Lecture 7 - Classes - UML - Abstraction

C++ fully supports **object-oriented programming** (OOP).
- Object-oriented programming uses **classes** to define **objects**.
  - The following are part of OOP: Encapsulation, Inheritance, and Polymorphism

class - class is a reserved word (lowercase).
- Classes are used to define objects.
  - Classes have data members and member functions (called methods in Java).

- **Data members** (called **instance variables**) - A class can have data members of different data types.
  - Data members are often declared as private.
  - Therefore, only class functions can act on the data members.

- **Member functions** - Functions defined in a class specification.
  - Class functions are the only way private data members of a class can be accessed.

- **Access Specifiers** - **public** and **private** are access specifiers.
  - They specify how class data members and class functions can be accessed.

- A class specifies what an object would be if one were declared.
  - In other words, a class specification, like the one below, does not declare an object, it just specifies what an object would be is one is declared in the program.

    (Declaring an object is like declaring a variable - discussed later.)

Ex #1:
```
class Dog
{
private:
   string breed;      // private data members
   int age;
public:
   void displayDog();     // public member function
};
```

Advantages of using a class:

1.) **Reusable code** - A class specification can be stored in a header file.
  - The header file can then be used by different programs.
  - Reusable code saves time in programming.

2.) **Private data members are protected** - Values held in private data members cannot be inadvertently changed in *main()*.
  - The only way their values can be changed is by member functions.

3) **Abstraction / Encapsulation**

- **Data abstraction** refers to providing only essential information about the data to the outside world, hiding the background details or implementation.
  Consider a real life example of a man driving a car. ... This is what abstraction is. He doesn't neet to know how the car is made, just how to drive it.

  Ex: cout << cout is an object defined in the iostream class. We use it without thinking about how it works.

  Ex: string name name is an object of the string class. It has functions like: length()

  Ex: class Stack A stack object can be created. The object can hold data in variables, and have functions that act on the data.
  - Abstract reduces programming complexity and increases efficiency.

- **Abstraction** - Abstraction mechanisms support the creation of reusable code.

Encapsulation is the property of being a self-contained unit.

- Class objects are encapsulated.
- Encapsulation means all details of an object are grouped together.
- The details are hidden from the user.
- The user only needs to know what functions to call, which makes programming much easier.

  Ex: #include <cmath> - We just need to know how to call the sqrt() function, but we don't need to know how it works.

Inheritance - Inheritance allows for the extension of an existing type.

- Data members and class functions defined in one class can be "inherited" by a new class.
- **Base class** - (superclass in Java) - A base class is an existing class.
- **Derived class** - (subclass in Java) - A subclass is a new class that inherits data members and functions from a parent (base) class.

Example of a program using a class

- Typically, each class has a specification file (.h file), and an implementation file (.cpp)

  Ex: Student.h - This file holds the Student class specification

  Student.cpp - This file holds the Student class function implementations

Class Specification - By convention, class names begin with an uppercase letter.

- All class members (data and functions) are private by default.

- Private members can be accessed only by functions of the class itself.
- As a general rule, keep data members of a class private.
- Create public functions to **set** and **get** the private member variables.
- The class specification is placed in a **header file**, with the same name as the class.

Ex #2:

```
//  File:  Cat.h

class Cat
{
   private:
      int age;
      float weight;
   public:
      Cat();
      Cat(int age, float weight);
      ~Cat();
      void setAge(int age);
      void setWeight(float weight);
      int getAge()const;
      float getWeight()const;
      void displayCat()const;
};
```

private section - Only public functions can access this private data.
- This data cannot be accessed directly from *main()*.

2 private data members

← default constructor
← overloaded constructor
← destructor

8 public functions (function prototypes)

These functions can be called from *main()* because they are public.

NOTE: Eight function definitions are in a **Cat.cpp** file starting near the bottom of page #4.

Member Functions - Two categories of member functions: Mutators and Accessors.

