move assignment. It should be used when creating objects by initialization from temporary objects. If a move assignment operator is added to a class, then a move constructor should also be added to the class.

Important Concepts

1. Move operations transfer resources from source objects to target objects. They are suitable for situations where the source object is temporary and will be deleted shortly after the move operation.

2. By assigning a value to an rvalue reference, you allow other code to access that value – even other rvalue references.

3. Any time a class contains a pointer or reference to outside data, a copy constructor, a move constructor, a copy assignment operator, and a move assignment operator should be included as well. These will reduce the number of operations performed and increase the performance of the code.

4. Move constructors and assignment operators are automatically called when they are appropriate, so you do not need to do any extra work beyond creating them to reap their benefits.

5. If you create your own version of a move constructor, move assignment operator, copy constructor, or copy assignment operator, the compiler will no longer automatically

generate the code for any of them. This means you will then have to create your own versions of each one.

1.3 Chapter 15

1.3.1 What is Inheritance?

Vocabulary

A. **Inheritance**: The ability of one class to inherit certain parts of a parent, or base class. These parts are limited to member variables and functions in versions of C++ before C++ 11, but in C++ 11 and onward, classes may inherit some of the constructors of their base class(es).

B. **Base Class**: These are the classes which other classes inherit data and functions from. They may also be referred to as parent classes.

C. **Derived Class**: These are the classes which inherit data and functions from other classes. They may also be referred to as child classes.

D. **"Is A" Relationship**: When a class is inheriting from a parent class, there is an "is a" relationship between them. Generally, parent classes are generalized versions of something, while the children, or derived classes, are more specialized versions of them. So, for example, a spider *is an* Arachnid. A human *is a* mammal. These relationships are created by inheritance.

E. **Base Class Access Specification**: This is a type of access specification that determines what and how derived classes will inherit the members of the base class.

1.3.2 Protected Members and Class Access

Vocabulary

A. **Protected Member**: A third type of member variable or function. They are like private members, except they can be accessed by functions in a derived class. Nothing else outside of the base and derived class' members can access these members.

Important Concepts

1. A table is given below that indicates how base class specification affects how derived classes have access to the base class' members. Something to remember is that private members of the base class are *always* inaccessible to its derived classes. Another thing to keep in mind is that all inherited members are *at least* as hidden as the base class specification that is used. For example, private base class access specification will make protected and public members private. Protected will make protected and public members protected. The one exception is public specification, where protected and public members remain as they are.

Access Spec. in Base Class → Base Class Access Spec. ↓	Private Member	Protected Member	Public Member
`: private BaseClass`	Inaccessible	Private	Private
`: protected BaseClass`	Inaccessible	Protected	Protected
`: public BaseClass`	Inaccessible	Protected	Public

1.3.3 Constructors and Destructors in Base and Derived Classes

Vocabulary

A. **Constructor Inheritance**: This is when derived classes inherit the constructors of their parent class(es). There are three constructors that cannot be inherited: the default constructor, the copy constructor, and the move constructor. Any others are able to be inherited. Even if a class inherits a constructor from another class, it can still have its own constructors.

Important Concepts

1. The base class constructors is called before the derived class constructor. Destructors are called the other way around, with the derived class' destructor being called before the base class' destructor.

2. When defining the constructor of a derived class, you can pass arguments to the base class' constructor. Any value that is in the proper scope may be passed as an argument to the base class.

1.3.4 Redefining Base Class Function

Important Concepts

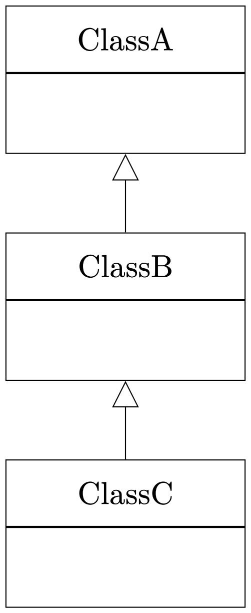
1. Base class member functions can be redefined in derived classes. This can allow for increased or more specialized functionality within the derived classes.

2. Recall that for member functions of a derived class that have the same name as the member functions in a base class, you must use the scope resolution operator for the compiler to be able to distinguish between them.

1.3.5 Class Hierarchies

Important Concepts

1. Base classes can also be derived from other classes. There is no limit to how many classes can be inherited from, so it is valuable to create charts like the one below. The more general the class, the higher it is towards the top of the chart.



