Constant Member Functions

- Does not modify its calling object
- Declared with keyword `const`
- When `const` appears in the parameter list, e.g.,
  ```cpp
  int setNum (const int num)
  ```
  the function is prevented from modifying the parameter. The parameter is read-only.
- When `const` follows the parameter list, e.g.,
  ```cpp
  int getX() const
  ```
  the function is prevented from modifying the object.
Static Members

- **Static member variable:**
  - One instance of variable for the entire class
  - Shared by all objects of the class

- **Static member function:**
  - Can be used to access static member variables
  - Can be called before any class objects are created
Static Member Variables

1) Must be declared in the class with keyword `static`:

```cpp
class IntVal {
    public:
        IntVal(int val = 0) {
            value = val;
            valCount++
        }
        int getVal();
        void setVal(int);
    private:
        int value;
        static int valCount;
};
```
2) Must be defined outside of the class:

```cpp
class IntVal
{
    //In-class declaration
    static int valCount;

    //Other members not shown
}

//Definition outside of class
int IntVal::valCount = 0;
```
3) Can be accessed or modified by any object of the class: Modifications by one object are visible to all objects of the class:

```c
IntVal val1, val2;
```
1) Declared with `static` before return type:

```cpp
class IntVal
{
    public:
        static int getValCount()
        {
            return valCount;
        }

    private:
        int value;
        static int valCount;
};
```
Static Member Functions

2) Can be called independently of class objects, through the class name:

\[
\text{cout} \ll \text{IntVal::getValCount}();
\]

3) Because of item 2 above, the this pointer cannot be used

4) Can be called before any objects of the class have been created

5) Used primarily to manipulate static member variables of the class
Friends of Classes

- **Friend function**: a function that is not a member of a class, but has access to private members of the class
- A friend function can be a stand-alone function or a member function of another class
- It is declared a friend of a class with the `friend` keyword in the function prototype
1) Friend function may be a stand-alone function:

```cpp
class Sample {
private:
    int x;
    friend void fSet(Sample &s, int a);
};

void fSet(Sample &s, int a) {
    s.x = a;
}
```
2) Friend function may be a member of another class:

class aClass
{
private:
    int x;
    friend void OtherClass::fSet(aClass &c, int a);
};

class OtherClass
{
public:
    void fSet(aClass &c, int a)
    {
        c.x = a;
    }
};
Friend Class Declaration

3) An entire class can be declared a friend of a class:

class aClass    {
   private:
      int x;
      friend class frClass;
   };

class frClass    {
   public:
      void fSet(aClass &c, int a) {
         c.x = a;
      }
      int fGet(aClass c) {
         return c.x;
      }
   };

Friend Class Declaration

• If FriendClass is a friend of ClassA, then all member functions of FriendClass have unrestricted access to all members of ClassA, including the private members.

• In general, restrict the property of Friendship to only those functions that must have access to the private members of a class.
Aggregation and Composition

- **Class aggregation**: An object of one class owns an object of another class.

- **Class composition**: A form of aggregation where the enclosing class controls the lifetime of the objects of the enclosed class.

- Supports the modeling of ‘has-a’ relationship between classes – enclosing class ‘has a(n)’ instance of the enclosed class.
Object Composition

• Occurs when an object is a member variable of another object.
• It is often used to design complex objects whose members are simpler objects.
• Example: Define a Course class. Then, define a Transcript class and use a Course object as a member of a Transcript object.
Object Composition

• Example: Define a Name class, a Date class, and an Address, then define a Person class and use a Name class as a member of the class, a Date object for the dob member, an Address object for the address member of the Person class.
Example of Object Composition

class MyName
{
    private:
        string firstName;
        string middle;
        string lastName;
}

class MyDate
{
    private:
        int month;
        int day;
        int year;
        ...
};
Example of Object Composition

class MyAddress
{
    private:
        string street;
        string city;
        string state;
        string zip;
    ...
};
Example of Object Composition

class Person
{
    private:
        MyName name;
        MyDate birthday;
        MyAddress address;
    
    ...

};
Another Example of Object Composition

class StudentInfo
{
   private:
      string firstName, LastName;
      string address, city, state, zip;
   ...
};

class Student
{
   private:
      StudentInfo personalData;
   ...
};
Member Initialization Lists

• Used in constructors for classes involved in aggregation.

• Allows constructor for enclosing class to pass arguments to the constructor of the enclosed class.

• Notation:

  owner_class(parameters):owned_class(parameters);
Use:

```cpp
class StudentInfo {

};

class Student {

private:
    StudentInfo personalData;

public:
    Student(string fname, lname):
        StudentInfo(fname, lname);

};
```
Member Initialization Lists

• Member Initialization lists can be used to simplify the coding of constructors
• Should keep the entries in the initialization list in the same order as they are declared in the class
Aggregation Through Pointers

• A ‘has-a’ relationship can be implemented by owning a pointer to an object

• Can be used when multiple objects of a class may ‘have’ the same attribute for a member
  – ex: students who may have the same city/state/ zipcode

• Using pointers minimizes data duplication and saves space
Aggregation, Composition, and Object Lifetimes

• Aggregation represents the owner/owned relationship between objects.
• Composition is a form of aggregation in which the lifetime of the owned object is the same as that of the owner object.
• Owned object is usually created as part of the owning object’s constructor, destroyed as part of owning object’s destructor.