- **Mutator functions** - (**set** functions)
  - A **set** function is a mutator function that receives one or more arguments and assigns them to private data members.

    Ex #3: `void setAge(int age);`

- **Accessor functions** - (**get** functions)
  - A **get** function is a accessor function that returns a value held in a private data member.

    Ex #4: `double getAge();`

Constructor – Every class has a special member function called a constructor.

- A constructor 'constructs' a new object in memory when an object is **instantiated** (created).

- When an object is instantiated, its constructor is automatically called.

- A constructor is a class function with the same name as the class itself.
  - The constructor for a Cat class is: `Cat();` // Constructor prototype
- A constructor never has a return value – not even void
- Every class must have at least one constructor. Sometimes a class will have two or three or more, depending on the program.

Default Constructor – By definition, a default constructor has no parameters:

Ex #5: `Cat();`

Overloaded Constructor – As mentioned, sometimes a class has more than one constructor.

- An **overloaded constructor** has the same name as the default constructor (the class name).
  - However, the overloaded constructor has one or more parameters.

Ex #6: Constructor prototype (2 parameters): `Cat(int age, float weight);`

- The correct constructor is called, depending on how the object is declared. (discussed later)

Destructor - Every class has a special member function called a destructor.

- When a class object ends or is deleted, a destructor frees any memory that was allocated.
- A destructor always has the name of the class, preceded by a tilde (~).
- Destructors have no parameters and do not return a value.
- The Cat class declaration includes: `~Cat();` // Prototype of a destructor.
- There can be only on destructor – never with parameters.

Implementation file - Function prototypes for class functions are listed in a class specification in a header file (Cat.h)

- Each function prototype listed in the class specification must have a function implementation (definition).
- Function implementations can be included within a class specification (in the header file), but are usually placed in a separate .cpp file.
- The .cpp file is usually given the same name as the class name (like: Cat.cpp).

Ex #7: `// File: Cat.cpp`

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

These are required because the displayCat function uses `cout <<`

```
// Cat.cpp

#include "Cat.h"


// === Default constructor ====================
Cat::Cat()
{
     age = 0;
     weight = 0;
}
// ============================================

// === Overloaded constructor =================
// Values passed as arguments from main() are assigned
//      to an object's data members.
Cat::Cat(int age, float weight)
{
     this->age = age;
     this->weight = weight;
}
// ============================================


// === Destructor =============================
// Does nothing
Cat::~Cat(){}
// ============================================

// === setAge method ==========================
// Values passed as arguments from main() are assigned
//      to an object's data members.
void Cat::setAge(int age)
{
     this->age = age;
}
// ============================================

// === getAge method ==========================
// A value representing an object's age is returned.
int Cat::getAge()const
{
```

Note: The 'this' pointer can also be used, and does the same thing:

```
this->age = 0;
this->weight = 0;
```

```
     return age;
}
// ========================================

// === setWeight method ====================
// Values passed as arguments from main() are assigned
//       to an object's data members.
void Cat::setWeight(float weight)
{
     this->weight = weight;
}
// ========================================

// === getWeight function ==================
// A value representing an object's weight is returned.
float Cat::getWeight()const
{
     return weight;
}
// ========================================
// === displayCat function ==================
// A Cat object is displayed on the screen.
void Cat::displayCat()const
{
     cout << "The cat is " << age << " years old,\n"
          << "and weighs " << weight << " pounds.\n\n";
}
// ========================================
```

To Declare and use class objects:

- Class objects are declared much like variables are declared.

Ex #8: `// File: main.cpp`

```
int main()
{
    int age = 3;
    float weight = 4.4;

    Cat fluffy;

    Cat tom(4, 5.5);


    fluffy.displayCat();
```

This statement declares a new object of *Cat* type.