1.3.6 Polymorphism and Virtual Member Function

Vocabulary

A. **Polymorphism**: Allows object reference variables or pointers of different types to call the correct member functions, depending on the type of the object being defined. It is named polymorphism because it allows the functions of base classes to take whatever form is necessary in their derived classes. In order to use polymorphism, functions must use reference variables or pointers – never objects passed by value.

B. **Virtual Member Function**: A type of member function that enables dynamic binding, allowing it to be overridden in derived classes.

C. **Binding**: The process that allows C++ to determine which function to call when a member function of an object is called.

D. **Static Binding**: The type of binding where function calls are matched to functions at *compile time.*

E. **Dynamic Binding**: The type of binding where function calls are matched to functions at *runtime*. This is enabled by making a function virtual.

F. **Redefining**: This process occurs when a non-virtual function is modified in a derived class.

G. **Overriding**: This process occurs when a virtual function is modified in a derived class.

H. **Virtual Destructor**: Reverses the order of execution of destructors in derived classes. Without the virtual destructor, pointers and dynamically allocated objects that exist in the derived class will not be properly deleted, causing many problems.

I. **`'override'` Keyword**: Tells the compiler that the function is supposed to override a function in the base class. If no overriding occurs, an error will occur.

J. **`'final'` Keyword**: Tells the compiler that a function cannot be overridden in a derived class. If a derived class attempts to modify a function declared `'final'`, the compiler will throw an error.

Important Concepts

1. Pointers to base classes can be assigned the address of a derived class object. The downside to this approach is that these pointers are only able to use base class members. If the derived class has extra member functions not found in the original, then you will not be able to use those extra functions.

2. If a class has virtual functions, it should also have a virtual destructor.

1.3.7 Abstract Base Classes and Pure Virtual Functions

Vocabulary

A. **Abstract Base Class**: A type of class that is not instantiated itself, but serves purely as a base class for other classes. It is a generic class from which other classes are defined. A class becomes an abstract base class by the inclusion of one or more pure virtual functions.

B. **Pure Virtual Function**: A special type of function that has no body or definition. They *must* be overridden in derived classes. Additionally, if a class contains a pure virtual function, it cannot be instantiated as an object.

1.3.8 Multiple Inheritance

Vocabulary

A. **Multiple Inheritance**: This is when a derived class inherits from two or more base classes. This is different to chains of inheritance, where there is ultimately only a single base class. To inherit from multiple classes, you simply separate each base class with a comma.

Important Concepts

1. There can be ambiguity when calling the member functions of base classes from a derived class, so the scope resolution operator (`::`)should be used when functions have the same name in order to inherit from the correct class. When this occurs, the derived class should also always redefine or override the member functions in question. If none of these precautions are taken, the compiler will throw an error.

2 Textbook Questions

2.1 Chapter 13

2.1.1 Short Answer

1. What is the difference between a class and an instance of the class?

2. What is the difference between the following Person structure and Person class?

3. What is the default access specification of class members?

4. Look at the following function header for a member function:

   `'void Circle::getRadius();'`

   (a) What is the name of the function?

   (b) Of what class is the function a member?

5. A contractor uses a blueprint to build a set of identical houses. Are classes analagous to the blueprint or the houses?

6. What is a mutator function? What is an accessor function?

7. Is it a good idea to make member variables private? Why or why not?

8. Can you think of a good reason to avoid writing statements in a class member function that use cout or cin?

9. Under what circumstances should a member function be private?

10. What is a constructor? What is a destructor?

11. Is it possible to have more than one constructor? Is it possible to have more than

one destructor?

12. What is a default constructor? Is it possible to have more than one default constructor?

13. If a class object is dynamically allocated in memory, does its constructor execute? If so, when?

14. When defining an array of class objects, how do you pass arguments to the constructor for each object in the array?

15. What are a class' responsibilities?

16. How do you identify the classes in a problem domain description?

2.1.2 Fill in the Blank

17. The two common programming methods in practice today are ________ and ________.

18. ________ programming is centered around functions or procedures.

19. ________ programming is centered around objects.

20. ________ is an object's ability to contain and manipulate its own data.

21. In C++, the ________ is the construct primarily used to create objects.

22. A class is very similar to a(n) ________.

23. A(n) ________ is a key word inside a class declaration that establishes a member's accessibility.

24. The default access specification of class members is ________.

25. The default access specification of a struct in C++ is ________.

26. Defining a class object is often called the ________ of a class.

27. Members of a class object may be accessed through a pointer to the object by using the ________ operator.

28. If you were writing the declaration of a class named `'Canine'`, what would you name

the file it was stored in?

29. If you were writing the external definitions of the Canine class' member functions, you would save them in a file named ________.

30. When a member function's body is written inside a class declaration, the function is ________.

31. A(n) ________ is automatically called when an object is created.

32. A(n) ________ is a member function with the same name as the class.

33. ________ are useful for performing initialization or setup routines in a class object.

34. Constructors cannot have a(n) ________ type.

35. A(n) ________ constructor is one that requires no arguments.

36. A(n) ________ is a member function that is automatically called when an object is destroyed.

37. A destructor has the same name as the class, but is preceded by a(n) ________ character.

38. Like constructors, destructors cannot have a(n) ________ type.

39. A constructor whose arguments all have default values is a(n) ________ constructor.

40. A class may have more than one constructor, as long as each has a different ________.

41. A class may only have one default ________ and one ________.

42. A(n) ________ may be used to pass arguments to the constructors of elements in an object array.

2.1.3 True or False

51. | T F | Private members must be declared before public members.

52. | T F | Class memmbers are private by default.

53. | T F | Members of a struct are private by default.

54. | T F | Classes and structures in C++ are very similar.

55. | T F | All private members of a classs must be declared together.

56. | T F | All public members of a class must be declared together.

57. | T F | It is legal to define a pointer to a class object.

58. | T F | You can use the new operator to dynamically allocate an instance of a class.

59. | T F | A private member function may be called from a statement outside of the class, as long as the statement is in the same program as the class declaration.

60. | T F | Constructors do not have to have the same name as the class.

61. | T F | Constructors may not have a return type.

62. | T F | Constructors cannot take arguments.

63. | T F | Destructors cannot take arguments.

64. | T F | Destructors may return a value.

65. | T F | Constructors may have default arguments.

66. | T F | Member functions may be overloaded.

67. | T F | Constructors may not be overloaded.

68. | T F | A class may not have a constructor with no parameter list and a constructor whose arguments all have default values.

69. | T F | A class may only have one destructor

70. | T F | When an array of objects is defined, the constructor is only called for the first element.

71. | T F | To find the classes needed for an object-oriented application, you identify all of the verbs in a description of the problem domain.

72. | T F | A class' responsibilities are the things the class is responsible for knowing and actions the class must perform.

2.1.4 Find the Errors

73.

```
class Circle:
{
private
  double centerX;
  double centerY;
  double radius;
public
  setCenter(double, double);
  setRadius(double);
}
```

74.

```
#include <iostream>
using namespace std;
Class Moon;
{
Private;
  double earthWeight;
  double moonWeight;
Public;
  moonWeight(double ew);
    { earthWeight = ew; moonWeight = earthWeight / 6; }
  double getMoonWeight();
    { return moonWeight; }
}

int main()
{
  double earth;
  cout >> "What is your weight? ";
  cin << earth;
  Moon lunar(earth);
  cout << "On the moon you would weight "
       <<lunar.getMoonWeight() << endl;
  return 0;
}
```

75.

```
#include <iostream>
using namespace std;

class DumbBell;
{
  int weight;
  public:
  void setWeight(int);
};
void setWeight(int w)
{
  weight = w;
}
int main()
{
  DumbBell bar;
  DumbBell(200);
  cout << "The weight is " << bar.weight << endl;
  return 0;
}
```

76.

```
class Change
{
public:
  int pennies;
  int nickels;
  int dimes;
  int quarters;
  Change()
    { pennies = nickels = dimes = quarters = 0; }
  Change(int p = 100, int n = 50, d = 50, q = 25);
};
void Change::Change(int p, int n, d, q)
{
  pennies = p;
  nickels = n;
  dimes = d;
  quarters = q;
}
```

2.2 Chapter 14

2.2.1 Short Answer

1. Describe the difference between an instance member variable and a static member variable.

2. Assume a class named Numbers has the following static member function declaration:

   `'static void showTotal();'`

   Write a statement that class the `'showTotal'` function.