The default constructor is called (see Ex #5)

When *tom* is created, the overloaded constructor is called. (see Ex #6)

The fluffy object is displayed, because *fluffy* is the calling Object. (The output is garbage in this case).

```
tom.displayCat();            // The tom object is displayed.  (see output)

fluffy.setAge(age);          // A value of 3 is passed to the setAge function.

fluffy.setWeight(weight);    // 4.4 is passed to setWeight function.

fluffy.displayCat();         // The fluffy object is displayed.  (see output)
```

```
/*  OUTPUT:

The cat is 0 years old,
and weighs 0 pounds.
```
} The output is values from the 1st constructor. `fluffy.displayCat();`

```
The cat is 4 years old,
and weighs 5.5 pounds.
```
} The output from `tom.displayCat();`

```
The cat is 3 years old,
and weighs 4.4 pounds.
```
} The output is values from the 2nd constructor. `fluffy.displayCat();`

Note: An object declaration is actually a function call, because a constructor is called when an object is created (instantiated).

- This is an exception to the rule that states all functions require parentheses, because the declaration of fluffy, above, does not have parentheses.

Note: **A class object can be a data member of a different class.**

- For example, a data member of the Student class could be an object named: ***_birthday***
- ***_birthday*** could be an object of a class named: ***Date***
- Add the following *Date* class to your project containing the *Student* class.

```
// ==== Date.h ==========

#include<iostream>
#include<string>
using namespace std;
class Date
{
private:
     int month;
     int day;
     int year;
public:
     Date()                    // Default Constructor
```

Note:

Usually, programmers create a header file for the class specification (like Cat.h), and then a .cpp file for the function definitions (like Cat.cpp).

However, the .cpp file is not required. This *Date* class does not have a separate *Date.cpp* file, because the function definitions are included in the header file. This is not usually done, but to save space in the lecture notes, it's done here.

```
     {
          month = 0;
          day = 0;
          year = 0;
     }
     // ------------------------

     // ------------------------
     ~Date(){};               // Destructor
     // ------------------------

     // ------------------------
     void displayDate()
     {
          cout << month << "/"
               << day << "/"
               << year << "\n\n";
     }
     // ------------------------

     // ------------------------
     void setDate(int month,int day,int year)
     {
          this->month = month;
          this->day = day;
          this->year = year;
     }
};
// =========================================================

// ==== Student.h ========================================

#include "Date.h"

class Student
{
private:
     int    id;
     string name;
     float  gpa;
     Date  birthday;
public:
     Student();
     Student(int id, string name,float gpa
                    ,int month,int day,int year);
     ~Student();
```

```
     void Student::displayRecord();
     void Student::setRecord(int id, string name,float gpa
                                   ,int month,int day,int year);
};
// ========================================================

// ==== Student.cpp ========================================
#include "Student.h"

Student::Student( )      // Default Constructor
{
     id = 0;
     name = "";
     gpa = 0.0;
     birthday.setDate(0,0,0);
}
// --------------------------------

// --------------------------------
Student::~Student(){}
// --------------------------------

// --------------------------------
void Student::setRecord(int id, string name, float gpa, int month
                              , int day, int year)
{
     this->id = id;
     this->name = name;
     this->gpa = gpa;
     this->birthday.setDate(month, day, year);
}
// --------------------------------

// --------------------------------
void Student::displayRecord()
{
     cout << "NAME:  " << name << "\nID:  " << _id
          << "\nGPA:  " << gpa << endl;

     cout << "DATE OF BIRTH:  ";
     birthday.displayDate();
}
// ========================================================

// ==== main.cpp ===========================================
```

```
#include "Student.h"

int main( )
{
     int month = day = year = 0;
     int id = 100;
     string name = "Tom Lee";
     float gpa = 3.5;

     Student s1;
     Student s2(1001, "Tom Lee", 3.55, 4, 12, 1988);

     cout << "Here is S1 showing the default constructor:\n";
     s1.displayRecord();

     cout << "Enter s1's Record:\n\n";
     cout << "ID:  ";
     cin >> id;
     cout << "NAME:  ";
     cin.ignore();
     getline(cin, name);
     cout << "GPA:  ";
     cin >> gpa;
     cout << "\nEnter s1's birthday (month, day, year)\n";
     cout << "MONTH:  ";
     cin >> month;
     cout << "DAY:  ";
     cin >> day;
     cout << "YEAR:  ";
     cin >> year;

     s1.setRecord(id,name,gpa, month, day, year);

     cout << "\n\nHere is S1 after the s1.setRecord:\n";
     s1.displayRecord();

     cout << "\nHere is S2 showing the overloaded constructor:\n";
     s2.displayRecord();

     return 0;
}
```