3. A static member variable is declared in a class. Where is the static member variable defined?

4. What is a friend function?

5. Why is it not always a good idea to make an entire class a friend of another class?

6. What is memberwise assignment?

7. When is a copy constructor called?

8. How can the compiler determine if a constructor is a copy constructor?

9. Describe a situation where memberwise assignment is not desirable.

10. Why must the parameter of a copy constructor be a reference?

11. What is a default copy constructor?

12. Why would a programmer want to overload operators rather than use regular member functions to perform similar operations?

13. What is passed to the parameter of a class' `'operator='` function?

14. Why shouldn't a class' overloaded = operator be implemented with a `'void'` operator function?

15. How does the compiler know whether an overloaded ++ operator should be used in prefix or postfix mode?

16. What is the `'this'` pointer?

17. What type of value should be returned from an overloaded relational operator function?

18. The class `'Stuff'` has both a copy constructor and an overloaded = operator. Assume `'blob'` and `'clump'` are both instances of the `'Stuff'` class. For each statement below, indicate whether the copy constructor or the overloaded = operator will be called.

```
Stuff blob = clump;
clump = blob;
blob.operator=(clump);
showValues(blob);
```

19. Explain the programming steps necessary to make a class' member variable static.

20. Explain the programming steps necessary to make a class' member function static.

21. Consider the following class declaration:

```
class Thing
{
private:
  int x;
  int y;
  static int z;
public:
  Thing()
    { x = y = z; }
  static void putThing(int a)
    { z = a; }
};
```

Assume a program containing the class declaration defines three `'Thing'` objects with the following statement:

`'Thing one, two, three;'`

(a) How many separate instances of the x member exist?

(b) How many separate instances of the y member exist?

(c) How many separate instances of the z member exist?

(d) What value will be stored in the the `'x'` and `'y'` members of each object?

(e) Write a statement that will call the `'putThing'` member *before* the objects above are defined.

22. Describe the difference between making a class a member of another class (object aggregation) and making a class a friend of another class.

23. What is the purpose of a forward declaration of a class?

24. Explain why memberwise assignment can cause problems with a class that contains a pointer member.

25. Why is a class' copy constructor called when an object of that class is passed by value into a function?

2.2.2 Fill in the Blank

26. If a member variable is declared ________, all objects of that class have access to the same variable.

27. Static member variables are defined ________ the class.

28. A(n) ________ member function cannot access any nonstatic member variable in its own class.

29. A static member function may be called ________ any instances of its class are defined.

30. A(n) ________ function is not a member of a class, but has access to the private members of the class.

31. A(n) ________ tells the compiler that a specific class will be declared later in the program.

32. ________ is the default behavior when an object is assigned the value of another object of the same class.

33. A(n) ________ is a special constructor, called whenever a new object is initialized with another object's data.

34. ________ is a special built-in pointer that is automatically passed as a hidden argument to all nonstatic member functions

35. An operator may be ________ to work with a specific class.

36. When overloading the ________ operator, its function must have a dummy parameter.

37. Making an instance of one class a member of another class is called ________.

38. Object aggregation is useful for creating a(n) ________ relationship between the two classes.

2.2.3 True or False

44. | T F | Static member variables cannot be accessed by nonstatic member functions

45. | T F | Static member variables are defined outside their class declaration.

46. | T F | A static member function may refer to nonstatic member variables of the same class, but only after an instance of the class has been defined.

47. | T F | When a function is declared a `'friend'` by a classs, it becomes a member of that class.

48. | T F | A `'friend'` function has access to the private members of the class declaring it a `'friend'`

49. | T F | An entire class may be declared a `'friend'` of another class.

50. | T F | In order for a function or class to become a friend of another class, it must be declared as such by the class granting it access.

51. | T F | If a class has a pointer as a member, it's a good idea to also have a copy constructor.

52. | T F | You cannot use the = operator to assign one object's values to another object, unless you overload the operator.

53. | T F | If a class doesn't have a copy constructor, the compiler generates a default copy constructor for it.

54. | T F | If a class has a copy constructor and an object of that class is passed by value into a function, the function's parameter will *not* call its copy constructor.

55. | T F | The `'this'` pointer is passed to static member functions.

56. | T F | All functions that overload unary operators must have a dummy parameter.

57. | T F | For an object to perform automatic type conversion, an operator function must be written.

58. | T F | It is possible to have an instance of one class as a member of another class.

2.2.4 Find the Errors

59.

```
class Box
{
  private:
    double width;
    double length;
    double height;
  public:
    Box(double w, l, h)
      { width = w; length = l; height = h; }
    Box(Box b) // Copy Constructor
      { width = b.width;
        length = b.length;
        height = b.height; }

    ... Other member functions follow ...
};
```

60.

```
class Circle
{
  private:
    double diameter;
    int centerX;
    int centerY;
  public:
    Circle(double d, int x, int y)
      { diameter = d; centerX = x; centerY = y; }
    void Circle=(Circle &right)
      { diameter = right.diameter;
        centerX = right.centerX;
        centerY = right.centerY; }

  ... Other member functions follow ...
};
```

61.

```
class Point
{
  private:
    int xCoord;
    int yCoord;
  public:
    Point (int x, int y)
      { xCoord = x; yCoord = y; }
    void operator+(const &Point right)
      { xCoord += right.xCoord;
        yCoord += right.yCoord;
      }

  ... Other member functions follow ...
};
```

62.

```
class Box
{
  private:
    double width;
    double length;
    double height;
  public:
    Box(double w, l, h)
      { width = w; length = l; height = h; }
    void operator++()
      { ++width; ++length; }
    void operator++()
      { width++; length++; }

  ... Other member functions follow ...
};
```

63.

```
class Yard
{
  private:
    float length;
  public:
    yard(float l)
      { length = l; }
    void operator float ()
      { return length; }

  ... Other member functions follow ...
};
```

2.3 Chapter 15

2.3.1 Short Answer

1. What is an “is a” relationship?

2. A program uses two class: `'Dog'` and `'Poodle'`. Which is the base class, and which is the derived class?

3. How does base class access specification differ from class member access specification?

4. What is the difference between a protected class member and a private class member?

5. Can a derived class ever directly access the private members of its base class?

6. Which constructor is called first, that of the derived class or the base class?

7. What is the difference between redefining a base class function and overriding a base class function?

8. When does static binding take place? When does dynamic binding take place?

9. What is an abstract base class?

10. A program has a class `'Potato'`, which is derived from the class `'Vegetable'`, which is derived from the class `'Food'`. Is this an example of multiple inheritance? Why or why not?

11. What base class is named in the line below?

```
'class Pet : public Dog'
```

12. What derived class is named in the line below?

```
'class Pet : public Dog'
```

13. What is the class access specification of the base class named below?

`'class Pet : public Dog'`

14. What is the class access specification of the base class named below?

`'class Pet : Fish'`

15. Protected members of a base class are like ________ members, except they may be accessed by derived classes.

16. Complete the table on the next page by filling in private, protected, public, or inaccessible in the right-hand column.

In a **private** base class, this base class MEMBER access specification...	...becomes this access specification in the derived class.
private protected public	

17. Complete the table on the next page by filling in private, protected, public, or inaccessible in the right-hand column.

In a **protected** base class, this base class MEMBER access specification...	...becomes this access specification in the derived class.
private	
protected	
public	

18. Complete the table on the next page by filling in private, protected, public, or inaccessible in the right-hand column.

In a **public** base class, this base class MEMBER access specification...	...becomes this access specification in the derived class.
private	
protected	
public	

2.3.2 Fill in the Blank

19. A derived class inherits the ________ of its base class.

20. When both a base class and a derived class have constructors, the base class' constructor is called ________ (first/last).

21. When both a base class and a derived class have destructors, the base class' destructor is called ________ (first/last)

22. An overridden base class function may be called by a function in a derived class by

using the ________ operator.

23. When a derived class redefines a function in a base class, which version of the function do objects that are defined of the base class call?

24. A(n) ________ member function in a base class expects to be overridden in a derived class.

25. ________ binding is when the compiler binds member function calls at compile time.

26. ________ binding is when a function call is bound at runtime.

27. ________ is when member functions in a class hierarchy behave differently, depending upon which object performs the call.

28. When a pointer to a base class is made to point to a derived class, the pointer ignores any ________ the derived class performs, unless the function is ________.

29. A(n) ________ class cannot be instantiated.

30. A(n) ________ functions has no body, or definition, in the class in which it is declared.

31. A(n) ________ of inheritance is where one class is derived from a second class, which in turn is derived from a third class.