NOTE: Declaring multiple variables on one line is bad coding style, but it works. It is done here to save space, but generally, it's not good.

UML - Unified Modeling Language

UML - UML provides a standard method for graphically depicting an object-oriented system.

UML Diagrams:

- Case diagram
- Interaction diagram
- State diagram
- Activity diagram
- **Class diagram**

- **UML 2.0** - UML 1.0 was released in 1997.
  UML 2.0 is now the current release.

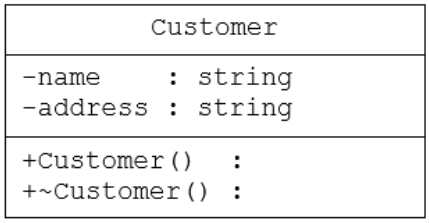
UML Diagrams - UML provides a set of standard diagrams for graphically depicting object-oriented systems.

- UML provides several different types of diagrams for depicting different facets of an information system.

Class diagram - Class diagrams are widely used to describe the types of objects in a system and their relationships.

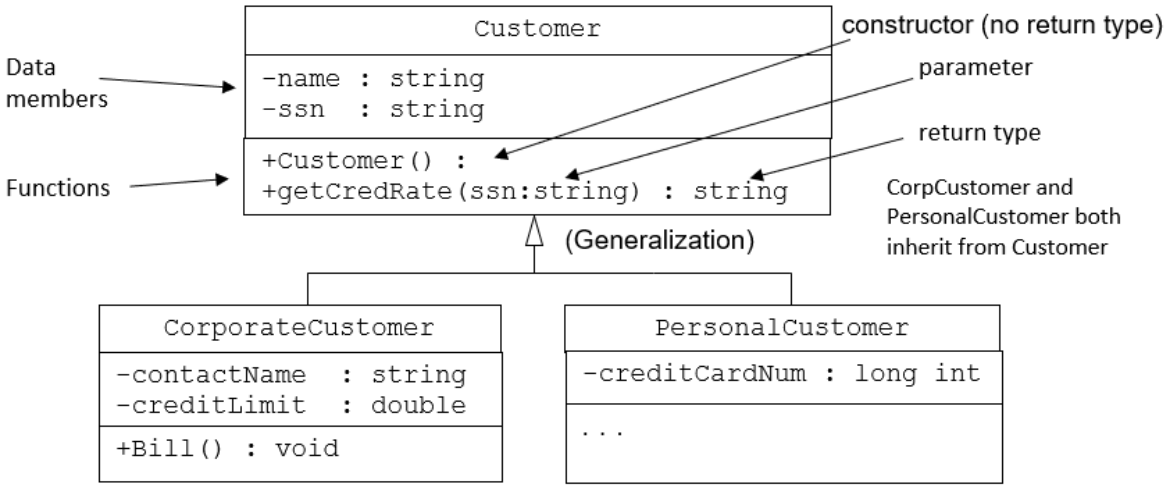
- UML 2 class diagrams show the classes of the system, their interrelationships and the operations and attributes of the classes.

- A class diagram shows 3 things:
  - Class name
  - Attributes (data members)
  - Operations (functions)



- means private
means protected
+ means public

  - **Generalization** - A generalization is used when two classes are similar, but have some differences.
    - Generalization diagram shows **inheritance** relationships. (see next page)



- Both CorporateCustomer and PersonalCustomer classes inherit from the Customer class.