32. ________ is where a derived class has two or more base classes.

33. In multiple inheritance, the derived class should always ________ a function that has the same name in more than one base class.

2.3.3 True or False

38. | T F | The base class' access specification affects the way base class member functions may access base class member variables.

39. | T F | The base class' access specification affects the way the derived class inherits members of the base class.

40. | T F | Private members of a private base class become inaccessible to the derived class.

41. | T F | Public members of a private base class become private members of the derived class.

42. | T F | Protected members of a private base class become public members of the derived class.

43. | T F | Public members of a protected base class become public members of the derived class.

44. | T F | Private members of a protected base class become inaccessible to the derived class.

45. | T F | Protected members of a public base class become public members of the derived class.

46. | T F | The base class constructor is called after the derived class constructor.

47. | T F | The base class destructor is called after the derived class destructor.

48. | T F | It isn't possible for a base class to have more than one constructor.

49. | T F | Arguments are passed to the base class constructor by the derived class constructor.

50. | T F | A member function of a derived class may not have the same name as a member function of the base class.

51. | T F | Pointers to a base class may be assigned the address of a derived class object.

52. | T F | A base class may not be derived from another class.

2.3.4 Find the Errors

53.

```
class Car, public Vehicle
{
  public:
    Car();
    ~Car();
  protected:
    int passengers;
}
```

54.

```
class Truck, public : Vehicle, protected
{
  private:
    double cargoWeight;
  public:
    Truck()
    ~Truck()
};
```

55.

```
class SnowMobile : Vehicle
{
  protected:
    int horsePower;
    double weight;
  public:
    SnowMobile(int h, double w), Vehicle(h)
      { horsePower = h; }
    ~SnowMobile();
};
```

56.

```
class Table : public Furniture
{
  protected:
    int numSeats;
  public:
    Table(int n) : Furniture(numSeats)
      { numSeats = n; }
    ~Table();
};
```

57.

```
class Tank : public Cylinder
{
  private:
    int fuelType;
    double gallons;
  public:
    Tank();
    ~Tank();
    void setContents(double);
    void setContents(double);
};
```

58.

```
class Three : public Two : public One
{
  protected:
    int x;
  public:
    Three(int a, int b, int c), Two(b), Three(c)
      { x = a; }
    ~Three();
};
```

3 Code Examples

3.1 Class Member Access Specification

```cpp
class Rectangle {
  private: // private spec.
    double someAttribute;
  protected: // protected spec.
    double length;
    double width;
  public: // public spec.
    virtual double getArea() const;
    void setLength(const double &);
};
```

3.2 Inline Functions and Constructors

```cpp
// Specifically in Rectangle.h
class Rectangle {
  // Inline function
  void setWidth(const double &w)
    { width = w; }

  // Inline constructor
  Rectangle() {
    width = 0;
    height = 0;
  }
};
```

3.3 Specification vs. Implementation File

```cpp
// Rectangle.h
// The specification file
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle {
  private:
    double length;
    double width;
  public:
    void setLength(const double &);
    double getLength() const;
};

#endif CIRCLE_H
```

```cpp
// Circle.cpp
// The implementation file

void Rectangle::setLength(const
    double &len)
  { length = len; }

double Rectangle::getLength() const
  { return length; }
```

3.4 Passing Arguments to Classes

```cpp
// The object has a constructor with two
// arguments - width and length
Rectangle object(10.0, 10.0);
```

3.5 Passing Arguments to Base Class Constructors

```cpp
Box::Box(const double &w, const double &l, const double &h) :
  Rectangle(w, l)
  { height = h; }
```

3.6 Accessor vs. Mutator Function

```
\\ Accessor function - notice the 'const' keyword at the end
double getLength() const;

\\ Mutator function
void setLength(const double &);
```

3.7 Pointer Objects and Dynamically Allocating Objects

```
// Note: Rectangle constructor is not actually
// called until the second statement
Rectangle *dynamicRectangle = NULL;
dynamicRectangle = new Rectangle(10, 10);

// Regular pointer
Rectangle dummy(15, 15);
Rectangle *rectanglePointer = NULL;
rectanglePointer = &dummy;

rectanglePointer->setLength(20);
dynamicRectangle->setWidth(13);

// Safely handle the dynamic pointer
delete dynamicRectangle;
dynamicRectangle = NULL;
```

3.8 Pointer from Base to Derived

```
// In Box.h (Rectangle is parent/base of Box)
class Box : public Rectangle {}

// In main.cpp                      w, l, h
Rectangle *rectPointer = new Box(1, 3, 5);
rectPointer->getLength(); // Will be 3
rectPointer->getHeight(); // Error: getHeight is not defined for Rectangle

// Safely handle the dynamic pointer
delete rectPointer;
rectPointer = NULL;
```

3.9 Member Initialization List

```
// In Rectangle.cpp
// A default constructor
Rectangle::Rectangle() : width(0.0), length(0.0) {}

// A constructor with arguments
Rectangle::Rectangle(const double &wid, const double &len) : length(wid),
    width(len) {}
```

3.10 Default Constructors

```cpp
// This is a default constructor
Rectangle::Rectangle() {}

// But this would also be a default constructor
Rectangle::Rectangle(const double &wid = 0, const double &len = 0)
```

3.11 Static Members and Functions

```cpp
// In Rectangle.h
// Important note: static functions CANNOT be const
static void getNumberOfInstances();
static double NumberOfInstances();

// In Rectangle.cpp
double Rectangle::numberOfInstances = 0;
void Rectangle::getNumberOfInstances()
  { return numberOfInstances; }

// main.cpp
// To call without using a class instances
Rectangle::getNumberOfInstances();

// To call using an instance
rectangleInstance.getNumberOfInstances();
```

3.12 Copy Constructor

```cpp
// In Box.cpp
Box::Box(const Box &object) {
  height = object.height;

  // Follow the same "formula" for other attributes
}

// In main.cpp
Box A(1, 2, 3);
Box B = A;
```

3.13 Move Constructor

```cpp
// In Box.cpp
Box::Box(Box &&temp) {
  // When using pointers you need to do
  // attribute = nullptr;
  // after stealing the data from temp.attribute
  height = temp.height;
  temp.height = 0;

  // Follow the same "formula" for other attributes
}

// In main.cpp
Box A = Box(3, 2, 1);
```

3.14 Copy Assignment

```cpp
// In Box.cpp
Box &Box::operator=(const Box &right) {
  // Self-assignment check (Not strictly necessary)
  if (this != &right) {
    height = right.height;

    // Follow the same "formula" for other attributes
  }

  return *this;
}

// In main.cpp
Box A(2, 2, 2);
Box B(3, 3, 3);
A = B;
```

3.15 Move Assignment

```cpp
// Move assignment
// In Box.cpp
Box &Box::operator=(Box &&right) {
  // Self-assignment check (Not strictly necessary)
  if (this != &right) {
    // swap is from #include <algorithm>
    // You need 'using namespace std;' to use it like this.
    swap(height, right.height);

    // Follow the same "formula" for other attributes
  }

  return *this;
}

// In main.cpp
Box A(1, 2, 3);
A = Box(2, 2, 1);
```

3.16 Self-Assignment Check

```cpp
if (this != &right) {
  // operations
}
```

3.17 All (Differing) Overloads

```cpp
// Assignment operator
const Box operator=(const Box &right) {
  if (this != &right) {
    height = right.height;

    // Follow the same "formula" for other attributes
  }

  return *this;
}
```

```cpp
// Regular math operators
Box operator+(const Box &right) {
  Box temp;

  temp.height = height + right.height;

  // Follow the same "formula" for other attributes

  return temp;
}
```

```
// Prefix Operators
// No parameters
// *this is returned
Box Box::operator++() {
  ++height;

  return *this;
}
```

```
// Postfix Operators
// int Dummy Parameter
// Temp object is returned
Box Box::operator++(int) {
  Box temp(width, length, height);

  height++;

  return temp;
}
```

```
// Relational operators
// Like math operators but returns 'bool'
bool Box::operator>(const Box &right) {
  if (height > right.height) {
    return true;
  } else {
    return false;
  }
}
```

```
// Stream INSERTION Operator (<<)
// 'ostream' is used
// The &strm parameter CANNOT
// be a constant reference
ostream &operator<<(ostream &strm,
        const Box &object) {
  strm << object.height;

  return strm;
}
```

```
// Stream EXTRACTION Operator (>>)
// 'istream' is used
// Both parameters are
// non-constant references
istream &operator>>(istream &strm,
        Box &object) {
  strm >> object.height;

  return strm;
}
```

```cpp
// How insertion and extraction should be declared in class specification file
#ifndef BOX_H
#define BOX_H
#include<iostream>

// Forward declaration
class Box;

// Overloads
ostream &operator<<(ostream &, const Box &);
istream &operator>>(istream &, Box &);

// Class declaration
class Box {};
```

```cpp
// Subscript operator - []
// Only really good for array-like structures

Cat &Box::operator[](const int &subscript) {
  if (subscript < 0 || sub >= numberOfCats) {
    // Error handling stuff
    return 0;
  }

  return listOfCatsInBox[subscript];
}
```