- The Customer class is a more general type of customer, and CorporateCustomer and PersonalCustomer are more specific types of customers.

Arrays and Class Objects

- An array can hold objects of a class.

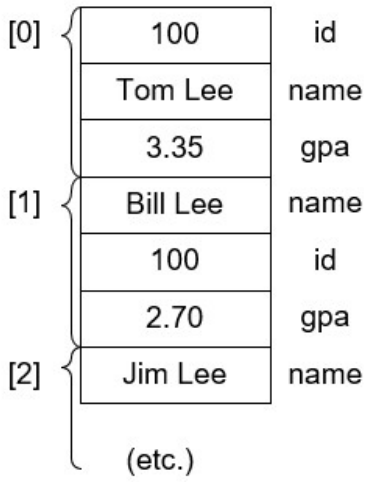
  Ex: Assume a **class Student** has been declared.

  - Each object has an **id**, **name**, and **gpa**.

    ```
    // In main(), declare an array of Student objects
    Student students[SIZE];
    ```

  - To access an object's data member, use a class method.

    ```
    students[i].getName();
    ```



Copy constructor - A copy constructor is a special constructor that creates a new object from an existing object.

- If you do not define a copy constructor, the compiler will generate one that performs a shallow copy of the existing object's member variables.

- **A copy constructor gets called when:**
  - An object is passed to a function by value or returned by value

```
Person p1("Kathy Jones",25);
Person p2 = p1;
```

- In the above case, a copy constructor is called when a new object (*p2*) is created, but the object is initialized with another object's (*p1*) data (see example below).

```
int main()
{
     Person p1("Kathy Jones", 25);
     Person p2 = p1;                    ← Problem
```

Problem: The constructor for Person *p2* is not called. Instead, a default copy constructor is called (even if a copy constructor has not been explicitly defined in the class specification).

- Therefore, a separate section of memory is not allocated for *p2*.
- Both object's name members point to the same location ("Kathy Jones").
- However, an instance of a class owns its own pointers. Therefore, p1 has its own pointers but p2 simply has a copy of p1's.
  - If two pointers call delete on the same pointer, heap corruption results.

Solution: Include a **copy constructor** in the class specification (see below).

```
//  File:  Person.h
```

```
#include <iostream>
#include <string>

using namespace std;

class Person
{
private:
     string name;
     int age;
public:
     // Constructor
     Person(string name, int age)
     {
        this->name = name;
        this->age = age;
     }
     // -------------------------------
```

```
    // Copy Constructor
```

Note: A copy constructor has the same form as a regular constructor, except a reference parameter of the same class as the object is included.

```
    Person(Person & obj)
    {
        this->name = obj.name;
        this->age = obj.age;
    }

    // ------------------------------
    . . . (other class functions)
};

  int main()
  {
     Person p1("Kathy Jones", 25);
     Person p2 = p1;
```

← No Problem now, because the copy constructor gets called.

Creating Objects on the Free Store

- A pointer to a class object can also be created on the **free store**,
- **new operator** - The *new* operator dynamically allocates memory on the heap (free store)

  Ex: `Cat *ptr = new Cat;`

- **delete operator** - The *delete* operator de-allocates memory on the heap (free store)
  - This statement deletes the *Cat* object, not the pointer (ptr)

  Ex: `delete *ptrCat;`

Memory Management - When objects are allocated dynamically on the heap, they will remain in memory until de-allocated.

Memory leak - When dynamically-allocated objects are not deleted, then the amount of available memory (RAM) is reduced. To prevent memory leak, memory must be managed.

Memory management

Memory management is a form of resource management applied to computer memory. The essential requirement of memory management is to provide ways to dynamically allocate portions of memory to programs at their request, and free it for reuse when no longer needed. This is critical to any advanced computer system where more than a single process might be underway at any time.