3.18 Aggregation

```
class Box {
  // There will be an instance of a Cat inside the Box class
  // Box is the Aggregate Class
  Cat catInTheBox;
  // Other Stuff
}
```

3.19 Lvalue and Rvalue References

```
// Lvalue References
double variable = 3.2;
double &lvalue = variable;

// Rvalue References
double &&rvalue = 9.3;
```

3.20 Constructor Inheritance

```
// Constructor Inheritance
// Constructors in Rectangle.h
class Rectangle {
  Rectangle();
  Rectangle(const double &);
  Rectangle(const double &, const double &);
};

// In Box.h
class Box : public Rectangle {
  using Rectangle::Rectangle;

  Box(const double &);
};

// In main.cpp
// Calls Rectangle::Rectangle()
Box boxOne;

// Will call Box::Box(const double &) because
// any constructors in the derived class
// with the same parameters as the inherited
// constructors will take precedence.
Box boxTwo(5);

// Calls Rectangle::Rectangle(const double &, const double &)
Box boxThree(2, 5);
```

3.21 Virtual Function

```
// In Rectangle.h
virtual double getWidth() const;

// In Rectangle.cpp
double Rectangle::getWidth() const
  { return width; }
```

3.22 Override and Final

```
// In Box.h
// Needs to be virtual to be final
virtual double getArea() const final;

// In Box.cpp
// Compiler will throw an error if a derived class of
// Box tried to override this function
double Box::getArea() const
  { return 2 * (length * width + height * width + length * height); }

// In Rectangle.h
// Needs to be virtual to be override
virtual double getArea() const override;

// In Rectangle.cpp
double getArea() const
  { return length * width; }
```

3.23 Abstract Base Class and Pure Virtual Function

```cpp
// In Shape.h
class Shape {
  public:
    // Pure virtual function -> Makes Shape Abstract
    // Instances of Shape cannot exist
    virtual double getArea() const = 0;
}
```

3.24 Multiple Inheritance vs. Chain of Inheritance

```cpp
// Multiple Inheritance
// Notice: It is inheriting from
// two classes at once
class Cat : public Pet, public
    Feline {}
```

```cpp
// Chain of Inheritance
// Notice: Only inheriting
// one at a time
class Mammal : public Organism {}
class Feline : public Mammal {}
class Cat : public Feline {}
```

3.25 Static vs. Dynamic Binding

```cpp
// Static Binding
// In Rectangle.h
// Area of Rectangle is
// length * width
double getArea() const;

// In Box.cpp
double getArea() const {
  return 2 * (length * width)
       + 2 * (length * height)
       + 2 * (height * width);
}

// In main.cpp
Rectangle *rectPtr = NULL;
rectPtr = new Box(1, 1, 1);

// This will only display 1, since
// getArea is not virtual and will
// use Rectangle's definition
cout << rectPtr->getArea() << '\n';
```

```cpp
// Dynamic Binding
// In Rectangle.h
// Area of Rectangle is
// length * width
virtual double getArea() const;

// In Box.cpp
double getArea() const {
  return 2 * (length * width)
       + 2 * (length * height)
       + 2 * (height * width);
}

// In main.cpp
Rectangle *rectPtr = NULL;
rectPtr = new Box(1, 1, 1);

// This will display 6, since
// getArea is virtual and has
// been overridden by Box
cout << rectPtr->getArea() << '\n';
```

3.26 Friend Functions and Classes

```cpp
// In Box.h
// Forward declaration of Cat
class Cat;

class Box {
  // Friend function of Box
  friend Cat getCatFromBox() const;

  // Friend class if Box
  friend Cat;
}
```