Virtual memory

Virtual memory is one method used to increase the effectiveness of memory management. It is a method of decoupling the memory organization from the physical hardware. Virtual memory systems separate the memory addresses used by a process from actual physical addresses, allowing separation of processes and increasing the size of the virtual address space beyond the available amount of RAM using paging or swapping to secondary storage. The quality of the virtual memory manager can have an extensive effect on overall system performance.

Application-level memory management is generally categorized as either:

- **Automatic memory management**, usually involving **garbage collection**, or,
- **Manual memory management**.

Manual memory management

When a program contains class objects that have been dynamically allocated using the new operator, a programmer can manage memory by including the delete operator in a class destructor. When an object goes out of scope, the destructor is called and the delete operator deallocates the object's memory.

Garbage collection

- Garbage collection is a strategy for automatically detecting memory allocated to objects that are no longer usable in a program, and returning that allocated memory to a pool of free memory locations. This method is in contrast to "manual" memory management where a programmer explicitly codes memory requests and memory releases in the program.
- Automatic garbage collection **advantages**:
  - Reduces programmer workload
  - Prevents certain kinds of memory allocation bugs
- Garbage collection **disadvantage**:
  - Requires memory resources of its own, and may take a significant proportion of total processing time in a program and, as a result, can have significant influence on performance.

Reference counting (Lifetime management) - **A type of garbage collection**.

Reference counting is a specific type of garbage collection that stores the number of references or pointers to an object. Reference counting uses reference counts to deallocate objects which are no longer referenced. As a collection algorithm, reference counting keeps track of the number of references to it. If an object's reference count reaches zero, the object has become inaccessible, and can be destroyed.

When an object is destroyed, any objects referenced by that object also have their reference counts decreased. Because of this, removing a single reference can potentially lead to a large number of objects being freed. A common modification allows reference counting to be made incremental: instead of destroying an object as soon as its reference count becomes zero, it is

added to a list of unreferenced objects, and periodically (or as needed) one or more items from this list are destroyed.

Simple reference counts require frequent updates. Whenever a reference is destroyed or overwritten, the reference count of the object it references is decremented, and whenever one is created or copied, the reference count of the object it references is incremented.

Passing by Reference

Note: If an object is passed to a function by value - A copy of the object is made.

- Each time an object is returned from a function by value, another copy is made.
- A constructor is called at the time of the function call, and a destructor is called at the end of the function.
- The size of a user-created object on the **stack** is the sum of each of its data member variables and member function parameters and local variables.
- With large objects, runtime performance suffers.

Therefore, objects should be passed by reference to functions.

const Member Functions

Class functions can be declared *constant*

- A function declared **const** won't change the value of any data members of the class.
- To declare a class function **constant**, put the keyword **const** (in the prototype) after the parenthesis but before the semicolon.

  Ex: `void displayResults()const;`

- *Cat* class has one **mutator** function: `void setAge(int anAge);`
- *Cat* class has one **accessor** function: `int getAge();`

Note: **setAge()** cannot be **const** because it changes the member variables **itsAge**.

getAge() can and should be **const** because it doesn't change the class at all.

- **getAge()** simply returns the current value of the member variable **itsAge**.
  - Therefore, the declaration of these

functions should be written like:

```
void setAge(int anAge);
int getAge() const;
```

- Use **const** whenever possible.

ifndef

Ex #7: The following is part of a class **Rectangle** specification.

- It is good coding style to use preprocessor directives to declare a named constant for each class.
- The name should be the same as the class name, but should be all uppercase, (as all constants should be).

```
// ===============================================================
// File:  Rectangle.h   -   Rectangle class specification
// ===============================================================

#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
private:
   . . .
public:
   . . .
};

#endif
```

This directive tells the preprocessor to determine whether a constant named RECTANGLE_H has been previously declared with a #define directive.

This line declares the RECTANGLE_H constant.

If the RECTANGLE_H constant has not been declared, these lines are included in the program.

Otherwise, these lines are not included.